Institute for Advanced Computing And Software Development (IACSD)
Akurdi, Pune

MS.Net Technologies

Dr. D.Y. Patil Educational Complex, Sector 29, Behind Akurdi Railway Station,
Nigdi Pradhikaran, Akurdi, Pune - 411044.

1. Introduction to .NET Framework

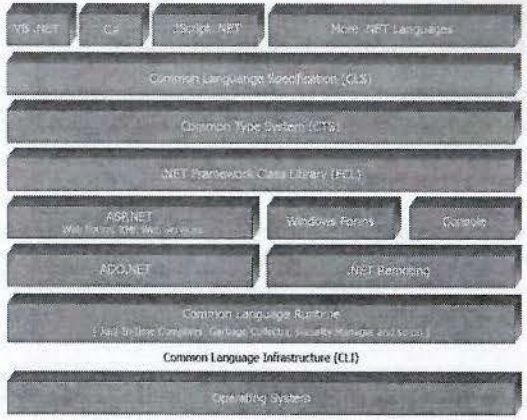
1.1 What is .NET?

.NET is a software framework that is designed and developed by Microsoft. The first version of the .Net framework was 1.0 which came in the year 2002. In easy words, it is a virtual machine for compiling and executing programs written in different languages like C#, VB.Net, etc.

It is used to develop Form-based applications, Web-based applications, and Web services. There is a variety of programming languages available on the .Net platform, VB.Net and C# being the most common ones. It is used to build applications for Windows, phones, web, etc. It provides a lot of functionalities and also supports industry standards.

.NET Framework supports more than 60 programming languages of which 11 programming languages are designed and developed by Microsoft. The remaining Non-Microsoft Languages are supported by .NET Framework but not designed and developed by Microsoft.

Diagram :NET Framework



Architecture of .NET Framework

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

- The **Common Language Runtime (CLR)** is the execution engine that handles running applications. It provides services like thread management, garbage collection, type-safety, exception handling, and more.

- The **Class Library** provides a set of APIs and types for common functionality. It provides types for strings, dates, numbers, etc. The Class Library includes APIs for reading and writing files, connecting to databases, drawing, and more.

.NET applications are written in the C#, F#, or Visual Basic programming language. Code is compiled into a language-agnostic Common Intermediate Language (CIL). Compiled code is stored in assemblies—files with a .dll or .exe file extension.

When an app runs, the CLR takes the assembly and uses a just-in-time compiler (JIT) to turn it into machine code that can execute on the specific architecture of the computer it is running on.

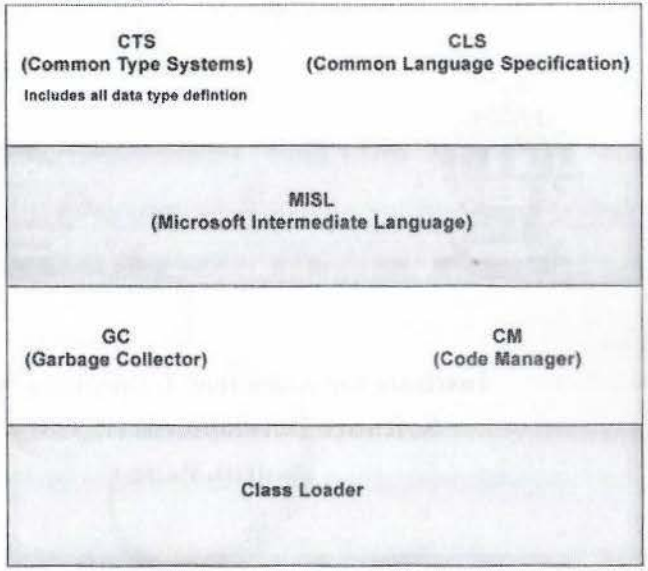
1.2Intermediate Language (IL):

Intermediate language (IL) is an object-oriented programming language designed to be used by compilers for the .NET Framework before static or dynamic compilation to machine code. The IL is used by the .NET Framework to generate machine-independent code as the output of compilation of the source code written in any .NET programming language.

IL is a stack-based assembly language that gets converted to bytecode during execution of a virtual machine. It is defined by the common language infrastructure (CLI) specification. As IL is used for automatic generation of compiled code, there is no need to learn its syntax.

This term is also known as Microsoft intermediate language (MSIL) or common intermediate language (CIL).

Diagram : Intermediate Language



(1)Common Type System (CTS):

Common type system defines how types are declared, used and managed in the common language runtime. It also plays and important part in cross language integration. CTS perform the following function as follows:

- Define rules that language must follow that help to ensure that object written different languages can interact each other.
- It ensure cross language integration, type safety.
- It helps to trace syntax and compile time error.
- Also improve code execution.

(2)Garbage Collector:

Garbage collector is most important part of DOT Net framework. As memory increase over the time programs size also tends to improve. It is always important to keep memory free for additional program to execute. Garbage collector is like demon program that have low priority and always runs in background. Main job as name suggest garbage collector. It check in the memory when any object

do not have any reference in runtime it will remove that object from the memory and free up some space.
Not only has that but garbage collector also taken care when user exit the application it will remove that program from the memory. So user can have memory for other program that need to be executed.

(3)MSIL (Microsoft Intermediate Language):
Before Going to the concept of MSIL you must know what the differences between Native Language and MSIL are:

a)Native Language:
Language that understood by targeting operating system to execute that program on machine known as native code. This code conversion is called **code compilation.** This code compilation is done by compiler. In DOT NET Frame work it is 2 stage processes. Fist program code get converted into MSIL And then compile by Just In Time Compiler.

b)MSIL
For security reason and language portability DOT framework convert the code into intermediate language called MSIL. That can be then converted to machine level language by Just In Time Compiler (JIT). Since Common language runtime provide many just in time compiler by using which it convert MSIL to machine level code.

(4)Just In Time Compiler:
Just time compiler performs compilation job that convert MSIL to operating system (OS) specific code to run it on target machine. Only at this point operating system can execute the program on targeted machine.
Just In Time as the name suggest this compiler compile code each time and convert MSIL to native code.

(5)Code Manager:
Code manger invokes class loader for execution.
There are two kinds of coding style supported by DOT Net framework.

a)**Managed Code**:
Managed code is the code that control and compiled by CLR. Any language that managed within DOT Net framework it is managed code. Advantage of manageable code CLR Have control over code so it can take an advantage of CLR functionality like managing memory, handling security, allowing across, language interportability.

b)**UnManaged Code**:
When code developed outside the .Net framework is called as UN managed code. Application that do not run under complete control of CLR. They cannot use advantage and functionality offered by CLR.

1.3Assemblies in .NET

Assemblies are the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications. An assembly is a collection of types and resources that are built to work together 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. They provide the common language runtime with the information it needs to be aware of type implementations.
In .NET and .NET Framework, you can build an assembly from one or more source code files. In .NET Framework, assemblies can contain one or more modules. This way, larger projects can be planned so that several developers can work on separate source code files or modules, which are combined to create a single assembly. For more information about modules.

Types of Assemblies of in .NET Framework:
In the .NET Framework, there are two types of assemblies.
They are as follows:
1.EXE (Executable)
2.DLL (Dynamic Link Library)
In .NET Framework when we compile a Console Application or a Windows Application, it generates EXE, whereas when we compile a Class Library Project or ASP.NET web application, then it generates DLL. In.NET framework, both EXE and DLL are called assemblies

What is a .Net Assembly?

The .NET assembly is the standard for components developed with the Microsoft.NET. Dot NET assemblies may or may not be executable, i.e., they might exist as the executable (.exe) file or dynamic link library (DLL) file. All the .NET assemblies contain the definition of types, versioning information for the type, meta-data, and manifest. The designers of .NET have worked a lot on the component (assembly) resolution.

An assembly can be a single file or it may consist of the multiple files. In the case of multi-file, there is one master module containing the manifest while other assemblies exist as non-manifest modules. A module in .NET is a subpart of a multi-file .NET assembly. Assembly is one of the most interesting and extremely useful areas of .NET architecture along with reflections and attributes.

.NET supports three kinds of assemblies:
A)private
B)shared
C)satellite

A)Private Assembly

Private assembly requires us to separately in all application folders where we want to use that assembly's functionalities; without in, we cannot access the private assembly features and power. Private assembly means every time we have one, we exclusively into the BIN folder of each application folder.

Public Assembly

Public assembly is not required to separately into all application folders. Public assembly is also called Shared Assembly. Only one is required in system level, there is no need to the assembly into the application folder.
Public assembly should install in GAC.
Shared assemblies (also called strong named assemblies) are copied to a single location (usually the Global assembly cache). For all calling assemblies within the same application, the same of the shared assembly is used from its original location. Hence, shared assemblies are not copied in the private folders of each calling assembly. Each shared assembly has a four-part name including its face name, version, public key token, and culture information. The public key token and version information makes it almost impossible for two different assemblies with the same name or for two similar assemblies with a different version to mix with each other.

GAC (Global Assembly Cache)

When the assembly is required for more than one project or application, we need to make the assembly with a strong name and keep it in GAC or in the Assembly folder by installing the assembly with the Actual command.

B)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 multiple copies will not be created even when used by multiple applications. The GAC folder is under the Windows folder.

C)Satellite Assembly:
Satellite assemblies are used for deploying language and culture-specific resources for an application.

1.4 CLR and its Functions

.NET CLR is a runtime environment that manages and executes the code written in any .NET programming language. CLR is the virtual machine component of the .NET framework. That language's compiler compiles the source code of applications developed using .NET compliant languages into CLR's intermediate language called MSIL, i.e., Microsoft intermediate language code. This code is platform-independent. It is comparable to byte code in java. Metadata is also generated during compilation and MSIL code and stored in a file known as the Manifest file. This metadata is generally about members and types required by CLR to execute MSIL code. A just-in-time compiler component of CLR converts MSIL code into native code of the machine. This code is platform-dependent. CLR manages memory, threads, exceptions, code execution, code safety, verification, and compilation.

Following are the functions of the CLR.

- It converts the program into native code.
- Handles Exceptions
- Provides type-safety
- Memory management
- Provides security
- Improved performance
- Language independent
- Platform independent
- Garbage collection
- Provides language features such as inheritance, interfaces, and overloading for object-oriented programs.

The code that runs with CLR is called managed code, whereas the code outside the CLR is called unmanaged code. The CLR also provides an Interoperability layer, which allows both the managed and unmanaged codes to interoperate.

(A)The main components of CLR are:

a)Common type system

b)Common language speciation

c)Garbage Collector

d)Just in Time Compiler

e)Metadata and Assemblies

(A)JIT Compilation:

Just-In-Time compiler (JIT) is a part of Common Language Runtime (CLR) in .NET which is responsible for managing the execution of .NET programs regardless of any .NET programming language. A language-specific compiler converts the source code to the intermediate language. This intermediate language is then converted into the machine code by the Just-In-Time (JIT) compiler. This machine code is specific to the computer environment that the JIT compiler runs on.

Working of JIT Compiler: The JIT compiler is required to speed up the code execution and provide support for multiple platforms. Its working is given as follows:

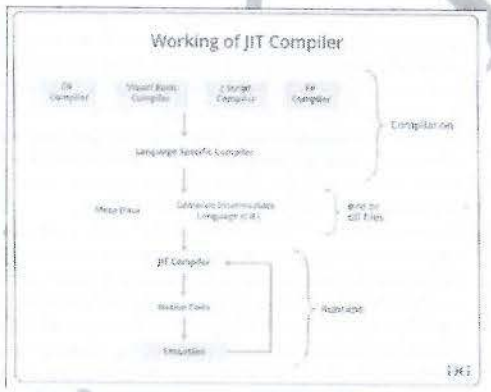


Diagram: Working of JIT Compiler

The JIT compiler converts the Microsoft Intermediate Language (MSIL) or Common Intermediate Language (CIL) into the machine code. This is done before the MSIL or CIL can be executed. The MSIL is converted into machine code on a requirement basis i.e. the JIT compiler compiles the MSIL or CIL as required rather than the whole of it. The compiled MSIL or CIL is stored so that it is available for subsequent calls if required.

Types of Just-In-Time Compiler:

There are 3 types of JIT compilers which are as follows:

(A)Pre-JIT Compiler: All the source code is compiled into the machine code at the same time in a single compilation cycle using the Pre-JIT Compiler. This compilation process is performed at application deployment time. And this compiler is always implemented in the Ngen.exe (Native Image Generator).

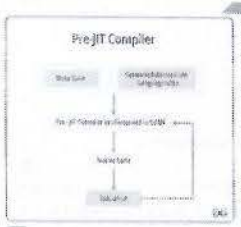


Diagram: Pre-JIT Compiler

(B)Normal JIT Compiler: The source code methods that are required at run-time are compiled into machine code the first time they are called by the Normal JIT Compiler. After that, they are stored in the cache and used whenever they are called again.

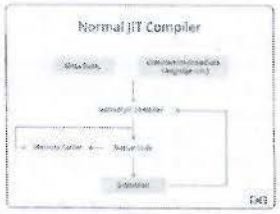
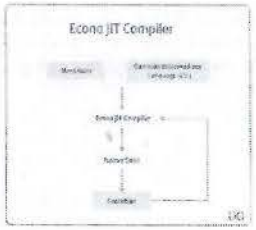


Diagram : Normal JIT Compiler

(C)Econo JIT Compiler: The source code methods that are required at run-time are compiled into machine code by the Econo JIT Compiler. After these methods are not required anymore, they are removed. This JIT compiler is obsolete starting from dotnet 2.0



Advantages of JIT Compiler:

- The JIT compiler requires less memory usage as only the methods that are required at run-time are compiled into machine code by the JIT Compiler.
- Page faults are reduced by using the JIT compiler as the methods required together are most probably in the same memory page.
- Code optimization based on statistical analysis can be performed by the JIT compiler while the code is running.

Disadvantages of JIT compiler:

- The JIT compiler requires more startup time while the application is executed initially.
- The cache memory is heavily used by the JIT compiler to store the source code methods that are required at run-time.

(B)Memory Management:

Automatic memory management is made possible by Garbage Collection in .NET Framework. When a class object is created at runtime, certain memory space is allocated to it in the heap memory.

Memory allocation

1. Garbage Collector (GC) is the part of the .NET framework that allocates and releases memory for your .NET applications.
2. When a new process is started, the runtime reserves a region of address space for the process called the managed heap.
3. Objects are allocated in the heap contiguously one after another.
4. Memory allocation is a very fast process as it is just the adding of a value to a pointer.

5. In addition to the managed heap, an app always consumes some amount of so-called unmanaged* memory which is not managed by GC.

(C)Appdomain

1. Application domains provide a unit of isolation for the common language runtime. They are created and run inside a process. Application domains are usually created by a runtime host, which is an application responsible for loading the runtime into a process and executing user code within an application domain.
2. An application domain is a .NET concept whereas the process is an operating system concept. Both have many characteristics in common. A process can contain multiple app domains and each one provides a unit of isolation within that process.
3. Application Domain or Appdomain is one of the most powerful features of the .NET framework. Appdomain can be considered as a light-weight process. An application domain is a .NET concept whereas the process is an operating system concept. Both have many characteristics in common. A process can contain multiple app domains and each one provides a unit of isolation within that process. Most importantly, only those applications are written in .net, or in other words, only managed applications will have application domain.
4. Application domain provides isolation between code running in different app domains. App Domain is a logical container for code and data just like process and has separate memory space and access to resources. App domain also serves as a boundary like a process that does avoid any accidental or illegal attempts to access the data of an object in one running application from another.
5. System.Appdomain class provides you ways to deal with the application domain. It provides methods to create a new application domain, unload domain from memory, etc.
6. A Process is much heavier and expensive to create and maintain. So, the server would have a tough time managing the processes. So, when Microsoft developed the .Net framework, they introduced the concept of Application Domain and it is an integral part of the .net framework.
7. While Appdomains still offer many of the features that a Windows process offers, they are more efficient than a process by enabling multiple assemblies to be run in separate application domains without the overhead of launching separate processes. An App Domain is relatively cheaper when compared to the process to create, and has relatively less overhead to maintain.

(D)Security in .NET

The common language runtime and .NET provide many useful classes and services that enable developers to write secure code, use cryptography, and implement role-based security.

A) Key Security Concepts :

.NET offers role-based security to help address security concerns about mobile code and to provide support that enables components to determine what users are authorized to do.

Type safety and security :

Type-safe code accesses only the memory locations it is authorized to access. (For this discussion, type safety specifically refers to memory type safety and should not be confused with type safety in a broader respect.) For example, type-safe code cannot read values from another object's private fields. It accesses types only in well-defined, allowable ways.

During just-in-time (JIT) compilation, an optional verification process examines the metadata and Microsoft intermediate language (MSIL) of a method to be JIT-compiled into native machine code to verify that they are type safe. This process is skipped if the code has permission to bypass verification. For more information about verification, see Managed Execution Process.

Although verification of type safety is not mandatory to run managed code, type safety plays a crucial role in assembly isolation and security enforcement. When code is type safe, the common language runtime can completely isolate assemblies from each other. This isolation helps ensure that assemblies cannot adversely affect each other and it increases application reliability. Type-safe components can execute safely in the same process even if they are trusted at different levels. When code is not type safe, unwanted side effects can occur. For example, the runtime cannot prevent managed code from calling into native (unmanaged) code and performing malicious operations. When code is type safe, the runtime's security enforcement mechanism ensures that it does not access native code unless it has permission to do so. All code that is not type safe must have been granted Security permission with the passed enum member skip verification to run.

B) Role-Based Security:

Roles are often used in financial or business applications to enforce policy. For example, an application might impose limits on the size of the transaction being processed depending on whether the user making the request is a member of a specified role. Clerks might have authorization to process transactions that are less than a specified threshold, supervisors might have a higher limit, and vice-presidents might have a still higher limit (or no limit at all). Role-based security can also be

used when an application requires multiple approvals to complete an action. Such a case might be a purchasing system in which any employee can generate a purchase request, but only a purchasing agent can convert that request into a purchase order that can be sent to a supplier.

To provide ease of use and consistency with code access security, .NET role-based security provides System.Security.Permissions.PrincipalPermission objects that enable the common language runtime to perform authorization in a way that is similar to code access security checks.
The PrincipalPermission class represents the identity or role that the principal must match and is compatible with both declarative and imperative security checks. You can also access a principal's identity information directly and perform role and identity checks in your code when needed.

.NET provides role-based security support that is flexible and extensible enough to meet the needs of a wide spectrum of applications. You can choose to interoperate with existing authentication infrastructures, such as COM+ 1.0 Services, or to create a custom authentication system. Role-based security is particularly well-suited for use in ASP.NET Web applications, which are processed primarily on the server. However, .NET role-based security can be used on either the client or the server.

The principal object represents the security context under which code is running. Applications that implement role-based security grant rights based on the role associated with a principal object. Similar to identity objects, .NET provides a GenericPrincipal object and a WindowsPrincipal object. You can also define your own custom principal classes.

The IPrincipal interface defines a property for accessing an associated **Identity** object as well as a method for determining whether the user identified by the **Principal** object is a member of a given role. All **Principal** classes implement the **IPrincipal** interface as well as any additional properties and methods that are necessary. For example, the common language runtime provides the **WindowsPrincipal** class, which implements additional functionality for mapping group membership to roles.

C) Cryptography Model:

.NET provides implementations of many standard cryptographic algorithms, and the .NET cryptography model is extensible.

Object inheritance

The .NET cryptography system implements an extensible pattern of derived class inheritance. The hierarchy is as follows:

Algorithm type class, such as SymmetricAlgorithm, AsymmetricAlgorithm, or HashAlgorithm. This level is abstract.

Algorithm class that inherits from an algorithm type class; for example, Aes, RSA, or ECDiffieHellman. This level is abstract.

Implementation of an algorithm class that inherits from an algorithm class; for example, AesManaged, RC2CryptoServiceProvider, or ECDiffieHellmanCng. This level is fully implemented.

This pattern of derived classes lets you add a new algorithm or a new implementation of an existing algorithm. For example, to create a new public-key algorithm, you would inherit from the AsymmetricAlgorithm class. To create a new implementation of a specific algorithm, you would create a non-abstract derived class of that algorithm.

Introduction to cryptography

Public networks such as the Internet do not provide a means of secure communication between entities. Communication over such networks is susceptible to being read or even modified by unauthorized third parties. Cryptography helps protect data from being viewed, provides ways to detect whether data has been modified, and helps provide a secure means of communication over otherwise nonsecure channels. For example, data can be encrypted by using a cryptographic algorithm, transmitted in an encrypted state, and later decrypted by the intended party. If a third party intercepts the encrypted data, it will be difficult to decipher.

D)Secure Coding Guidelines:

(A)Securing resource access :

When designing and writing your code, you need to protect and limit the access that code has to resources, especially when using or invoking code of unknown origin. So, keep in mind the following techniques to ensure your code is secure:

1. Do not use Code Access Security (CAS).
2. Do not use partial trusted code.
3. Do not use the AllowPartiallyTrustedCaller attribute (APTCA).
4. Do not use .NET Remoting.
5. Do not use Distributed Component Object Model (DCOM).
6. Do not use binary formatters.

Code Access Security and Security-Transparent Code are not supported as a security boundary with partially trusted code. We advise against loading and executing code of unknown origins without putting alternative security measures in place.

The alternative security measures are:

- Virtualization
- AppContainers
- Operating system (OS) users and permissions
- Hyper-V containers

Securing State Data

1. Applications that handle sensitive data or make any kind of security decisions need to keep that data under their own control and cannot allow other potentially malicious code to access the data directly. The best way to protect data in memory is to declare the data as private or internal (with scope limited to the same assembly) variables. However, even this data is subject to access you should be aware of:
2. Using reflection mechanisms, highly trusted code that can reference your object can get and set private members.
3. Using serialization, highly trusted code can effectively get and set private members if it can access the corresponding data in the serialized form of the object.
4. Under debugging, this data can be read.
5. Make sure none of your own methods or properties exposes these values unintentionally.

Security and On-the-Fly Code Generation

- Some libraries operate by generating code and running it to perform some operation for the caller. The basic problem is generating code on behalf of lesser-trust code and running it at a higher trust. The problem worsens when the caller can influence code generation, so you must ensure that only code you consider safe is generated.
- You need to know exactly what code you are generating at all times. This means that you must have strict controls on any values that you get from a user, be they quote-enclosed strings (which should be escaped so they cannot include unexpected code elements), identifiers (which should be checked to verify that they are valid identifiers), or anything else. Identifiers can be dangerous because a compiled assembly can be modified so that its identifiers contain

strange characters, which will probably break it (although this is rarely a security vulnerability).

- It is recommended that you generate code with reflection emit, which often helps you avoid many of these problems.
- When you compile the code, consider whether there is some way a malicious program could modify it. Is there a small window of time during which malicious code can change source code on disk before the compiler reads it or before your code loads the .dll file? If so, you must protect the directory containing these files, using an Access Control List in the file system, as appropriate.

Security and User Input:

- User data, which is any kind of input (data from a Web request or URL, input to controls of a Microsoft Windows Forms application, and so on), can adversely influence code because often that data is used directly as parameters to call other code. This situation is analogous to malicious code calling your code with strange parameters, and the same precautions should be taken. User input is actually harder to make safe because there is no stack frame to trace the presence of the potentially untrusted data.
- These are among the subtlest and hardest security bugs to find because, although they can exist in code that is seemingly unrelated to security, they are a gateway to pass bad data through to other code. To look for these bugs, follow any kind of input data, and imagine what the range of possible values might be, and consider whether the code seeing this data can handle all those cases. You can fix these bugs through range checking and rejecting any input the code cannot handle.

- Some important considerations involving user data include the following:

  Any user data in a server response runs in the context of the server's site on the client. If your Web server takes user data and inserts it into the returned Web page, it might, for example, include a **<script>** tag and run as if from the server.

1. Remember that the client can request any URL.
2. Consider tricky or invalid paths:
3. ..\ , extremely long paths.
4. Use of wild card characters (*).
5. Token expansion (%token%).
6. Strange forms of paths with special meaning.

7. Alternate file system stream names such as filename::$DATA.
8. Short versions of file names such as longfi~1 for longfilename.

- Remember that Eval(userdata) can do anything.
- Be wary of late binding to a name that includes some user data.
- If you are dealing with Web data, consider the various forms of escapes that are permissible, including:
  - Hexadecimal escapes (%nn).
  - Unicode escapes (%nnn).
  - Overlong UTF-8 escapes (%nn%nn).
  - Double escapes (%nn becomes %mmnn, where %mm is the escape for '%').
- Be wary of user names that might have more than one canonical format. For example, you can often use either the MYDOMAIN*username* form or the *username*@mydomain.example.com form.

  NET Framework also provides security on code and this is referred to as code access security (also referred to as evidence-based security). With code access security, a user may be trusted to access a resource but if the code the user executes is not trusted, then access to the resource will be denied.

2.NET Framework

.NET is a framework to develop software applications. It is designed and developed by Microsoft and the first beta version released in 2000.

It is used to develop applications for web, Windows, phone. Moreover, it provides a broad range of functionalities and support.

This framework contains a large number of class libraries known as Framework Class Library (FCL). The software programs written in .NET are executed in the execution environment, which is called CLR (Common Language Runtime). These are the core and essential parts of the .NET framework.

This framework provides various services like memory management, networking, security, memory management, and type-safety.

The .Net Framework supports more than 60 programming languages such as C#, F#, VB.NET, J#, VC++, JScript.NET, APL, COBOL, Perl, Oberon, ML, Pascal, Eiffel, Smalltalk, Python, Cobra, ADA, etc.

Following is the .NET framework Stack that shows the modules and components of the Framework.

The .NET Framework is composed of four main components:

- Common Language Runtime (CLR)
- Framework Class Library (FCL),
- Core Languages (WinForms, ASP.NET, and ADO.NET), and
- Other Modules (WCF, WPF, WF, Card Space, LINQ, Entity Framework, Parallel LINQ, Task Parallel Library, etc.)

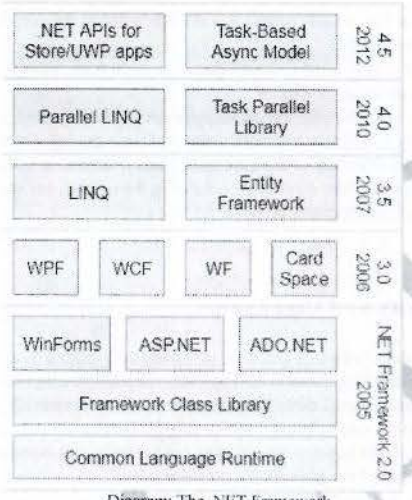
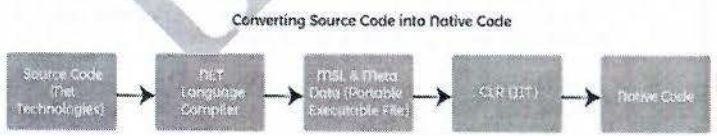


Diagram: The .NET Framework

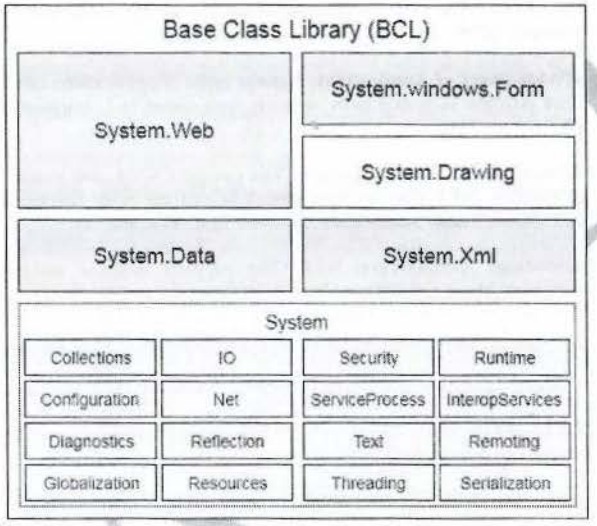
1)CLR (Common Language Runtime)

It is a program execution engine that loads and executes the program. It converts the program into native code. It acts as an interface between the framework and operating system. It does exception handling, memory management, and garbage collection. Moreover, it provides security, type-safety, interoperability, and portability. A list of CLR components are given below:



2) FCL (Framework Class Library)

It is a standard library that is a collection of thousands of classes and used to build an application. The BCL (Base Class Library) is the core of the FCL and provides basic functionalities.



3) WinForms

Windows Forms is a smart client technology for the .NET Framework, a set of managed libraries that simplify common application tasks such as reading and writing to the file system.

4) ASP.NET

ASP.NET is a web framework designed and developed by Microsoft. It is used to develop websites, web applications, and web services. It provides a fantastic integration of HTML, CSS, and JavaScript. It was first released in January 2002.

5) ADO.NET

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

6) WPF (Windows Presentation Foundation)

Windows Presentation Foundation (WPF) is a graphical subsystem by Microsoft for rendering user interfaces in Windows-based applications. WPF, previously known as "Avalon", was initially released as part of .NET Framework 3.0 in 2006. WPF uses DirectX.

7) WCF (Windows Communication Foundation)

It is a framework for building service-oriented applications. Using WCF, you can send data as asynchronous messages from one service endpoint to another.

8) WF (Workflow Foundation)

Windows Workflow Foundation (WF) is a Microsoft technology that provides an API, an in-process workflow engine, and a rehostable designer to implement long-running processes as workflows within .NET applications.

9) LINQ (Language Integrated Query)

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

10) Entity Framework

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

11) Parallel LINQ

Parallel LINQ or PLINQ is a parallel implementation of LINQ to objects. It combines the simplicity and readability of LINQ and provides the power of parallel programming.

It can improve and provide fast speed to execute the LINQ query by using all available computer capabilities.Apart from the above features and libraries, .NET includes other APIs and Model to improve and enhance the .NET framework.

.NET Core Overview

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

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

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

Why .NET Core?

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

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

.NET Core Characteristics

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

You can view, download, or contribute to the source code using the following GitHub repositories:

- Language compiler platform Roslyn: https://github.com/dotnet/roslyn
- .NET Core runtime: https://github.com/dotnet/runtime
- .NET Core SDK repository. https://github.com/dotnet/sdk
- ASP.NET Core repository. https://github.com/dotnet/aspnetcore

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

3)Consistent across Architectures:

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

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

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

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

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

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

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

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

.NET Core Version History

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

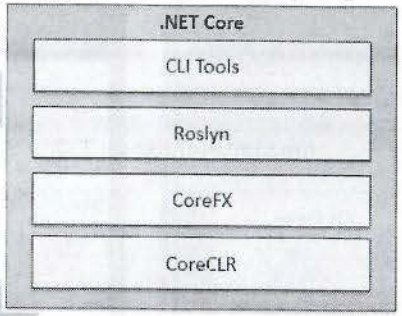
Both, .NET 3.1, and .NET Core 2.1 will have long term support.

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

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

.NET Core Composition

The .NET Core Framework composed of the following parts:



- CLI Tools: A set of tooling for development and deployment.
- Roslyn: Language compiler for C# and Visual Basic
- CoreFX: Set of framework libraries.
- CoreCLR: A JIT based CLR (Command Language Runtime).

Mono

The open source development platform based on the .NET Framework, allows developers to build cross-platform applications with improved developer productivity. Mono's .NET implementation is based on the ECMA standards for C# and the Common Language Infrastructure.

Supported previously by Novell, Xamarin and now Microsoft and the .NET Foundation, the Mono project has an active and enthusiastic contributing community. Mono includes both developer tools and the infrastructure needed to run .NET client and server applications.

The Components

There are several components that make up Mono:

- **C# Compiler** - Mono's C# compiler is feature complete for C# 1.0, 2.0, 3.0, 4.0, 5.0 and 6.0 (ECMA).
- **Mono Runtime** - The runtime implements the ECMA Common Language Infrastructure (CLI). The runtime provides a Just-in-Time (JIT) compiler, an Ahead-of-Time compiler (AOT), a library loader, the garbage collector, a threading system and interoperability functionality.
- **.NET Framework Class Library** - The Mono platform provides a comprehensive set of classes that provide a solid foundation to build applications on. These classes are compatible with Microsoft's .NET Framework classes.
- **Mono Class Library** - Mono also provides many classes that go above and beyond the Base Class Library provided by Microsoft. These provide additional functionality that are useful, especially in building Linux applications. Some examples are classes for Gtk+, Zip files, LDAP, OpenGL, Cairo, POSIX, etc.

Mono Feature Highlights

- Multi-Platform
  Runs on Linux, macOS, BSD, and Microsoft Windows, including x86, x86-64, ARM, s390, PowerPC and much more
- Multi-Language
  Develop in C# 4.0 (including LINQ and `dynamic`), VB 8, Java, Python, Ruby, Eiffel, F#, Oxygene, and more
- Binary Compatible
  Built on an implementation of the ECMA's Common Language Infrastructure and C#

- Microsoft Compatible API
  Run ASP.NET, ADO.NET, Silverlight and Windows.Forms applications without recompilation
- Open Source, Free Software
  Mono's runtime, compilers, and libraries are distributed using the MIT license.
- Comprehensive Technology Coverage
  Bindings and managed implementations of many popular libraries and protocols

There are many benefits to choosing Mono for application development:

- **Popularity** - Built on the success of .NET, there are millions of developers that have experience building applications in C#. There are also tens of thousands of books, websites, tutorials, and example source code to help with any imaginable problem.
- **Higher-Level Programming** - All Mono languages benefit from many features of the runtime, like automatic memory management, reflection, generics, and threading. These features allow you to concentrate on writing your application instead of writing system infrastructure code.
- **Base Class Library** - Having a comprehensive class library provides thousands of built in classes to increase productivity. Need socket code or a hashtable? There's no need to write your own as it's built into the platform.
- **Cross Platform** - Mono is built to be cross platform. Mono runs on Linux, Microsoft Windows, macOS, BSD, and Sun Solaris, Nintendo Wii, Sony PlayStation 3, Apple iPhone and Android. It also runs on x86, x86-64, IA64, PowerPC, SPARC (32), ARM, Alpha, s390, s390x (32 and 64 bits) and more. Developing your application with Mono allows you to run on nearly any computer in existence.
- **Common Language Runtime (CLR)** - The CLR allows you to choose the programming language you like best to work with, and it can interoperate with code written in any other CLR language. For example, you can write a class in C#, inherit from it in VB.NET, and use it in Eiffel. You can choose to write code in Mono in a variety of programming languages.

- **Other Uses**

  Scripting and Embedding - The Mono runtime can also be used to script your applications by embedding it inside other applications, to allow managed code and scripts to run in a native application.See Embedding Mono for details on how to embed Mono.See Scripting With Mono for strategies on how to script your application using the Mono runtime.

What is Xamarin?

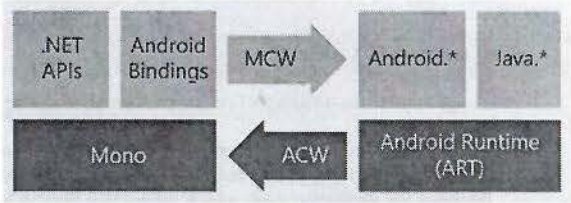
.NET is a developer platform made up of tools, programming languages, and libraries for building many different types of applications.

Xamarin extends the .NET developer platform with tools and libraries specifically for building apps for Android, iOS, tvOS, watchOS, macOS, and Windows.

Xamarin is an open-source platform for building modern and performant applications for iOS, Android, and Windows with .NET. Xamarin is an abstraction layer that manages communication of shared code with underlying platform code. Xamarin runs in a managed enviro54nment that provides conveniences such as memory allocation and garbage collection.

access common resources across all three platforms. Shared code can significantly reduce both development costs and time to market for mobile developers.

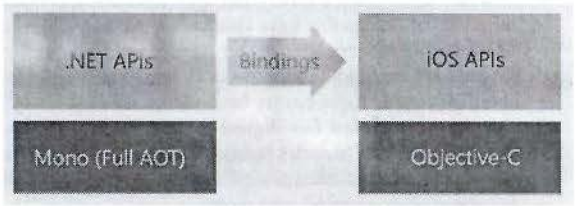
Xamarin.Android



Xamarin.Android applications compile from C# into **Intermediate Language (IL)** which is then **Just-in-Time (JIT)** compiled to a native assembly when the application launches. Xamarin.Android applications run within the Mono execution environment, side by side with the Android Runtime (ART) virtual machine. Xamarin provides .NET bindings to the Android.* and Java.* namespaces. The Mono execution environment calls into these namespaces via **Managed Callable Wrappers (MCW)** and provides **Android Callable Wrappers (ACW)** to the ART, allowing both environments to invoke code in each other.

For more information, see Xamarin.Android architecture.

Xamarin.iOS



Xamarin.iOS applications are fully **Ahead-of-Time (AOT)** compiled from C# into native ARM assembly code. Xamarin uses **Selectors** to expose Objective-C to managed C# and Registrars to expose managed C# code to Objective-C. Selectors and Registrars collectively are called "bindings" and allow Objective-C and C# to communicate.

Xamarin.Essentials

Xamarin.Essentials is a library that provides cross-platform APIs for native device features. Like Xamarin itself, Xamarin.Essentials is an abstraction that simplifies the process of accessing native functionality. Some examples of functionality provided by Xamarin.Essentials include:

- Device info
- File system
- Accelerometer
- Phone dialer
- Text-to-speech
- Screen lock

Xamarin.Forms

Xamarin.Forms is an open-source UI framework. Xamarin.Forms allows developers to build Xamarin.iOS, Xamarin.Android, and Windows applications from a single shared codebase. Xamarin.Forms allows developers to create user interfaces in XAML with code-behind in C#. These user interfaces are rendered as performant native controls on each platform. Some examples of features provided by Xamarin.Forms include:

- XAML user-interface language
- Databinding
- Gestures
- Effects
- Styling

Xamarin enables developers to share an average of 90% of their application across platforms. This pattern allows developers to write all of their business logic in a single language (or reuse existing application code) but achieve native performance, look, and feel on each platform.

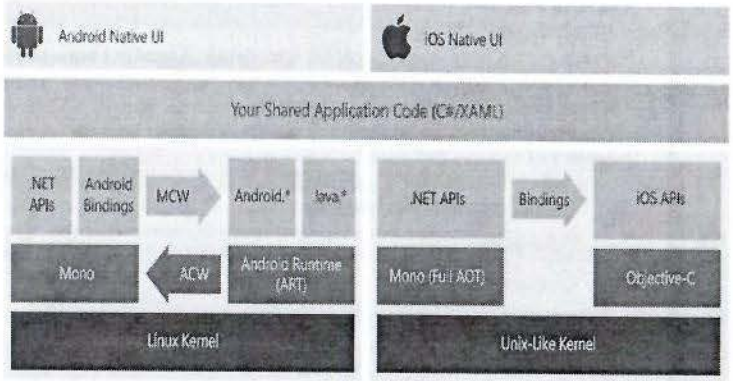
Xamarin applications can be written on PC or Mac and compile into native application packages, such as an .apk file on Android, or an .ipa file on iOS.

Who Xamarin is for

Xamarin is for developers with the following goals:

- Share code, test and business logic across platforms.
- Write cross-platform applications in C# with Visual Studio.

How Xamarin works



- The diagram shows the overall architecture of a cross-platform Xamarin application. Xamarin allows you to create native UI on each platform and write business logic in C# that is shared across platforms. In most cases, 80% of application code is sharable using Xamarin.

- Xamarin is built on top of .NET, which automatically handles tasks such as memory allocation, garbage collection and interoperability with underlying platforms.
- For more information about platform-specific architecture, see Xamarin.Android and Xamarin.iOS.

Added features of Xamarin :

Xamarin combines the abilities of native platforms, while adding features that include:

1. **Complete binding for the underlying SDKs** – Xamarin contains bindings for nearly the entire underlying platform SDKs in both iOS and Android. Additionally, these bindings are strongly-typed, which means that they're easy to navigate and use, and provide robust compile-time type checking and during development. Strongly-typed bindings lead to fewer runtime errors and higher-quality applications.
2. **Objective-C, Java, C, and C++ Interop** – Xamarin provides facilities for directly invoking Objective-C, Java, C, and C++ libraries, giving you the power to use a wide array of third party code. This functionality lets you use existing iOS and Android libraries written in Objective-C, Java, or C/C++. Additionally, Xamarin offers binding projects that allow you to bind native Objective-C and Java libraries using a declarative syntax.
3. **Modern language constructs** – Xamarin applications are written in C#, a modern language that includes significant improvements over Objective-C and Java such as dynamic language features, functional constructs such as lambdas, LINQ, parallel programming, generics, and more.
4. **Robust Base Class Library (BCL)** – Xamarin applications use the .NET BCL, a large collection of classes that have comprehensive and streamlined features such as powerful XML, Database, Serialization, IO, String, and Networking support, and more. Existing C# code can be compiled for use in an app, which provides access to thousands of libraries that add functionality beyond the BCL.
5. **Modern Integrated Development Environment (IDE)** – Xamarin uses Visual Studio, a modern IDE that includes features such as code auto completion, a sophisticated project and solution management system, a comprehensive project template library, integrated source control, and more.
6. **Mobile cross-platform support** – Xamarin offers sophisticated cross-platform support for the three major platforms of iOS, Android, and Windows. Applications can be written to share up to 90% of their code, and Xamarin.Essentials offers a unified API to

Difference between .NET Core and .NET Framework

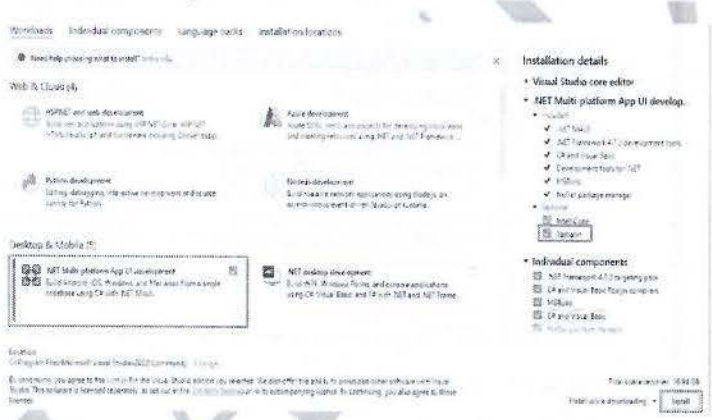
BASED ON	.NET Core	.NET Framework
Open Source	.Net Core is an open source.	Certain components of the .Net Framework are open source.
Cross-Platform	Works on the principle of "build once, run anywhere". It is compatible with various operating systems — Windows, Linux, and Mac OS as it is cross-platform.	.NET Framework is compatible with the windows operating system. Although, it was developed to support software and applications on all operating systems.
Application Models	.Net Core does not support desktop application development and it rather focuses on the web, windows mobile, and windows store.	.Net Framework is used for the development of both desktop and web applications as well as it supports windows forms and WPF applications.
Installation	.NET Core is packaged and installed independently of the underlying operating system as it is cross-platform.	.NET Framework is installed as a single package for Windows operating system.
Support for Micro-Services and REST Services	.Net Core supports the development and implementation of micro-services and the user has to create a REST API for its implementation.	.Net Framework does not support the development and implementation of microservices but it supports the REST API services.
Performance and Scalability	.NET Core offers high performance and scalability.	.Net Framework is less effective in comparison to .Net Core in terms of performance and scalability of applications.
Compatibility	.NET Core is compatible with various operating systems — Windows, Linux, and Mac OS.	.NET Framework is compatible only with the Windows operating system.

BASED ON	.NET Core	.NET Framework
Android Development	.NET Core is compatible with open-source mobile application platforms, i.e. Xamarin, through the .NET Standard Library. Developers use Xamarin's tools to configure the mobile app for specific mobile devices such as iOS, Android, and Windows phones.	.NET Framework does not support any framework for mobile application development.
Packaging and Shipping	.Net Core is shipped as a collection of Nugget packages.	All the libraries of .Net Framework are packaged and shipped together.
Deployment Model	Whenever the updated version of .NET Core gets initiated; it is updated instantly on one machine at a time, thereby getting updated in new directories/folders in the existing application without affecting it. Thus, .NET Core has a good and flexible deployment model.	In the case of .Net Framework, when the updated version is released it is first deployed on the Internet Information Server only.
Support	It has support for microservices.	It does not support creation and microservices.
WCF Services	It has no support for WCF services.	It has excellent support for WCF services.
Rest APIs	It does not support REST APIs.	It also supports REST Services.
CLI Tools	.NET Core provides light-weight editors and command-line tools for all supported platforms.	.Net Framework is heavy for Command Line Interface and developers prefer to work on the lightweight Command Line Interface.

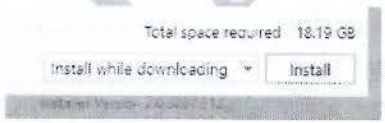
Installing Xamarin on Windows

Xamarin can be installed as part of a *new* Visual Studio 2022 installation, with the following steps:

1. Download Visual Studio 2022 Community, Visual Studio Professional, or Visual Studio Enterprise from the Visual Studio page.
2. Double-click the downloaded package to start installation.
3. Select the **.NET Multi-platform App UI development** workload from the installation screen, and under **Optional** check **Xamarin**:



4. When you are ready to begin Visual Studio 2022 installation, click the **Install** button in the lower right-hand corner:

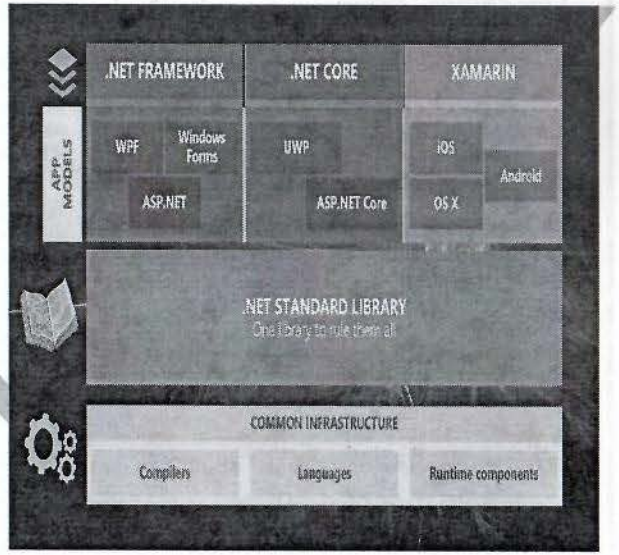


5. When Visual Studio 2022 installation has completed, click the **Launch** button to start Visual Studio.

Adding Xamarin to Visual Studio 2022

If Visual Studio 2022 is already installed, add Xamarin by re-running the Visual Studio 2022 installer to modify workloads (see Modify Visual Studio for details). Next, follow the steps listed above to install .NET Multi-platform App UI development and the optional Xamarin install.

.Net Framework, .Net Core, Mono, Xamarin differences



BASED ON	.NET Core	.NET Framework
Security	.NET Core does not have features like Code Access Security.	Code access security feature is present in .NET Framework.

- **.NET Framework**

The .NET Framework is the oldest implementation of the Common Language Runtime. In short, it is the Windows-only implementation of .NET that is the base for many existing applications. As a result, the .NET Framework has a host of tools and third-party libraries already in existence. Additionally, many existing .NET applications are already written on the .NET Framework.
Microsoft recommends using .NET Framework when your existing application already uses it or when you need access to .NET Framework-only third-party libraries. Furthermore, some features of .NET Framework itself are not available in .NET Core yet. For instance, some applications of ASP.NET, Visual Basic, and F# aren't yet supported in .NET Core.

- **.NET Core**

.NET Core is the latest version of the CLR and the currently preferred option. It's a slimming down of the .NET Framework with greater accessibility for various devices.
Whereas .NET Framework was Windows-only, .NET Core is cross-platform and can deploy applications on Mac or Linux. In addition, Microsoft Azure cloud infrastructure offers the most support for .NET Core applications. Because they're lighter weight than .NET Framework applications, .NET Core is faster and more scalable. Indeed, .NET Core applications can deploy easily as microservices on Azure. They also deploy well inside containers like Docker.
If you're starting a new project, you likely should use .NET Core. Even if you're extending an existing .NET Framework app, you can add .NET Core gradually as you build new features.

- **Xamarin**

Xamarin is not technically a .NET option. Instead, it runs on top of Mono, an earlier cross-platform version of .NET that Microsoft built before deciding to go with .NET Core. Xamarin is a mobile-only implementation. If you're planning to write a mobile app for Android or iOS, then Xamarin is how you deploy C#, Visual Basic, F#, etc on mobile platforms. In fact, Xamarin is your only option for those languages on mobile.

.NET Framework versions and dependencies

Each version of .NET Framework contains the common language runtime (CLR), the base class libraries, and other managed libraries. This article describes the key features of .NET Framework by version, provides information about the underlying CLR versions and associated development environments, and identifies the versions that are installed by the Windows operating system (OS).

Each new version of .NET Framework adds new features but retains features from previous versions.

.NET Framework **is serviced monthly** with security and reliability bug fixes. .NET Framework will continue to be included with Windows, with no plans to remove it. You don't need to migrate your .NET Framework apps, but for new development, use **.NET 6 or later**.

The CLR is identified by its own version number. The .NET Framework version number is incremented at each release, but the CLR version is not always incremented. For example, .NET Framework 4, 4.5, and later releases include CLR 4, but .NET Framework 2.0, 3.0, and 3.5 include CLR 2.0. (There was no version 3 of the CLR.)

Version information

The tables that follow summarize .NET Framework version history and correlate each version with Visual Studio, Windows, and Windows Server. Visual Studio supports multi-targeting, so you're not limited to the version of .NET Framework that's listed.

- The check mark icon ☒ denotes OS versions on which .NET Framework is installed by default.
- The plus sign icon + denotes OS versions on which .NET Framework doesn't come installed but can be installed.

- The asterisk * denotes OS versions on which .NET Framework (whether preinstalled or not) must be enabled in Control Panel or, for Windows Server, through the Server Manager.

Jump to:

WHAT IS MANAGED CODE IN C#?

As we have seen in the beginning, the code which is developed using the Common Language Runtime (CLR) of the .NET framework is known as Managed Code. In other words, it is the code that gets executed directly in C#. The runtime environment for managed code provides a variety of services to the programmer. The services provided are exception handling, garbage collection, type checking, etc. The above-mentioned services are provided to the programmer automatically. Apart from the above-mentioned services the runtime environment also provides memory allocation, object disposal, object creation, etc.

EXECUTION OF MANAGED CODE

The managed code is always compiled by the CLR into an intermediate language known as MSIL and after that, the executable of the code is created. The intermediate language (MSIL) in the source code is compiled by the compiler named 'Just In Time Compiler' when the

executable is run by the programmer. Here the whole process is carried under a runtime environment, hence the runtime environment is the reason behind the working of the program.

ADVANTAGES OF MANAGED CODE

- Security of the code is improved as we are using a runtime environment, which protects from buffer overflow by checking memory buffers.
- Garbage collection is automatically implemented.
- Dynamic type checking or runtime type checking is also provided
- The runtime environment does the reference checking too. It checks if the reference point of the object is valid or not and also checks if the duplicate object is present or not.

DISADVANTAGES OF MANAGED CODE

- Memory cannot be directly allocated.
- In managed code, low-level access to the CPU architecture cannot be obtained.

WHAT IS UNMANAGED CODE IN C#?

The Unmanaged Code is also known as Unsafe Code. Unmanaged code depends on the computer architecture as it aims for the processor architecture. In C#, the activities like managing stacks, allocation, and release of memory, etc. are taken care of by the CLR and hence these activities are out of the purview of the programmer. The programmer tells the compiler that the management of the code will be done by him/her when he/she uses the keyword "unsafe". However, issues like memory leaks can happen if a programmer writes bad code.

EXECUTION OF UNMANAGED CODE

For executing unmanaged code wrapper classes are used in C#. In C# the unmanaged code is directly executed by the operating system. Generally, the executable files of unmanaged or unsafe code are in the form of binary images which are loaded into the memory. The steps of execution of the code can be seen in the above image.

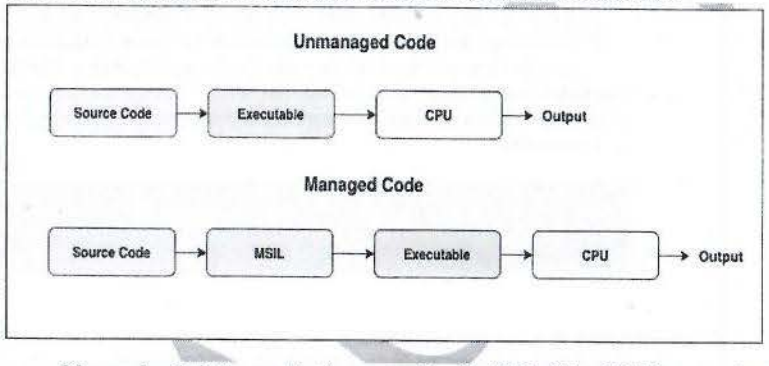
ADVANTAGES OF UNMANAGED CODE

- Unsafe code increases stability and the performance of the program
- Low-level access to the programmer is provided
- Unsafe code provides a medium to interface with the memory

DISADVANTAGES OF UNMANAGED CODE

- Security is not provided to the application
- The programmer has to do exception handling
- No automatic implementation of garbage collection
- Making use of unsafe code can lead to errors that might occur as type checking is bypassed.

DIFFERENCE BETWEEN MANAGED AND UNMANAGED CODE?



- Managed code is the one that is executed by the CLR of the .NET framework while unmanaged or unsafe code is executed by the operating system.
- The managed code provides security to the code while undamaged code creates security threats.
- In unsafe or unmanaged code the unsafe modifier is used to write the block of code while any other code written outside the unsafe code block is managed code.
- Memory buffer overflow problems do not occur in managed code as it is taken care of by the runtime environment but these problems occur in unmanaged code.
- Runtime services are provided in managed code while they are not provided in unmanaged code.
- Source code is first converted to intermediate language and then to native language in managed code while it is directly converted into the native language in unmanaged code.
- Unmanaged code provides low-level access while managed code does not provide that.
- In unsafe or unmanaged code the unsafe modifier is used to write the block of code while any other code written outside the unsafe code block is managed code.

Introduction to Visual Studio

Visual Studio is an **Integrated Development Environment(IDE)** developed by Microsoft to develop GUI(Graphical User Interface), console, Web applications, web apps, mobile apps, cloud, and web services, etc. With the help of this IDE, you can create managed code as well as native code. It uses the various platforms of Microsoft software development software like Windows store, Microsoft Silverlight, and Windows API, etc. It is not a language-specific IDE as you can use this to write code in C#, C++, VB(Visual Basic), Python, JavaScript, and many more languages. It provides support for 36 different programming languages. It is available for Windows as well as for macOS. **Evolution of Visual Studio:** The first version of VS(Visual Studio) was released in 1997, named as Visual Studio 97 having version number 5.0. The latest version of Visual Studio is 15.0 which was released on March 7, 2017. It is also termed as Visual Studio 2017. The supported *.Net Framework Versions* in latest Visual Studio is 3.5 to 4.7. Java was supported in old versions of Visual Studio but in the latest version doesn't provide any support for Java language.

Visual Studio Editions

There are 3 editions of Microsoft Visual Studio as follows:

1. Community: It is a **free** version which is announced in 2014. *All other editions are paid.* This contains the features similar to Professional edition. Using this edition, any individual developer can develop their own free or paid apps like *.Net applications*, Web applications and many more. In an enterprise organization, this edition has some limitations. For example, if your organization have more than 250 PCs and having annual revenue greater than $1 Million(US Dollars) then you are not permitted to use this edition. In a non-enterprise organization, up to five users can use this edition. Its main purpose is to provide the Ecosystem (Access to thousands of extensions) and Languages (You can code in C#, VB, F#, C++, HTML, JavaScript, Python, etc.) support.

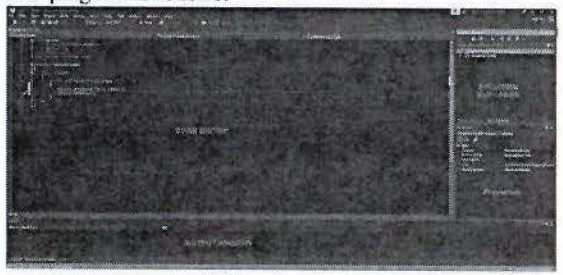
2. Professional: It is the commercial edition of Visual Studio. It comes in Visual Studio 2010 and later versions. It provides the support for XML and XSLT editing and includes the tool like Server Explorer and integration with Microsoft SQL Server. Microsoft provides a free trial of this edition and after the trial period, the user has to pay to continue using it. Its main purpose is to provide Flexibility (Professional developer tools for building any application type), Productivity (Powerful features such as CodeLens improve your team's productivity), Collaboration (Agile project planning tools, charts, etc.) and Subscriber benefits like Microsoft software, plus Azure, Plural sight, etc.

3. Enterprise: It is an integrated, end to end solution for teams of any size with the demanding quality and scale needs. Microsoft provides a 90-days free trial of this edition and after the trial period, the user has to pay to continue using it. The main benefit of this edition is that it is highly scalable and deliver high-quality software.

Getting Started with Visual Studio 2017

- First, you have to download and install the Visual Studio. For that, you can refer to **Downloading and Installing Visual Studio 2017**. Don't forget to select the .NET core workload during the installation of VS 2017. If you forget then you have to **modify** the installation.

- You can see a number of tool windows when you will open the Visual Studio and start writing your first program as follows:



1. **Code Editor:** Where the user will write code.
2. **Output Window:** Here the Visual Studio shows the outputs, compiler warnings, error messages and debugging information.
3. **Solution Explorer:** It shows the files on which the user is currently working.
4. **Properties:** It will give additional information and context about the selected parts of the current project.

- A user can also add windows as per requirement by choosing them from **View** menu. In Visual Studio the tool windows are customizable as a user can add more windows, remove the existing open one or can move windows around to best suit.
- **Various Menus in Visual Studio:** A user can find a lot of menus on the top screen of Visual Studio as shown below
    1. Create, Open and save projects commands are contained by **File** menu.
    2. Searching, Modifying, Refactoring code commands are contained by the **Edit** menu.
    3. **View** Menu is used to open the additional tool windows in Visual Studio.
    4. **Project** menu is used to add some files and dependencies in the project.
    5. To change the settings, add functionality to Visual Studio via extensions, and access various Visual Studio tools can be used by using **Tools** menu.
- The below menu is known as the **toolbar** which provide the quick access to the most frequently used commands. You can add and remove the commands by going to **View → Customize**

Advantages of using Visual Studio IDE:

- A full-featured programming platform for several operating systems, the web, and the cloud, Visual Studio IDE is available. Users can easily browse the UI so they can write their code quickly and precisely.
- To help developers quickly identify potential errors in the code, Visual Studio offers a robust debugging tool.
- Developers can host their application on the server with confidence because they have eliminated anything that could lead to performance issues.
- No matter what programming language developers are using, users of Visual Studio can get live coding support. For faster development, the Platform offers an autocomplete option. The built-in intelligent system offers descriptions and tips for APIs.
- Through Visual Studio IDE you can easily collab with your teammates in a same project. This IDE helps the developers to share, push and pull their code with their teammates.

- Every user of Visual Studio has the ability to customize it. They have the option to add features based on their needs. For example, they can download add-ons and install extensions in their IDE. Even programmers can submit their own extensions.

Using ILDASM :

What happens when we compile a .NET Application?

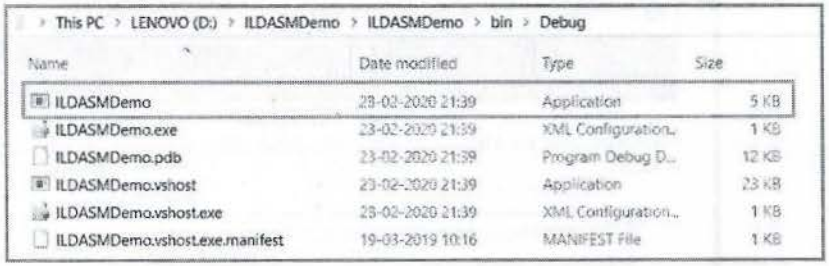
When we compile any .NET application. it will generate an assembly with the extension of either a .DLL or an .EXE. For example, if you compile a Windows or Console application, then you will get an .EXE, whereas if you compile a Web or Class library project, then you will get a .DLL. Irrespective of whether it is a .DLL or .EXE, an assembly consists of two things i.e. **Manifest and Intermediate language**. Let us understand how the Intermediate Language and Manifest look like in .NET Framework with an example.

Understanding Intermediate Language (ILDASM and ILASM) Code in C#:

In order to understand Intermediate Language Code (ILDASM and ILASM) in C#, let us create a simple console application. Once you create the console application, please modify the Program class as shown below.

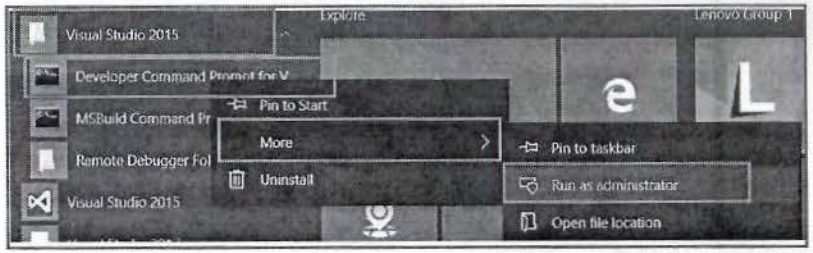
```
using System;
namespace ILDASMDemo
{
class Program
{
staticvoidMain(string[] args)
{
Console.WriteLine("Understanding ILDASM and ILASM");
Console.Read();
}
}
}
```

Now, build the application. Once you build the application, the above source code is compiled and intermediate language code generated and packaged into an assembly. In order to see the assembly, just right-click on the Project and select **Open Folder in File Explorer** option and then go to the **bin => Debug** folder and you should see an assembly with .exe extension as shown in the below image as it is a console application.

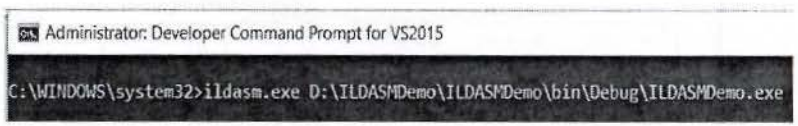


How to view the Intermediate Language Code in .NET Framework?

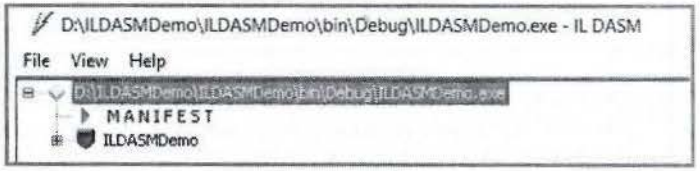
The .NET framework provides a nice tool called **ILDASM (Intermediate Language DisAssember)** to view the code of the intermediate language in C#.NET. In order to use the ILDASM tool, you need to follow the below steps. Open Visual Studio Command Prompt in Administrator mode as shown in the below image.



Once you open the visual studio command prompt in administrative mode, then type the "**Ildasm.exe C:\YourDirectoryPath\YourAssembly.exe**" command and press enter. Here, you need to provide the exe path where your exe is generated. My exe is generated in the path "**D:\ILDASMDemo\ILDASMDemo\bin\Debug\ILDASMDemo.exe**", so I execute the following code in the command prompt:

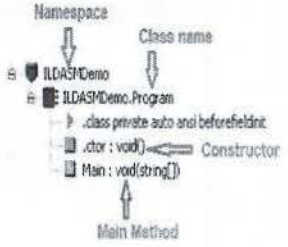


Once you type the above command and press enter, it should open the following ILDASM window.



As you can see, the assembly is consists of two things (**Manifest and Intermediate language**). Let us first discuss the intermediate language code and then we will discuss what Manifest is. Now, let us expand the ILDASMDemo and compare it with our code. For better understanding, please have a look at the below image. There is a constructor in ILDASM and this is because by default the .NET Framework provides a default constructor when there is no constructor in your class. You can also the Main method in the intermediate language code

```
namespace ILDASMDemo
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Understanding ILDASM and ILASM");
      Console.Read();
    }
  }
}
```



Now, double click on the Main method on the ILDASM window to see the intermediate language generated for the Main method as shown below.

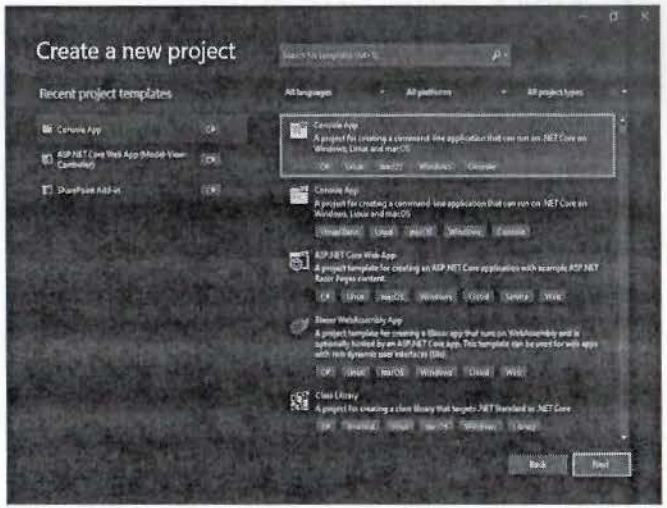
3.Console Application and class libraries (Framework and .net core)

Console Application (Framework)

Create Console App in .NET 6

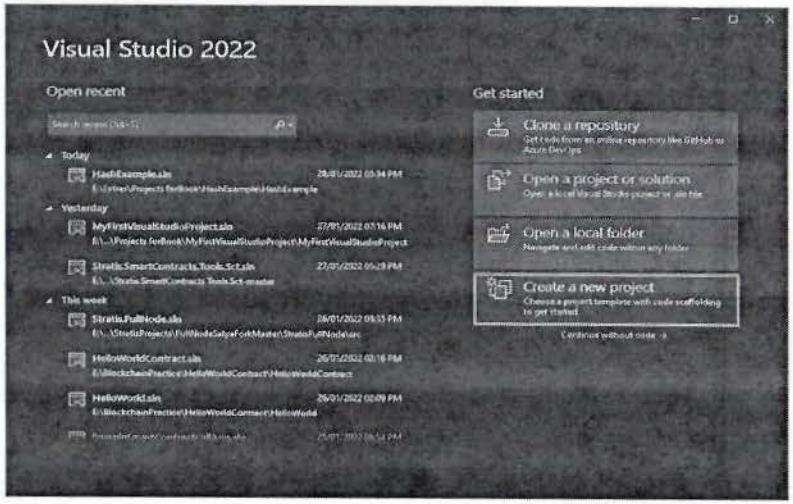
Step 1

Open Visual Studio 2022 and click Create a new project.



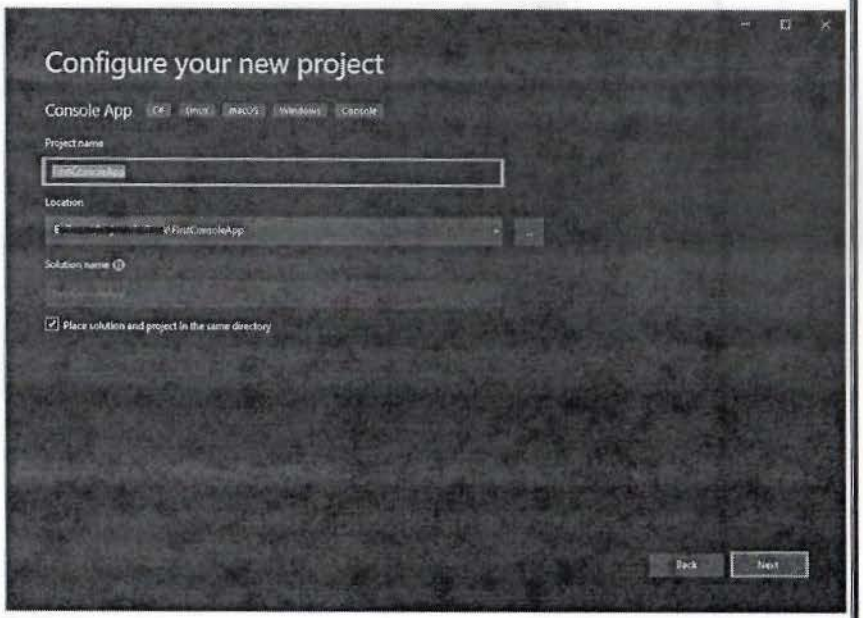
Step 2

Select Console App and click Next.



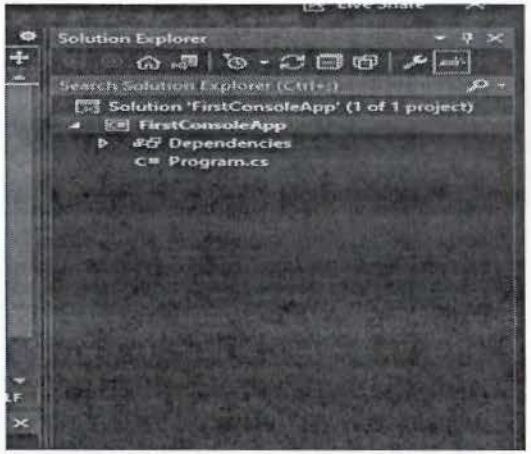
Step 3

Give the project name and location of the project.



Step 4
Select framework: **.NET 6.0 (Long-term support).**

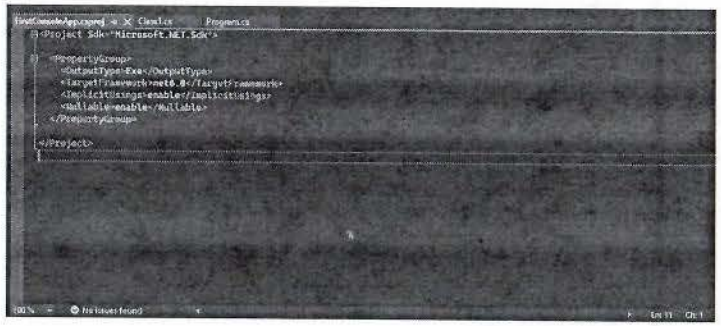
This creates the console app which looks like below.



Default Program.cs file is listed below.

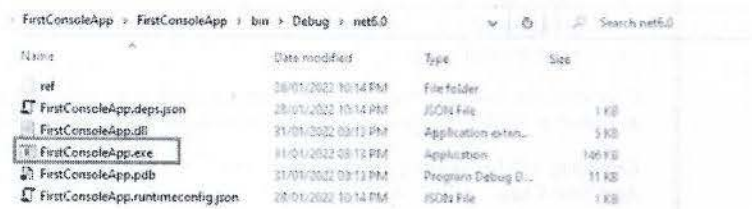
```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

C# The project.csproj file is given below.



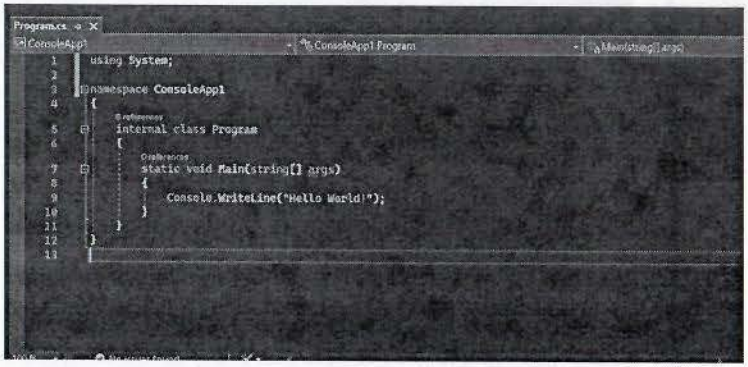
Now, let's compile and run the program. Click on Strat without debugging option or press Ctrl+F5 as illustrated below.

You can see the FirstConsoleApp.exe is created in the folder location of the project under the directory: "E:\SampleProject\FirstConsoleApp\bin\Debug\net6.0" as shown below.



We can run this .exe file which displays the **"Hello, World!"** message.
Create Project in .NET 5
Let's create a console application in Visual Studio using .NET 5. Follow all steps as per previous, only choose framework .NET 5 in **Step 4.**

Below is Program.cs of console app using .NET 5.



```
using System;

namespace ConsoleApp1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Comparing .NET6 and .NET 5

We can see here, Program.cs of .NET 5 contains:

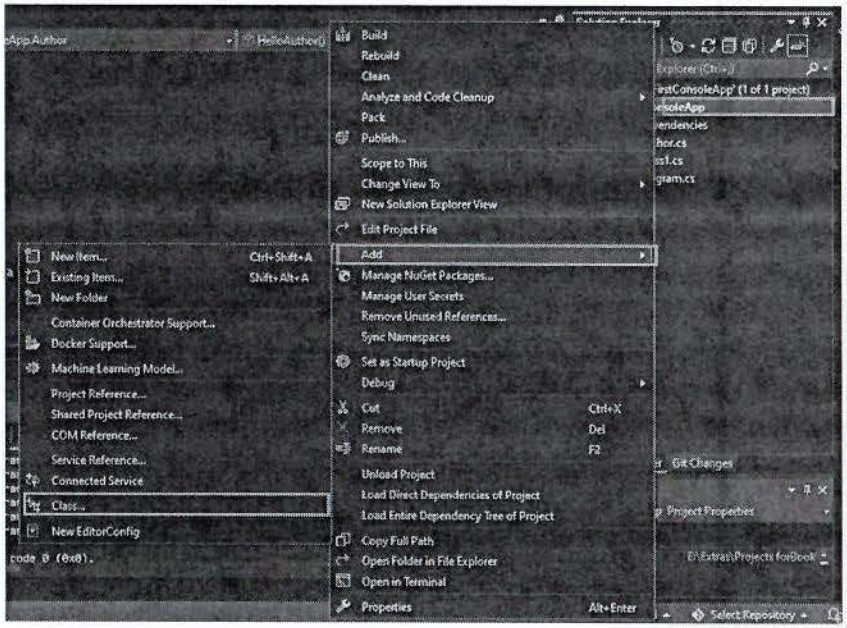
- Using system

- Namespace
  class keyword
- Main method

If we compare both the Program.cs file from the above two console apps, these are not available in .NET 6. so, this is the difference between .NET5 and .NET6.

Extending .NET 6 Console Application
Add New Class

Right-click on the project—Go to Add-->Class and click add.



We have added a class with the name: Class1.

Now, will create a public void method named Sum: in this Class1.


```
publicvoidSum(){
int a =5;
int b =6;
int Sum = a + b;
   Console.WriteLine("Sum : {0}", Sum);
}
C#
```

Complete code of Class1.cs

```
namespaceFirstConsoleApp{
internalclassClass1{
publicvoidSum(){
int a =5;
int b =6;
int Sum = a + b;
        Console.WriteLine("Sum : {0}", Sum);
}
}
}
C#
```

You might have noticed that this class contains:

- the namespace
- internal Class
- using statements (by default)

Whereas Program.cs doesn't contain those and you can write logic directly.

Call Sum() Method

In Program.cs class we will call the sum method of the Class1.cs. Furthermore, to call a method from class, here we need to add namespace: using FirstConsoleApp;

```
usingFirstConsoleApp;//need to call method from Class1
Class1 class1 =newClass1 ();
class1.Ssum();
C#
```

Now, if we run the app then we will get **Sum:11** in the console.

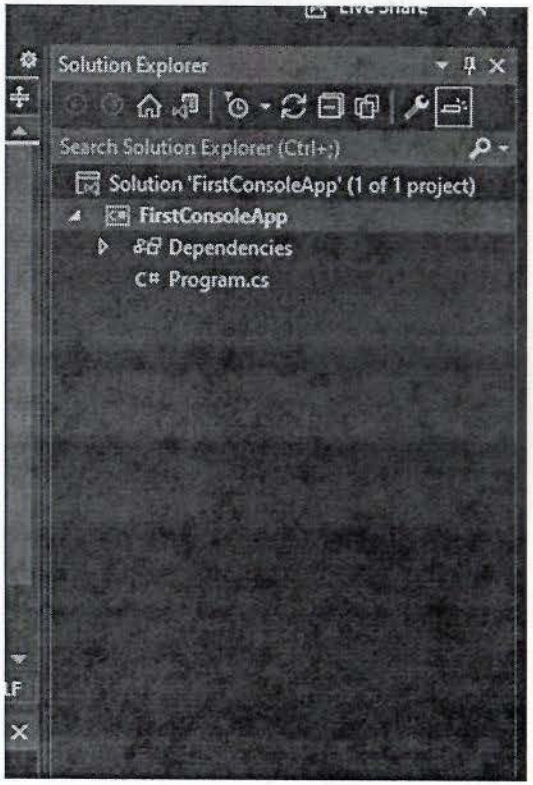
Complete code of Program.cs

```
// See https://aka.ms/new-console-template for more information
usingFirstConsoleApp;
Console.WriteLine("Hello, World!");
Class1 class1 =newClass1();
class1.Sum();
C#
```

When we run the console application, the output looks like below.

Conclusion

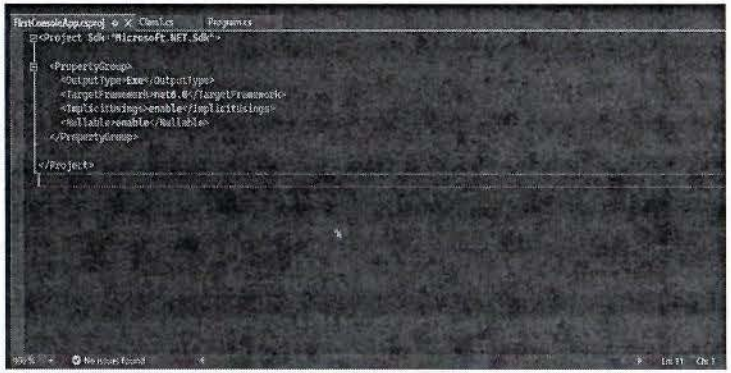
Hence, in this article, we created our first Console application in .NET 6 and also created a console app in .NET 5 and compared the differences between them. Additionally, we added the new class in .NET 6 console application and learned to call the method from class in Program.cs file. I hope it helps you to understand the differences between .NET 5 and .NET 6 and get start your journey in the latest .NET framework.



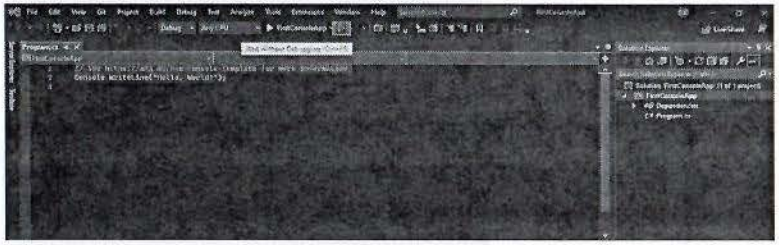
Default Program.cs file is listed below.

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

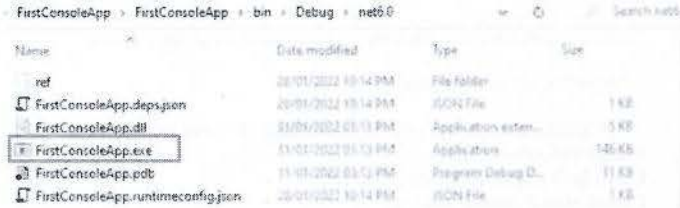
The project .csproj file is given below.



Now, let's compile and run the program. Click on Strat without debugging option or press Ctrl+F5 as illustrated below.



You can see the FirstConsoleApp.exe is created in the folder location of the project under the directory: "E:\SampleProject\FirstConsoleApp\bin\Debug\net6.0" as shown below.



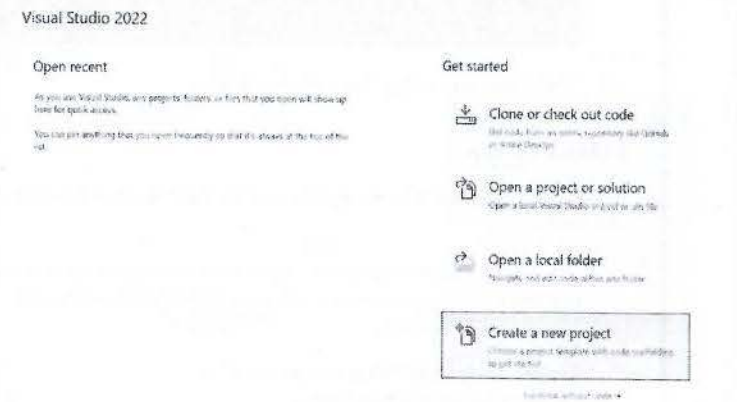
We can run this .exe file which displays the **"Hello, World!"** message.

Console Application in (.Net Core)

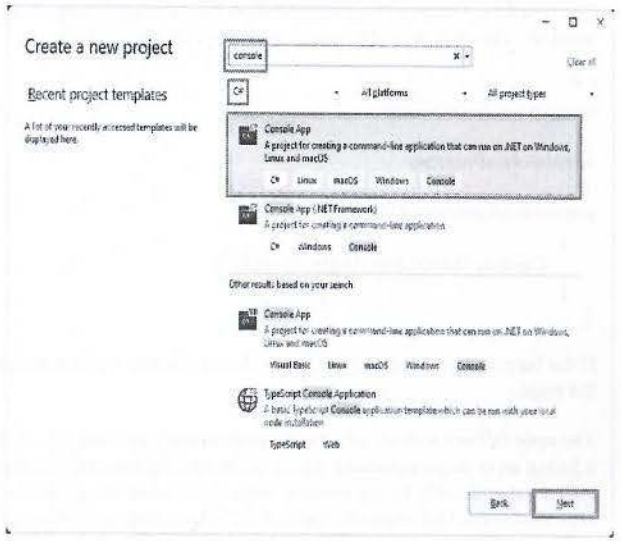
Create the app

Create a .NET console app project named "HelloWorld".

1. Start Visual Studio 2022.
2. On the start page, choose **Create a new project**.



3. On the **Create a new project** page, enter **console** in the search box. Next, choose **C#** or **Visual Basic** from the language list, and then choose **All platforms** from the platform list. Choose the **Console App** template, and then choose **Next**.



Tip

If you don't see the .NET templates, you're probably missing the required workload. Under the **Not finding what you're looking for?** message, choose the **Install more tools and features** link. The Visual Studio Installer opens. Make sure you have the **.NET desktop development** workload installed.

4. In the **Configure your new project** dialog, enter **HelloWorld** in the **Project name** box. Then choose **Next**.

5. In the **Additional information** dialog:
   - Select **.NET 7 (Standard-term support)**.
   - Select **Do not use top-level statements**.
   - Select **Create**.

The template creates a simple application that displays "Hello, World!" in the console window. The code is in the *Program.cs* or *Program.vb* file:

```
C#
namespaceHelloWorld;

internalclassProgram
 {
staticvoidMain(string[] args)
    {
       Console.WriteLine("Hello, World!");
    }
 }
```

If the language you want to use is not shown, change the language selector at the top of the page.

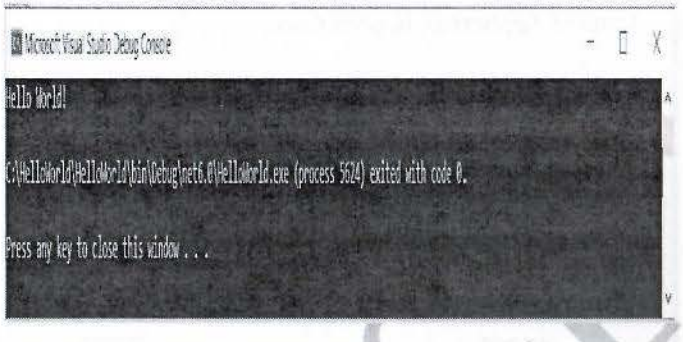
The code defines a class, `Program`, with a single method, `Main`, that takes a String array as an argument. `Main` is the application entry point, the method that's called automatically by the runtime when it launches the application. Any command-line arguments supplied when the application is launched are available in the *args* array.

In the latest version of C#, a new feature named top-level statements lets you omit the `Program` class and the `Main` method. Most existing C# programs don't use top-level statements, so this tutorial doesn't use this new feature. But it's available in C# 10, and whether you use it in your programs is a matter of style preference.

Run the app

1. Press `Ctrl+F5` to run the program without debugging.

   A console window opens with the text "Hello, World!" printed on the screen. (Or "Hello World!" without a comma in the Visual Basic project template.)



2. Press any key to close the console window.

Enhance the app

Enhance the application to prompt the user for their name and display it along with the date and time.

1. In *Program.cs* or *Program.vb*, replace the contents of the `Main` method, which is the line that calls `Console.WriteLine`, with the following code:

   ```
   C#
   Console.WriteLine("What is your name?");
   var name = Console.ReadLine();
   var currentDate = DateTime.Now;
   Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at
   {currentDate:t}!");
   Console.Write($"{Environment.NewLine}Press any key to exit...");
   Console.ReadKey(true);
   ```

   This code displays a prompt in the console window and waits until the user enters a string followed by the `Enter` key. It stores this string in a variable named `name`. It also retrieves the value of the DateTime.Now property, which contains the current local time, and assigns it to a variable named `currentDate`. And it displays these values in the console window. Finally, it displays a prompt in the console window and calls the Console.ReadKey(Boolean) method to wait for user input.

Environment.NewLine is a platform-independent and language-independent way to represent a line break. Alternatives are `\n` in C# and `vbCrLf` in Visual Basic.

The dollar sign (`$`) in front of a string lets you put expressions such as variable names in curly braces in the string. The expression value is inserted into the string in place of the expression. This syntax is referred to as interpolated strings.

2. Press `Ctrl+F5` to run the program without debugging.
3. Respond to the prompt by entering a name and pressing the `Enter` key.

```
What is your name?
Maira

Hello, Maira, on 12/3/2019 at 3:36 AM!

Press any key to exit...
```

4. Press any key to close the console window.

Class Libraries in Framework

The Framework Class Library or FCL provides the system functionality in the **.NET Framework** as it has various classes, data types, interfaces, etc. to perform multiple functions and build different types of applications such as desktop applications, web applications, mobile applications, etc. The Framework Class Library is integrated with the Common Language Runtime (CLR) of the .NET framework and is used by all the .NET languages such as **C#**, F#, Visual Basic .NET, etc.

Categories in the Framework Class Library:

The functionality of the Framework Class Library can be broadly divided into **three** categories

- utility features written in .NET
- wrappers around the OS functionality
- Frameworks.

These categories are not rigidly defined and there are many classes that may fit into more than one category.

Namespaces in the Framework Class Library :

Namespaces in the Framework Class Library are a group of related classes and interfaces that can be used by all the *.NET framework* languages. Some of the namespaces in the FCL along with their description is given as follows:

Namespace	Description
Accessibility	The Accessibility namespace is a part of the managed wrapper for the COM accessibility interface.
Microsoft.Activities	The Microsoft.Activities namespace provides support for Windows Workflow Foundation applications.
Microsoft.CSharp	The Microsoft.CSharp namespace has support for compilation and code generation for the C# source code.
Microsoft.JScript	The Microsoft.JScript namespace has support for compilation and code generation for the JScript source code.
Microsoft.VisualBasic	The Microsoft.VisualBasic namespace has support for compilation and code generation for the VisualBasic source code.
System	The System namespace has base classes for definition of interfaces, data types, events, event handlers, attributes, processing exceptions etc.
System.Activities	The System.Activities namespace handles the creation and working with activities in the Window Workflow Foundation using various classes.
System.Collections	The System.Collections namespace has multiple standard, specialized, and generic collection objects that are defined using various types.

Namespace	Description
System.Configuration	The System.Configuration namespace handles configuration data using various types. This may include data in machine or application configuration files.
System.Data	The System.Data namespace accesses and manages data from various sources using different classes.
System.Drawing	The System.Drawing namespace handles GDI+ basic graphics functionality. Various child namespaces also handle vector graphics functionality, advanced imaging functionality, etc.
System.Globalization	The System.Globalization namespace handles language, country, calendars used, format patterns for dates, etc. using various classes.
System.IO	The System.IO namespaces support IO like data read/write into streams, data compression, communicate using named pipes etc. using various types.
System.Linq	The System.Linq namespace supports Language-Integrated Query (LINQ) using various types.
System.Media	The System.Media namespace handles sound files and accessing the sounds provided by the system using various classes.
System.Net	The System.Net namespace provides an interface for network protocols, cache policies for web resources, composing and sending e-mail etc. using various classes.

Namespace	Description
System.Reflection	The System.Reflection namespace gives a managed view of loaded methods, types, fields, etc. It can also create and invoke types dynamically.
System.Security	The System.Security namespace has the .NET security system and permissions. Child namespaces provide authentication, cryptographic services etc.
System.Threading	The System.Threading namespace allows multithreaded programming using various types.
XamlGeneratedNamespace	The XamlGeneratedNamespace has compiler-generated types that are not used directly from the code.

Class Libraries (.Net Core)

Class libraries are the shared library concept for .NET. They enable you to componentize useful functionality into modules that can be used by multiple applications. They can also be used as a means of loading functionality that is not needed or not known at application startup. Class libraries are described using the .NET Assembly file format.

There are three types of class libraries that you can use:

- **Platform-specific** class libraries have access to all the APIs in a given platform (for example, .NET Framework on Windows, Xamarin iOS), but can only be used by apps and libraries that target that platform.
- **Portable** class libraries have access to a subset of APIs, and can be used by apps and libraries that target multiple platforms.
- **.NET Standard** class libraries are a merger of the platform-specific and portable library concept into a single model that provides the best of both.

Platform-specific class libraries

Platform-specific libraries are bound to a single .NET platform (for example, .NET Framework on Windows) and can therefore take significant dependencies on a known execution environment. Such an environment exposes a known set of APIs (.NET and OS APIs) and maintains and exposes expected state (for example, Windows registry).

Developers who create platform-specific libraries can fully exploit the underlying platform. The libraries will only ever run on that given platform, making platform checks or other forms of conditional code unnecessary (modulo single sourcing code for multiple platforms).

Platform-specific libraries have been the primary class library type for the .NET Framework. Even as other .NET implementations emerged, platform-specific libraries remained the dominant library type.

Portable class libraries

Portable libraries are supported on multiple .NET implementations. They can still take dependencies on a known execution environment, however, the environment is a synthetic one that's generated by the intersection of a set of concrete .NET implementations. Exposed APIs and platform assumptions are a subset of what would be available to a platform-specific library.

You choose a platform configuration when you create a portable library. The platform configuration is the set of platforms that you need to support (for example, .NET Framework 4.5+, Windows Phone 8.0+). The more platforms you opt to support, the fewer APIs and fewer platform assumptions you can make, the lowest common denominator. This characteristic can be confusing at first, since people often think "more is better" but find that more supported platforms results in fewer available APIs.

Many library developers have switched from producing multiple platform-specific libraries from one source (using conditional compilation directives) to portable libraries. There are several approaches for accessing platform-specific functionality within portable libraries, with bait-and-switch being the most widely accepted technique at this point.

.NET Standard class libraries

.NET Standard libraries are a replacement of the platform-specific and portable libraries concepts. They are platform-specific in the sense that they expose all functionality from the underlying platform (no synthetic platforms or platform intersections). They are portable in the sense that they work on all supporting platforms.

.NET Standard exposes a set of library *contracts*. .NET implementations must support each contract fully or not at all. Each implementation, therefore, supports a set of .NET Standard contracts. The corollary is that each .NET Standard class library is supported on the platforms that support its contract dependencies.

.NET Standard does not expose the entire functionality of .NET Framework (nor is that a goal), however, the libraries do expose many more APIs than Portable Class Libraries.

The following implementations support .NET Standard libraries:

- .NET Core
- .NET Framework
- Mono
- Universal Windows Platform (UWP)
- Class libraries are supported on Mono, including the three types of libraries described previously. Mono is often viewed as a cross-platform implementation of .NET Framework. In part, this is because platform-specific .NET Framework libraries can run on the Mono runtime without modification or recompilation. This characteristic was in place before the creation of portable class libraries, so was an obvious choice to enable binary portability between .NET Framework and Mono (although it only worked in one direction).

C# Basics :

- C++ language, C# is an Object Oriented programming language. Generally many people spell C# as C#.Net (C Sharp dot Net), but here Microsoft developed the .Net environment mainly for distributed applications (the sharing of processing between client and server) and in C#.Net "Net" indicates that C# is used to develop only Distributed Applications but using C# we can develop any kind of software applications including Windows applications.

- C# is a new language especially developed from scratch to work with the .NET environment. Using C# we can write a webpage, XML based applications like web services, Components for distributed applications as well as desktop applications.
- Writing First C# Program
- Writing a program in the C# language is similar to writing in the traditional C++ language. If you are familiar with the C++ language its easy you to write and understand C# Code. Anyway I will explain every line in our basic First C# program where we will cover the things used for writing and understanding simple to complex programs. Let's look at our first C# program.

Program :

```
using System;
namespace sai.CS
{
  class FirstCSProgram
  {
    static void Main()
    {
      Console.WriteLine("This Is Our First CSharp Program.");
      Console.ReadLine();
      return;
    }
  }
}
```

C# Compiling and Executing the Program

- The above Program can be written using any text editor like notepad, editplus, vi editor (in Linux) etc.., or we can use the Visual Studio's .NET IDE (Integrated Development Environment) designed by Microsoft especially to write, compile and execute the .NET compatible languages. C# is one of the .NET compatible languages; the other compatible .NET languages are VB.NET, J# etc.
- There are two ways of compiling the above C# Program.
- If you write this Program in Visual Studio's .NET IDE, then there is no additional work to do to compile and execute the application. Just use function key F5 or go to the <Debug> Menu and select <Start Debugging>. The IDE internally complies and executes the application without any user interaction.
- In this method we manually compile and execute the above application using the Command Prompt. Now open your command prompt (Start->Run->cmd or command). You can compile the program by simply using the C# compiler Tool (csc.exe) as shown below:

- Csc FirstCSProgram.cs
- When you press the <Enter> Key this csc.exe tool will compile our application named FirstCSProgram.cs and create an exe file with the same file name, such as FirstCSProgram.exe.
- Now type "FirstCSProgram.exe" (with or without quotes) in your command prompt to execute our sample first C# program.

- **Note: cs is the file extension for C# applications.**

```
1. using System;
2. namespace sai.CS
3. {
4. class FirstCSProgram
5. {
6. static void Main()
7. {
      a. Console.WriteLine("This Is Our First CSharp Program.");
      b. Console.ReadLine();
      c. return;
8. }
9. }
10.}
```

A Close Look at the Code

Line 1: The first line of our sample program is:
using System;
C#

Here we are importing the namespaces using the <using> keyword. In the above statement, system is the namespace and we are importing it into our program. I will explain what this namespace is and how to use it and its importance in our next session; for now just remember that a namespace is a group of similar types of items and every class should belong to a specific namespace.

Line 2: Our next line in our program defines a namespace to our class as shown below:
namespace sai.CS
C#

Here we define a namespace forour class by simply writing a user-defined namespace name preceded with the <namespace> keyword.

Line 3: The Opening flower brace ({) indicates to the compiler that the block is opened or started; it is similar to its use in the C++ programming Language. When the compiler encounters this opening brace it will crease a new space on the stack memory where it will declare the variable which is scoped to that block only (in the next chapter I will explain about variables, declaring variables and their scopes) and allocating some memory from that newly created space for that block. Like when the compiler maintains n number of variables in memory.
Line 4: In line 4 of our program we are declaring the class and its name as shown below:
class FirstCSProgram
C#

In C# programming, whatever the business logic is that you want to write, it should be in a class block. I will explain what a lass is and its uses in our future chapters so don't worry abou that; for now just remember that whatever you want to write should be written in a class block and every program should contain at least one class. A class optionally contains variables and optionally methods or functions; here we can define a class by simply writing a user-defined class name preceded with the "class" keyword.

Line 5: is an opening flower brace ({) as I explained above in line 3, but this opening brace indicates to the compiler a class block to be opened.
Line 6: As I said in the above line, that is a class optionally containing variables and optionally containing methods or functions; here now in line 6 we defined a function or method as shown below:
static void Main()
C#

One important point you should note is that a program can contain multiple classes under one namespace but in that class at least one class should contain this Main() method because the compiler starts its job from this Main() function; if you do not have this function in your program then the compiler cannot compile your program because it doesn't know from where it should start compiling and instead just raises an error. So you can define a Main() method as we defined in our program.

Note: A deep discussion on this Main() method and their uses will be in future chapters.
Line 7: An opening flower brace ({) as I explained above for lines 3 & 5, but this brace indicates to the compiler that the Main() method block has opened.
Line 8: This is the first statement in our Main() method. Note that every statement should end with a semicolon (;). Here in the following statement, Console is the class in the System namespace as explained above and WriteLine() is a static method. I will explain later what the difference is between a static method and a normal method; for now just remember that static methods are called directly by its class name as in this case. In the code WriteLine() is the method we are sending the text "This Is Our First CSharp Program.". When the compiler reads this line it just prints the text "This Is Our First CSharp Program." on the command prompt at runtime.
Console.WriteLine("This Is Our First CSharp Program.");
C#

Note: A Console class is designed to read and print or write a text on the console; i.e., in a command prompt, this class contains functions that are used to read and print text on the console.

In the Console class we have two methods to print text in the command prompt. There are WriteLine() and Write() methods. Both print the text on the screen but the WriteLine() method prints the text followed by a new line character (\n); with this, every WriteLine() method writes text on separate new lines. For example:

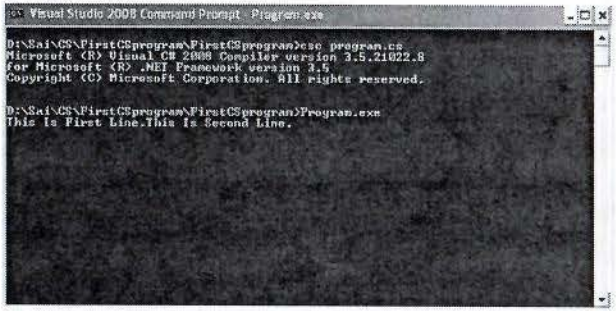
Console.Write ("This Is First Line.");
Console.Write ("This Is Second Line.");
C#

The output will be

In the same example, replace the WriteLine() method with the Write() method as shown below:

```
Console.Write ("This Is First Line.");
Console.Write ("This Is Second Line.");
C#
```

The output will be:



Line 9: As we know by the above discussion of the Console class we said that we can read and print the text on the console; above we came to know how we can print a text on the console screen but how will we read text written by a user at runtime? The answer is, as I said, that the Console class has functionality for reading and printing text, as we have seen to print the text on the screen we used the WriteLine() and Write() methods in the Console class. To read we have another method, ReadLine(); the name itself resembles the functionality of

this method i.e.., it reads the text which is entred by the user at runtime.

```
Console.ReadLine();
C#
```

Let's take one example program to understand how the ReadLine() method works and its importance. Here in this example we accept two numbers from the user ar runtime and we print the sum of the two numbers. Here is the Program:

```
using System;
namespace FirstCSprogram
{
  class Program
  {
    static void Main()
    {
      int a, b;
      Console.Write("Enter First Number : ");
      a = int.Parse(Console.ReadLine());
      Console.Write("Enter First Number : ");
      b = int.Parse(Console.ReadLine());
      Console.WriteLine("The Sum Of {0} and {1} Is : {2}", a, b, (a + b).ToString());
      Console.ReadLine();
    }
  }
}
C#
```

When you compile and run this program in a console it will ask you to enter the first number; after entering the fist number and pressing enter, again it prompts for the second number; after giving the second number it will calculate the sum of these two numbers and display the result on the console.

Here is how the output screen will look like:

Note: Since we have the ReadLine() to read a line of text, similarly we have another method Read() which reads the next character on the input stream.
Line 10: The next statement is:
return;
C#

This statement indicates to the compiler that the end of the scope of its belonging Block. It returns the control to its calling method which is waiting for this control on the stack memory.

Line 11, 12, and 13: These three lines indicate the end of their corresponding block. As I explained, the opening of the block is represented by opening flower braces ({); similarly the closing of the block is represented by closing flower braces (}).

Project references are a new feature in TypeScript 3.0 that allow you to structure your TypeScript programs into smaller pieces.

By doing this, you can greatly improve build times, enforce logical separation between components, and organize your code in new and better ways.

We're also introducing a new mode for `tsc`, the `--build` flag, that works hand in hand with project references to enable faster TypeScript builds.

An Example Project :

Let's look at a fairly normal program and see how project references can help us better organize it. Imagine you have a project with two modules, `converter` and `units`, and a corresponding test file for each:

```
/
├── src/
│   ├── converter.ts
│   └── units.ts
├── test/
│   ├── converter-tests.ts
│   └── units-tests.ts
└── tsconfig.json
```

The test files import the implementation files and do some testing:

```
// converter-tests.ts
import * as converter from "../converter";
assert.areEqual(converter.celsiusToFahrenheit(0), 32);
```

Previously, this structure was rather awkward to work with if you used a single tsconfig file:

- It was possible for the implementation files to import the test files
- It wasn't possible to build `test` and `src` at the same time without having `src` appear in the output folder name, which you probably don't want
- Changing just the *internals* in the implementation files required *typechecking* the tests again, even though this wouldn't ever cause new errors
- Changing just the tests required typechecking the implementation again, even if nothing changed

You could use multiple tsconfig files to solve *some* of those problems, but new ones would appear:

- There's no built-in up-to-date checking, so you end up always running `tsc` twice
- Invoking `tsc` twice incurs more startup time overhead

- `tsc -w` can't run on multiple config files at once

Project references can solve all of these problems and more.

What is a Project Reference?

`tsconfig.json` files have a new top-level property, `references`. It's an array of objects that specifies projects to reference:

```
{
  "compilerOptions": {
    // The usual
  },
  "references": [
    { "path": "../src" }
  ]
}
```

The `path` property of each reference can point to a directory containing a `tsconfig.json` file, or to the config file itself (which may have any name).

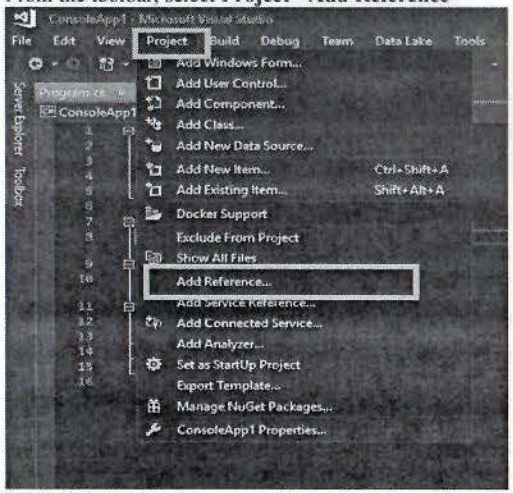
When you reference a project, new things happen:

- Importing modules from a referenced project will instead load its *output* declaration file (`.d.ts`)
- If the referenced project produces an `outFile`, the output file `.d.ts` file's declarations will be visible in this project
- Build mode (see below) will automatically build the referenced project if needed

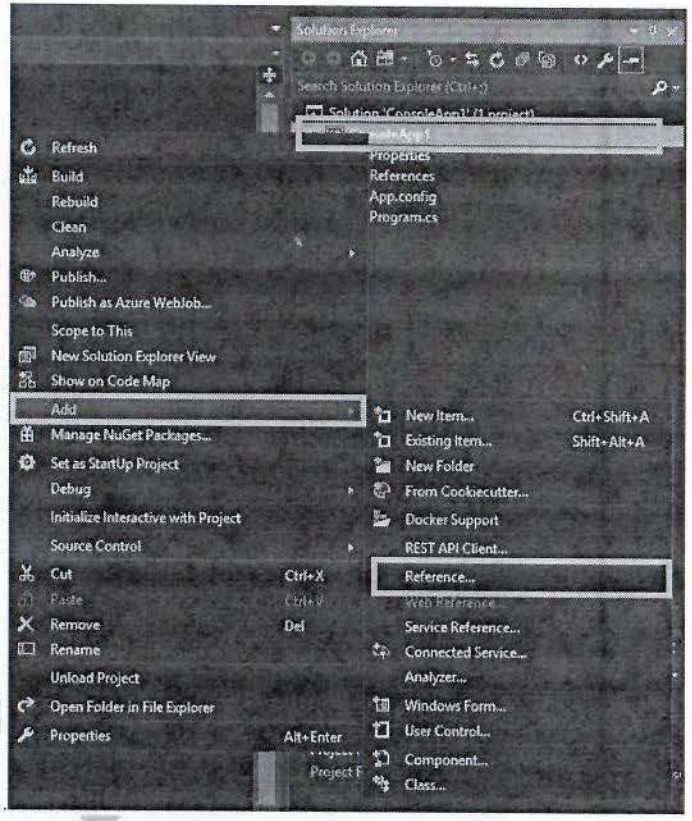
By separating into multiple projects, you can greatly improve the speed of typechecking and compiling, reduce memory usage when using an editor, and improve enforcement of the logical groupings of your program.

Project - Add Reference :

1. Within your project in Visual Studio, you must first open the **Reference Manage** dialog box. This can be accomplished two ways:
   - From the toolbar, select **Project - Add Reference**



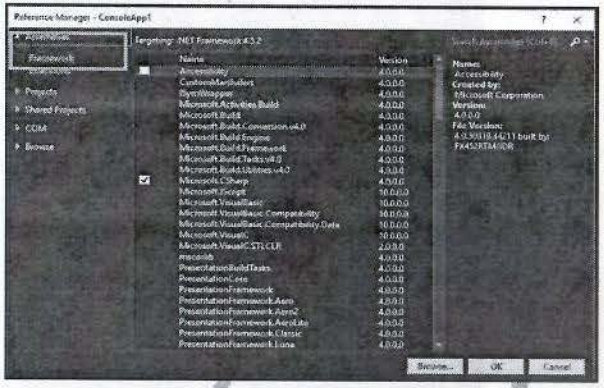
   - In the Solution Explorer pane, Right click the project then select **Add - Reference**



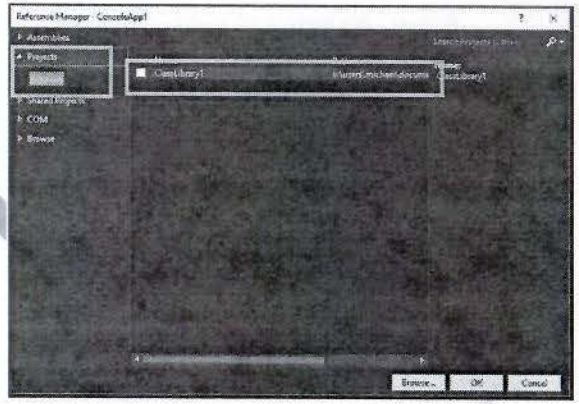
2. Within the Reference Manager, you select the appropriate category.

   Categories include:

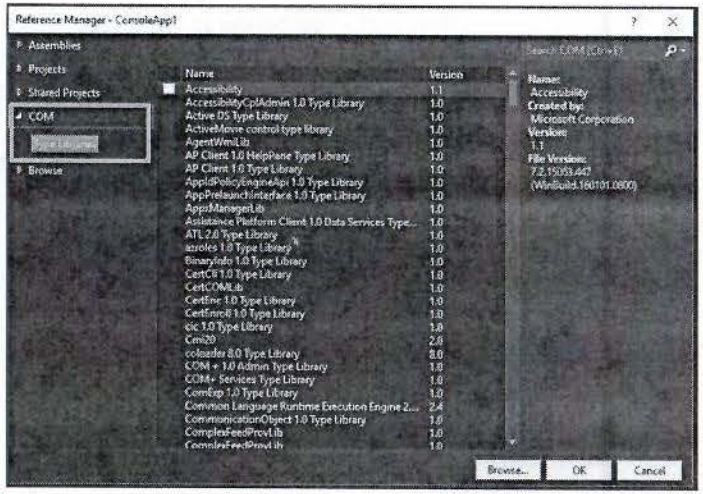
Framework - This includes all .Net libraries installed on the local machine



Projects - This includes all class library projects in your Visual Studio Solution



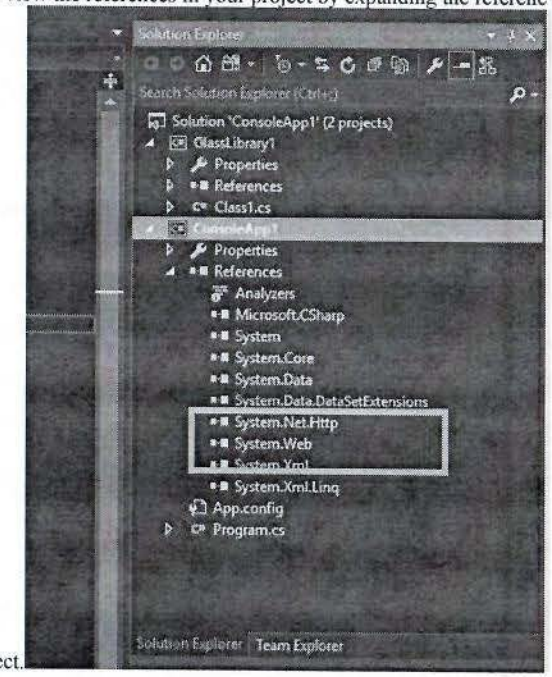
Com - This includes any Com components registered on the local machine.



Browse - this opens a file explorer dialog box and allows you to locate the appropriate dll or exe.

3. Using the Reference Manager, Choose the category and library within that category. In the example below, I have chosen the **System.Web** library in the Framework category.

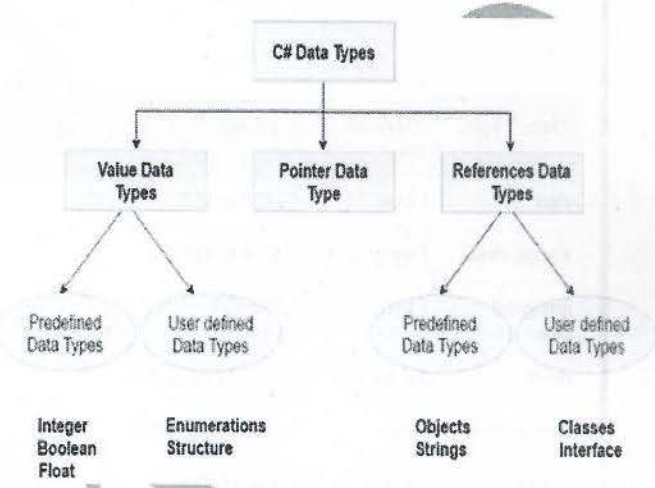
Once an item is selected, click the **OK** button and the reference will be added to your project. You can view the references in your project by expanding the referenc category within your project.



You can also use the object explorer to browse the classes contained within the referenced library.

Data types in .Net and CTS equivalents:

A **data type** specifies the type of data that a variable can store such as integer, floating, character etc.



Types	Data Types
Value Data Type	short, int, char, float, double etc
Reference Data Type	String, Class, Object and Interface

Pointer Data Type	Pointers

- **Value Data Type :**

Data Types	Memory Size	Range
char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	4 byte	-2,147,483,648 to -2,147,483,647
signed int	4 byte	-2,147,483,648 to -2,147,483,647
unsigned int	4 byte	0 to 4,294,967,295
long	8 byte	?9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
signed long	8 byte	?9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

unsigned long	8 byte	0 - 18,446,744,073,709,551,615
float	4 byte	$1.5 * 10^{-45}$ - $3.4 * 10^{38}$, 7-digit precision
double	8 byte	$5.0 * 10^{-324}$ - $1.7 * 10^{308}$, 15-digit precision
decimal	16 byte	at least $-7.9 * 10^{?28}$ - $7.9 * 10^{28}$, with at least 28-digit precision

The value data types are integer-based and floating-point based. C# language supports both signed and unsigned literals.

There are 2 types of value data type in C# language.

1) Predefined Data Types - such as Integer, Boolean, Float, etc.

2) User defined Data Types - such as Structure, Enumerations, etc.

The memory size of data types may change according to 32 or 64 bit operating system.

Let's see the value data types. It size is given according to 32 bit OS.

- **Reference Data Type**

The reference data types do not contain the actual data stored in a variable, but they contain a reference to the variables.

If the data is changed by one of the variables, the other variable automatically reflects this change in value.

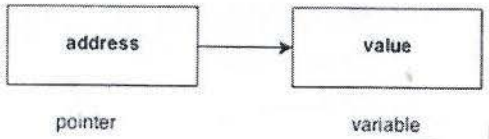
There are 2 types of reference data type in C# language.

1) Predefined Types - such as Objects, String.

2) User defined Types - such as Classes, Interface.

▪ Pointer Data Type

The pointer in C# language is a variable, it is also known as locator or indicator that points to an address of a value.



Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Declaring a pointer

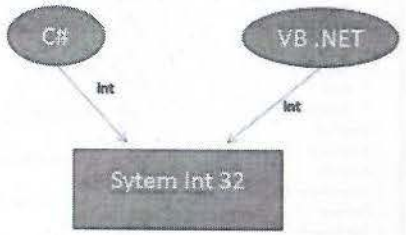
The pointer in C# language can be declared using * (asterisk symbol).

- **int** * a; //pointer to int
- **char** * c; //pointer to char

The Common Type System (CTS) standardizes the data types of all programming languages using .NET under the umbrella of .NET to a common data type for easy and smooth communication among these .NET languages.

How CTS converts the data type to a common data type

To implement or see how CTS is converting the data type to a common data type, for example, when we declare an int type data type in C# and VB.Net then they are converted to int32. In other words, now both will have a common data type that provides flexible communication between these two languages.



Let's take a live example using a Double data type and we will see how .NET helps to make the data types of C# and VB.NET common to each other for easy communication.

Live Example

We will create a new Visual Basic (Console Application) project.

Now we will add one more project of Visual C# (Console Application).

Now to see how CTS is helping with the data types of the programming languages and providing a common environment we will declare a Double data type (*note you can use other variables for your reference) variable in both the C# program.cs and Module1.vb.

C# Data type Declaration

```
Csharp.Program
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Csharp
{
    class Program
    {
        static void Main(string[] args)
        {
            double CustomerValue;
        }
    }
}
```

Visual Basic Data type Declaration

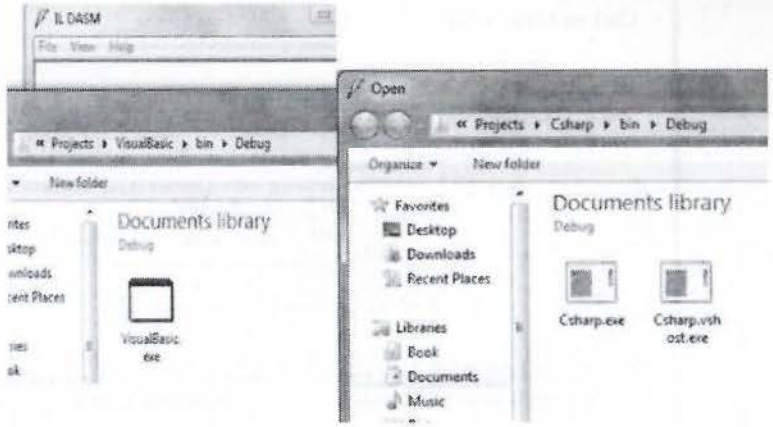
```
Module1
Module Module1

    Sub Main()

        Dim CustomerValue As Double

    End Sub

End Module
```

Now start debugging both of these Console Applications. Once you are done with Debugging the application, we will now use an important tool provided with Visual Studio Called "*ILDASM*".

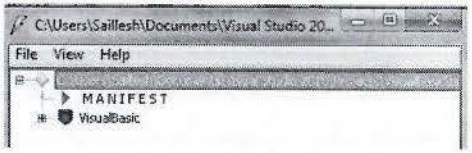
You can learn about "*ILDASM*" by Control + Click on the "*ILDASM*" link and gain information on how *ILDASM* is helpful. So returning back to CTS, now open "ILDASM" from the Visual Studio x64 Command Prompt (2010).

After execution of the command, the "*ILDASM*" window will open as in the following:

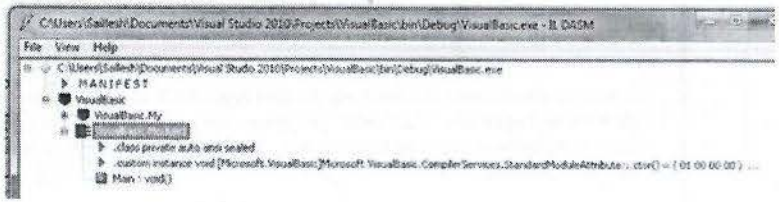
So now to check how CTS converts the data type of C# and VB.Net to be common to each other we will open two "ILDASM" programs then click on File then open the exe file of the desired application. Here we have created VBCONSOLE and C#, so now we will open the .exe in "ILDASM".



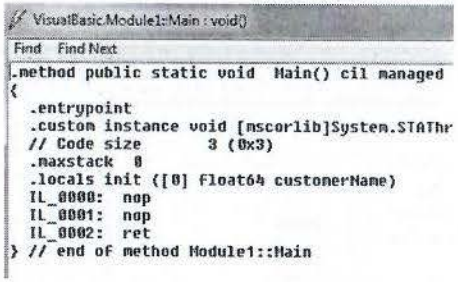
Now once we have opened both of the Console Applications the "*ILDASM*" window will appear as in the following snapshot.



Click on Visual Basic and then on VisualBasic.Module1, you can see there is a void main function as shown below.



Click on Main : void()



You will find that the Double variable is converted to float64 and the same in case of the C# Console Application the Double variable is converted to float64.

CTS converting data type to Common type for both programming languages

Hence we can clearly see how the .NET Framework is converting the data type to a Common Type System, so it is a flexible communication among languages that are compliant with .NET Framework. I hope this article was helpful and it is easy to understand the concept of CTS and how using "ILDASM" we figured that the Common type System helps to bring the data types of all programming languages to a common type.

Methods:

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method.

The `Main` method is the entry point for every C# application and it's called by the common language runtime (CLR) when the program is started. In an application that uses top-level statements, the `Main` method is generated by the compiler and contains all top-level statements.

Method signatures

Methods are declared in a class, struct, or interface by specifying the access level such as `public` or `private`, optional modifiers such as `abstract` or `sealed`, the return value, the name of the method, and any method parameters. These parts together are the signature of the method.

A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters. This class contains four methods:

```
C#
abstractclassMotorcycle
{
// Anyone can call this.
publicvoidStartEngine() {/* Method statements here */ }

// Only derived classes can call this.
protectedvoidAddGas(int gallons) { /* Method statements here */ }

// Derived classes can override the base class implementation.
publicvirtualintDrive(int miles, int speed) { /* Method statements here */return1; }

// Derived classes must implement this.
publicabstractdoubleGetTopSpeed();
}
```

Method access

Calling a method on an object is like accessing a field. After the object name, add a period, the name of the method, and parentheses. Arguments are listed within the parentheses, and are separated by commas. The methods of the `Motorcycle` class can therefore be called as in the following example:

```
C#
classTestMotorcycle : Motorcycle
{

publicoverridedoubleGetTopSpeed()
  {
return108.4;
```



```
  }

staticvoidMain()
  {

    TestMotorcycle moto = new TestMotorcycle();

    moto.StartEngine();
    moto.AddGas(15);
    moto.Drive(5, 20);
double speed = moto.GetTopSpeed();
    Console.WriteLine("My top speed is {0}", speed);
  }
}
```

Method parameters vs. arguments

The method definition specifies the names and types of any parameters that are required. When calling code calls the method, it provides concrete values called arguments for each parameter. The arguments must be compatible with the parameter type but the argument name (if any) used in the calling code doesn't have to be the same as the parameter named defined in the method. For example:

```
C#
publicvoidCaller()
{
int numA = 4;
// Call with an int variable.
int productA = Square(numA);

int numB = 32;
// Call with another int variable.
int productB = Square(numB);

// Call with an integer literal.
int productC = Square(12);

// Call with an expression that evaluates to int.
```

```
    productC = Square(productA * 3);
}

intSquare(int i)
{
// Store input argument in a local variable.
int input = i;
return input * input;
}
```

Passing by reference vs. passing by value

By default, when an instance of a value type is passed to a method, its is passed instead of the instance itself. Therefore, changes to the argument have no effect on the original instance in the calling method. To pass a value-type instance by reference, use the `ref` keyword. For more information, see Passing Value-Type Parameters.

When an object of a reference type is passed to a method, a reference to the object is passed. That is, the method receives not the object itself but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the argument in the calling method, even if you pass the object by value.

You create a reference type by using the `class` keyword, as the following example shows:

```
C#
publicclassSampleRefType
{
publicintvalue;
}
```

Now, if you pass an object that is based on this type to a method, a reference to the object is passed. The following example passes an object of type `SampleRefType` to method `ModifyObject`:

```
C#
publicstaticvoidTestRefType()
{
   SampleRefType rt = new SampleRefType();
   rt.value = 44;
```



```
   ModifyObject(rt);
   Console.WriteLine(rt.value);
}

staticvoidModifyObject(SampleRefType obj)
{
   obj.value = 33;
}
```

The example does essentially the same thing as the previous example in that it passes an argument by value to a method. But, because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `value` field of the parameter, `obj`, also changes the `value` field of the argument, `rt`, in the `TestRefType` method. The `TestRefType` method displays 33 as the output.

Method overloading allows programmers to use multiple methods with the same name. The methods are differentiated with their number and type of method arguments. Method overloading is an example of the polymorphism feature of an object oriented programming language.

Method overloading can be achieved by the following:

- By changing number of parameters in a method
- By changing the order of parameters in a method
- By using different data types for parameters

Here is an example of method overloading.

```
1.  public class Methodoveloading
2.   {
3.     public int add(int a, int b)  //two int type Parameters method
4.     {
5.        return a + b;
6.
7.     }
8.     public int add(int a, int b,int c)  //three int type Parameters with same method same
   as above
9.     {
10.       return a + b+c;
11.
```

```
12.    }
13.    public float add(float a, float b,float c,float d)  //four float type Parameters with sam
   e method same as above two method
14.    {
15.        return a + b+c+d;
16.
17.    }
18.  }
```

In the above example, there are three methods with the same names but each method differs in number of parameters and also types of parameters. This method is called method overloading. One of the common example of method overloading is formatting a string using the Format method. The string can be formatted using various input types such as int, double, numbers, chars, and other strings.

As the name suggests 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. This concept is introduced in *C# 4.0*.

Important Points:

- You can use optional parameters in **Methods**, Constructors, Indexers, and Delegates.
- Each and every optional parameter contains a default value which is the part of its definition.
- If we do not pass any parameter to the optional arguments, then it takes its default value.
- The default value of an optional parameter is a constant expression.
- The optional parameters are always defined at the end of the parameter list. Or in other words, the last parameter of the method, constructor, etc. is the optional parameter.

Example:

static public void scholar(string fname, string lname, int age = 20, string branch = "Computer Science")

In the above example, *int age = 20* and *string branch = "Computer Science"* is the optional parameter and has its default value.

Optional Parameters

```
// C# program to illustrate the
// concept of optional parameters
using System;

class GFG {

  // This method contains two regular
  // parameters, i.e. fname and lname
  // And two optional parameters, i.e.
  // age and branch
  static public void scholar(string fname,
                             string lname,
                             int age = 20,
    string branch = "Computer science")
```

```
{
    Console.WriteLine("First name: {0}", fname);
    Console.WriteLine("Last name: {0}", lname);
    Console.WriteLine("Age: {0}", age);
    Console.WriteLine("Branch: {0}", branch);
}

// Main Method
static public void Main()
{

    // Calling the scholar method
    scholar("Ankita", "Saini");
    scholar("Siya", "Joshi", 30);
    scholar("Rohan", "Joshi", 37,
      "Information Technology");
```



```
    }
}
```

Output:

First name: Ankita

Last name: Saini

Age: 20

Branch: Computer science

First name: Siya

Last name: Joshi

Age: 30

Branch: Computer science

First name: Rohan

Last name: Joshi

Age: 37

Branch: Information Technology

Explanation: In the above example, the scholar method contains four parameters out of these four parameters two are regular parameters, i.e. *fname* and *lname* and two are optional parameters, i.e. *age* and *branch*. These optional parameters contain their default values. When you do not pass the value of these parameters, then the optional parameters use their default value. And when you pass the parameters for optional parameters, then they will take the passed value not their default value.

What will happen if we use optional parameters other than the last parameter?
It will give the compile-time error "**Optional parameter cannot precede requiredparameters**" as optional parameters are always defined at the end of the parameter list. Or in other words, the last parameter of the method, constructor, etc. is the optional parameter.
Example:

```
// C# program to illustrate the concept
// of optional parameters
using System;

class GFG {

  // This method contains two regular
  // parameters, i.e. fname and lname
  // And one optional parameters, i.e. age
  static public void scholar(string fname,
          int age = 20, string lname)
  {
    Console.WriteLine("First name: {0}", fname);
    Console.WriteLine("Last name: {0}", lname);
    Console.WriteLine("Age: {0}", age);
  }
```



```
  // Main Method
  static public void Main()
  {

    // Calling the scholar method
    scholar("Ankita", "Saini");
  }
}
```

Compile-time Error:
prog.cs(11,26): error CS1737: Optional parameter cannot precede required parameters

Positional vs. Named Parameters

1. A positional parameter is linked by its position.
2. Positional parameters must be specified in the order in which they appear.
3. Named parameters are specified by assigning values to their names.
4. For an attribute, you can also create named parameters
5. named parameters can be assigned initial values by using their names.
6. A named parameter is supported by either a public field or property, which must not be read-only.

Named parameters free you from matching the order of arguments to the order of parameters in the parameter lists of called methods. The argument for each parameter can be specified by parameter name. For example, a function that prints order details (such as, seller name, order number & product name) can be called by sending arguments by position, in the order defined by the function.

C#
PrintOrderDetails("Gift Shop", 31, "Red Mug");

If you don't remember the order of the parameters but know their names, you can send the arguments in any order.

C#
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);

Named arguments also improve the readability of your code by identifying what each argument represents. In the example method below, the sellerName can't be null or white space. As both sellerName and productName are string types, instead of sending arguments by position, it makes sense to use named arguments to disambiguate the two and reduce confusion for anyone reading the code.

Named arguments, when used with positional arguments, are valid as long as

- they're not followed by any positional arguments, or

  C#
  PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");

- they're used in the correct position. In the example below, the parameter orderNum is in the correct position but isn't explicitly named.

  C#
  PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");

Positional arguments that follow any out-of-order named arguments are invalid.

C#
// This generates CS1738: Named argument specifications must appear after all fixed arguments have been specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");

What are the positional parameters?
A positional parameter is a parameter denoted by one or more digits, other than the single digit 0 . Positional parameters are assigned from the shell's arguments when it is invoked, and may be reassigned using the set builtin command.

Why function parameters are positional?
When defining custom functions, the variable(s) that hold values being passed in are called parameters. Traditionally, the order of parameter lists are critical in that the order of the parameters and arguments match up one-to-one. These are called positional parameters.

What is the difference between positional parameters and named parameters?
You can use either positional or named parameter notation when passing parameters to a function or procedure. If you specify parameters using positional notation, you must list the parameters in the order that they are declared. If you specify parameters with named notation, the order of the parameters doesn't matter.

▪ Using Params:

Params is an important keyword in C#. It is used as a parameter which can take the **variable number of arguments.**
Important Point About Params Keyword :

- 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.

Examples: To illustrate the use of **params keyword**

- Simple program to show the params keyword usage

```
// C# program to illustrate the
// use of params keyword
usingSystem;
namespaceExamples {

classGeeks {

    // function containing params
parameters
    publicstaticintAdd(paramsint[]
ListNumbers)
    {
        inttotal = 0;
```



```
        // foreach loop
        foreach(inti inListNumbers)
        {
            total += i;
        }
        returntotal;
    }

// Driver Code
staticvoidMain(string[] args)
{

    // Calling function by passing 5
    // arguments as follows
    inty = Add(12,13,10,15,56);
```

```
        // Displaying result
        Console.WriteLine(y);
    }
    }
    }
```

- **Output :**

106

▪ Local Functions:

Local functions are methods of a type that are nested in another member. They can only be called from their containing member. Local functions can be declared in and called from:
Methods, especially iterator methods and async methods

a. Constructors
b. Property accessors
c. Event accessors
d. Anonymous methods
e. Lambda expressions
f. Finalizers
g. Other local functions

Local function syntax

A Local function is defined as a nested method inside a containing member. Its definition has the following syntax:

C#
<modifiers><return-type><method-name><parameter-list>
You can use the following modifiers with a local function:

- async
- unsafe
- static

A static local function can't capture local variables or instance state.
extern An external local function must be static.
All local variables that are defined in the containing member, including its method parameters, are accessible in a non-static local function.
Unlike a method definition, a local function definition cannot include the member access modifier. Because all local functions are private, including an access modifier, such as the private keyword, generates compiler error CS0106, "The modifier 'private' is not valid for this item."
The following example defines a local function named AppendPathSeparator that is private to a method named GetText:
C#

```
private static string GetText(string path, string filename)
{
    var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
    var text = reader.ReadToEnd();
    return text;
```

```
    string AppendPathSeparator(string filepath)
    {
      return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
    }
}
```

▪ Properties

A property is like a combination of a variable and a method, and it has two methods: a `get` and a `set` method:

Example

```
classPerson
{
privatestring name;// field

publicstring Name  // property
{
get{return name;}// get method
set{ name =value;}// set method
}
}
```


Example explained

The `Name` property is associated with the `name` field. It is a good practice to use the same name for both the property and the private field, but with an uppercase first letter.

The `get` method returns the value of the variable `name`.

The `set` method assigns a `value` to the `name` variable. The `value` keyword represents the value we assign to the property.

we can use the `Name` property to access and update the `private` field of the `Person` class:

```
Example
classPerson
{
privatestring name;// field
publicstring Name   // property
{
get{return name;}
set{ name =value;}
}
}
```

```
classProgram
{
staticvoidMain(string[] args)
{
Person myObj =newPerson();
    myObj.Name ="Liam";
    Console.WriteLine(myObj.Name);
}
}
```

The output will be:

Liam

A property looks a bit like a crossover between a field and a method, because it's declared much like a field with visibility, a data type and a name, but it also

has a body, like a method, for controlling the behavior:

```
public string Name
{
    get { return _name; }
    set { _name = value; }
}
```

Notice the special get and set keywords. They are used exclusively for properties, to control the behavior when reading (get'ing) and writing (set'ing) the field. You can have properties with only a get OR a set implementation, to create read-only or write-only properties, and you can even control the visibility of the get or set implementations, e.g. to create a property which can be read from anywhere (public) but only modified from inside the declaring class (private).

You will also notice that I refer to a field called _name. You will have to declare that in your class as well, so that your property can use it. A common usage pattern for fields and properties will look like this:

```
private string _name = "John Doe";

public string Name
{
```

```
    get { return _name; }
    set { _name = value; }
}
```

You can now see how the field and the property works together: The get method will return the value of the _name field, while the set method will assign the passed value to the _name field. In the set method, we use the special keyword value which, in this specific situation, will refer to the value passed to the property.

So, this is pretty much as basic as it gets and at this point, we don't do anything that couldn't be accomplished with a simple public field. But at a later point, you may decide that you want to take more control of how other classes can work with the name and since you have implemented this as a property, you are free to modify the implementation without disturbing anyone using your class. For instance, the Name property could be modified to look like this:

Download, Edit & Run this example!

```
private string _name = "John Doe";


public string Name
{
    get
```



```
    {
        return _name.ToUpper();
    }
    set
    {
        if(!value.Contains(" "))
            throw new Exception("Please specify both
first and last name!");
        _name = value;
    }
}
```

The **get method** now enforces that the value returned is always in UPPERCASE, no matter which case the backing field (_name) is in. In the **set method**, we have added a couple of lines of code to check whether the passed value contains a space, because we have decided that the name should always consist of both a first and a last name - if this is not the case, an exception is thrown. This is all very crude and simplified, but it should illustrate the full level of control you get when using properties.

- **Readonly Properties :**

The readonly keyword is a modifier that can be used in four contexts:

In a field declaration, readonly indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class. A readonly field can be assigned and reassigned multiple times within the field declaration and constructor.

A readonly field can't be assigned after the constructor exits. This rule has different implications for value types and reference types:

Because value types directly contain their data, a field that is a readonly value type is immutable.

Because reference types contain a reference to their data, a field that is a readonly reference type must always refer to the same object. That object isn't immutable. The readonly modifier prevents the field from being replaced by a different instance of the reference type. However, the modifier doesn't prevent the instance data of the field from being modified through the read-only field.

Readonly field example

In this example, the value of the field year can't be changed in the method ChangeYear, even though it's assigned a value in the class constructor:


```
class Age
{
    private readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if
uncommented.
    }
}
```

You can assign a value to a readonly field only in the following contexts:

When the variable is initialized in the declaration, for example:

```
public readonly int y = 5;
```

In an instance constructor of the class that contains the instance field declaration.

In the static constructor of the class that contains the static field declaration.

These constructor contexts are also the only contexts in which it's valid to pass a readonly field as an out or ref parameter.

Ref readonly return example

The readonly modifier on a ref return indicates that the returned reference can't be modified. The following example returns a reference to the origin. It uses the readonly modifier to indicate that callers can't modify the origin:

C#

```
private static readonly SamplePoint s_origin = new SamplePoint(0, 0, 0);

public static ref readonly SamplePoint Origin => ref s_origin;
```

The type returned doesn't need to be a readonly struct. Any type that can be returned by ref can be returned by ref readonly.

C# readonly property

The property that has only the `get` accessor and no `set` accessor is called `readonly` property.

A readonly property allows you to only read the value of a private field. The readonly modifier specifies that the member is not allowed to modify. When you try to set the value of the readonly field/property, a compilation error occurs.

```
1  classEmployee
2     {
3          privatestringname;
4
5          // ReadOnly Property
6          publicstringName
7          {
8            get{
9                  returnname;
10               }
11         }
12         // Constructor
13         publicEmployee(stringname)
14         {
15             this.name = name;
16         }
17    }
18    publicclassPrograms
19    {
20         publicstaticvoidMain(string[] args)
21         {
22       Employee employee = newEmployee("Shekh Ali"); //
Constructor
23 stringname = employee.Name; // Calling get block of the
24 ReadOnly property
25
```

```
26          Console.WriteLine($"Name : {name}");
27      employee.Name = "Mark Adams"; // Calling set block to
28  assign the new value.
29              Console.WriteLine($"Name : {employee.Name }")
30
31              Console.ReadLine();
            }
        }
```

```
public class Programs
{
    public static void Main(string[] args)
    {
        Employee employee = new Employee("Shekh Ali"); // Constructor
        string name = employee.Name; // Calling get block of the ReadOnly property

        Console.WriteLine($"Name : {name}");
        employee.Name = "Mark Adams"; // Calling set block to assign the new value.
        Console.W  (local variable) Employee employee
        Console.R  Property or indexer 'Employee.Name' cannot be assigned to -- it is read only
    }
```

C#

ReadOnly Property

As we can see in the above example, It throws an error when we try to set the value to the property outside the constructor.

Conclusion

This article has provided us with the following insights:

- Use of the properties and its Syntax.
- About the write-only and `readonly` property in C#
- Use of static property
- Use of Auto implemented properties in C#

Constructors

A constructor is a special method of the class which gets automatically invoked whenever an instance of the class is created. Like methods, a constructor also contains the collection of instructions that are executed at the time of Object creation. It is used to assign initial values to the **data members** of the same class.

Example :

```
class Geek
{
 .....
 // Constructor
 public Geek() {}
 .....
}

// an object is created of Geek class,
// So above constructor is called
Geek obj = new Geek();
```

A constructor is a **special method** that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

Example

Create a constructor:

```
// Create a Car class
classCar
{
publicstring model;// Create a field

// Create a class constructor for the Car class
publicCar()
{
  model ="Mustang";// Set the initial value for model
}

staticvoidMain(string[] args)
{
```



```
Car Ford =newCar();// Create an object of the Car Class (this will call the constructor)
  Console.WriteLine(Ford.model);// Print the value of model
}
}

// Outputs "Mustang"
```

- **Importance of Constructor :**

1) Constructor of a class must have the same name as the class name in which it resides.
2) A constructor can not be abstract, final, and Synchronized.
3) Within a class, you can create only one static constructor.
4) A constructor doesn't have any return type, not even void.
5) A static constructor cannot be a parameterized constructor.
6) A class can have any number of constructors.
7) Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

- **Types of Constructor**

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

Default Constructor

A constructor with no parameters is called a default constructor. A default constructor has every instance of the class to be initialized to the same values. The default constructor initializes all numeric fields to zero and all string and object fields to null inside a class.

Example :

C#

```
// C# Program to illustrate calling
// a Default constructor
usingSystem;
namespaceDefaultConstructorExample {

classGeek {

    intnum;
    stringname;
```



```
    // this would be invoked while the
    // object of that class created.
    Geek()
    {
        Console.WriteLine("Constructor
Called");
    }

    // Main Method
    publicstaticvoidMain()
    {

        // this would invoke default
        // constructor.
        Geek geek1 = newGeek();

        // Default constructor provides
```

```
        // the default values to the
        // int and object as 0, null
        // Note:
        // It Give Warning because
        // Fields are not assign
        Console.WriteLine(geek1.name);
        Console.WriteLine(geek1.num);
    }
}
}
```

Output :
Constructor Called
0

Note This will also show some warnings as follows:
prog.cs(8, 6): warning CS0649: Field `DefaultConstructorExample.Geek.num' is never assigned to, and will always have its default value `0'prog.cs(9, 9): warning CS0649: Field `DefaultConstructorExample.Geek.name' is never assigned to, and will always have its default value `null'

Parameterized Constructor

A constructor having at least one parameter is called as parameterized constructor. It can initialize each instance of the class to different values.

Example :

- C#

```
// C# Program to illustrate calling of
// parameterized constructor.
usingSystem;
namespaceParameterizedConstructorExample {

classGeek {

    // data members of the class.
    String name;
    intid;

    // parameterized constructor would
    // initialized data members with
    // the values of passed arguments
```

```
    // while object of that class created.
    Geek(String name, intid)
    {
        this.name = name;
        this.id = id;
    }

    // Main Method
    publicstaticvoidMain()
    {

        // This will invoke parameterized
        // constructor.
        Geek geek1 = newGeek("GFG", 1);
        Console.WriteLine("GeekName = "+
geek1.name +
                          " and GeekId = "+
geek1.id);
```



```
    }
}
}
```

Output :
GeekName = GFG and GeekId = 1

Copy Constructor

This constructor creates an object by copying variables from another object. Its main use is to initialize a new instance to the values of an existing instance.

Example :

C#

```
// C# Program to illustrate calling
// a Copy constructor
usingSystem;
namespacecopyConstructorExample {

classGeeks {
```

```
privatestringmonth;
privateintyear;

// declaring Copy constructor
publicGeeks(Geeks s)
{
    month = s.month;
    year = s.year;
}

// Instance constructor
publicGeeks(stringmonth, intyear)
{
    this.month = month;
    this.year = year;
}
```



```
    // Get details of Geeks
    publicstringDetails
    {
        get
        {
            return"Month: "+ month.ToString() +
                        "\nYear: "+
year.ToString();
        }
    }

    // Main Method
    publicstaticvoidMain()
    {

        // Create a new Geeks object.
        Geeks g1 = newGeeks("June", 2018);
```

```
        // here is g1 details is copied to g2.
        Geeks g2 = newGeeks(g1);

        Console.WriteLine(g2.Details);
    }
}
}
```

Output :

```
Month: June
Year: 2018
```

Private Constructor

If a constructor is created with private specifier is known as Private Constructor. It is not possible for other classes to derive from this class and also it's not possible to create an instance of this class.

Points To Remember :

- It is the implementation of a singleton class pattern.
- use private constructor when we have only static members.

- Using private constructor, prevents the creation of the instances of that class.

Example :

- C#

```
// C# Program to illustrate calling
// a Private constructor
usingSystem;
namespaceprivateConstructorExample {

publicclassGeeks {

    // declare private Constructor
    privateGeeks()
    {
    }

    // declare static variable field
```

```
publicstaticintcount_geeks;

// declare static method
publicstaticintgeeks_Count()
{
    return++count_geeks;
}

// Main Method
publicstaticvoidMain()
{
    // If you uncomment the following
    // statement, it will generate
    // an error because the constructor
    // is unaccessible:
```



```
    // Geeks s = new Geeks(); // Error

    Geeks.count_geeks = 99;

    // Accessing without any
    // instance of the class
    Geeks.geeks_Count();

    Console.WriteLine(Geeks.count_geeks);

    // Accessing without any
    // instance of the class
    Geeks.geeks_Count();

    Console.WriteLine(Geeks.count_geeks);
}
```

```
}

}
```

Output :
100
101

Static Constructor

Static Constructor has to be invoked only once in the class and it has been invoked during the creation of the first reference to a static member in the class. A static constructor is initialized static fields or data of the class and to be executed only once.

Points To Remember :

- It can't be called directly.
- When it is executing then the user has no control.
- It does not take access modifiers or any parameters.
- It is called automatically to initialize the class before the first instance created.

Example :

- C#

```
// C# Program to illustrate calling

// a Static constructor
```



```
usingSystem;

namespacestaticConstructorExample {


classgeeks {


    // It is invoked before the first

    // instance constructor is run.

    staticgeeks()

    {


        // The following statement produces

        // the first line of output,

        // and the line occurs only once.

        Console.WriteLine("Static Constructor");

    }


    // Instance constructor.
```

```
publicgeeks(inti)
{
    Console.WriteLine("Instance Constructor "+ i);
}

// Instance method.
publicstringgeeks_detail(stringname, intid)
{
    return"Name:"+ name + " id:"+ id;
}

// Main Method
publicstaticvoidMain()
{

    // Here Both Static and instance
    // constructors are invoked for
```



```
    // first instance
    geeks obj = newgeeks(1);

    Console.WriteLine(obj.geeks_detail("GFG", 1));

    // Here only instance constructor
    // will be invoked
    geeks obj1 = newgeeks(2);

    Console.WriteLine(obj1.geeks_detail("GeeksforGeeks",
2));
    }
}
}
```

Output :

Static Constructor

Instance Constructor 1

Name:GFG id:1

Instance Constructor 2

Name:GeeksforGeeks id:2

- **Object initializers**

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to invoke a constructor followed by lines of assignment statements. The object initializer syntax enables you to specify arguments for a constructor or omit the arguments (and parentheses syntax). The following example shows how to use an object initializer with a named type, Cat and how to invoke the parameterless constructor. Note the use of auto-implemented properties in the Cat class

```
public class Cat
{
   // Auto-implemented properties.
   public int Age { get; set; }
   public string Name { get; set; }

   public Cat()
   {
   }

   public Cat(string name)
   {
      this.Name = name;
   }
```



```
}
```

C#

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

The object initializers syntax allows you to create an instance, and after that it assigns the newly created object, with its assigned properties, to the variable in the assignment.

Object initializers can set indexers, in addition to assigning fields and properties. Consider this basic Matrix class:

C#

```
public class Matrix
{
   private double[,] storage = new double[3, 3];

   public double this[int row, int column]
   {
      // The embedded array will throw out of range exceptions as appropriate.
      get { return storage[row, column]; }
      set { storage[row, column] = value; }
   }
}
```

How to initialize objects by using an object initializer (C# Programming Guide)

You can use object initializers to initialize type objects in a declarative manner without explicitly invoking a constructor for the type.

The following examples show how to use object initializers with named objects. The compiler processes object initializers by first accessing the parameterless instance constructor and then processing the member initializations. Therefore, if the parameterless constructor is declared as private in the class, object initializers that require public access will fail.You must use an object initializer if you're defining an anonymous type.

- **Destructors**

Destructors are used to destruct instances of classes. In this article, you will understand how different C# destructors are when compared to C++ destructors.

In C# you can never call them, the reason is one cannot destroy an object. So who has the control over the destructor (in C#)? it's the .NET frameworks Garbage Collector (GC).

Syntax of Destructor(~)

~ ClassName()

```
usingSystem;
namespacedestructorex
{
classProgram
{
~Program()// destructor define
{
// clean up statement
}
}
}
```


Characteristics of Destructor :

- Destructors (~) cannot be defined in Structs.
- Destructors (~) are only used with classes.
- Destructor cannot be inherited or overloaded.
- Destructor does not take modifiers or have parameters.
- Destructor cannot be called. They are invoked automatically.
- An instance becomes eligible for destruction when it is no longer possible for any code to use the instance.
- The Programmer has no control over when destructor is called because this is determined by Garbage Collector.
- Destructor is called when program exits.
- Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction.
- Destructor implicitly calls Finalize on the base class of object.

Example: The above given code is implicitly translated to the following code:

```
protectedoverridevoidFinalize()
{
try
{
// to clean conditions
}
finally
{
base.Finalize();
}
}
```

Explanation of code: Finalize() is called recursively for all instances in the inheritance chain, from most derived to least derived.

C# Constructor and Destructor Example

Let's see an example of constructor and destructor in C# which is called automatically.

```
1.      using System;
2.        public class Employee
3.        {
4.           public Employee()
5.           {
6.              Console.WriteLine("Constructor Invoked");
7.           }
8.           ~Employee()
9.           {
10.             Console.WriteLine("Destructor Invoked");
11.          }
12.       }
13.       class TestEmployee{
14.          public static void Main(string[] args)
15.          {
16.             Employee e1 = new Employee();
17.             Employee e2 = new Employee();
18.          }
19.       }
```

Output:

Constructor Invoked

Constructor Invoked

Destructor Invoked

Destructor Invoked

- **Dispose**

The Dispose method performs all object cleanup, so the garbage collector no longer needs to call the objects' Object. Finalize override. Therefore, the call to the SuppressFinalize method prevents the garbage collector from running the finalizer. If the type has no finalizer, the call to GC.

Working of dispose() Function:

1. To free and reset the resources that are unmanaged like connections to the databases, files, etc., and to perform a cleanup of the memory, we make use of a function called dispose of () function in C#.
2. The dispose() function in C# must implement the IDisposable interface.
3. The dispose of () function in C# must be called by the user explicitly and not by the garbage collector.
4. The cost with respect to performance while using dispose of () method is null because the dispose of () method cleans up the memory immediately which optimizes the memory and hence usage of this function to clean the unmanaged resources like files and connections to the databases is very much recommended.

Why should we use Dispose And Finalize in C#?

- Garbage collector takes care of all the hard work of freeing up the managed objects in the .NET framework. And that work is taken care by CLR by running a garbage collector on a separate thread.
- The thread keeps watching the roots in the program. The roots are the objects which are still in the scope of program execution and once these objects are out of scope of program execution, they are ready for garbage collection.

- That was about the managed objects. But what about the non managed objects like SQL connection objects, file handlers, http client request object which are not managed by .NET framework.
- We need to explicitly close or dispose the above mentioned objects.
- If we let these objects to remain in memory of the application, there would be a situation where your application would be consuming way too much memory and causing memory leaks.

Static Members of a Class

We can define class members as static using the static keyword. When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.

The keyword static implies that only one instance of the member exists for a class. Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it. Static variables can be initialized outside the member function or class definition. You can also initialize static variables inside the class definition.

The following is an example –

Example

```
usingSystem;

namespaceStaticVarApplication{
  classStaticVar{
    publicstaticint num;

    publicvoid count(){
      num++;
```



```
    }

    publicint getNum(){
      return num;
    }
  }

  classStaticTester{
    staticvoidMain(string[] args){
      StaticVar s1 =newStaticVar();
      StaticVar s2 =newStaticVar();

      s1.count();
      s1.count();
      s1.count();

      s2.count();
      s2.count();
      s2.count();

      Console.WriteLine("Variable num for s1: {0}", s1.getNum());
      Console.WriteLine("Variable num for s2: {0}", s2.getNum());
      Console.ReadKey();
    }
  }
}
```

Output

```
Variable num for s1: 6
```

Variable num for s2: 6

Class Members:

Fields and methods inside classes are often referred to as "Class Members":

Example:

Create a Car class with three class members: **two fields** and **one method.**

```
// The class
class MyClass
{
 // Class members
 string color = "red";      // field
 int maxSpeed = 200;        // field
 public void fullThrottle()  // method
 {
  Console.WriteLine("The car is going as fast as it can!");
 }
}
```

Fields :

In the previous chapter, you learned that variables inside a class are called fields, and that you can access them by creating an object of the class, and by using the dot syntax (.).

The following example will create an object of the Car class, with the name myObj. Then we print the value of the fields color and maxSpeed:

Example

```
class Car
{
 string color = "red";
 int maxSpeed = 200;

 static void Main(string[] args)
 {
  Car myObj = new Car();
  Console.WriteLine(myObj.color);
  Console.WriteLine(myObj.maxSpeed);
 }
}
```

Creating multiple objects of one class:

```
class Car
{
 string model;
 string color;
```

```
int year;

static void Main(string[] args)
{
 Car Ford = new Car();
 Ford.model = "Mustang";
 Ford.color = "red";
 Ford.year = 1969;

 Car Opel = new Car();
 Opel.model = "Astra";
 Opel.color = "white";
 Opel.year = 2005;

 Console.WriteLine(Ford.model);
 Console.WriteLine(Opel.model);
}
}
```

Properties:

Before we start to explain properties, you should have a basic understanding of "**Encapsulation**".

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare fields/variables as private
- provide public get and set methods, through **properties**, to access and update the value of a private field

Properties

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, sometimes we need to access them - and it can be done with properties.

A property is like a combination of a variable and a method, and it has two methods: a get and a set method:

Example

```
class Person
{
 private string name; // field

 public string Name  // property
 {
  get { return name; }  // get method
  set { name = value; }  // set method
 }
}
```

Example explained

The Name property is associated with the name field. It is a good practice to use the same name for both the property and the private field, but with an uppercase first letter.

The get method returns the value of the variable name.

The set method assigns a value to the name variable. The value keyword represents the value we assign to the property.

Using automatic properties:

```
class Person
{
 public string Name  // property
 { get; set; }
}

class Program
{
 static void Main(string[] args)
 {
  Person myObj = new Person();
  myObj.Name = "Liam";
  Console.WriteLine(myObj.Name);
 }
}
```

The output will be:

Liam

Constructors

A constructor is a **special method** that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

Example

Create a constructor:

```
// Create a Car class
class Car
{
 public string model;  // Create a field

 // Create a class constructor for the Car class
 public Car()
 {
  model = "Mustang"; // Set the initial value for model
 }

 static void Main(string[] args)
 {
  Car Ford = new Car();  // Create an object of the Car Class (this will call the constructor)
  Console.WriteLine(Ford.model);  // Print the value of model
 }
}
```

// Outputs "Mustang"

The constructor name must **match the class name**, and it cannot have a **return type** (like void or int).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, C# creates one for you. However, then you are not able to set initial values for fields.

- **Static Classes :**

In C#, a static class is a class that cannot be instantiated. The main purpose of using static classes in C# is to provide blueprints of its inherited classes. Static classes are created using the static keyword in C# and .NET. A static class can contain static members only. You can't create an object for the static class.

Advantages of Static Classes

1. If you declare any member as a non-static member, you will get an error.
2. When you try to create an instance to the static class, it again generates a compile time error, because the static members can be accessed directly with its class name.
3. The static keyword is used before the class keyword in a class definition to declare a static class.
4. A static class members are accessed by the class name followed by the member name.

Syntax of static class

```
static class classname
{
  //static data members
  //static methods
```



```
}
```

C#

Copy

```
Static Class Demo
namespace StaticConstructorsDemo
{
  class MyCollege
  {
    //static fields
    public static string CollegeName;
    public static string Address;

    //static constructor
    static MyCollege()
    {
      CollegeName = "ABC College of Technology";
      Address = "Hyderabad";
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine(MyCollege.CollegeName);
```

```
        Console.WriteLine(MyCollege.Address);
        Console.Read();
    }
  }
}
```

- **Static local function**

1. A local function declared static cannot capture state from the enclosing scope. As a result, locals, parameters, and this from the enclosing scope are not available within a static local function.
2. A static local function cannot reference instance members from an implicit or explicit this or base reference.A static local function may reference static members from the enclosing scope.A static local function may reference constant definitions from the enclosing scope.
3. nameof() in a static local function may reference locals, parameters, or this or base from the enclosing scope.
4. Accessibility rules for private members in the enclosing scope are the same for static and non-static local functions.
5. A static local function definition is emitted as a static method in metadata, even if only used in a delegate.
6. A non-static local function or lambda can capture state from an enclosing static local function but cannot capture state outside the enclosing static local function.
7. A static local function cannot be invoked in an expression tree.

8. A call to a local function is emitted as call rather than callvirt, regardless of whether the local function is static.
9. Overload resolution of a call within a local function not affected by whether the local function is static.
10. Removing the static modifier from a local function in a valid program does not change the meaning of the program.

- **Inheritance** (Derived and Base Class)

  In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories: **Derived Class (child) - the class that inherits from another class. Base Class (parent) - the class being inherited from**.

To inherit from a class, use the : symbol.

In the example below, the Car class (child) inherits the fields and methods from the Vehicle class (parent):

Example

```
class Vehicle  // base class (parent)
{
 public string brand = "Ford";  // Vehicle field
 public void honk()          // Vehicle method
 {
  Console.WriteLine("Tuut, tuut!");
 }
```

```
}

class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
}


class Program
{
  static void Main(string[] args)
  {
    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (From the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class
    Console.WriteLine(myCar.brand + " " + myCar.modelName);
  }
}
```


Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

- **Access modifiers or specifiers :**

C# Access modifiers or specifiers are the keywords that are used to specify accessibility or scope of variables and functions in the C# application.

C# provides five types of access specifiers.

1. Public
2. Protected
3. Internal
4. Protected internal
5. Private

We can choose any of these to protect our data. Public is not restricted and Private is most restricted. The following table describes about the accessibility of each.

Access Specifier	Description
Public	It specifies that access is not restricted.
Protected	It specifies that access is limited to the containing class or in derived class.
Internal	It specifies that access is limited to the current assembly.
protected	It specifies that access is limited to the current assembly

internal	or types derived from the containing class.
Private	It specifies that access is limited to the containing type.

Now, let's create examples to check accessibility of each access specifier.

C# Public Access Specifier

It makes data accessible publicly. It does not restrict data to the declared block.

Example

```
1. using System;
2. namespace AccessSpecifiers
3. {
4.    class PublicTest
5.    {
6.       public string name = "Joy";
7.       public void Msg(string msg)
8.       {
9.          Console.WriteLine("Hello " + msg);
10.      }
11.   }
12.   class Program
13.   {
14.      static void Main(string[] args)
15.      {
16.         PublicTest publicTest = new PublicTest();
```



```
17.         // Accessing public variable
18.         Console.WriteLine("Hello " + publicTest.name);
19.         // Accessing public function
20.         publicTest.Msg("Emily");
21.      }
22.   }
23. }
```

Output:

Hello Joy

Hello Emily

Constructors in a hierarchy:

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

What is overloading in derived class?

Method Overloading means creating multiple methods in a class with same names but different signatures (Parameters). It permits a class, struct, or interface to declare multiple methods with the same name with unique signatures

Is overloading possible in inheritance C#?

A method that is defined in the parent class can also be overloaded under its child class. It is called Inheritance Based Method Overloading in C#.

In method hiding, you can hide the implementation of the methods of a base class from the derived class using the new keyword. Or in other words, in method hiding, you can redefine the method of the base class in the derived class by using the new keyword.

Hiding, using new :

In method hiding, you can hide the implementation of the methods of a base class from the derived class using the new keyword. Or in other words, in method hiding, you can redefine the method of the base class in the derived class by using the new keyword.

Introduction to the C# new modifier

In inheritance, if a member of a subclass has the same name as the member of a base class, the C# compiler will issue a warning. For example:

```
class Person
{
   public string Name { get; set; }

   public string Introduce() => $"Hi, I'm {Name}.";
}

class Employee : Person
{
   public string JobTitle { get; set; }

   public string Introduce() => $"Hi, I'm {Name}. I'm a {JobTitle}.";
}
```


Code language: C# (cs)

In this example:

The Person is the base class and the Employee is a subclass that inherits from the Person class.

Both Person and Employee classes have the same method Introduce() that returns a string.

The override modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

In the following example, the Square class must provide an overridden implementation of GetArea because GetArea is inherited from the abstract Shape class:

```
abstract class Shape
{
   public abstract int GetArea();
}

class Square : Shape
{
   private int _side;

   public Square(int n) => _side = n;

   // GetArea method is required to avoid a compile-time error.
   public override int GetArea() => _side * _side;

   static void Main()
   {
      var sq = new Square(12);
```

```
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

An override method provides a new implementation of the method inherited from a base class. The method that is overridden by an override declaration is known as the overridden base method. An override method must have the same signature as the overridden base method. override methods support covariant return types. In particular, the return type of an override method can derive from the return type of the corresponding base method.

You cannot override a non-virtual or static method. The overridden base method must be virtual, abstract, or override.

An override declaration cannot change the accessibility of the virtual method. Both the override method and the virtual method must have the same access level modifier.

You cannot use the new, static, or virtual modifiers to modify an override method.

An overriding property declaration must specify exactly the same access modifier, type, and name as the inherited property. Beginning with C# 9.0, read-only overriding properties support covariant return types. The overridden property must be virtual, abstract, or override.

This example defines a base class named Employee, and a derived class named SalesEmployee. The SalesEmployee class includes an extra field, salesbonus, and overrides the method CalculatePay in order to take it into account.

```
class TestOverride
{
    public class Employee
    {
        public string Name { get; }
```



```
        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal _basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            Name = name;
            _basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return _basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal _salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
```

```
    public SalesEmployee(string name, decimal basepay, decimal salesbonus)
        : base(name, basepay)
    {
        _salesbonus = salesbonus;
    }

    // Override the CalculatePay method
    // to take bonus into account.
    public override decimal CalculatePay()
    {
        return _basepay + _salesbonus;
    }
}

static void Main()
{
    // Create some new employees.
    var employee1 = new SalesEmployee("Alice", 1000, 500);
    var employee2 = new Employee("Bob", 1200);

    Console.WriteLine($"Employee1 {employee1.Name} earned:
{employee1.CalculatePay()}");
    Console.WriteLine($"Employee2 {employee2.Name} earned:
{employee2.CalculatePay()}");
  }
}
/*
```



```
  Output:
  Employee1 Alice earned: 1500
  Employee2 Bob earned: 1200
*/
```

Sealed Methods :

When applied to a class, the sealed modifier prevents other classes from inheriting from it. In the following example, class B inherits from class A, but no class can inherit from class B.

```
class A {}
sealed class B : A {}
```

You can also use the sealed modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.

Example

In the following example, Z inherits from Y but Z cannot override the virtual function F that is declared in X and sealed in Y.

```
class X
{
  protected virtual void F() { Console.WriteLine("X.F"); }
  protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
  sealed protected override void F() { Console.WriteLine("Y.F"); }
  protected override void F2() { Console.WriteLine("Y.F2"); }
```

```
}

class Z : Y
{
  // Attempting to override F causes compiler error CS0239.
  // protected override void F() { Console.WriteLine("Z.F"); }

  // Overriding F2 is allowed.
  protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

When you define new methods or properties in a class, you can prevent deriving classes from overriding them by not declaring them as virtual.

It is an error to use the abstract modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

When applied to a method or property, the sealed modifier must always be used with override.

Because structs are implicitly sealed, they cannot be inherited.

Abstract Classes and Methods :

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The abstract keyword is used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal
{
 public abstract void animalSound();
 public void sleep()
 {
  Console.WriteLine("Zzz");
 }
}
```

From the example above, it is not possible to create an object of the Animal class:

Animal myObj = new Animal(); // Will generate an error (Cannot create an instance of the abstract class or interface 'Animal')

What does Sealed Class Mean?
A sealed class, in C#, is a class that cannot be inherited by any class but can be instantiated.

The design intent of a sealed class is to indicate that the class is specialized and there is no need to extend it to provide any additional functionality through inheritance to override its behavior. A sealed class is often used to encapsulate a logic that needs to be used across the program but without any alteration to it.

A sealed class is mostly used for security reasons by preventing unintended derivation by which the derived class may corrupt the implementation provided in the sealed class. As the sealed class cannot form a base class, calls to sealed classes are slightly faster because they enable certain runtime optimizations such as invocation of virtual member functions on instances of sealed class into non-virtual invocations. Sealed class helps in versioning by not breaking compatibility while changing a class from sealed type to unsealed.

Some of the key classes in the .NET framework library are designed as sealed classes, mainly to limit the extensibility of these classes.

Interface

An interface is a specification for a set of class members, not an implementation. An Interface is a reference type and it contains only abstract members such as Events, Methods, and Properties etc. It contain only declaration for its members and implementation defined as separate entities from classes. It can't contain constants, data fields, constructors, destructors and static members and all the member declarations inside interface are implicitly public.

You can think of an interface as an abstract class that contains only pure virtual functions.The implementation of the methods is done in the class that implements the interface.

Interface sample

- **Implementing an Interface**

A class or a Struct can implement one or more interfaces using colon :. On implementing an interface, you must override all the members of an interface.

Syntax:

```
class ClassName : InterfaceName
{

}
```


For example, the following FileInfo class implements the IFile interface, so it should override all the members of IFile.

Example: Interface Implementation

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);
}

class FileInfo : IFile
{
   public void ReadFile()
   {
      Console.WriteLine("Reading File");
   }

   public void WriteFile(string text)
   {
      Console.WriteLine("Writing to file");
   }
}
```

In the above example, the FileInfo class implements the IFile interface. It overrides all the members of the IFile interface with public access modifier. The FileInfo class can also contain members other than interface members.

Note:

Interface members must be implemented with the public modifier; otherwise, the compiler will give compile-time errors.

You can create an object of the class and assign it to a variable of an interface type, as shown below.

Example: Interface Implementation

```
public class Program
{
   public static void Main()
   {
      IFile file1 = new FileInfo();
      FileInfo file2 = new FileInfo();

      file1.ReadFile();
      file1.WriteFile("content");

      file2.ReadFile();
      file2.WriteFile("content");
   }
}
```

Try it

Above, we created objects of the FileInfo class and assign it to IFile type variable and FileInfo type variable. When interface implemented implicitly, you can access IFile members with the IFile type variables as well as FileInfo type variable.

- **Explicit Implementation**

An interface can be implemented explicitly using <InterfaceName>.<MemberName>. Explicit implementation is useful when class is implementing multiple interfaces; thereby, it is more readable and eliminates the confusion. It is also useful if interfaces have the same method name coincidently.

Note:

Do not use *public* modifier with an explicit implementation. It will give a compile-time error.

Example: Explicit Implementation

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);
}

class FileInfo : IFile
{
   void IFile.ReadFile()
   {
      Console.WriteLine("Reading File");
   }

   void IFile.WriteFile(string text)
   {
      Console.WriteLine("Writing to file");
   }
}
```

When you implement an interface explicitly, you can access interface members only through the instance of an interface type.

Example: Explicit Implementation

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);
}
```

```
class FileInfo : IFile
{
   void IFile.ReadFile()
   {
      Console.WriteLine("Reading File");
   }

   void IFile.WriteFile(string text)
   {
      Console.WriteLine("Writing to file");
   }

   public void Search(string text)
   {
      Console.WriteLine("Searching in file");
   }
}

public class Program
{
   public static void Main()
   {
      IFile file1 = new FileInfo();
      FileInfo file2 = new FileInfo();
```



```
      file1.ReadFile();
      file1.WriteFile("content");
      //file1.Search("text to be searched")//compile-time error

      file2.Search("text to be searched");
      //file2.ReadFile(); //compile-time error
      //file2.WriteFile("content"); //compile-time error
   }
}
```

Try it

In the above example, file1 object can only access members of IFile, and file2 can only access members of FileInfo class. This is the limitation of explicit implementation.

Implementing Multiple Interfaces

A class or struct can implement multiple interfaces. It must provide the implementation of all the members of all interfaces.

Example: Implement Multiple Interfaces

```
interface IFile
{
   void ReadFile();
}

interface IBinaryFile
{
   void OpenBinaryFile();
   void ReadFile();
}
```

```
class FileInfo : IFile, IBinaryFile
{
   void IFile.ReadFile()
   {
      Console.WriteLine("Reading Text File");
   }

   void IBinaryFile.OpenBinaryFile()
   {
      Console.WriteLine("Opening Binary File");
   }

   void IBinaryFile.ReadFile()
   {
      Console.WriteLine("Reading Binary File");
   }

   public void Search(string text)
   {
      Console.WriteLine("Searching in File");
   }
}

public class Program
{
   public static void Main()
```



```
   {
      IFile file1 = new FileInfo();
      IBinaryFile file2 = new FileInfo();
      FileInfo file3 = new FileInfo();

      file1.ReadFile();
      //file1.OpenBinaryFile(); //compile-time error
      //file1.SearchFile("text to be searched"); //compile-time error

      file2.OpenBinaryFile();
      file2.ReadFile();
      //file2.SearchFile("text to be searched"); //compile-time error

      file3.Search("text to be searched");
      //file3.ReadFile(); //compile-time error
      //file3.OpenBinaryFile(); //compile-time error
   }
}
```

Try it

Above, the FileInfo implements two interfaces IFile and IBinaryFile explicitly. It is recommended to implement interfaces explicitly when implementing multiple interfaces to avoid confusion and more readability.

- Default Interface Methods :

Till now, we learned that interface can contain method declarations only. C# 8.0 added support for virtual extension methods in interface with concrete implementations.

The virtual interface methods are also called default interface methods that do not need to be implemented in a class or struct.

Example: Default Interface Methods

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);

   void DisplayName()
   {
      Console.WriteLine("IFile");
   }
}
```

In the above IFile interface, the DisplayName() is the default method. The implementation will remain same for all the classes that implements the IFile interface. Note that a class does not inherit default methods from its interfaces; so, you cannot access it using the class instance.

Example: Interface Implementation

```
class FileInfo : IFile
{
   public void ReadFile()
   {
      Console.WriteLine("Reading File");
   }

   public void WriteFile(string text)
   {
      Console.WriteLine("Writing to file");
```



```
   }
}

public class Program
{
   public static void Main()
   {
      IFile file1 = new FileInfo();
      file1.ReadFile();
      file1.WriteFile("content");
      file1.DisplayName();

      FileInfo file2 = new FileInfo();
      //file2.DisplayName(); //compile-time error
   }
}
```

Operator Overloading Techniques in C#

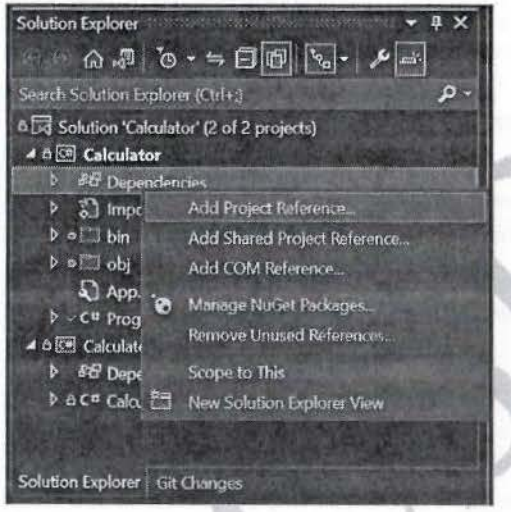
Operator overloading in C# can be done using different forms of operators. But before proceeding to the techniques, let's consider the validations of operators and how are they used while performing operator overloading.

Operators	Overloadability
+, -, *, /, %, &, |, <<, >>	All C# binary operators can be overloaded.
+, -, !, ~, ++, --, true, false	All C# unary operators can be overloaded.
==, !=, <, >, <=, >=	All relational operators can be overloaded, but only as pairs.
&&, ||	They can't be overloaded.
[] (Array index operator)	They can't be overloaded.
() (Conversion operator)	They can't be overloaded.
+=, -=, *=, /=, %=	These compound assignment operators can be overloaded. But in C#, these operators are automatically overloaded when the respective binary operator is overloaded.
=, ., ?:, ->, new, is, as, sizeof	These operators can't be overloaded in C#.

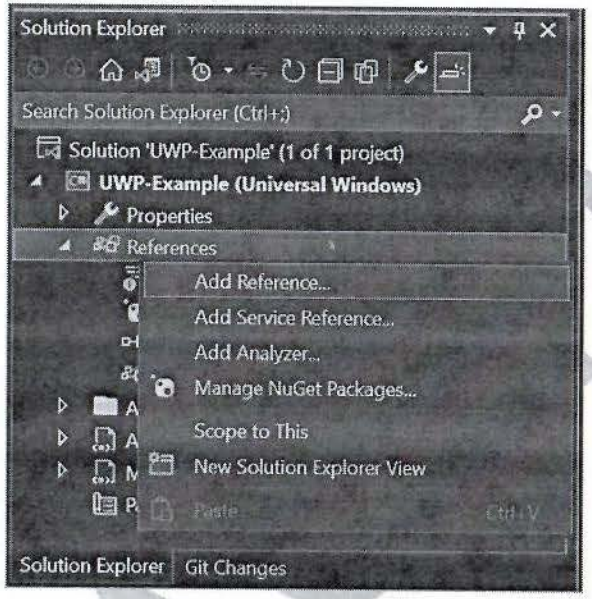
References and Value Types

How you add a reference depends on the project type for the code you're working on:

- If you see a Dependencies node in Solution Explorer, you can use the right-click context menu to choose Add Project Reference. You can also right-click the project node and select Add > Project Reference.



- If you see a References node in Solution Explorer, you can use the right-click context menu to choose Add Reference. Or, right-click the project node and select Add > Reference.



For more information, see How to: Add or remove references.

You can add a reference to the following types of components and services:

- .NET class libraries or assemblies
- UWP apps
- COM components
- Other assemblies or class libraries of projects in the same solution
- Shared projects
- XML Web service

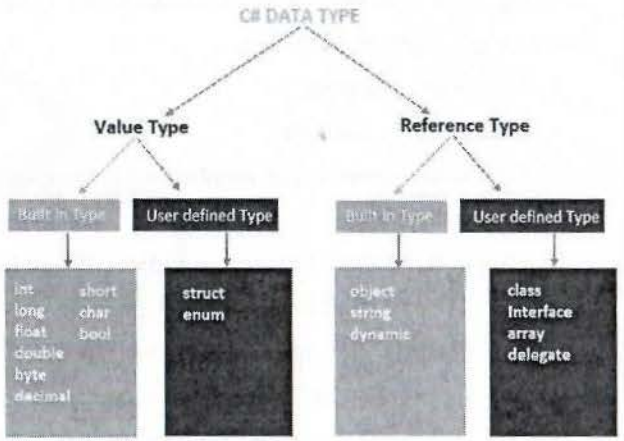
▪ Value Types

A data type is a *value type* if it holds the data within its own memory allocation.

Value types include the following:

- All numeric data types
- Boolean, Char, and Date
- All structures, even if their members are reference types
- Enumerations,since their underlying type is always SByte, Short, Integer, Long, Byte, UShort, UInteger, or ULong
  - Every structure is a value type, even if it contains reference type members. For this reason, value types such as Char and Integer are implemented by .NET Framework structures.
  - You can declare a value type by using the reserved keyword, for example, Decimal. You can also use the New keyword to initialize a value type. This is especially useful if the type has a constructor that takes parameters. An example of this is the Decimal(Int32, Int32, Int32, Boolean, Byte) constructor, whichbuilds a new Decimal value from the supplied parts.

Diagram: Value Type



Kinds of value types and type constraints

A value type can be one of the two following kinds:

- a structure type, which encapsulates data and related functionality
- an enumeration type, which is defined by a set of named constants and represents a choice or a combination of choices

A nullable value type `T?` represents all values of its underlying value type `T` and an additional null value. You cannot assign `null` to a variable of a value type, unless it's a nullable value type.

You can use the `struct` constraint to specify that a type parameter is a non-nullable value type. Both structure and enumeration types satisfy the `struct` constraint. You can

use `System.Enum` in a base class constraint (that is known as the enum constraint) to specify that a type parameter is an enumeration type.

Built-in value types

C# provides the following built-in value types, also known as *simple types*:

- Integral numeric types
- Floating-point numeric types
- bool that represents a Boolean value
- char that represents a Unicode UTF-16 character

All simple types are structure types and differ from other structure types in that they permit certain additional operations:

- You can use literals to provide a value of a simple type. For example, 'A' is a literal of the type char and 2001 is a literal of the type int.
- You can declare constants of the simple types with the const keyword. It's not possible to have constants of other structure types.
- Constant expressions, whose operands are all constants of the simple types, are evaluated at compile time.

A value tuple is a value type, but not a simple type.

C# ref vs out

Ref and out keywords in C# are used to pass arguments within a method or function. Both indicate that an argument/parameter is passed by reference. By default parameters are passed to a method by value. By using these keywords (ref and out) we can pass a parameter by reference.

Ref Keyword

The ref keyword passes arguments by reference. It means any changes made to this argument in the method will be reflected in that variable when control returns to the calling method.

Example code

```
public static string GetNextName(ref int id)
{
   string returnText = "Next-" + id.ToString();
   id += 1;
   return returnText;
}
static void Main(string[] args)
{
   int i = 1;
   Console.WriteLine("Previous value of integer i:" + i.ToString());
   string test = GetNextName(ref i);
   Console.WriteLine("Current value of integer i:" + i.ToString());
}
```

- **Nullable value types**

A nullable *value type* T? represents all values of its underlying value type T and an additional null value. For example, you can assign any of the following three values to a bool? variable: true, false, or null. An underlying value type T cannot be a nullable value type itself.

Any nullable value type is an instance of the generic System.Nullable<T> structure. You can refer to a nullable value type with an underlying type T in any of the following interchangeable forms: Nullable<T> or T?.

You typically use a nullable value type when you need to represent the undefined value of an underlying value type. For example, a Boolean, or bool, variable can only be either true or false. However, in some applications a variable value can be undefined or missing. For example, a database field may contain true or false, or it may contain no value at all, that is, NULL. You can use the bool? type in that scenario.

Declaration and assignment

As a value type is implicitly convertible to the corresponding nullable value type, you can assign a value to a variable of a nullable value type as you would do that for its underlying value type. You can also assign the null value. For example:

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

The default value of a nullable value type represents null, that is, it's an instance whose Nullable<T>.HasValue property returns false.

Examination of an instance of a nullable value type

You can use the is operator with a type pattern to both examine an instance of a nullable value type for null and retrieve a value of an underlying type:

```
Run
int? a = 42;
if (a is int valueOfA)
{
   Console.WriteLine($"a is {valueOfA}");
}
else
{
   Console.WriteLine("a does not have a value");
}
// Output:
// a is 42
```

You always can use the following read-only properties to examine and get a value of a nullable value type variable:

- Nullable<T>.HasValue indicates whether an instance of a nullable value type has a value of its underlying type.
- Nullable<T>.Value gets the value of an underlying type if HasValue is true. If HasValue is false, the Value property throws an InvalidOperationException.

The following example uses the HasValue property to test whether the variable contains a value before displaying it:


```
Run
int? b = 10;
if (b.HasValue)
{
   Console.WriteLine($"b is {b.Value}");
}
else
{
   Console.WriteLine("b does not have a value");
}
// Output:
// b is 10
```

- **Nullable reference types :**

Nullable reference types are available in code that has opted in to a *nullable aware context*. Nullable reference types, the null static analysis warnings, and the null-forgiving operator are optional language features. All are turned off by default. A *nullable context* is controlled at the project level using build settings, or in code using pragmas.

Important

All project templates starting with .NET 6 (C# 10) enable the *nullable context* for the project. Projects created with earlier templates don't include this element, and these features are off unless you enable them in the project file or use pragmas.

In a nullable aware context:

- A variable of a reference type T must be initialized with non-null, and may never be assigned a value that may be null.
- A variable of a reference type T? may be initialized with null or assigned null, but is required to be checked against null before de-referencing.
- A variable m of type T? is considered to be non-null when you apply the null-forgiving operator, as in m!.

The distinctions between a non-nullable reference type T and a nullable reference type T? are enforced by the compiler's interpretation of the preceding rules. A variable of type T and a variable of type T? are represented by the same .NET type. The following example declares a non-nullable string and a nullable string, and then uses the null-forgiving operator to assign a value to a non-nullable string:

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

The variables notNull and nullable are both represented by the String type. Because the non-nullable and nullable types are both stored as the same type, there are several locations where using a nullable reference type isn't allowed. In general, a nullable reference type can't be used as a base class or implemented interface. A nullable reference type can't be used in any object creation or type testing expression. A nullable reference type can't be the type of a member access expression. The following examples show these constructs:

```
public MyClass : System.Object? // not allowed
{
}
```



```
var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
  if (thing is string? nullableString) // not allowed
    Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
  Console.WriteLine("error");
}
```

The goal of this feature is to:

- Allow developers to express whether a variable, parameter or result of a reference type is intended to be null or not.
- Provide warnings when such variables, parameters and results are not used according to that intent.

Expression of intent

- The language already contains the T? syntax for value types. It is straightforward to extend this syntax to reference types.
- It is assumed that the intent of an unadorned reference type T is for it to be non-null.

Type inference

- In type inference, if a contributing type is a nullable reference type, the resulting type should be nullable. In other words, nullness is propagated.

- We should consider whether the null literal as a participating expression should contribute nullness. It doesn't today: for value types it leads to an error, whereas for reference types the null successfully converts to the plain type.

```
string? n = "world";
var x = b ? "Hello" : n; // string?
var y = b ? "Hello" : null; // string? or error
var z = b ? 7 : null; // Error today, could be int?
```

- **Working with Arrays (single, multidim, jagged)**

You create a single-dimensional array using the new operator specifying the array element type and the number of elements. The following example declares an array of five integers:

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

This array contains the elements from array[0] to array[4]. The elements of the array are initialized to the default value of the element type, 0 for integers.

Arrays can store any element type you specify, such as the following example that declares an array of strings:

```
string[] stringArray = new string[6];
```

Array Initialization

You can initialize the elements of an array when you declare the array. The length specifier isn't needed because it's inferred by the number of elements in the initialization list. For example:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

The following code shows a declaration of a string array where each array element is initialized by a name of a day:

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

You can avoid the new expression and the array type when you initialize an array upon declaration, as shown in the following code. This is called an implicitly typed array:

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

- **Multidimensional Arrays**

Arrays can have more than one dimension. For example, the following declaration creates a two-dimensional array of four rows and two columns.

```
int[,] array = new int[4, 2];
```

The following declaration creates an array of three dimensions, 4, 2, and 3.

```
int[,,] array1 = new int[4, 2, 3];
```

Array Initialization

You can initialize the array upon declaration, as is shown in the following example.

```
// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                        { "five", "six" } };

// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[,,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                      { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
```



```
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
   total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

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

```
// 7
// three
// 8
// 12
// 12 equals 12
```

You can also initialize the array without specifying the rank.

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

If you choose to declare an array variable without initialization, you must use the new operator to assign an array to the variable. The use of new is shown in the following example.

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };  // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}};  // Error
```

The following example assigns a value to a particular array element.

```
array5[2, 1] = 25;
```

Similarly, the following example gets the value of a particular array element and assigns it to variable elementValue.

```
int elementValue = array5[2, 1];
```


The following code example initializes the array elements to default values (except for jagged arrays).

```
int[,] array6 = new int[10, 10];
```

Jagged Arrays

A jagged array is an array whose elements are arrays, possibly of different sizes. A jagged array is sometimes called an "array of arrays." The following examples show how to declare, initialize, and access jagged arrays.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[][] jaggedArray = new int[3][];
```

Before you can use jaggedArray, its elements must be initialized. You can initialize the elements like this:

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

Each of the elements is a single-dimensional array of integers. The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.

A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to null.

You can access individual array elements like these examples:

```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

It's possible to mix jagged and multidimensional arrays. The following is a declaration and initialization of a single-dimensional jagged array that contains three two-dimensional array elements of different sizes.

```
int[][,] jaggedArray4 = new int[3][,]
{
   new int[,] { {1,3}, {5,7} },
   new int[,] { {0,2}, {4,6}, {8,10} },
   new int[,] { {11,22}, {99,88}, {0,9} }
};
```

You can access individual elements as shown in this example, which displays the value of the element [1,0] of the first array (value 5):


```
System.Console.Write("{0}", jaggedArray4[0][1, 0]);
```

The method Length returns the number of arrays contained in the jagged array. For example, assuming you have declared the previous array, this line:

```
System.Console.WriteLine(jaggedArray4.Length);
```

returns a value of 3.

	Property & description
1	**IsFixedSize** Gets a value indicating whether the Array has a fixed size.
2	**IsReadOnly** Gets a value indicating whether the Array is read-only.
3	**Length** Gets a 32-bit integer that represents the total number of elements in all the dimensions of the Array.
4	**LongLength** Gets a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
5	**Rank** Gets the rank (number of dimensions) of the Array.

Array Class Members:

The Array class is the base class for all the arrays in C#. It is defined in the System namespace. The Array class provides various properties and methods to work with arrays.

Properties of the Array Class

The following table describes some of the most commonly used properties of the Array class –

Sr.No.	Property & description
1	**IsFixedSize** Gets a value indicating whether the Array has a fixed size.
2	**IsReadOnly** Gets a value indicating whether the Array is read-only.
3	**Length** Gets a 32-bit integer that represents the total number of elements in all the dimensions of the Array.
4	**LongLength** Gets a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
5	**Rank** Gets the rank (number of dimensions) of the Array.

Sr.No.	Methods & Description
1	**Clear** Sets a range of elements in the Array to zero, to false, or to null, depending on the element type.
2	**Copy(Array, Array, Int32)** Copies a range of elements from an Array starting at the first element and pastes them into another Array starting at the first element. The length is specified as a 32-bit integer.
3	**CopyTo(Array, Int32)** Copies all the elements of the current one-dimensional Array to the specified one-dimensional Array starting at the specified destination Array index. The index is specified as a 32-bit integer.
4	**GetLength** Gets a 32-bit integer that represents the number of elements in the specified dimension of the Array.
5	**GetLongLength** Gets a 64-bit integer that represents the number of elements in the specified dimension of the Array.
6	**GetLowerBound** Gets the lower bound of the specified dimension in the Array.
7	**GetType** Gets the Type of the current instance. (Inherited from Object.)
8	**GetUpperBound** Gets the upper bound of the specified dimension in the Array.

9 **GetValue(Int32)**

Gets the value at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.

10 **IndexOf(Array, Object)**

Searches for the specified object and returns the index of the first occurrence within the entire one-dimensional Array.

11 **Reverse(Array)**

Reverses the sequence of the elements in the entire one-dimensional Array.

12 **SetValue(Object, Int32)**

Sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.

13 **Sort(Array)**

Sorts the elements in an entire one-dimensional Array using the IComparable implementation of each element of the Array.

14 **ToString**

Returns a string that represents the current object. (Inherited from Object.)

What Are Indices?

In C# indices represent an index in a sequence. Starting from C# 8.0, the ^ operator can be used to specify an index relative to the end of a sequence. The constructor Index is composed of two parameters.

Let's look at the constructor definition:

```
public Index (int value, bool fromEnd = false);
```


The value parameter specifies the index value. Its value must be greater than or equal to zero. The second parameter fromEnd is optional. It indicates if the value of the index is from the start or end.

To better understand indices in C#, let's create the IndexExamples class and see the different ways we can use the index:

```
public class IndexExamples
{
public static string GetFirst(string[] names)
{
var index = new Index(0);
return names[index];
}
public static string GetLastMethod1(string[] names)
{
var index = new Index(1, true);
return names[index];
}
}
```

GetFirst returns the first element of an array. GetLastMethod1 returns the last element of an array. Although the code works fine, we can further simplify the syntax.

Let's add new methods to the IndexExamples class:

```
public static string GetLastMethod2(string[] names)
{
return names[^1];
}
```

```
public static string GetSecondLast(string[] names)
{
return names[^2];
}
```

In both GetLastMethod2 and GetSecondLast we use the hat operator to indicate the index is from the end. GetLastMethod2 and GetLastMethod1 result in the same output, however the syntax in GetLastMethod2 is cleaner.

When we use the index from the end, the index value must be greater than zero. Otherwise, it will throw the System.IndexOutOfRangeException exception. For the index value, the index ^i is equivalent to names.Length - i.

For example:

- names[^1] is equivalent to names[names.Length - 1]
- names[^2] is equivalent to names[names.Length - 2]

What Are Ranges?

Ranges in C# are a very concise way of representing a subset of a sequence using the .. operator. The range operator specifies the start and the end of a slice. Let's take x..y as an example to understand the range operator:

- x specifies the start index
- y specifies the end index
- Both x and y are optional values

- The range operator (..) is inclusive of the starting index (x) and exclusive of the end index (y)

Let's inspect the constructor of the Range class:

```
public Range(Index start, Index end);
```

As the name suggests the first parameter defines the start index and the second parameter specifies the end index.

Let's create the RangeExamples class and see the different ways we can use the range operator(..):

```
public class RangeExamples
{
public static string[] GetAll(string[] arr)
{
return arr[..];
}
public static string[] GetFirstTwoElements(string[] arr)
{
var start = new Index(0);
var end = new Index(2);
var range = new Range(start, end);
return arr[range];
}
public static string[] GetFirstThreeElements(string[] arr)
{
return arr[..3];
}
```

```
public static string[] GetLastThreeElements(string[] arr)
{
return arr[^3..];
}
public static string[] GetThreeElementsFromMiddle(string[] arr)
{
return arr[3..6];
}
}
```

The RangeExamples is a class that contains different methods that take advantage of the ..operator. The index feature added in C# 8.0 complements the range feature. It provides an easier way to specify the start and end of the range. For example, in the GetLastThreeElements() method, the hat operator is used to define the range start position.

Limitations of Ranges and Indices :

Ranges and indices provide a succinct syntax for accessing a single element or ranges in a sequence. However, both range and index have required specifications.

To use the index language feature on an instance:

- The class must have publicly accessible Length or Count property
- Also, it must have an accessible indexer which takes a single argument of type int
- If the class contains an indexer with more than one parameter the first parameter should not be Index. If the first parameter is Index, the remaining parameters must be optional

To use the range language feature on an instance:

- A class must have publicly accessible Length or Count property

- Also, it must have an accessible method named Slice which has two parameters of type int
- If a class contains an indexer with more than one parameter the first parameter should not be Range. If it is, the remaining parameters must be optional

Indexers :

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble properties except that their accessors take parameters.

The following example defines a generic class with simple get and set accessor methods to assign and retrieve values. The Program class creates an instance of this class for storing strings.

C#

Copy

```
using System;

class SampleCollection<T>
{
  // Declare an array to store the data elements.
  private T[] arr = new T[100];
```

```
    // Define the indexer to allow client code to use [] notation.
    public T this[int i]
    {
      get { return arr[i]; }
      set { arr[i] = value; }
    }
}

class Program
{
  static void Main()
  {
    var stringCollection = new SampleCollection<string>();
    stringCollection[0] = "Hello, World";
    Console.WriteLine(stringCollection[0]);
  }
}
// The example displays the following output:
//      Hello, World.
```


- **Generic Classes**

Generic classes encapsulate operations that are not specific to a particular data type. The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on. Operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored.

For most scenarios that require collection classes, the recommended approach is to use the ones provided in the .NET class library. For more information about using these classes, see Generic Collections in .NET.

Typically, you create generic classes by starting with an existing concrete class, and changing types into type parameters one at a time until you reach the optimal balance of generalization and usability. When creating your own generic classes, important considerations include the following:

- Which types to generalize into type parameters.

  As a rule, the more types you can parameterize, the more flexible and reusable your code becomes. However, too much generalization can create code that is difficult for other developers to read or understand.

- What constraints, if any, to apply to the type parameters

  A good rule is to apply the maximum constraints possible that will still let you handle the types you must handle. For example, if you know that your generic class is intended for use only with reference types, apply the class constraint. That will prevent unintended use of your class with value types, and will enable you to use the as operator on T, and check for null values.

- Whether to factor generic behavior into base classes and subclasses.

Because generic classes can serve as base classes, the same design considerations apply here as with non-generic classes. See the rules about inheriting from generic base classes later in this topic.

- Whether to implement one or more generic interfaces.

  For example, if you are designing a class that will be used to create items in a generics-based collection, you may have to implement an interface such as IComparable<T> where T is the type of your class.

  For an example of a simple generic class, see Introduction to Generics.

  The rules for type parameters and constraints have several implications for generic class behavior, especially regarding inheritance and member accessibility. Before proceeding, you should understand some terms. For a generic class Node<T>, client code can reference the class either by specifying a type argument - to create a closed constructed type (Node<int>); or by leaving the type parameter unspecified - for example when you specify a generic base class, to create an open constructed type (Node<T>). Generic classes can inherit from concrete, closed constructed, or open constructed base classes:

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }
```



```
//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Non-generic, in other words, concrete, classes can inherit from closed constructed base classes, but not from open constructed classes or from type parameters because there is no way at run time for client code to supply the type argument required to instantiate the base class.

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Generic classes that inherit from open constructed types must supply type arguments for any base class type parameters that are not shared by the inheriting class, as demonstrated in the following code:

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }
```

```
//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

Generic classes that inherit from open constructed types must specify constraints that are a superset of, or imply, the constraints on the base type:

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>,
new() { }
```

Generic types can use multiple type parameters and constraints, as follows:

```
class SuperKeyType<K, V, U>
   where U : System.IComparable<U>
   where V : new()
{ }
```

Open constructed and closed constructed types can be used as method parameters:

```
void Swap<T>(List<T> list1, List<T> list2)
{
   //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
   //code to swap items
}
```


If a generic class implements an interface, all instances of that class can be cast to that interface.

Generic classes are invariant. In other words, if an input parameter specifies a List<BaseClass>, you will get a compile-time error if you try to provide a List<DerivedClass>.

We can create a method which defer the parameter data type until the method is called. These parameters are called Type parameters that means we can pass the actual data type later.

Below is the example of Generic Method.

```
static void Swap<T>(ref T input1, ref T input2)
{
   T temp = default(T);

   temp = input2;
   input2 = input1;
   input1 = temp;
}

static void Main(string[] args)
{
   int first = 4;
   int second = 5;

   Swap<int>(ref first, ref second);
```

```
}
```

- **Generic Methods and Generic Classes**

Generic method type parameter hides the type parameter of Generic classes. CLR will not issue any warnings or errors, when we use the same type parameter name with both generic method and generic class.
Below is the example:

```
public class Helper<T>
{
   public void Method<T>(T input)
   {
      Console.WriteLine(input);   //Result Hello
   }
}

class Program
{
   static void Main(string[] args)
   {
      Helper<int> helper = new Helper<int>();
      helper.Method<string>("Hello");

      //Result:
      // Hello
   }
```



```
}
```

In the above generic method example, we have taken the same type parameter name which is used by generic class. In the main method, we have passed the <int> type argument in the Helper class. In the method, we have passed the <string> argument.

- **Generic Overloaded Methods**

C# allows you to define generic overloaded methods with many type parameters.
Below is the example:

```
static void Swap<T>(T input) { }

static void Swap<T, U>(T input, U input2) { }

static void Swap<T, U, W>(T input, U input2, W input3) { }
```

Generic Methods Constraints

Constraints are validations on type arguments which type of parameter we can pass in generic method. Generic class and Generic methods follow the same constraints. There are six types of constraints.

1. **where T : struct** – Type argument must be a value type
2. **where T : class** – Type argument must be a reference type
3. **where T : new()** – Type argument must have a public parameterless constructor.
4. **where T : <base class>** – Type argument must inherit from <base class> class.
5. **where T : <interface>** – Type argument must implement from <interface> interface.
6. **where T : U** – There are two type arguments T and U. T must be inherit from U.

Below is the example of usage of above constraints in Generic methods.

```
static void Swap<T>(ref T input1, ref T input2) where T : struct { }

static void Swap<T>(ref T input1, ref T input2) where T : class { }

static void Swap<T>(ref T input1, ref T input2) where T : new() { }

static void Swap<T>(ref T input1, ref T input2) where T : BaseEmployee { }

static void Swap<T>(ref T input1, ref T input2) where T : IEmployee { }

static void Swap<T, U>(ref T input1, ref U input2) where T : U { }
```

Generic Constraints

C# allows you to use constraints to restrict client code to specify certain types while instantiating generic types. It will give a compile-time error if you try to instantiate a generic type using a type that is not allowed by the specified constraints.

You can specify one or more constraints on the generic type using the where clause after the generic type name.

Syntax:

GenericTypeName<T> where T : contraint1, constraint2

The following example demonstrates a generic class with a constraint to reference types when instantiating the generic class.

Example: Declare Generic Constraints

```
class DataStore<T> where T : class
{
```



```
    public T Data { get; set; }
}
```

Above, we applied the class constraint, which means only a reference type can be passed as an argument while creating the DataStore class object. So, you can pass reference types such as class, interface, delegate, or array type. Passing value types will give a compile-time error, so we cannot pass primitive data types or struct types.

```
DataStore<string> store = new DataStore<string>(); // valid
DataStore<MyClass> store = new DataStore<MyClass>(); // valid
DataStore<IMyInterface> store = new DataStore<IMyInterface>(); // valid
DataStore<IEnumerable> store = new DataStore<IMyInterface>(); // valid
DataStore<ArrayList> store = new DataStore<ArrayList>(); // valid
//DataStore<int> store = new DataStore<int>(); // compile-time error
```

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.

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. These classes create collections of objects of the Object class, which is the base class for all data types in C#.

Various Collection Classes and Their Usage

The following are the various commonly used classes of the **System.Collection** namespace. Click the following links to check their detail.

Sr.No.	Class & Description and Useage
1	ArrayList It represents ordered collection of an object that can be **indexed** individually. It is basically an alternative to an array. However, unlike array you can add and remove items from a list at a specified position using an **index** and the array resizes itself automatically. It also allows dynamic memory allocation, adding, searching and sorting items in the list.
2	Hashtable It uses a **key** to access the elements in the collection. A hash table is used when you need to access elements by using key, and you can identify a useful key value. Each item in the hash table has a **key/value** pair. The key is used to access the items in the collection.
3	SortedList It uses a **key** as well as an **index** to access the items in a list. A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key , it is a Hashtable. The collection of items is always sorted by the key value.

4	Stack It represents a **last-in, first out** collection of object. It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called **pushing** the item and when you remove it, it is called **popping** the item.
5	Queue It represents a **first-in, first out** collection of object. It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called **enqueue** and when you remove an item, it is called **deque**.
6	BitArray It represents an array of the **binary representation** using the values 1 and 0. It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an **integer index**, which starts from zero.

Generic collections hold elements of same datatypes.

For example –

- List
- Dictionary
- Hashset

Dictionary – Dictionary is a collection of keys and values in C#. Dictionary <TKey, TValue> is included in the System.Collection.Generics namespace.

Hashset – HashSet in C# eliminates duplicate strings or elements in an array.In C#, it is an optimized set collection.

Non-Generics in C#

Non-generic collections hold elements of different datatypes.

The following are the non-generic collections: ArrayList, BitArray.

ArrayList – It represents ordered collection of an object that can be indexed individually. ArrayList is an alternative to an array. However, unlike array you can add and remove items from a list at a specified position using an index and the array resizes itself automatically.

BitArray – It represents an array of the binary representation using the values 1 and 0. It is used when you need to store the bits but do not know the number of bits in advance.

What is Generic Collection in C#?

a)The Non-generic collections such as ArrayList, Queue, Stack, etc. can store elements of different data types. When obtaining the items, the programmer should type cast them to the correct data type. Else, it can cause a runtime exception.

b)The generic collection classes can be used to overcome this issue. Generic collections store elements internally in arrays of their actual types. Therefore, type casting is not required.

c) They can be used to store elements of the specified type or types. Some Generic collection classes are List<T>, Dictionary <TKey, TValue>, SortedList <TKey, TValue>, HashSet<T>, Queue<T>, Stack<T>.

d)Arrays can be used to store multiple elements. One drawback is that it can store elements of the same data type. There are classes in C# that can be used to store many values or objects known as collections. Collections help to store, update, delete, search, sort objects. The size of the collection can be increased or decreased dynamically.

What is the Difference Between Generic and Non-Generic Collection in C#?

Generic	Non-Generic Collection in C#
A Generic collection is a class that provides type safety without having to derive from a base collection type and implement type-specific members.	A Non-generic collection is a specialized class for data storage and retrieval that provides support for stacks, queues, lists and hash tables.
Namespace	
The Generic Collection classes are in the System. Collections. Generics namespace.	The Non -generic Collection classes are in the System. Collections namespace.
Type	
A Generic Collection is strongly typed.	A Non-Generic Collection is not strongly typed.
Storing Elements	
The Generic Collections store elements internally in arrays of their actual types.	The Non-generic collections store elements internally in object arrays so it can store any type of data.

- **Collection Examples based on ICollection, IList, IDictionary (both generic and non-generic)**

(A)**ICollection interface** is the base interface of all collection types in the .NET library. And the uses of the ICollection interface is every now and then in application development. You might think, no I have never used the ICollection interface in my day-to-day coding life. Let me ask a small question. Have you used List or any other type of collection in .NET? Haha, your answer is yes. Then you have used the ICollection interface without your knowledge.

There are the following two versions of the ICollection interface:

- ICollection: without parameter
- ICollection<T>: with parameter of type T

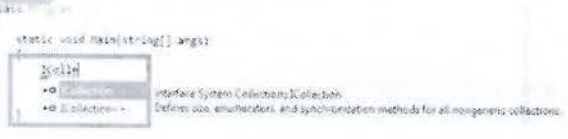
The ICollection interface is located in the following place of the .NET class library:

- Namespace: System.Collections
- Assembly: mscorlib.dll

And the ICollection<T> is located in the following location:

- Namespace: System.collection.Generics
- Assembly: mscorlib.dll

Here is a different version of the ICollection interface.



Fine, so far we have understood the various versions of the ICollection interface and their location in the .NET class library.

Definition of ICollection and the ICollection<T> interface

There are three properties (all are read only in nature) and one method in the ICollection interface.

```
public interface ICollection : IEnumerable
{
  int Count { get; }
  bool IsSynchronized { get; }
  Object SyncRoot { get; }
  void CopyTo(Array array, int index);
}
```

C#

Copy

ICollection inherits from IEnumerable, so that we can iterate an object collection of the ICollection interface (for example List and List<T>) using a foreach loop.

(B)IList is a descendant of the ICollection interface and is the base interface of all non-generic lists. IList implementations fall into three categories: read-only, fixed-size, and variable-size. A read-only IList cannot be modified. A fixed-size IList does not allow the addition or removal of elements, but it allows the modification of existing elements. A variable-size IList allows the addition, removal, and modification of elements.

(C) Dictionary Collection

The dictionary collection represents Key-Value pairs. It is similar to an English dictionary with different words and their meanings. The dictionary collection is included in the System.Collections.Generic namespace.

Properties in Dictionary Collection

The different properties and their description is given as follows:

Table: Properties in Dictionary Collection in C#

Source: MSDN

Sr.No	Properties	Description
1	Comparer	This property gets the IEqualityComparer<T> that is used to determine equality of keys for the dictionary.
2	Count	This property gets the number of key/value pairs contained in the Dictionary<TKey,TValue>.
3	Item[TKey]	This property gets or sets the value associated with the specified key
4	Item[TKey]	This property gets a collection containing the keys in the Dictionary<TKey,TValue>.
5	Values	This property gets a collection containing the values in the Dictionary<TKey,TValue>.

Methods in Dictionary Collection

The different methods and their description is given as follows:

Table: Methods in Dictionary Collection in C#

Source: MSDN

Methods	Description
Add(TKey, TValue)	This method adds the specified key and value to the dictionary.
Clear()	This method removes all keys and values from the Dictionary<TKey,TValue>.
ContainsKey(TKey)	This method determines whether the Dictionary<TKey,TValue> contains the specified key.

Methods	Description
ContainsValue(TValue)	This method determines whether the Dictionary<TKey,TValue> contains a specific value.
Equals(Object)	This method determines whether the specified object is equal to the current object.
GetEnumerator()	This method returns an enumerator that iterates through the Dictionary<TKey,TValue>
GetHashCode()	This method serves as the default hash function.
GetObjectData(SerializationInfo, StreamingContext)	This method implements the ISerializable interface and returns the data needed to serialize the Dictionary<TKey,TValue>

Methods	Description
	instance.
GetType()	This method gets the Type of the current instance.
MemberwiseClone()	This method creates a shallow copy of the current Object.
OnDeserialization(Object)	This method implements the ISerializable interface and raises the deserialization event when the deserialization is complete.
Remove(TKey)	This method removes the value with the specified key from the Dictionary<TKey,TValue>.

Methods	Description
ToString()	This method returns a string that represents the current object.
TryGetValue(TKey, TValue	This method gets the value associated with the specified key.

Iterating collections using foreach :

- The foreach statement executes a statement or a block of statements for each element in a collection which implements IEnumerable.

```
public void ForEach (Action<T> action);
```

- The ForEach method performs the specified action on each element of the List<T>.
- we show how to loop over data in C# with foreach statement and forEach method.

```
foreach (var val in vals)
{
  ...
}
```


- **C# foreach array**

In the following example, we go over elements of an array using the foreach statement.

Program.cs

```
int[] vals = {1, 2, 3, 4, 5};

foreach (var val in vals)
{
  Console.WriteLine(val);
}
```

The example prints all elements of the array of integers.

```
$ dotnet run
1
2
3
4
5
```

C# foreach List

In the following example, we loop over a list with the foreach statement.

Program.cs

```
var words = new List<string> {"tea", "falcon", "book", "sky"};
foreach (var word in words)
{
    Console.WriteLine(word);
}
```

We print all elements of the list of strings.

```
$ dotnet run
tea
falcon
book
sky
```

C# foreach list of lists

In the next example, we loop over a list of lists.

Program.cs

```
var words = new List<List<string>>
```



```
{
    new List<string>{"tea", "falcon", "book", "sky"},
    new List<string>{"cup", "crown", "borrow", "moore"},
    new List<string>{"arm", "nice", "frost", "sea"}
};

foreach (var nested in words)
{
    foreach (var word in nested)
    {
        Console.WriteLine(word);
    }
}
```

To iterate over a list of lists, we use two foreach loops.

C# forEach array

In the following example, we go over elements of an array using the forEach method.

Program.cs

```
int[] vals = {1, 2, 3, 4, 5};
```

```
Array.ForEach(vals, e => Console.WriteLine(e));
```

The example prints all elements of the array of integers. We use the Array.ForEach method.

- **Using Tuples to pass multiple values to a function**

C# tuple is a data structure in C#. In this article, learn how to use tuples in C# applicaitons. If you need to regurn multiple values from a method in C#, there are three ways to do so without using tuples.

1. Using 0ut parameters
2. Using a class, struct, or a record type
3. Anonymous types returned through a dynamic return type

In C#, Tuples solve this problem. C# tuple is a data structure that provides an easy way to represent a single set of data. The System.Tuple class provides static methods to create tuple objects.

Tuples allow us to,

1. Create, access, and manipulate a data set

2. Return a data set from a method without using out parameter
3. Pass multiple values to a method through a single parameter

We can create a Tuple<> using its constructor, or the "Create" method. The code snippet in Listing 1 creates a 3-tuple using a constructor. The tuple is a set of three data types, including two strings and one int, that represents an author's name, book title, and year of publication.

```
// Create a 3-tuple

var author = new Tuple<string, string, int>("Mahesh Chand", "ADO.NET Programming", 2003);

// Display author info

System.Console.WriteLine("Author {0} wrote his first book titled {1} in {2}.", author.Item1, author.Item2, author.Item3);
```

How to return tuples in C#?

A tuple can be used to return a data set as a single variable of a method. The code snippet in following program returns a tuple with three values.

```
public static Tuple < string, string, int > GetTupleMethod() {
  // Create a 3-tuple and return it
  var author = new Tuple < string,
    string, int > ("Schildt, Herbert", "The Complete ReferenceC#", 2018);
  return author;
}
```

WHAT IS DELEGATE?

A delegate is reference type that basically encapsulates the reference of methods.It is a similar to class that store the reference of methods which have same signature as delegate has.

It is very similar to a function pointer in C++,but delegate is type safe,object oriented and secure as compare to C++ function pointer 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

```
namespace Session8Demo
{
    public delegate int MyDelegate(int x,int y);//Declare a Delegate

    class TestProgram
    {

        public int calculateFunction(int a,int b)//Target method
        {
            return (a + b);
        }

        static void Main(string[] args)
        {
            TestProgram obj=new TestProgram();
            //Create Delegate Instance that points to the Method
            MyDelegate d = new MyDelegate(obj.calculateFunction);
            //Invoke the Delegate, which calls the method
            Console.WriteLine(d(12, 22));
        }
    }
}
```


WHYDOWEUSE DELEGATE?

There are following reason because of we use delegate.

1. It is usedfortypesafety.
2. Forexecutingmultiplemethodsthroughoneexecution.
3. It allowsustopassmethodsasparameter.
4. Forasynchronousprogramming.
5. ForCallbackmethodimplementation.
6. Itisalsousedwhenworkingwitheventbasedprogramming.
7. When creating an onymousmethod.
8. When working with lambda expression.
9. A Delegate is a function pointer that allows you to reference a method.
10. A Function that is added to delegates must have same return type and same signature as delegate.

HOW MANY TYPES OF DELEGATE IN C#?

Therearethreetypesofdelegatesavailablein C#.

1. Simple/SingleDelegate
2. MulticastDelegate
3. GenericDelegate

WHAT ARE THE WAYS TO CREATE AND USE DELEGATE?

Thereareonlyfive stepstocreateanduseadelegate andthese areasfollowing.

1. Declareadelegatetype.
2. Createorfindamethodwhich hassametypeofsignature.
3. Createobject/instanceofdelegate
4. Passthereferenceofmethodwithdelegateinstance.
5. At lastInvokethedelegate object.

WHAT IS SINGLE DELEGATE?

WhenDelegatetakesreferencewithsinglemethodonlythenitiscalledSingleDelegate.Itisusedforinvocation ofonly onereferencemethod.

WHATISMULTICASTDELEGATE?

When Delegate takes reference with multiple methods then it is called multicast delegate. It isusedforinvocationofmultiplereferencemethods.Wecanaddorremovereferencesofmultiple methods with same instance of delegate using (+) or (-) sign respectively. It need toconsiderherethat allmethodswillbeinvokedinoneprocessandthatwillbein sequence.

WHATISMULTICASTDELEGATE

When Delegate takes reference with multiple methods then it is called multicast delegate. It isusedforinvocationofmultiplereferencemethods.Wecanaddorremovereferencesofmultiple methods with same instance of delegate using (+) or (-) sign respectively. It need toconsiderherethat allmethodswillbeinvokedinoneprocessandthatwillbein sequence.

Followingexamplehasadelegatenameas"TestMultiCastDelegate"withoneparameterasint.

```
usingSystem;
using
System.Collections.Generic;using
System.Linq;
usingSystem.Text;

namespaceEventAndDelegateExamples
{
  delegate void TestMultiCastDelegate(int
  a);classMultiCastDelegateExample
  {
    internalvoidGetSalary(intAnnualSalary)
    {
      Console.WriteLine("MonthlySalaryis"+AnnualSalary/12);
    }
```

Below code snippet shows that first we are creating a instance of TestMultiCastDelegate and passing the reference of methods [GetSalary] and [GetSalaryAfterDeduction]with"objDelegate2".

```
MultiCastDelegateExampleobjMultiCastDelegateExample=newMultiCastDelegateExample();


//Creating the instance of the

delegateTestMultiCastDelegateobjDelegate

2=null;


//Referencing the multiple methods using +

signobjDelegate2+=objMultiCastDelegateExample.GetSalary
```



```
objDelegate2.Invoke(60000);
```

HOWTOACHIEVECALLBACKINDELEGATE?

Callbackistermwhereaprocessisgoingonandinbetweenittargetssomeachievementthenitreturntomainmethod.Forcallback,wejustneedtoencapsulatethemethodwithdelegate.

```
objCallBackMethodExample.CheckEvenEvent+=newOnEvenNumberHandler(objCallBackMethodExample.CallBackMethod);
```

Let see an example, CallBackMethodExample is a class which have a method CallBackMethod. Itwillbeexecutedwhen somecriteriawillbefulfill.

```
publicdelegatevoidOnEvenNumberHandler(objectsender,EventArgse);


publicclassCallBackMethodExample

{

    publicvoidCallBackMethod(objectsender,EventArgse)

    {

      Console.WriteLine("EvenNumberhasfound!");

    }

}
```

Whenwearegoingtocallthismethodusing Delegateforcallback,onlyneedtopassthismethodnameasareference.

```
CallBackMethodExampleobjCallBackMethodExample=newCallBackMethodExample();


objCallBackMethodExample.CheckEvenEvent +=new
OnEvenNumberHandler(objCallBackMethodExample.CallBackMethod);

Random random = new

Random();for (inti=0; i<6;i++)

{
```

```
var randomNumber = random.Next(1,
10);Console.WriteLine(randomNumber);
if (randomNumber%2== 0)
{
  objCallBackMethodExample.OnCheckEvenNumber();
}
}
```

HOWTOACHIEVEASYNCCALLBACKINDELEGATE?

Async callback means, process will be going on in background and return back when it will becompleted. In C#, Async callback can be achieved using AsyncCallback predefined delegate asfollowing.

```
publicdelegatevoidAsyncCallback(IAsyncResultar);
```

Kindly follow the following example to understand the Async callback. There is a class name asCallBackMethodExamplewhichhavetwomethod,oneisTestMethodandotheroneisAsyncCallBac kFunction. But as we can see AsyncCallBackFunction is taking IAsyncResult as aparameter.

```
public delegate void OnEvenNumberHandler(object sender, EventArgs
e);publicclass CallBackMethodExample
{
  publicvoidTestMethod(objectsender,EventArgse)
```



```
  {
    Console.WriteLine("Thisissimpletestmethod");
  }

  publicvoidAsynCallBackFunction(IAsyncResultasyncResult)
  {
    OnEvenNumberHandlerdel=(OnEvenNumberHandler)asyncResult.AsyncState;del.EndInv
    oke(asyncResult);
    Console.WriteLine("Wait For 5
    Seconds");Console.WriteLine("...");
    Console.WriteLine("...");
    Console.WriteLine("...");
    Console.WriteLine("...");System.Threading.Thread.Sleep(5000
    );Console.WriteLine("AsyncCallBackFunctionCompleted");
  }
}
```

Whenwearegoingtoinvokethedelegate,onlyjustneedtopassAsyncCallBackFunctionwithdelegate instance.

```
CallBackMethodExampleobjCallBackMethodExample=newCallBackMethodExample();
OnEvenNumberHandler objDelegate3 = new
OnEvenNumberHandler(objCallBackMethodExample.TestMethod);
objDelegate3.BeginInvoke(null, EventArgs.Empty, new
AsyncCallback(objCallBackMethodExample.AsynCallBackFunction),objDelegate3);
Console.WriteLine("EndofMainFunction");
Console.WriteLine("WaitingforAsyncCallBackFunction");
```

```
End of Main Function
Waiting for Async Call Back Function
This is simple test method
Wait For 5 Seconds
...
...
...
...
Async Call Back Function Completed
```

WhatareFunc 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 SessionDemo
{
    class Program
    {
        static int addNum(int x, int y)
        {
            return x + y;
        }
        static string display(string msg)
        {
            return msg;
        }
        static float calculateTax()
        {
            return 0.5f * 1200 ;
        }
    }
```

```
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);
}
```


WhatareAction 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");
        }
    }
}
```

What are 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.


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

WhatareAnonymous 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

     class TestProgram
     {

        static void Main(string[] args)
        {
            //Anonymous method
            MyDelegate d = delegate (int val) {
                Console.WriteLine(val);
            };

            d(10);
        }
    }
}
```

What is 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

     class TestProgram
     {

        static void Main(string[] args)
        {
            //Lambda expressions
            MyDelegate d1 = x =>
             {
                Console.WriteLine(x);
             };
            d1(8);
        }
    }
}
```

WhatareEvents ?

Events are higherlevelofencapsulationoverdelegates.Eventsusedelegatesinternally.Delegates are naked and when passed to any other code, the client code can invoke thedelegate.Eventprovidesa publisher/subscribermechanismmodel.

Sosubscriberssubscribetotheeventandpublisherthenpushmessagestoallthesubscribers.Belowis a simplecodesnippet forthesame:-

Createadelegateanddeclaretheeventforthesame.

public delegate void CallEveryone();publiceventCall EveryoneMyEvent;

Raisethe event.

MyEvent();

Attached clientmethodstotheeventarefired/notified.

obj.MyEvent+=Function1;

Difference between delegate and events:-

Theycannotbecomparedbecauseonederivesfromtheother.

- Actually,eventsusedelegatesin bottom.Buttheyaddanextralayerofsecurityonthedelegates,thusforming thepublisherand subscribermodel.
- As delegates are function to pointers, they can move across any clients. So any ofthe clients can add or remove events, which can be confusing. But events give theextra protection / encapsulation by adding the layer and making it a publisher andsubscribermodel.

Exception Handling

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

Try block

A try block is used by C# programmers to partition code that might be affected by an exception. Associated catch blocks are used to handle any resulting exceptions.
A finally block contains code that is run whether or not an exception is thrown in the `try` block, such as releasing resources that are allocated in the `try` block.
A `try` block requires one or more associated `catch` blocks, or a `finally` block, or both.

The following examples show a `try-catch` statement, a `try-finally` statement, and a `try-catch-finally` statement.


```
C#

try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

```
C#

try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

```
C#

try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

A `try` block without a `catch` or `finally` block causes a compiler error.

Catch Blocks

A `catch` block can specify the type of exception to catch. The type specification is called an *exception filter*. The exception type should be derived from Exception. In

general, don't specify Exception as the exception filter unless either you know how to handle all exceptions that might be thrown in the `try` block, or you've included a throw statement at the end of your `catch` block.

Multiple catch blocks with different exception classes can be chained together. The catch blocks are evaluated from top to bottom in your code, but only one catch block is executed for each exception that is thrown. The first catch block that specifies the exact type or a base class of the thrown exception is executed. If no catch block specifies a matching exception class, a catch block that doesn't have any type is selected, if one is present in the statement. It's important to position catch blocks with the most specific (that is, the most derived) exception classes first.

Catch exceptions when the following conditions are true:

You have a good understanding of why the exception might be thrown, and you can implement a specific recovery, such as prompting the user to enter a new file name when you catch a FileNotFoundException object.

You can create and throw a new, more specific exception

```
C#

int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

You can also specify *exception filters* to add a boolean expression to a catch clause. Exception filters indicate that a specific catch clause matches only when that condition is true. In the following example,

both catch clauses use the same exception class, but an extra condition is checked to create a different error message:

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be greater than the array size.", e);
    }
}
```

Finally Blocks

A `finally` block enables you to clean up actions that are performed in a `try` block. If present, the `finally` block executes last, after the `try` block and any matched `catch` block. A `finally` block always runs, whether an exception is thrown or a `catch` block matching the exception type is found.

The `finally` block can be used to release resources such as file streams, database connections, and graphics handles without waiting for the garbage collector in the runtime to finalize the objects. For more information See the using Statement.

In the following example, the `finally` block is used to close a file that is opened in the `try` block. Notice that the state of the file handle is checked before the file is closed. If the `try` block can't open the file, the file handle still has the value `null` and the `finally` block doesn't try to close it. Instead, if the file is opened successfully in the `try` block, the `finally` block closes the open file.

C#

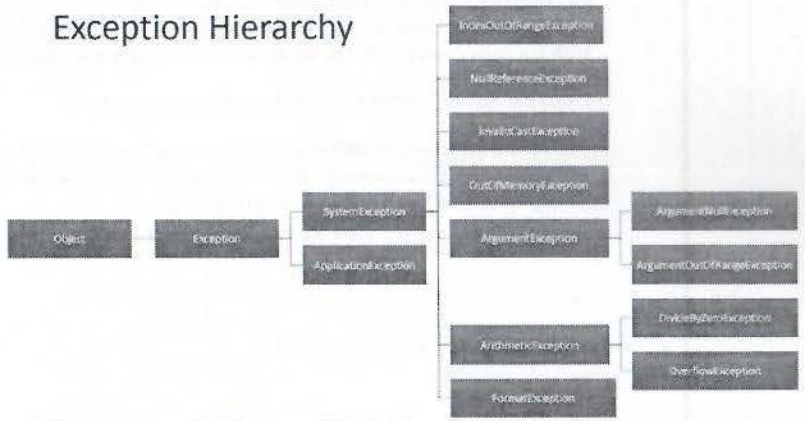
```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

Compiler-generated exceptions

Exception	Description
ArithmeticException	A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException.
ArrayTypeMismatchException	Thrown when an array can't store a given element because the actual type of the element is incompatible with the actual type of the array.
DivideByZeroException	Thrown when an attempt is made to divide an integral value by zero.
IndexOutOfRangeException	Thrown when an attempt is made to index an array when the index is less than zero or outside the bounds of the array.
InvalidCastException	Thrown when an explicit conversion from a base type to an interface or to a derived type fails at run time.
NullReferenceException	Thrown when an attempt is made to reference an object whose value is null.
OutOfMemoryException	Thrown when an attempt to allocate memory using the new operator fails. This exception indicates that the memory available to the common language runtime has been exhausted.
OverflowException	Thrown when an arithmetic operation in a checked context overflows.
StackOverflowException	Thrown when the execution stack is exhausted by having too many pending method calls; usually indicates a very deep or infinite recursion.

TypeInitializationException	Thrown when a static constructor throws an exception and no compatible catch clause exists to catch it.

Exception Hierarchy



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.

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.

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
{
    class MyException : Exception
    {
        public MyException(string message)
        {
            Console.WriteLine(message);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            int x = 0;
            try
            {
                if(cond)
                throw new MyException("This is incorrect...");
            }catch(MyException ex)
            {
                Console.WriteLine("Error: " + ex.Message);
            }
        }
    }
}
```

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

```
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 p) { Console.WriteLine("M4 method"); }
        public static void M5(this ExistingClass p,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

- Its 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

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.

```
namespace Session10Demo
{
    public partial class Student
    {
        public int studentID;
        public string studentName;

        public Student(int id, string name)
        {
            this.studentID = 1;
            this.studentName = name;
        }
    }
    public partial class Student
    {

        public void displayStudents()
        {
            Console.WriteLine("ID:" + studentID);
            Console.WriteLine("Name:" + studentName);
        }
    }

    class TestProgram
    {

        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


```
namespace Session10Demo
{
    public partial class Student
    {
        public int studentID;
        public string studentName;

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


    }
    class TestProgram
    {

        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.

Advantages of LINQ

- We do not need to learn new query language syntaxes for different sources of data because it providesthe standard query syntax for the various data sources.
- In LINQ, we have to write the 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.

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:

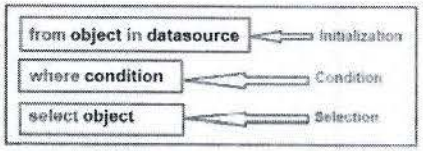
1. Initialization(to work with a particular data source)
2. Condition(where, filter, sorting condition)
3. Selection (single selection, group selection or joining)

- There are two basic ways to write a LINQ query to IEnumerable collection or IQueryable data sources.

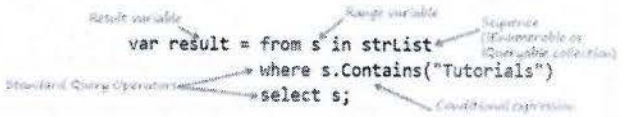
1. Using Query Syntax
2. Using Method Syntax

Writing LINQ Query:

1. Using Query Syntax



Example :



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);
            }
        }
    }
}
```

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);
            }
        }
    }
}
```

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 opertors 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.

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"));
```

Extension method — Lambda expression

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);
            }
        }
    }
}
```

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.

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.dll4 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

Whatareattributes?

Attributes provide a powerful method of associating metadata, or declarative information, withcode (assemblies, types, methods, properties, and so forth). After an attribute is associatedwith aprogram entity,theattribute canbequeriedatruntimebyusingatechnique called*reflection.*

Attributeshavethefollowingproperties:

- Attributes add metadata to your program. *Metadata* is information about the typesdefined in a program. All .NET assemblies contain a specified set of metadata thatdescribes the types and type members defined in the assembly. You can add customattributestospecifyanyadditionalinformationthat is required.
- Youcanapplyoneormoreattributestoentireassemblies,modules,orsmallerprogramelements suchas classesandproperties.

- Attributescanacceptargumentsinthesamewayasmethodsandproperties.
- Yourprogramcanexamineitsownmetadataorthemetadatainotherprogramsbyusingreflection.

Therearetwotypesofattributes:

1. Predefinedattributes:theseareprovidedbyFCL.Ex.Obsolete,Serializable,Conditionaletc.
2. Customattributes:thesearedevelopedbydevelopersasperthерequirements.

Note:Alltheattributesarederiveddirectlyorindirectlyfrom System.Attributeclass.

Howtocreatecustomattributes?

Theprimarystepstoproperlydesigncustomattributeclassesareasfollows:

- ApplyingtheAttributeUsageAttribute
- Declaringtheattributeclass
- Declaringconstructors

Declaringproperties

ApplyingtheAttributeUsageAttribute

A custom attribute declaration begins with the System.AttributeUsageAttribute, which definessome of the key characteristics of your attribute class. For example, you can specify whetheryour attribute can be inherited by other classes or specify which elements the attribute can beappliedto.Thefollowingcodefragmentdemonstrateshowtousethe AttributeUsageAttribute.

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

The AttributeUsageAttributehas three members that are important for the creation of customattributes:AttributeTargets,Inherited,andAllowMultiple.

AttributeTargetsMember

In the previous example, AttributeTargets.Allis specified, indicating that this attribute can beappliedto allprogramelements.Alternatively,you canspecifyAttributeTargets.Class,

indicating that your attribute can be applied only to a class, or AttributeTargets.Method,indicatingthatyourattributecanbeappliedonlytoamethod.Allprogram elementscanbemarkedfordescriptionby acustom attributeinthis manner.

You can also pass multiple AttributeTargetsvalues. The following code fragment specifies that acustom attributecanbeappliedtoany class ormethod.

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Method)]

InheritedProperty

The AttributeUsageAttribute.Inheritedproperty indicates whether your attribute can beinherited by classes that are derived from the classes to which your attribute is applied. Thispropertytakeseither atrue(thedefault)or falseflag. In thefollowing example, MyAttribute has a default Inheritedvalue of true, while YourAttribute hasanInheritedvalueof false.

```
// This defaults to Inherited =
true.publicclassMyAttribute:Attrib
ute
{
  //...
}
[AttributeUsage(AttributeTargets.Method, Inherited =
false)]publicclass YourAttribute: Attribute
{
  //...
```



```
}
```

Thetwoattributesarethenappliedtoamethodinthebaseclass MyClass.

```
publicclassMyClass
{
  [MyAttribute][
  YourAttribute]
  publicvirtualvoidMyMethod()
  {
    //...
  }}
```

Finally, the class YourClass is inherited from the base class MyClass. ThemethodMyMethod shows MyAttribute,butnotYourAttribute.

```
publicclassYourClass:MyClass
{
  //MyMethodwillhaveMyAttributebutnotYourAttribute.

  publicoverridevoidMyMethod()
  {
    //...
  }

}
```


AllowMultipleProperty

TheAttributeUsageAttribute.AllowMultiplepropertyindicateswhethermultipleinstancesofyourattributecanexistonanelement.Ifsettotrue,multiple instancesareallowed;ifset tofalse(thedefault),onlyoneinstance isallowed.

In the following example, MyAttribute has a default AllowMultiplevalue of false,while YourAttribute hasa valueoftrue.

```
publicclassMyAttribute:Attribute
{
}

[AttributeUsage(AttributeTargets.Method, AllowMultiple =
true)]publicclass YourAttribute:Attribute
{
}
```

When multiple instances of these attributes are applied, MyAttribute produces a compilererror. The following code example shows the valid use of YourAttribute and the invalid useofMyAttribute.

```
publicclassMyClass
{
  //Thisproducesanerror.
  // Duplicates are not
  allowed.[MyAttribute]
  [MyAttribute]
  publicvoidMyMethod()
  {
    //...
  }

  // This is
  valid.[YourAttr
  ibute][YourAtt
  ribute]
  publicvoidYourMethod()
  {
    //...
  }
}
```

If both the AllowMultipleproperty and the Inheritedproperty are set to true, a class that isinherited from another class can inherit an attribute and have another instance of the sameattribute applied in the same child class. If AllowMultipleis set to false, the values of anyattributesintheparentclasswillbeoverwrittenbynewinstancesofthesameattributeinthechild class.

DeclaringtheAttributeClass

After you apply the AttributeUsageAttribute, you can begin to define the specifics of yourattribute.Thedeclarationofanattributeclasslookssimilartothedeclarationofatraditionalclass,as demonstrated by thefollowing code.

```
[AttributeUsage(AttributeTargets.Method)]public
classMyAttribute : Attribute
{
  //...
}
```

Thisattributedefinitiondemonstratesthefollowingpoints:

- Attributeclassesmustbedeclaredaspublicclasses.
- By convention, the name of the attribute class ends with the word **Attribute**. While notrequired,thisconventionisrecommendedforreadability.Whentheattributeisapplied,thein clusionoftheword Attributeisoptional.
- Allattributeclassesmustinheritdirectlyorindirectlyfrom System.Attribute.
- In Microsoft Visual Basic, all custom attribute classes must haveto be System.AttributeUsageAttributeattribute.

DeclaringConstructors

Attributesareinitializedwithconstructorsinthesamewayastraditionalclasses.Thefollowingcode fragment illustrates a typical attribute constructor. This public constructor takes aparameter andsetsa membervariableequaltoits value.

```
publicMyAttribute(boolmyvalue)
{
  this.myvalue=myvalue;
}
```


You can overload the constructor to accommodate different combinations of values. If you alsodefine a propertyfor your custom attribute class, you can use a combination of named andpositional parameters when initializing the attribute. Typically, you define all requiredparameters as positional and all optional parameters as named. In this case, the attributecannot be initialized without the required parameter. All other parameters are optional. NotethatinVisualBasic,constructorsforanattributeclassshouldnotuseaParamArrayargument.

The following code example shows how an attribute that uses the previous constructor can beapplied using optional and required parameters. It assumes that the attribute has one requiredBoolean valueandoneoptional stringproperty.

```
// One required (positional) and one optional (named) parameter are
applied.[MyAttribute(false,OptionalParameter="optional data")]
publicclassSomeClass
{
  //...
}
[MyAttribute(false)]
publicclassSomeOtherClass
{
  //...
}
```

DeclaringProperties

Ifyouwanttodefineanamedparameterorprovidean easywaytoreturnthevalues storedbyyour attribute, declare a property. Attribute properties should be declared as public entitieswith a description of the data type that will be returned. Define the variable that will hold thevalue of your property and associate it with the **get** and **set** methods. The following codeexampledemonstrateshowtoimplement asimplepropertyinyourattribute.

```
publicboolMyProperty
{
  get {return
  this.myvalue;}set{this.my
  value=value;}
}
```

CustomAttributeExample

Thissectionincorporatesthepreviousinformationandshowshowtodesignasimpleattributethat documents information about the author of a section of code. The attribute in thisexample storesthename and levelof theprogrammer, and whetherthecodehasbeen

reviewed.Itusesthreeprivatevariablestostoretheactualvaluestosave.Eachvariableisrepresented by a public property that gets and sets the values. Finally, the constructor isdefinedwithtworequiredparameters.

```
[AttributeUsage(AttributeTargets.All)]pub
licclassDeveloperAttribute:Attribute
{
  // Private fields.private
  string name;private
  string
  level;privateboolrevie
  wed;

  // This constructor defines two required parameters: name and

  level.publicDeveloperAttribute(stringname,string level)
  {
    this.name =
    name;this.level =
    level;this.reviewed=f
    alse;
  }

  //DefineNameproperty.
  //Thisisaread-onlyattribute.

  publicvirtualstringName
  {
    get{returnname;}
  }
  //DefineLevelproperty.
  //Thisisaread-onlyattribute.

  publicvirtualstringLevel
  {
    get{returnlevel;}
  }

  //DefineReviewedproperty.
  //Thisisaread/writeattribute.

  publicvirtualboolReviewed
  {
```



```
    get{returnreviewed;}
```

```
    set{reviewed=value;}
  }
}
```

You can apply this attribute using the full name, DeveloperAttribute, or using the abbreviatedname,Developer,inoneofthefollowing ways.

```
[Developer("JoanSmith","1")]
```

-or-

```
[Developer("JoanSmith","1",Reviewed=true)]
```

Thefirstexampleshowstheattributeappliedwithonlytherequirednamedparameters,whilethe second example shows the attribute applied with both the required and optionalparameters.

Whatisreflection?

Reflection provides objects (of type Type) that describe assemblies, modules and types. You canuse reflection to dynamically create an instance of a type, bind the type to an existing object, orget the type from an existing object and invoke its methods or access its fields and properties. Ifyou areusingattributes inyourcode, reflectionenablesyoutoaccess them.

UsesforReflectionC#

Thereareseveralusesincluding:

1. UseModuletogetallglobalandnon-globalmethodsdefinedinthemodule.
2. UseMethodInfotolookatinformationsuchasparameters,name,returntype,accessmodifiers andimplementationdetails.
3. UseEventInfotofindouttheevent-handlerdatatype,thename,declaringtypeandcustom attributes.
4. UseConstructorInfotogetdataontheparameters,accessmodifiers,andimplementationdetailsof a constructor.
5. UseAssemblytoloadmoduleslistedintheassemblymanifest.
6. Use PropertyInfo to get the declaring type, reflected type, data type, name and writablestatusofaproperty ortogetandsetproperty values.
7. Use CustomAttributeData to find out information on custom attributes or to reviewattributes withouthavingtocreatemore instances.

OtherusesforReflectionincludeconstructingsymboltables,todeterminewhichfieldstopersistandt hroughserialization.

ReflectionExamples

Thisexample showshowtodynamicallyloadassembly,howtocreateobject instance,howtoinvoke methodorhowto getandsetpropertyvalue.

Createinstancefromassemblythatisinyourproject References

ThefollowingexamplescreateinstancesofDateTimeclassfromtheSystem assembly:

```
//createinstanceofclassDateTime
DateTimedateTime=(DateTime)Activator.CreateInstance(typeof(DateTime));

// create instance of DateTime, use constructor with parameters (year, month,
day)DateTimedateTime= (DateTime)Activator.CreateInstance(typeof(DateTime),
                                newobject[]{2008,7,4});
```

Createinstancefromdynamicallyloadedassembly

All the following examples try to access to **sample class Calculator** from Test.dll assembly. Thecalculator class canbedefinedlike this.

```
namespaceTest
{
  publicclassCalculator
  {
    public Calculator() { ...
    }privatedouble_number;
    public double Number { get { ... } set { ...
    } }publicvoid Clear(){...}
    privatevoidDoClear(){...}
    public double Add(double number) { ...
    }publicstatic doublePi{...}
    publicstaticdoubleGetPi(){...}
  }
}
```

Examples of using reflection to load the Test.dll assembly, to create instance of the Calculatorclassandtoaccess itsmembers(public/private,instance/static).

```
//dynamicallyloadassemblyfromfileTest.dll
    AssemblytestAssembly=Assembly.LoadFile(@"c:\Test.dll");

//gettypeofclassCalculatorfromjustloadedassembly
    TypecalcType=testAssembly.GetType("Test.Calculator");

    //createinstanceofclassCalculator
    objectcalcInstance=Activator.CreateInstance(calcType);

    //getinfoaboutproperty:publicdoubleNumber
    PropertyInfonumberPropertyInfo=calcType.GetProperty("Number");

    //getvalueofproperty:publicdoubleNumber
    doublevalue=(double)numberPropertyInfo.GetValue(calcInstance,null);

    // set value of property: public double
    NumbernumberPropertyInfo.SetValue(calcInstance,10.0,null);

    // get info about static property: public static double
    PiPropertyInfopiPropertyInfo=calcType.GetProperty("Pi");

    //getvalueofstaticproperty:publicstaticdoublePi
    doublepiValue=(double)piPropertyInfo.GetValue(null,null);

    // invoke public instance method: public void
    Clear()calcType.InvokeMember("Clear",
      BindingFlags.InvokeMethod | BindingFlags.Instance |
      BindingFlags.Public,null,calcInstance, null);

    // invoke private instance method: private void
    DoClear()calcType.InvokeMember("DoClear",
      BindingFlags.InvokeMethod | BindingFlags.Instance |
      BindingFlags.NonPublic,null,calcInstance, null);

    // invoke public instance method: public double Add(double
    number)doublevalue=(double)calcType.InvokeMember("Add",
      BindingFlags.InvokeMethod | BindingFlags.Instance |
      BindingFlags.Public,null,calcInstance, new object[]{20.0});

    // invoke public static method: public static double
    GetPi()doublepiValue=(double)calcType.InvokeMember("
    GetPi",
```



```
      BindingFlags.InvokeMethod | BindingFlags.Static |
      BindingFlags.Public,null, null,null);
    //getvalueofprivatefield: privatedouble_number
    double value =
      (double)calcType.InvokeMember("_number",BindingFlags.GetField |
      BindingFlags.Instance | BindingFlags.NonPublic,null,calcInstance,
      "");
```

FileHandlinginC#

All the objects created at runtime reside on heap section primary memory. Once they are outof scope the objects are released by garbage collector. So the data stored inan object cannotbe reused once the program is terminated. Therefore we store the data on secondary storagedevices in the form of file. File handling is an unmanaged resource in your application system. Itisoutsideyourapplicationdomain(unmanaged resource).It isnot managed byCLR.

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...");
            }
        }
    }
}
```

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}

```
namespace Session11Demo
{
    public class Program
    {
        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...");
            }
        }
    }
}
```

Whatisastream?

A stream is a sequence of bytes. In the file system, streams contain the data that is written to afile, and that gives more information about a file than attributes and properties. When youopen a file for reading or writing, it becomes stream. Stream is a sequence of bytes travelingfrom a source to a destination over a communication path. The two basic streams are input andoutputstreams. Inputstream isused toread and output stream is used towrite. TheSystem.IOnamespace includesvariousclasses forfilehandling.

Streamis the abstract base class of all streams. A stream is an abstraction of a sequence ofbytes, such as a file, an input/output device, an inter-process communication pipe, or a TCP/IPsocket. The Streamclass and its derived classes provide a generic view of these different typesof input and output, and isolate the programmer from the specific details of the operatingsystem andtheunderlying devices.

Streamsinvolvethreefundamentaloperations:

- You can read from streams. Reading is the transfer of data from a stream into a datastructure,suchasanarrayofbytes.
- You can write to streams. Writing is the transfer of data from a data structure into astream.
- Streamscansupportseeking.Seekingreferstoqueryingandmodifyingthecurrentposition within a stream. Seek capability depends on the kind of backing store a streamhas. For example, network streams have no unified concept of a current position, andthereforetypically

donotsupportseeking.

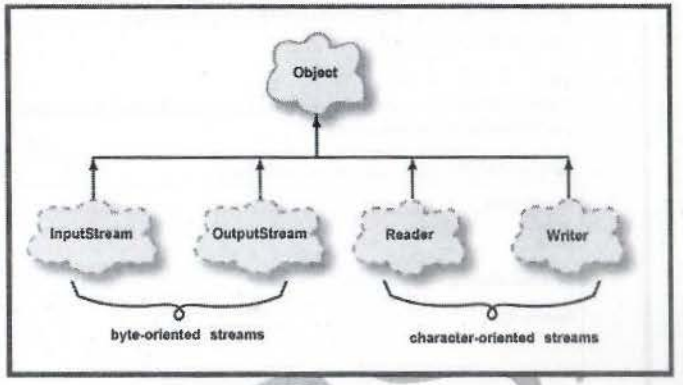
Someofthe more commonly used streams that inherit from Streamare FileStream,andMemoryStream.

WhatisFileStreamclass?

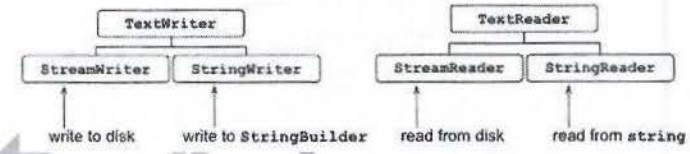
Use the FileStreamclass to read from, write to, open, and close files on a file system, and tomanipulate other file-related operating system handles, including pipes, standard input, andstandardoutput.Youcan usetheRead, Write,CopyTo,andFlushmethodstoperform

synchronous operations, or the ReadAsync, WriteAsync, CopyToAsync, and FlushAsyncmethodstoperformasynchronous operations.

Why there is a need of reader and writer classes?



As shown in the picture above, FileStream is byte oriented. But if we want to read and write characters then we need character readerandwriter streams.



TextReaderis the abstract base class of StreamReaderand StringReader, which read charactersfromstreamsandstrings,respectively.Usethesederivedclassestoopenatextfileforreadingas pecified rangeofcharacters, orto createareaderbasedonanexistingstream.

TextWriteris the abstract base class of StreamWriterand StringWriter, which write charactersto streams and strings, respectively. Create an instance of TextWriterto write an object to astring,writestringstoafile,ortoserializeXML.

The BinaryReaderclassisusedtoreadbinarydatafromafile.ABinaryReaderobjectiscreatedbypassingaFile Streamobjecttoitsconstructor.

The BinaryWriterclassisusedtowritebinarydatatoastream.ABinaryWriterobjectiscreatedbypassinga FileStreamobjecttoitsconstructor.

Programofreadingafileusing StreamReaderclass:

```
usingSystem;
using
System.Collections.Generic;using
System.IO;

classProgram
{
  staticvoidMain()
  {
    List<string>list=newList<string>();
    using(StreamReaderreader=newStreamReader("file.txt"))
    {
      stringline;
      while((line=reader.ReadLine())!=null)
      {
        list.Add(line); // Add to
        list.Console.WriteLine(line);//Writetoconsol
        e.
      }
    }
  }
```


Programforreadingandwriting binarydatausingBinaryReaderand BinaryWriter:

```
classProgram
  {
    staticvoidMain(string[]args)
    {

      // Create the new, empty data
      file.stringfileName=@"C:\Temp.dat
      a";if(File.Exists(fileName))
      {
        Console.WriteLine(fileName + " already
        exists!");return;
      }
      FileStreamfs=newFileStream(fileName,FileMode.CreateNew);
      // Create the writer for
      data.BinaryWriterw=newBinaryWriter(
      fs);
      // Write data to
      Test.data.for(inti =0; i<
      11;i++)
      {
        w.Write((int)i);
      }
      w.Close();
      fs.Close();
      //Createthereaderfordata.
      fs=newFileStream(fileName,FileMode.Open,FileAccess.Read);BinaryReader r
      = new BinaryReader(fs);
      // Read data from
      Test.data.for(int
      i=0;i<11;i++)
      {
        Console.WriteLine(r.ReadInt32());
      }
      r.Close();
      fs.Close();
    }
  }
```

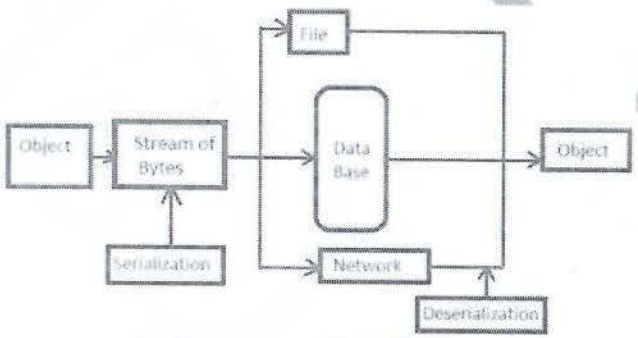
WhatisSerialization?

Serializationisconvertingobjecttostreamofbytes.Itisaprocessofbringinganobjectintoaformthatitcan bewritten onstream.It'stheprocessofconvertingtheobjectintoaformso

thatitcanbestoredonafile,databaseormemory.It istransferred acrossthenetwork.Mainpurposeistosavethestateoftheobject.

Whatisdeserialization?

Deserialization is converting stream of byte to object. Deserialization is the reverse process ofserialization. It is the process of getting back the serialization object (so that) it can be loadedintomemory.It livesthestateoftheobjectbysettingproperties,fieldsetc.



Whataredifferenttypesof Serialization techniques?

DifferentTypesofSerialization

The Microsoft .NET Framework provides an almost bewildering variety of ways to serialize anobject. This chapter focuses on XML serialization, but before we get into the details, I'd like tobrieflyexamineandcomparethe various serializationmethods offered.

XMLSerialization

XML serialization allows the public properties and fields of an object to be reduced to an XMLdocument that describes the publicly visible state of the object. This method serializes onlypublic properties and fields—private data will not be persisted, so XML serialization does notprovidefullfidelitywiththeoriginalobjectinallcases.However,becausethepersistence

format is XML, the data being saved can be read and manipulated in a variety of ways and onmultipleplatforms.

ThebenefitsofXMLserializationincludethefollowing:

Allows for complete and flexible control over the format and schema of the XML produced byserialization.

Serializedformatisbothhuman-readableandmachine-readable.

Easy to implement. Does not require any custom serialization-related code in the object to beserialized.The XML Schema Definition tool (xsd.exe) can generate an XSD Schema from a set ofserializable classes, and generate a set of serializable classes from an XSD Schema, making iteasy to programmatically consume and manipulate nearly any XML data in an object-oriented(ratherthanXML-oriented)fashion.

Objectstobeserializeddonotneedtobeexplicitlyconfiguredforserialization,eitherbytheSerializableAttributeor by implementingtheISerializableinterface.

TherestrictionsofXMLserializationincludethefollowing:

Theclasstobeserializedmusthaveadefault(parameterless)publicconstructor.Read-

onlypropertiesarenotpersisted.

Onlypublicpropertiesandfieldscanbeserialized.

SOAPSerialization

SOAPserialization issimilartoXMLserializationinthattheobjectsbeingserializedarepersisted as XML. The similarity, however, ends there. The classes used for SOAP serializationreside in theSystem.Runtime.Serializationnamespace rather thanthe System.Xml.Serializationnamespace used by XML serialization. The run-time serializationclasses(whichincludeboththe SoapFormatter andthe BinaryFormatter classes)useacompletelydifferent mechanismforserializationthan theXmlSerializerclass.

ThebenefitsofSOAPserializationincludethefollowing:

Produces a fully SOAP-compliant envelope that can be processed by any system or service thatunderstands SOAP.Supports either objects that implement the ISerializable interface to controltheirownserialization,orobjectsthat aremarkedwiththeSerializableAttributeattribute.

CandeserializeaSOAPenvelopeintoacompatiblesetofobjects.

Canserializeandrestorenon-publicandpublicmembersofanobject.

TherestrictionsofSOAPserializationincludethefollowing:

The class to be serialized must either be marked with the SerializableAttributeattribute, or mustimplementtheISerializableinterfaceandcontrolitsownserializationanddeserialization.

OnlyunderstandsSOAP.ItcannotworkwitharbitraryXMLschemas.

BinarySerialization

Binary serialization allows the serialization of an object into a binary stream, and restorationfrom a binary stream into an object. This method can be faster than XML serialization, and thebinary representation is usually much more compact thanan XML representation. However,thisperformance comesat thecostof cross-platformcompatibilityandhumanreadability.

Thebenefitsofbinaryserializationincludethefollowing:

It'sthefastestserializationmethodbecauseitdoesnothavetheoverheadofgeneratinganXMLdocument duringtheserializationprocess.

The resulting binary data is more compact than an XML string, so it takes up less storage spaceand canbetransmitted quickly.

Supportseitherobjectsthatimplementthe ISerializable interfacetocontrolitsownserialization,orobjectsthatare markedwiththe SerializableAttributeattribute.

Canserializeandrestorenon-publicandpublicmembersofanobject.

Therestrictionsofbinaryserializationincludethefollowing:

The class to be serialized must either be marked with the SerializableAttributeattribute, or mustimplementtheISerializableinterfaceandcontrolitsownserializationanddeserialization.

The binary format produced is specific to the .NET Framework and it cannot be easily used fromother systemsorplatforms.

The binary format is not human-readable, which makes it more difficult to work with if theoriginalprogramthatproducedthedata isnotavailable.

Programforbinaryserialization:

```
using
System;usingSy
stem.IO;
usingSystem.Runtime.Serialization.Formatters.Binary;

namespaceConsoleApplication1
{
  classProgram
  {
    publicstaticvoidSerializeData()
    {
      stringstr="helloworld.";
      //Createfiletosavethedata.
      FileStreamfs=newFileStream(@"D:\MyDataFile.txt",FileMode.Create);
      //BinaryFormatterobjectwillperformtheserializationBinaryFormatt
      erbf=newBinaryFormatter();
      //Serialize()methodserializesthedatatothefilebf.Serializ
      e(fs,str);
      //Closethefilefs
      .Close();
    }
    publicstaticvoidDeSerializeData()
    {
      //Openfiletoread thedata
      FileStreamfs=newFileStream(@"D:\MyDataFile.txt",FileMode.Open);
      //BinaryFormatterobjectperformsthedeserializationBinaryFormatte
      rbf=newBinaryFormatter();
      //Createtheobjecttostorethedeserializeddatastringdata=
      "";
      data=(string)bf.Deserialize(fs);f
      s.Close(); //closefile
      // Display the deserialized
      stringsConsole.WriteLine("Yourdeserializedat
      ais");Console.WriteLine(data);
    }
    staticvoidMain(string[]args)
    {
      SerializeData();D
      eSerializeData();
```



```
      Console.ReadLine();
    }
  }
```

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 obselete.
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 obselete.
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.

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

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.

```
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
{
    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(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();
        }
    }
}
```

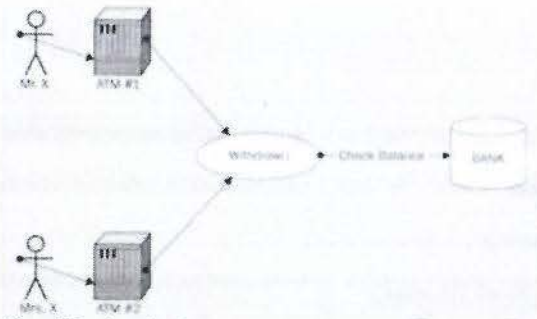
Thread Synchronization

• 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



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
{
    public class Program
    {
        public void getThreadInformation()
        {
            lock(this)
            {
                for (int i = 1; i <=10; i++)
                {
                Thread.Sleep(1000);
                Console.WriteLine("is" +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
{
    public class Program
    {
        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..");
        }
        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
{
    public class Program
    {
        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();}

        }
    }
}
```


Introduction to Asp.Net MVC 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.

Why choose ASP.NET Core?

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Architected for testability.
- Razor Pages makes coding page-focused scenarios easier and more productive.
- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused.
- Integration of modern, client-side frameworks and development workflows.
- Support for hosting Remote Procedure Call (RPC) services using gRPC.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on the following:
    - Kestrel

- IIS
- HTTP.sys
- Nginx
- Apache
- Docker

- Side-by-side versioning.
- Tooling that simplifies modern web development.

Build web APIs and web UI using ASP.NET Core MVC

ASP.NET Core MVC provides features to build web APIs and web apps:

- The Model-View-Controller (MVC) pattern helps make your web APIs and web apps testable.
- Razor Pages is a page-based programming model that makes building web UI easier and more productive.
- Razor markup provides a productive syntax for Razor Pages and MVC views.
- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.
- Built-in support for multiple data formats and content negotiation lets your web APIs reach a broad range of clients, including browsers and mobile devices.
- Model binding automatically maps data from HTTP requests to action method parameters.
- Model validation automatically performs client-side and server-side validation.

Client-side development

ASP.NET Core integrates seamlessly with popular client-side frameworks and libraries, including Blazor, Angular, React, and Bootstrap. For more information, see ASP.NET Core Blazor and related topics under *Client-side development*.

ASP.NET MVC Architecture

The MVC architectural pattern has existed for a long time in software engineering. All most all the languages use MVC with slight variation, but conceptually it remains the same.

Let's understand the MVC architecture supported in ASP.NET.

MVC stands for Model, View, and Controller. MVC separates an application into three components - Model, View, and Controller.

Model: Model represents the shape of the data. A class in C# is used to describe a model. Model objects store data retrieved from the database.

Model represents the data.

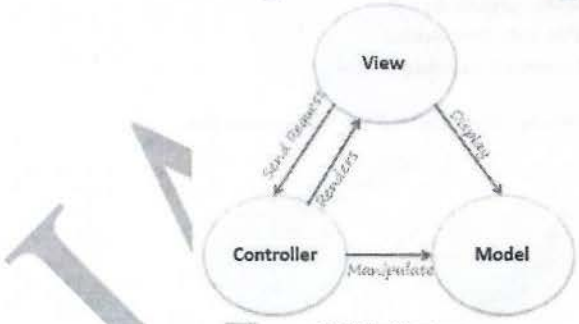
View: View in MVC is a user interface. View display model data to the user and also enables them to modify them. View in ASP.NET MVC is HTML, CSS, and some special syntax (Razor syntax) that makes it easy to communicate with the model and the controller.

View is the User Interface.

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.

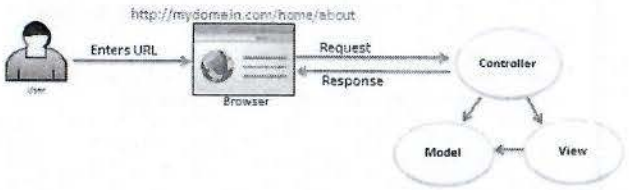
Controller is the request handler.

The following figure illustrates the interaction between Model, View, and Controller.



MVC Architecture

The following figure illustrates the flow of the user's request in ASP.NET MVC.



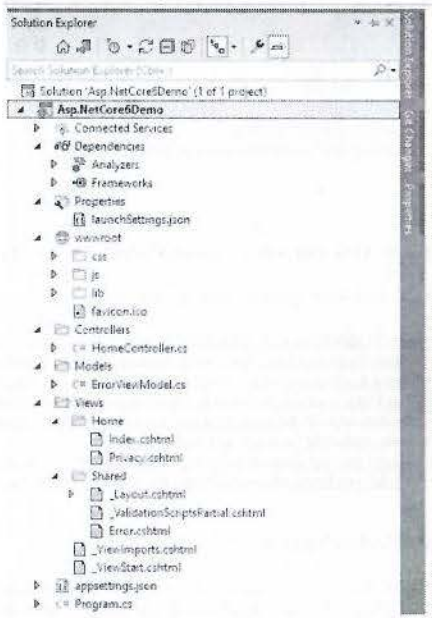
Request Flow in MVC Architecture

As per the above figure, when a user enters a URL in the browser, it goes to the webserver and routed to a controller. A controller executes related view and models for that request and create the response and sends it back to the browser.

Points to Remember

1. MVC stands for Model, View and Controller.
2. Model represents the data
3. View is the User Interface.
4. Controller is the request handler.

Understanding Folder structures and configuration files

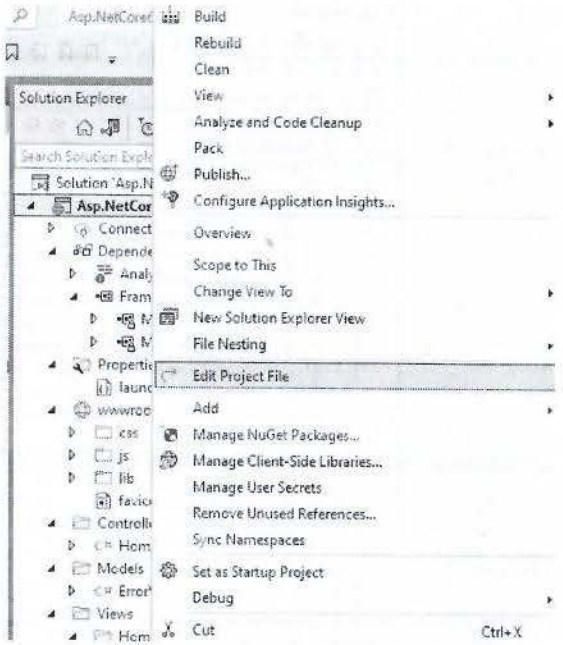


Project Folder Structure Description

The details of the default project structure can be seen in the solution explorer, it displays all the projects related to a single solution.

.csproj File

Double click on the project name in Solution Explorer to open .csproj file in the editor. Right-click on the project and then click on Edit Project File in order to edit the .csproj file. As shown in the following image.



Once clicked on Edit Project File, .csproj file will be opened in Visual Studio as shown below.


```
Program.cs   Asp.NetCore6Demo.csproj   _ViewStart.cshtml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

As you can see the project's SDK is Microsoft.NET.Sdk.Web. The target framework is net6.0 indicating that we are using .NET 6. Notice the Nullable and ImplicitUsings elements.

The <Nullable> elements decide the project wide behaviour of Nullable of Nullable reference types. The value of enable indicates that the Nullable reference types are enabled for the project.

To get more detailed about Nullable reference types Click here.

The < ImplicitUsings > element can be used to enable or disable. When

< ImplicitUsings > is set to enable, certain namespaces are implicitly imported for you.

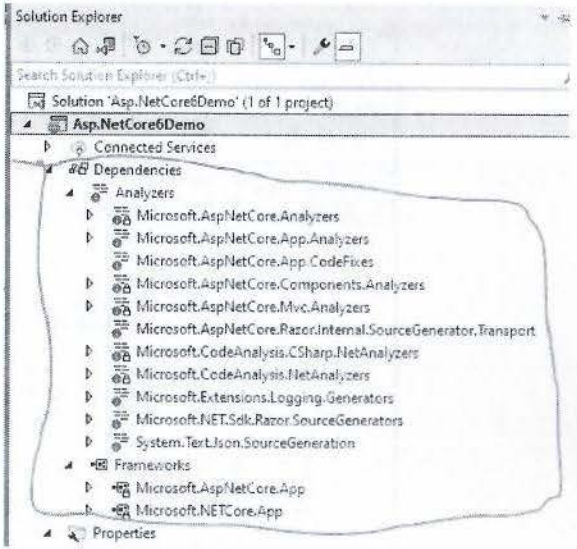
To get more detailed about common namespaces Click here.

Connected Services

It contains the details about all the service references added to the project. A new service can be added here, for example, if you want to add access to Cloud Storage of Azure Storage you can add the service here. As shown in the following image

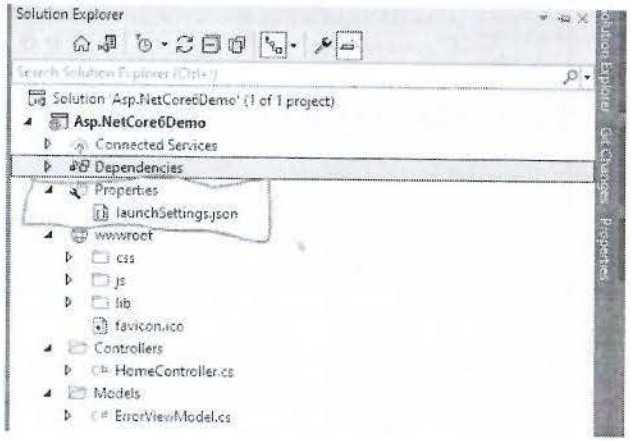
Dependencies

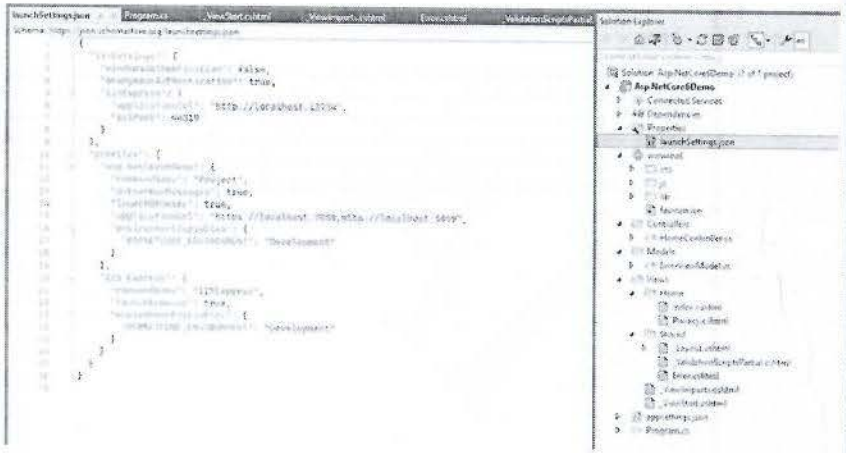
The **Dependencies** node contains all the references of the NuGet packages used in the project. Here the **Frameworks** node contains reference two most important dotnet core runtime and asp.net core runtime libraries. Project contains all the installed server-side NuGet packages, as shown below.



Properties

Properties folder contains a **launchSettings.json** file, which containing all the information required to lunch the application. Configuration details about what action to perform when the application is executed and contains details like IIS settings, application URLs, authentication, SSL port details, etc.

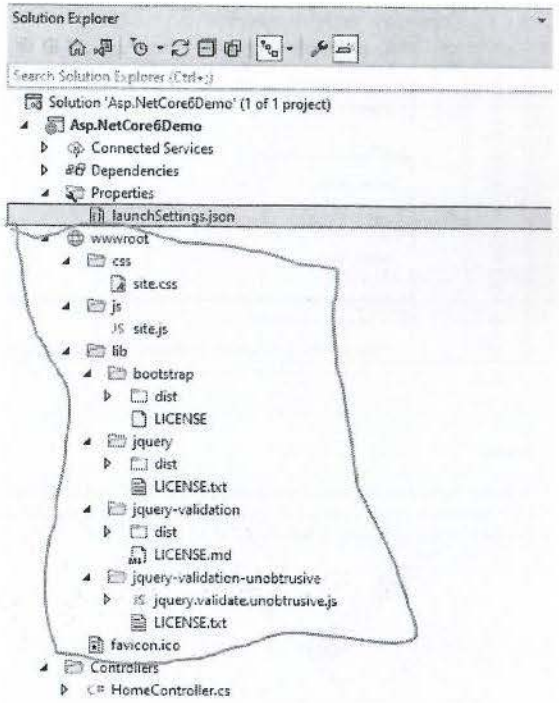




WWWroot

This is the webroot folder and all the static files required by the project are stored and served from here. The webroot folder contains a sub-folder to categorize the static file types, like all the Cascading Stylesheet files, are stored in the **CSS folder**, all the javascript files are stored in the **js folder** and the external libraries like bootstrap, jquery are kept in the **library folder**.

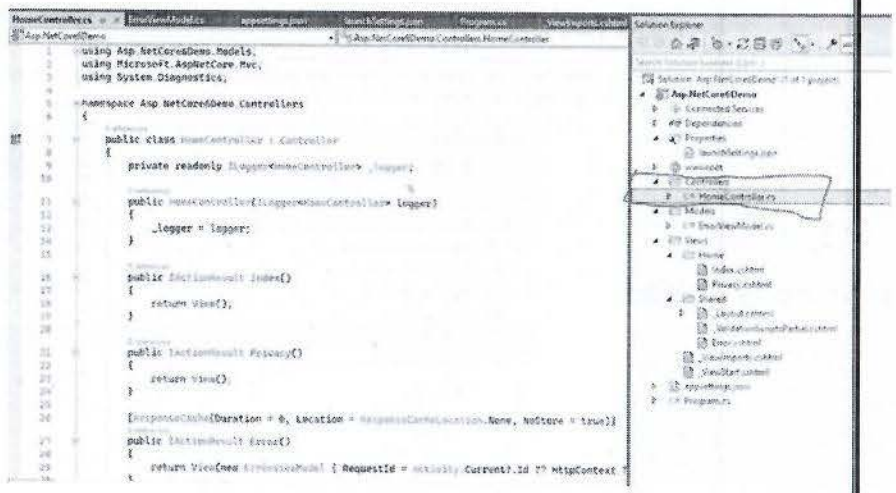
Generally, there should be separate folders for the different types of static files such as JavaScript, CSS, Images, library scripts, etc. in the wwwroot folder as shown below.



Controllers

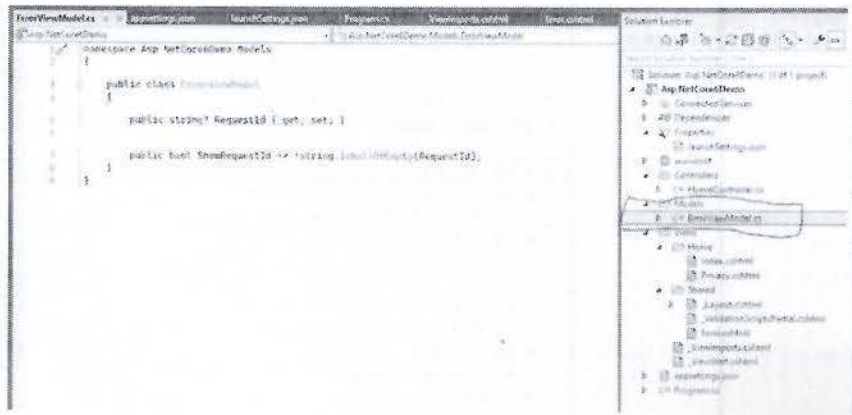
Controller handles all the incoming requests. All the controllers needed for the project are stored here. Controllers are responsible for handling end user interaction, manipulating the model and choose a view to display the UI. Each controller class inherits a Controller class or ControllerBase class. Each controller class

has "Controller" as a suffix on the class name, for example, the default "HomeController.cs" file can be found here. As shown below.



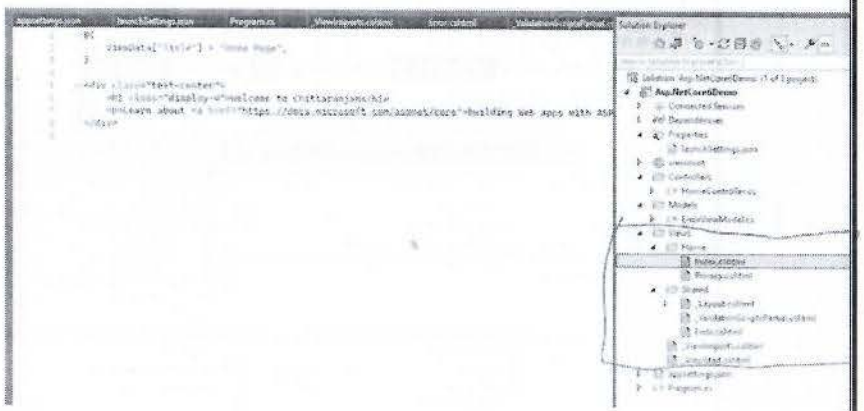
Models

A Model represents the data of the application and a ViewModel represents data that will be displayed in the UI. The models folder contains all the domain or entity classes. Please note user can add folders of his choice to create logical grouping in the project.



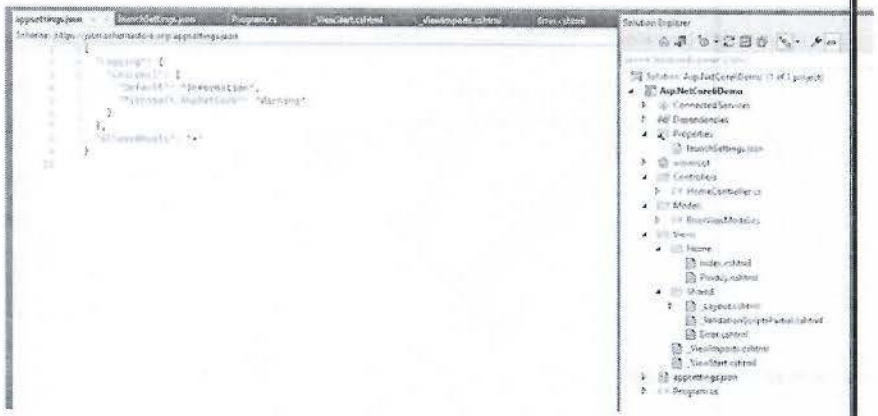
Views

A view represents the user interface that displays ViewModel or Model data and can provide an option to let the user modify them. Mostly a folder in name of the controller is created and all the views related to it are stored in it. Here HomeController related view Index.cshtml and Privacy.cshtml is stored in Home folder and all the shared view across the application is kept in Shared folder. Under this shared folder there are _Layout.cshtml, _ValidationScriptsPartial.cshtml and Error.cshtml view files. There are two more view files, _ViewImports.cshtml and _ViewStart.cshtml under the view folder.



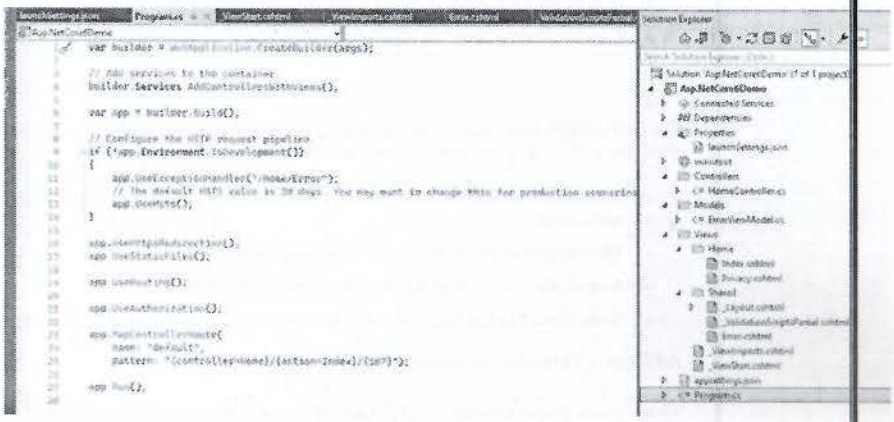
appsettings.json

This file contains the application settings, for example, configuration details like logging details, database connection details.



Program.CS

This class is the entry point of the web application. It builts the host and executes the run method.



Understanding Controllers and Action

What is Controller in asp.net core

A controller is the main pillar in any asp.net core mvc application.

- A controller is a special class with a [redacted] (If using C# programming language) extension.
- By default, All the controllers reside in the Controllers folder.
- In MVC template a controller class is inherited from the Controller class.

Role of Controller in asp.net core

Let's understand how the request work in asp.net core –

When a client sends a request to the server (for asp.net core MVC application) then the request goes to HTTP-Pipeline (middleware) and once it passes through the HTTP-Pipeline then the request hits a controller.

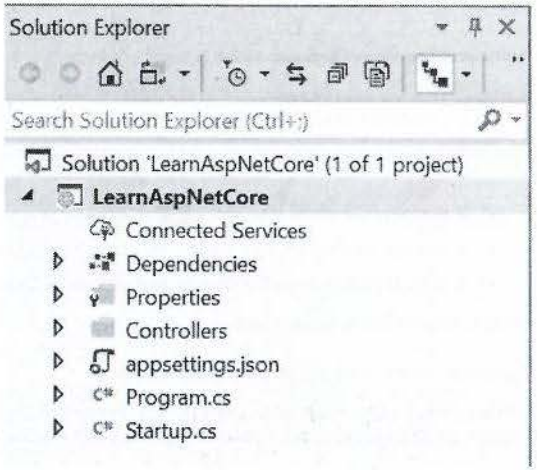
Inside a controller, there might be one or more methods (called action methods). Based on the routing (resource mapping to a URL) the request will hit a particular action method of the controller.

The flow of application depends on the logic written in this action method. The action method may return a view, file, call database, some third-party API, etc.

- A controller is used to define the flow by handling the HTTP-Pipeline of an Asp.Net core MVC application.
- The mapping of HTTP-Request is done using routing.
- A controller is used to group the actions (Action methods).
- Some filters like Caching, Security, etc. can also be applied on the controller level.

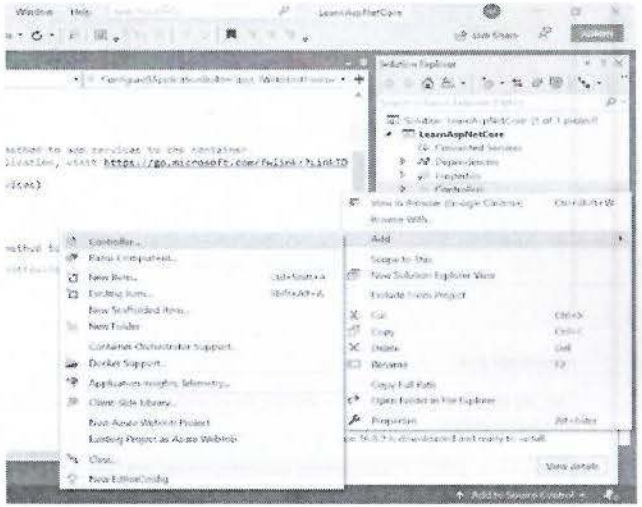
Add a new controller in asp.net core application

To add a new controller using Visual studio first we need to open the solution explorer by clicking on [redacted]



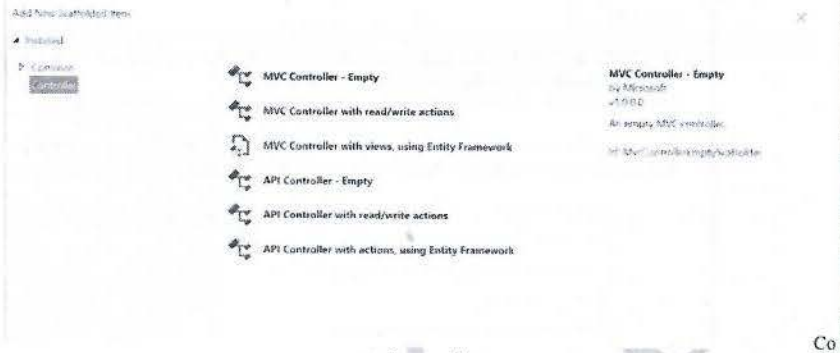
Solution explorer

Now, Right click on the [redacted] folder choose [redacted] and then [redacted]



Add controller in asp.net core

Now you will see following window –



Co

ntroller template

Here we have three options –

- **MVC Controller – Empty**: This will add an empty controller.
- **MVC Controller with read/write actions:** This will generate some action methods required for read and write operations.
- **MVC Controller with views, using entity framework:** This will add a new controller, all basic required action methods, their corresponding views based on the entity framework's context class.

Choose any one of the options and click on [redacted] button.

On the next window enter the name of the controller (Make sure to suffix the name by [redacted] keyword) and click on [redacted] button.

```
using Microsoft.AspNetCore.Mvc;

namespace LearnAspNetCore.Controllers
{
publicclassBookController:Controller
{
publicIActionResultIndex()
{
returnView();
}
}
}
```


- The controller is inherited from [redacted] class.

What is action method in asp.net core?

- All public methods of the controller class are known as the Action Method.
- They are called Action because these action methods are created for a particular action (example: Add employee, edit employee, delete employee, etc.)

How to call an action method in asp.net core

When we get an HTTP call on the controller, we actually are getting it on an action method.

Following is the default pattern to hit the action method.

[redacted]

Pass parameter in action method

The default URL pattern for asp.net core application is –

[redacted]

or

[redacted]

If we need to pass only one value, then we can pass it directly in the URL at the last part.

Example: Let's say we have an action method that returns the details of a book based on the book id –

```
publicIActionResultGetBook(int id)
{
// get book details from DB or other resources from DB based on the book id.

returnView();
}
```

We can call this action method using following URL

[redacted]

The value 3 will get assign to the [redacted] parameter.

If you want to pass more than one parameter in the URL then either you need to use routing or use query string.

Pass parameter using query string in asp.net core mvc application

Let's create a new action method with two parameters.

```
publiclActionResultGetBooks(string bookName,string authorName)
{
// get book details from DB or other resources from DB based on the bookName and/or authorName.

returnView();
}
```

To pass the values in this action method we can use the query string –

There is no limitation on the number of query strings until the URL has a valid length.

When should we create a new controller in asp.net core

Diving the asp.net core application into multiple controllers is always a good practice. Let's see when we should create a new controller in asp.net core –

- Whenever we need to define a new group of operations in the application then always use a new controller.
- For example –
- BookController: For all book-related operations like add, delete, update, edit, etc.
- AuthorController: All operations related to the author

Etc.

How many controllers can we have in one application?

The number of controllers depends on the features of the application. But there should be a minimum of one controller to perform the operations.

HttpGet , HttpPost , NoAction Attributes

HttpGet and HttpPost are both the methods of posting client data or form data to the server. HTTP is a HyperText Transfer Protocol that is designed to send and receive the data between client and server using

web pages.

HTTPGET and HTTPPOST attributes encode request parameters as key and value pairs in the HTTP request.

The HttpGet protocol and the HttpPost protocol provide backward compatibility.

Find the below example. You may want two versions of the Search method, one that renders the Search form and the other that handles the request when that form is posted.

```
// GET api/values
[HttpGet]
public IEnumerable<string> Search()
{
    return new string[] { "value1", "value2" };
}
// POST api/values
[HttpPost]
public void Search([FromBody]string value)
{
    //Do actions
}
```

Whenever the POST request for /Values/Search is received, the action invoker creates a list of all methods of the Controller that match the Search action name. In this case, you would end up with a list of two methods. Immediately, the invoker looks at all of the ActionSelectorAttribute instances applied to each method and calls the IsValidForRequest method on each. If each attribute returns true, then the method is considered valid for the current action.

Let us consider one case, when you ask the first method if it can handle a POST request, it will respond with false because it only handles GET requests. The second method responds with true because it can handle the POST request, and it is the one selected to handle the action for post request.

While doing this action, no method is found that meets these criteria, the invoker will call the HandleUnknownAction method on the Controller, supplying the name of the missing action. If more than one action method meeting these criteria is found, an InvalidOperationException is thrown.

What is Action Method in ASP.NET Core MVC?

Actions are the methods in controller class which are responsible for returning the view or Json data. Action will mainly have return type "ActionResult" and it will be invoked from method InvokeAction called by controller. All the public methods inside a controller which respond to the URL are known as Action Methods. When creating an Action Method we must follow these rules. I have divided all the action methods into the following categories:

1. ActionResult
2. RedirectActionResult
3. FileResult
4. Security

```
1.  public ViewResult ViewResult()
2.  {
3.      return View("About","Home");
4.  }
```

ViewComponentResult

ViewComponentResult is an IActionResult which renders a view component to the response. We use view component by calling Component.InvokeAsync in the view. We can use it to return HTML form a view component. If we want to reuse our business logic or refresh the HTML parts of the page that are loaded with view component we can do that using view component.

```
1.  public ViewComponent ViewComponent()
2.  {
3.      return ViewComponent();
4.  }
```

Action results from previous version of ASP.NET MVC that are either renamed or deleted.

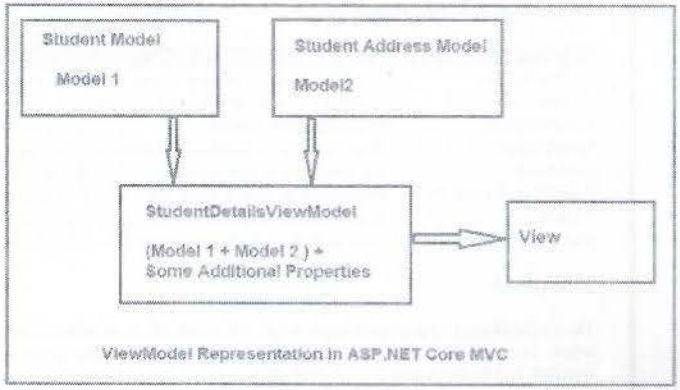
- JavaScriptResult - This action result does not exist anymore we can use ContentResult.
- FilePathResult - We can use VirtualFileResult or PhysicalFileResult instead of FilePathResult
- HttpNotFoundResult - We can use NotFoundResult instead HttpNotFoundResult
- HttpStatusCodeResult - We can use StatusCodeResult instead HttpStatusCodeResult
- HttpUnauthorizedResult - We can use UnauthorizedResult instead HttpUnauthorizedResult

What is a ViewModel in ASP.NET Core MVC?

In real-time applications, a single model object may not contain all the data required for a view. In such situations, we need to use ViewModel in the ASP.NET Core MVC application. So in simple words, we can say that a ViewModel in ASP.NET Core MVC is a model that contains more than one model data required for a particular view. Combining multiple model objects into a single view model object provides us better optimization.

Understanding the ViewModel in ASP.NET Core MVC:

The following diagram shows the visual representation of a view model in the ASP.NET Core MVC application.



Let say we want to display the student details in a view. We have two different models to represent the student data. The Student Model is used to represent the student basic details where the Address model is used to represent the address of the student. Along with the above two models, we also required some static information like page header and page title in the view. If this is our requirement then we need to create a view model let say StudentDetailsViewModel and that view model will contain both the models (Student and Address) as well as properties to store the page title and page header.

Creating the Required Models:

First, create a class file with the name **Student.cs** within the Models folder of your application. This is the model that is going to represent the basic information of a student such as a **name, branch, section**, etc. Once you create the **Student.cs** class file, then copy and paste the following code in it.

```
namespace FirstCoreMVCApplication.Models
{
    public class Student
    {
        public int StudentId { get; set; }
        public string Name { get; set; }
        public string Branch { get; set; }
        public string Section { get; set; }
        public string Gender { get; set; }
    }
}
```

Next, we need to create the Address model which is going to represent the Student Address such as **City, State, Country,** etc. So, create a class file with the name Address.cs within the Models folder and then copy and paste the following code in it.

```
namespace FirstCoreMVCApplication.Models
{
    public class Address
    {
        public int StudentId { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string Country { get; set; }
        public string Pin { get; set; }
    }
}
```

Creating the View Model:

Now we need to create the View Model which will store the required data that is required for a particular view. In our case its student's Details view. This View Model is going to represent the Student Model + Student Address Model + Some additional data like page title and page header.

You can create the View Models anywhere in your application, but it is recommended to create all the View Models within **a folder** called **ViewModels** to keep the things organized.

So first create a folder at the root directory of your application with the name **ViewModels** and then create a class file with the name **StudentDetailsViewModel.cs** within the **ViewModels** folder. Once you create the **StudentDetailsViewModel.cs** class file, then copy and paste the following code in it.

```
using FirstCoreMVCApplication.Models;
namespace FirstCoreMVCApplication.ViewModels
{
    public class StudentDetailsViewModel
    {
        public Student Student { get; set; }
        public Address Address { get; set; }
        public string Title { get; set; }
        public string Header { get; set; }
    }
}
```

We named the **ViewModel** class as **StudentDetailsViewModel**. Here the word **Student** represents the **Controller** name, the word **Details** represent the **action method name** within the Student Controller. As it is a view model so we prefixed the word **ViewModel**. Although it is not mandatory to follow this naming

convention, I personally prefer it to follow this naming convention to organize view models.

Creating Student Controller:

Right-click on the Controllers folder and then add a new class file with the name StudentController.cs and then copy and paste the following code in it.

HTML Helpers

An HTML Helper is just a method that returns a string. The string can represent any type of content that you want. For example, you can use HTML Helpers to render standard HTML tags like HTML `<input>` and `<img>` tags. You also can use HTML Helpers to render more complex content such as a tab strip or an HTML table of database data.

The ASP.NET MVC framework includes the following set of standard HTML Helpers (this is not a complete list):

- Html.ActionLink()
- Html.BeginForm()
- Html.CheckBox()
- Html.DropDownList()
- Html.EndForm()
- Html.Hidden()
- Html.ListBox()
- Html.Password()
- Html.RadioButton()
- Html.TextArea()
- Html.TextBox()

What is ViewBag in ASP.NET Core MVC?

The ViewBag in ASP.NET Core MVC is one of the mechanisms to pass the data from a controller action method to a view. If you go the Controller base class, then you will find the following signature of the ViewBag property.

```
[Dynamic]
public dynamic ViewBag { get; }
```

So the ViewBag is a dynamic property of the Controller base class. The dynamic type is introduced in C# 4.0. It is very much similar to the var keyword that means we can store any type of value in it but the type will be decided at run time rather than compile-time.

The ViewBag transfers the data from the controller action method to a view only, the reverse is not possible.

How to Pass and Retrieve data From ViewBag in ASP.NET Core MVC?

The point that you need to keep in mind is, ViewBag is operating on the dynamic data type. So we don't require typecasting while accessing the data from a ViewBag. It does not matter whether the data that we are accessing is of type string or any complex type.

ViewBag in ASP.NET Core MVC with String Type:

Storing the data in ViewBag

```
ViewBag.Header = "Student Details";
```

Controller

Accessing the Data

```
@ViewBag.Header
```

View

Type Casting is not Required

ViewBag in ASP.NET Core MVC with Complex Type:

```
Student student = new Student()
{
    StudentId = 101,
    Name = "James",
    Branch = "CSE",
    Section = "A",
    Gender = "Male"
};

ViewBag.Student = student;
```

Controller

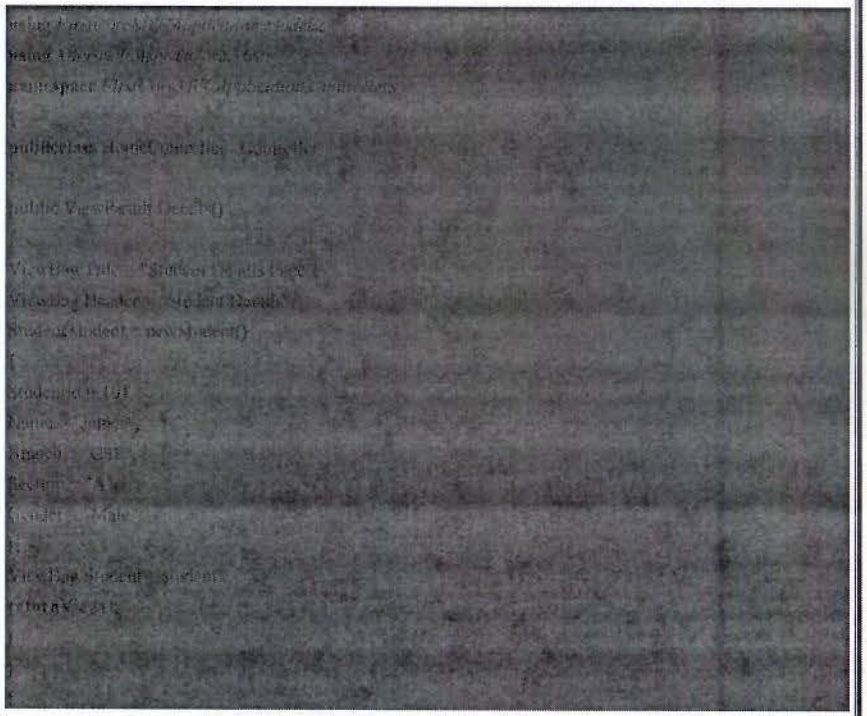
```
@{
    var student = ViewBag.Student;
}
```

Type Casting is not Required even though the data we are accessing is of Complex Type

View

Example of ViewBag in ASP.NET Core MVC:

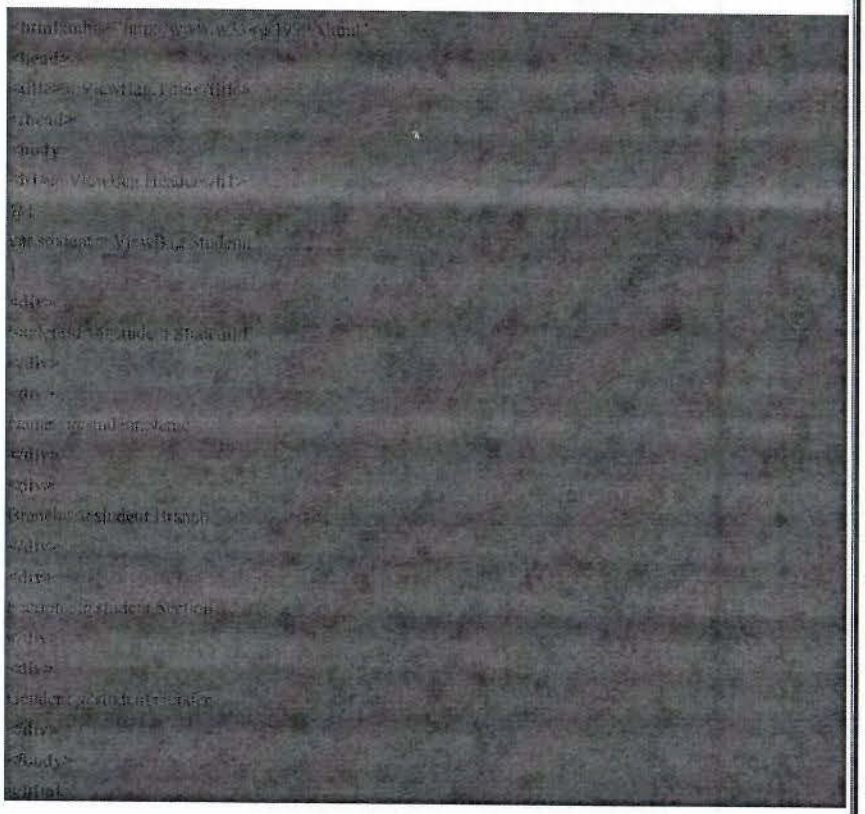
Let us see an example to understand how to use ViewBag to pass data from a controller to a view. We are going to work with the same example that we worked in our previous article with ViewData. So, modify the Details action method of HomeController class as shown below.



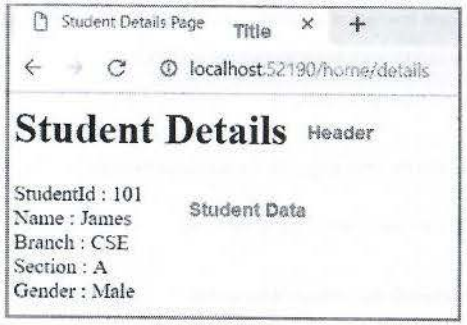
As you can see in the above example, here we are using the dynamic properties Title, Header, and Student on the ViewBag.

Accessing the ViewBag in a View in ASP.NET Core MVC

Now we will see how to access the ViewBag data within an ASP.NET Core MVC view. So, modify the Details.cshtml view file as shown below.



As you can see, here we are accessing the data from the ViewBag using the same dynamic properties Title, Header, and Student. Now run the application and navigate to the "**/Home/Details**" URL and you will see the data as expected on the webpage as shown in the below image.



The ViewBag is a dynamic property that is resolved at runtime; as a result, here also it will not provide compile-time error checking as well as intelligence support. For example, if we miss-spell the property names of the ViewBag, then we wouldn't get any compile-time error rather we came to know about the error at runtime.

Difference between ViewData and ViewBag in ASP.NET Core MVC

1. In ASP.NET Core MVC, we can use both ViewData and ViewBag to pass the data from a Controller action method to a View.
2. The ViewData is a weakly typed dictionary object whereas the ViewBag is a dynamic property. Both ViewData and ViewBag are used to create a loosely typed view in MVC.
3. In ViewData, we use string keys to store and retrieve the data from the ViewData dictionary whereas in ViewBag we use the dynamic properties to store and retrieve data.
4. Both the ViewData keys and ViewBag dynamic properties are resolved only at runtime. As a result, both do not provide compile-time error checking and because of this, we will not get intelligence support.
5. So if we misspell the key names or dynamic property names then we will not get any compile-time error rather we came to know about the error only at run time. This is the reason why we rarely used ViewBag and ViewData in our application.

Data Annotations

Data Annotations are nothing but certain validations that we put in our models to validate the input from the user. ASP.NET MVC provides a unique feature in which we can validate the models using the Data Annotation attribute. Import the following namespace to use data annotations in the application.

System.ComponentModel.DataAnnotations

It is very easy to use and the code becomes much cleaner as compared to normal ASP.NET validators.

Let us understand some of the validator attributes that we can use in MVC.

Types of Data Annotations in ASP.NET MVC

1. **Required**
   This attribute specifies that the value is mandatory and cannot be skipped.

Syntax

[Required(ErrorMessage = "Please enter name"),MaxLength(30)]

2. **DataType**
   This attribute is used to specify the datatype of the model.

Syntax

[DataType(DataType.Text)]

3. **Range**
   Using this attribute we can set a range between two numbers.

Syntax

[Range(100,500,ErrorMessage = "Please enter correct value")]

4. **StringLength**
   Using this attribute we can specify maximum and minimum length of the property.

Syntax

[StringLength(30,ErrorMessage = "Do not enter more than 30 characters")]

5. **DisplayName**
   Using this attribute we can specify property name to be displayed on view.

Syntax

[Display(Name = "Student Name")]

6. **MaxLength**

   Using this attribute we can specify maximum length of property.

Syntax

[MaxLength(3)]

7. **Bind**
   This attribute specifies fields to include or exclude for model binding.

Syntax

[Bind(Exclude = "StudentID")]

8. **DisplayFormat**
   This attribute allows us to set date in the format specified as per the attribute.

Syntax

[DisplayFormat(DataFormatString = "{0:dd.MM.yyyy}")]

9. **RegularExpression**
   We can set a regex pattern for the property. For ex: Email ID.

Syntax

[RegularExpression(@"^\w+([-+.']\w+)@\w+([-.]\w+)*\.\w+([-.]\w+)*$", ErrorMessage = "Email is not valid.")]*

Ex.

```
public class Student

[ScaffoldColumn(false)]
publicintStudentID { get; set; }
[DataType(DataType.Text)]
[Required(ErrorMessage = "Please enter name"), MaxLength(30)]
[Display(Name = "Student Name")]

public string Name { get; set; }
[MaxLength(3),MinLength(1)]
[Required(ErrorMessage = "Please enter marks")]

publicint Marks { get; set; }
[DataType(DataType.EmailAddress)]
[Required(ErrorMessage = "Please enter Email ID")]
[RegularExpression(@"^\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$", ErrorMessage = "Email is not
valid.")]

public string Email { get; set; }
[Required(ErrorMessage = "Please enter department")]

public string Department { get; set; }
[Required(ErrorMessage = "Please enter Mobile No")]
[Display(Name = "Contact Number")]
```

```
[DataType(DataType.PhoneNumber)]
```

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

```
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

1. **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 TestTempData()
    {
        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>
```

2. 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();
}
```


3. 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

```
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>
```

ViewData

It helps us to maintain your data when sending the data from Controller to View. It is a dictionary object and derived from ViewDataDictionary. As it is a dictionary object, it takes the data in a key-value pair.

Once you create ViewData object, pass the value, and make redirection; the value will become null. The data of ViewData is not valid for next subsequent request. Take care of two things when using ViewData, first, check for null and second, check for typecasting for complex data types.

```
1.  public ActionResult Index()
2.  {
3.      Employee emp = new Employee()
4.      {
5.          Id = 1001,
6.          Name = "Mukesh Kumar",
7.          Address = "New Delhi",
8.          Age = 25
9.      };
10.
11.     ViewData["Message"] = "This is ViewData";
12.     ViewData["Emp"] = emp;
13.
14.     return View();
15. }
```

The above code contains two ViewData dictionary objects - ViewData["Message"] and ViewData["Emp"]. The first one is a simple string value but the next one contains complex employee data. When the View is going to render, we need to first check the ViewData for null and if it is not, then we can get the value.

```
1.  @{
2.      ViewBag.Title = "Home Page";
3.  }
4.
5.  <div class="row">
6.      <div class="col-md-4">
7.          <h2>Employee Details</h2>
8.          <br />
9.          <p>
10.             @if(ViewData["Message"] !=null)
11.             {
12.                 <b>@ViewData["Message"].ToString();</b>
13.             }
14.         </p>
15.         <br />
```



```
16.         @if (ViewData["Emp"] != null)
17.         {
18.
19.             var emp = (MVCStateManagement.Models.Employee)ViewData["Emp"];
20.             <table>
21.                 <tr>
22.                     <td>
23.                         Name :
24.                     </td>
25.                     <td>
26.                         @emp.Name
27.                     </td>
28.                 </tr>
29.                 <tr>
30.                     <td>
31.                         Address :
32.                     </td>
33.                     <td>
34.                         @emp.Address
35.                     </td>
36.                 </tr>
37.                 <tr>
38.                     <td>
39.                         Age :
40.                     </td>
41.                     <td>
42.                         @emp.Age
43.                     </td>
44.                 </tr>
45.             </table>
46.         }
47.     </div>
48. </div>
```

ViewBag

The ViewBag's task is same as that of ViewData. It is also used to transfer the data from Controller to View. However, the only difference is that ViewBag is an object of Dynamic property introduced

in C# 4.a. It is a wrapper around ViewData. If you use ViewBag rather than ViewData, you will not have to do typecasting with the complex objects and do not need to check for null.

If we consider the same above code with ViewBag, the output will be same.

In Controller

```
public ActionResult Index()
{
    Employee emp = new Employee()
    {
        Id = 1001,
        Name = "Mukesh Kumar",
        Address = "New Delhi",
        Age = 25
    };

    ViewBag.Message = "This is ViewBag";
    ViewBag.Emp = emp;

    return View();
}
```

In View


```
@{
    ViewBag.Title = "Home Page";
}

<div class="row">
    <div class='col-md-4'>
        <h2>Employee Details</h2>
        <br />
        <p>

            <b>@ViewBag.Message</b>

        </p>
        <br />
        @{
            var emp = ViewBag.Emp;
            <table>
                <tr>
                    <td>
                        Name :
                    </td>
                    <td>
                        @emp.Name
                    </td>
                </tr>
                <tr>
                    <td>
                        Address :
                    </td>
                    <td>
                        @emp.Address
                    </td>
                </tr>
                <tr>
                    <td>
                        Age :
                    </td>
                    <td>
                        @emp.Age
                    </td>
                </tr>
            </table>
        }
    </div>
</div>
```

Server-Side State Management

1. **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

```
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"]))
    }
```

2. **Application State :**

is stored in the memory of the server and is faster than storing and retrieving information in a database.

- Session sate 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

Partial views in ASP.NET Core

A partial view is a Razor markup file (`.cshtml`) without an `@page` directive that renders HTML output *within* another markup file's rendered output.
The term *partial view* is used when developing either an MVC app, where markup files are called *views*, or a Razor Pages app, where markup files are called *pages*. This topic generically refers to MVC views and Razor Pages pages as *markup files*.

When to use partial views

- Break up large markup files into smaller components.

  In a large, complex markup file composed of several logical pieces, there's an advantage to working with each piece isolated into a partial view. The code in the markup file is manageable because the markup only contains the overall page structure and references to partial views.

- Reduce the duplication of common markup content across markup files.

  When the same markup elements are used across markup files, a partial view removes the duplication of markup content into one partial view file. When the markup is changed in the partial view, it updates the rendered output of the markup files that use the partial view.

  Partial views shouldn't be used to maintain common layout elements. Common layout elements should be specified in _Layout.cshtml files.

  Don't use a partial view where complex rendering logic or code execution is required to render the markup.

Data Management with ADO.NET

Connection Object

The Connection object is the first component of ADO.NET that you should be looking at. A connection sets a link between a data source and ADO.NET. A Connection object sits between a data source and a DataAdapter (via Command). You need to define a data provider and a data source when you create a connection. With these two, you can also specify the user ID and password depending on the type of data source. Figure 3-3 shows the relationship between a connection, a data source, and a data adapter.

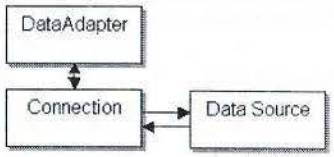


Figure 1: The relationship between connection, data adapter, and a data source

Connection can also be connected to a Command object to execute SQL queries, which can be used to retrieve, add, update, and delete data to a data source. Figure 2 shows the relationship between the Command and Connection objects.

Figure 2: The relationship between the command object and the connection object

The Connection object also plays a useful role in creating a transaction. Transactions are stored in transaction objects, and transaction classes have all those nice features for dealing with transactions such as commit and rollback. Figure 3 shows the relationship between the connection object and the transaction.

Figure 3: Creating a transaction from a connection object

Each data provider has a Connection class. Below Table shows the name of various connection classes for data providers.

Command Object

The command object is one of the basic components of ADO .NET.

1. The Command Object uses the connection object to execute SQL queries.
2. The queries can be in the Form of Inline text, Stored Procedures or direct Table access.
3. An important feature of Command object is that it can be used to execute queries and Stored Procedures with Parameters.
4. If a select query is issued, the result set it returns is usually stored in either a DataSet or a DataReader object.

Associated Properties of SqlCommand class

The properties associated with SqlCommand class are shown in the Table below.

Property	Type of Access	Description
Connection	Read/Write	The SqlConnection object that is used by the command object to execute SQL queries or Stored Procedure.
CommandText	Read/Write	Represents the T-SQL Statement or the name of the Stored Procedure.
CommandType	Read/Write	This property indicates how the CommandText property should be interpreted. The possible values are: 1. Text (T-SQL Statement) 2. StoredProcedure (Stored Procedure Name) 3. TableDirect
CommandTimeout	Read/Write	This property indicates the time to wait when executing a particular command.

	Default Time for Execution of Command is 30 Seconds. The Command is aborted after it times out and an exception is thrown.

Now, Let us have a look at various Execute Methods that can be called from a Command Object.

Property	Description
ExecuteNonQuery	This method executes the command specifies and returns the number of rows affected.
ExecuteReader	The ExecuteReader method executes the command specified and returns an instance of SqlDataReader class.
ExecuteScalar	This method executes the command specified and returns the first column of first row of the result set. The remaining rows and column are ignored.
ExecuteXMLReader	This method executes the command specified and returns an instance of XmlReader class. This method can be used to return the result set in the form of an XML document

DataReader

ADO.NET DataReader object is used for accessing data from the data store and is one of the two mechanisms that ADO.NET provides.
As we will remember DataReader object provides a read only, forward only, high performance mechanism to retrieve data from a data store as a data stream, while staying connected with the data source.
The DataReader is restricted but highly optimized. The .NET framework provides data providers for SQL Server native OLE DB providers and native ODBC drivers,

DataAdapter

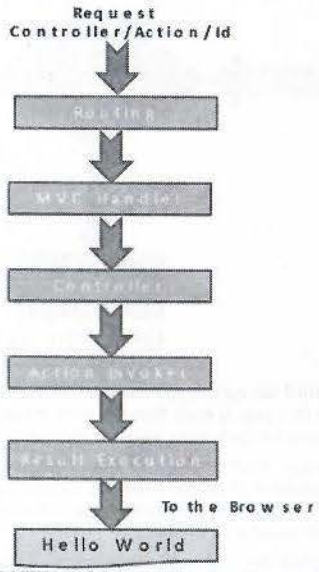
The DataAdapter works as a bridge between a DataSet and a data source to retrieve data. DataAdapter is a class that represents a set of SQL commands and a database connection. It can be used to fill the DataSet and update the data source.

Routing Engine & Routing Table

ASP.NET MVC Request Life Cycle

The URL in MVC Application generally follows the Pattern **Controller/Action/id.** This URL is automatically mapped to Action Method inside the MVC Controller. The job of Action Method is to

create the Response and send it back to the client. The Image below shows the various stages that a request goes through before the results are returned to the client.

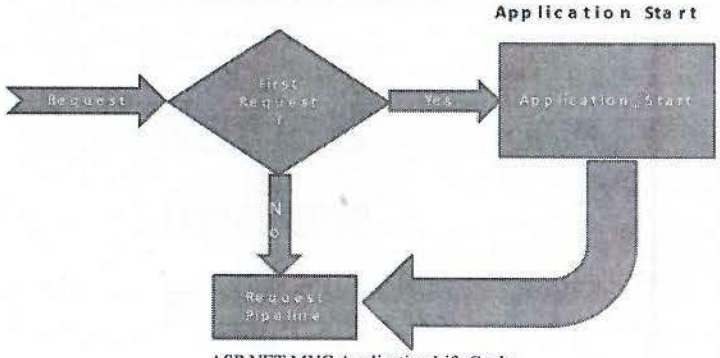


ASP.NET MVC Request Pipe line / Request life cycle

The **ASP.NET MVC Request life cycle** starts when the client requests for a resource. The request received from the client first goes through the *Routing module*. Routing Module then passes it to the *MVC Handler*. MVC Handler is responsible for creating the *Controller*. The Controller then invokes the action method using the *Action Invoker*. Finally, the *Result Execution* takes place and the response is sent to the Clients Browser.

let us look at each of these components in detail.

Application Life Cycle



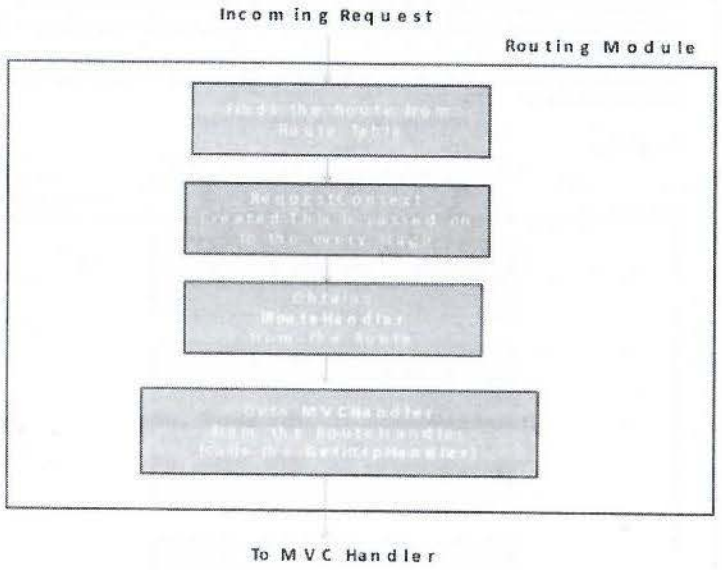
ASP.NET MVC Application Life Cycle

Application life cycle is different from the Request Life Cycle. Request Life Cycle starts when the Request for a page is made from the client and ends when the response is sent to the client. Application life cycle starts when the first request is made to the Application.

Application_Start event is fired when the **first request is made**. You will find the **Application_Start** event in global.asax.cs file. **Application_Start** event is used to set up the initial configuration of the Application. It fills up the route table with routes which specify the which controller action to be invoked.

The **Application_End** event fires when the Web server recycles the application or after a certain period of inactivity or when the CPU/memory thresholds are exceeded. Once the Application_End is fired the next request coming in is treated as the first request and **Application_Start** event is fired again.

Routing Module

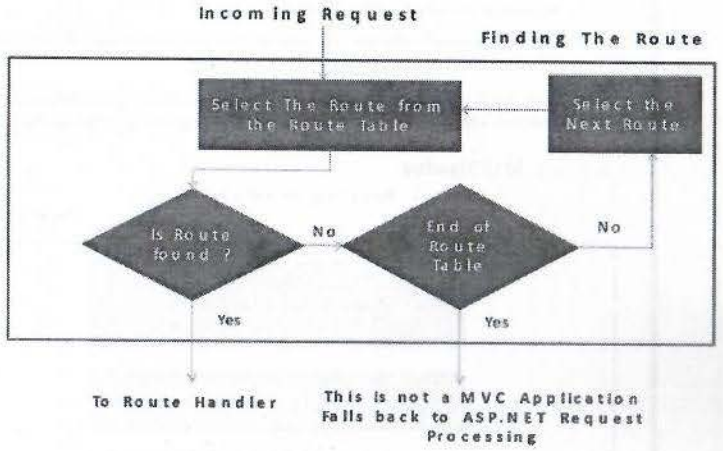


Routing Module in MVC Request life Cycle

The Purpose of the Routing Module map the incoming URL to the Route and then get the HttpHandler to Service the Request.

Routing is the first module in ASP.NET MVC Request life cycle that receives the request. It is handled by UrlRoutingModule module which is an HTTP Module. The UrlRoutingModule also creates **RequestContext** (HttpContext) and passes it to the next layer.

Finding the Route



Finding the Route

The UrlRoutingModule parses each request and makes the route selection. It loops through the RouteData from a **RouteTable** and selects the first route which matches the current URL. If no routes found then the UrlRoutingModule sends the request to regular ASP.NET or IIS request processing.

You can read about Routing in MVC from this link

Route Handler

The purpose of a route handler is to create the HTTP handler object which that will serve the requested URL

Route handler is attached to the every route that we add to the routes collection. Routes are added using the function **MapRoute.**

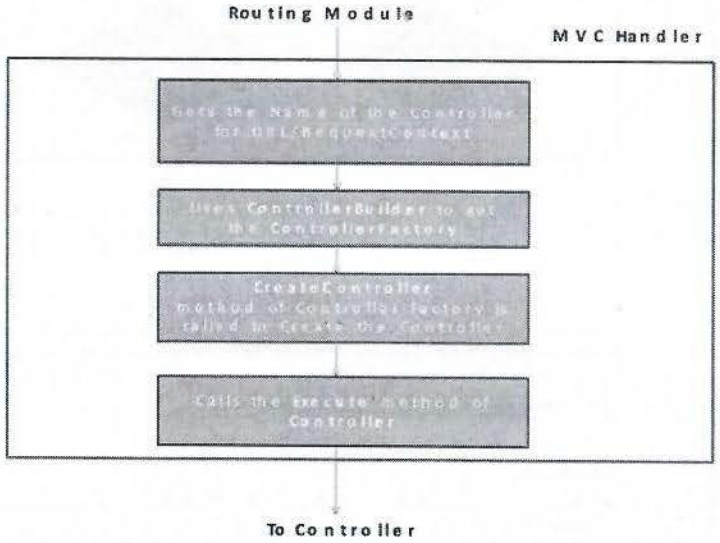
```
1
2   Routes.MapRoute(
3       name:"Default",
4       url:"{controller}/{action}/{id}",
5       defaults:new{controller="Home",action="Index",id=UrlParameter.Optional}
6       );
7
```

The MapRoute is an extension method. This method adds the default Route Handler **MvcRouteHandler.** A route handler is something that must implement **IRouteHandler** interface.

MVC Allows us to add our own custom route Handler instead of using the Default Route Handler.

Once the RouteHandler is obtained from the RouteData, the **GetHandler** method of RouteHandler is executed which will Create **MVCHandler** corresponding to the RouteHandler

MVCHandler



MVC Handler in MVC Request life Cycle

The Purpose of the MvcHandler is to Create the Controller Instance, that Process the current request.

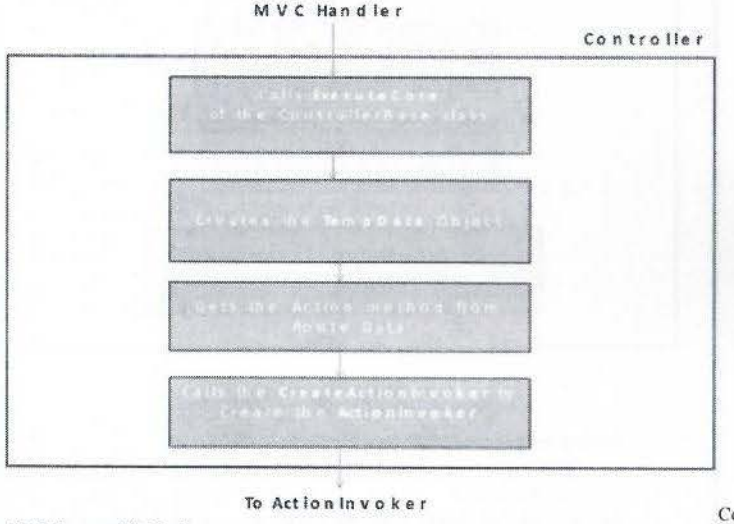
MVC handler is responsible for initiating MVC applications. This is where the actual MVC Application begins. It takes **RequestContext** as its parameter. MVCHandler is a class that implements the **IHttpHandler** interface. It Operates in both sync and async, depending on its caller.

MVCHandler inspects the RequestContext and gets the name of the Controller from the URL. Then it uses the **ControllerBuilder** for **ControllerFactory** instance. It then passes the name of the Controller and

RequestContext to the **CreateController.** CreateController creates the controller from the **ControllerFactory**. Finally, it calls the controller's **Execute** method and passes the **RequestContext** to Controller.

By default MVC creates a **DefaultControllerFactory**. We can also create our own **ControllerFactory** to override the behavior.

Controller



Controller in MVC Request life Cycle

The Purpose of the Controller is to Call the ControllerActionInvoker to deal with the request

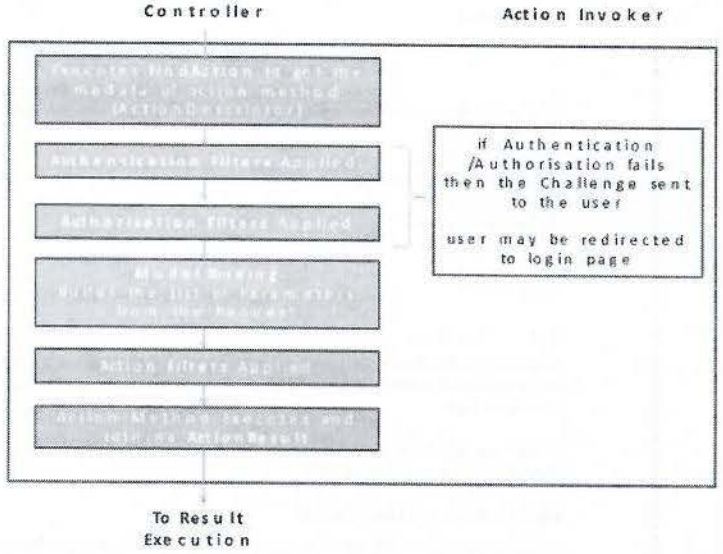
The **Controller** class implements the **IController** interface. It also inherits from the **ControllerBase**. The Controller exposes the method **Execute** which gets the RequestContext as its parameter.

The Execute method creates the **TempData** object. Then it calls the **ExecuteCore** of the **ControllerBase** class. ExecuteCore then finds out the Action Name from the URL (Using RequestContext).

Finally, ExecuteCore method calls **CreateActionInvoker** to Create the Action Invoker

We can create our own version of the controller by overriding the ExecuteCore () method of ControllerBase

Action Invoker



Action Invoker in MVC Request life Cycle

The Role of Action Invokers is to execute action Method passed to to it by the controller.

ActionInvoker implements the **IActionInvoker** interface. The interface has only one method called **InvokeAction**. The Default Implementation is *ControllerActionInvoker*

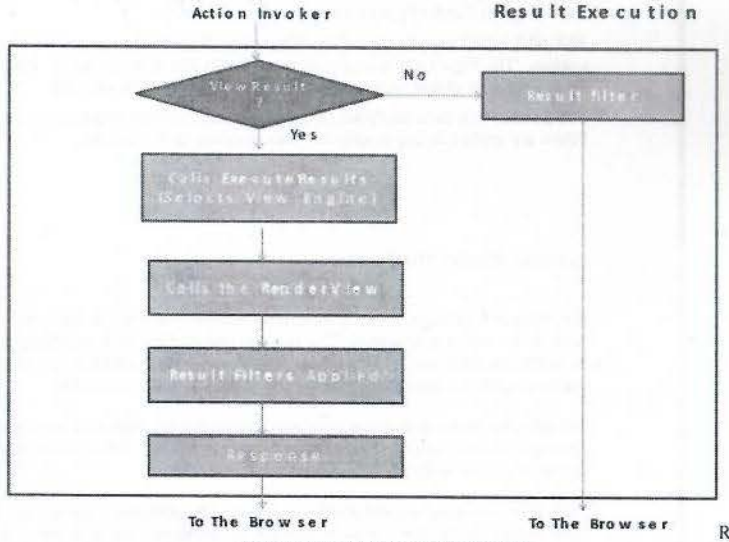
ControllerActionInvoker uses **FindAction** to find the information about the Action method to be executed. Then it builds the list of Parameter from the ControllerDescriptor and ActionDescriptor. ControllerDescriptor and ActionDescriptor provide the metadata about the controller and the action method like ControllerName, ControllerType, ActionName, Filters and Parameters etc.

ActionInvoker calls Authentication filters to verify to see if the user is authenticated and then it invokes the authorization filters to check to see if the user is authorized

ActionInvoker then calls the Model Binder object to map the parameters from the form data. These parameters are passed to the Action Method. This Process is called **Model Binding.**

ControllerActionInvoker then calls **InvokeAction** method to execute the action method. The action method returns the **ActionResult**

Result Execution



Execution in MVC Request life Cycle

The Purpose of View Engine to render the view as a web page.

ActionInvoker in the previous stage executes the Action Method in the controller and returns the ActionResult.

ActionResult is is an abstract class. There are many types of ActionResults can be returned by the ActionMethod. For example ViewResult, ContentResult, JsonResult, FileResult, RedirectToRouteResult, RedirectResult, , and EmptyResult. Etc. ViewResult is one of the common Return type, which returns an HTML page to the browser

Each of these ActionResults implements their own ExecuteResults method. So the ExecuteResult of the method of the ActionResult is called.

ViewResult is one of the most common return types used. ViewResults renders the HTML view from the ViewResult. When ExecuteResult is called on ViewResult calls the RenderView Method of the ViewResult. View Engines must implement the IViewEngine interface

The View Engine takes over when the **RenderView** method is executed. This will call the Current View Engines RenderView method

ASP.NET MVC uses the **RazorViewEngine** by default. We can possibly use the alternative view engines. The View Engine used must be configured in **Global.asax.cs**. You don't need to Register the View Engine in **global.asax.cs** file if you are using **RazorViewEngine**

The RenderView calls the ProcessRequest to create the final response to be sent to the client. Result Filters are applied before sending the final response to the browser

Layouts , Bundle , Minification

Bundling is a technique that combines multiple files into a single file to reduce the number of requests made to the server to download files. Bundles mainly for CSS & JavaScript can be created. Fewer .css & .js files means fewer HTTP requests from browser for these files. Any number of individual bundles can be created but again these bundles also should be small in number.

Minification removes unnecessary characters, comments, spaces & new lines from code without altering core functionality. This reduces the size of mainly CSS & js files significantly. Reduced size means less amount of data to be downloaded over HTTP.

ASP.NET Core bundling and minification should be generally enabled only in the production environment and for the non-production environments used original files which makes it easier to debug.

MVC Security

Using Authorize & Allow Anonymous attributes

Using [AllowAnonymous] to allow access for all users

Firstly, add a reference to the Microsoft.AspNetCore.Authorization package to the top of the controller. You may need to download the nuget package first.

using Microsoft.AspNetCore.Authorization;

In the standard ASP.NET Core 3 MVC template, the method for the index page looks like this before we make any changes;

```
public IActionResult Index()
{
return View();
}
```

We can add the [AllowAnonymous] attribute to the HomeController class **OR** the Index method.

```
[AllowAnonymous]
publicclassHomeController : Controller
{
 [AllowAnonymous]
public IActionResult Index()
  {
return View();
  }
}
```

Applying the attribute to the class will grant non-authenticated users the ability to access any of the pages within the controller, while applying the attribute to the method will only grant access to the individual page.

You'd typically use this attribute on the Index, Login or Registration pages of your website, however it depends on your specific use case.

Using [Authorize] to restrict access

[Authorize] works in the opposite way to the [AllowAnonymous] attribute.

We might not have any Identity redirects setup in the startup.cs file, but we want to restrict access to a specific page to authenticated users only!

In the same way we use the [AllowAnonymous] attribute, just apply the [Authorize] attribute on the controller or view.

For example;

```
[Authorize]
publicclassHomeController : Controller
{
 [Authorize]
public IActionResult Index()
 {
return View();
 }
}
```


Configure security settings in the Web.config File

This section demonstrates how to add and modify the `<authentication>` and `<authorization>` configuration sections to configure the ASP.NET application to use forms-based authentication.

1. In Solution Explorer, open the *Web.config* file.
2. Change the authentication mode to **Forms**.
3. Insert the `<Forms>` tag, and fill the appropriate attributes. Copy the following code, and then select **Paste as HTML** on the **Edit** menu to paste the code in the `<authentication>` section of the file:

```
XMLCopy
<authenticationmode="Forms">
<formsname=".ASPXFORMSDEMO"loginUrl="logon.aspx"
protection="All"path="/"timeout="30" />
</authentication>
```

4. Deny access to the anonymous user in the `<authorization>` section as follows:

```
XMLCopy
<authorization>
<denyusers ="?" />
<allowusers = "*" />
</authorization>
```

Create a sample database table to store users details

This section shows how to create a sample database to store the user name, password, and role for the users. You need the role column if you want to store user roles in the database and implement role-based security.

1. On the **Start** menu, select **Run**, and then type notepad to open Notepad.
2. Highlight the following SQL script code, right-click the code, and then select **Copy**. In Notepad, select **Paste** on the **Edit** menu to paste the following code:

```
SQLCopy
if exists (select * from sysobjects whereid =
object_id(N'[dbo].[Users]') and OBJECTPROPERTY(id, N'IsUserTable') = 1)
droptable [dbo].[Users]
GO
CREATETABLE [dbo].[Users] ([uname] [varchar] (15) NOTNULL,
```

```
    [Pwd] [varchar] (25) NOTNULL,
    [userRole] [varchar] (25) NOTNULL,
) ON [PRIMARY]
GO
ALTERTABLE [dbo].[Users] WITHNOCHECKADD
CONSTRAINT [PK_Users] PRIMARY KEY NONCLUSTERED
    ([uname]
    ) ON [PRIMARY]
GO

INSERTINTOUsersvalues('user1','user1','Manager')
INSERTINTOUsersvalues('user2','user2','Admin')
INSERTINTOUsersvalues('user3','user3','User')
GO
```

3. Save the file as *Users.sql*.
4. On the SQL Server computer, open *Users.sql* in Query Analyzer. From the list of databases, select **pubs**, and run the script. This operation creates a sample users table and populates the table in the Pubs database to be used with this sample application.

Create a Logon.aspx page

1. Add a new Web Form to the project named *Logon.aspx*.
2. Open the Logon.aspx page in the editor, and switch to HTML view.
3. Copy the following code, and use the **Paste as HTML** option on the **Edit** menu to insert the code between the `<form>` tags:

```
HTMLCopy
<h3>
<fontface="Verdana">Logon Page</font>
</h3>
<table>
<tr>
<td>Email:</td>
<td><inputid="txtUserName"type="text"runat="server"></td>
<td><ASP:RequiredFieldValidatorControlToValidate="txtUserName"
Display="Static"ErrorMessage="*"runat="server"
ID="vUserName" /></td>
</tr>
<tr>
<td>Password:</td>
<td><inputid="txtUserPass"type="password"runat="server"></td>
<td><ASP:RequiredFieldValidatorControlToValidate="txtUserPass"
Display="Static"ErrorMessage="*"runat="server"
ID="vUserPass" />
```



```
</td>
</tr>
<tr>
<td>Persistent Cookie:</td>
<td><ASP:CheckBoxid="chkPersistCookie"runat="server"autopostback="false" /></td>
<td></td>
</tr>
</table>
<inputtype="submit"Value="Logon"runat="server"ID="cmdLogin"><p></p>
<asp:Labelid="lblMsg"ForeColor="red"Font-Name="Verdana"Font-Size="10"runat="server" />
```

This Web Form is used to present a logon form to users so that they can provide their user name and password to log on to the application.

4. Switch to Design view, and save the page.

Code the event handler so that it validates the user credentials

This section presents the code that is placed in the code-behind page (Logon.aspx.cs).

1. Double-click **Logon** to open the *Logon.aspx.cs* file.
2. Import the required namespaces in the code-behind file:

```
C#Copy
using System.Data.SqlClient;
using System.Web.Security;
```

3. Create a `ValidateUser` function to validate the user credentials by looking in the database. Make sure that you change the `Connection` string to point to your database.

```
C#Copy
privatebool ValidateUser(string userName, string passWord )
{
  SqlConnection conn;
  SqlCommand cmd;
string lookupPassword = null;

// Check for invalid userName.
// userName must not be null and must be between 1 and 15 characters.
if ( ( null == userName ) || ( 0 == userName.Length ) || ( userName.Length >15 ))
  {
    System.Diagnostics.Trace.WriteLine( "[ValidateUser] Input validation of userName failed." );
returnfalse;
  }
```

```
// Check for invalid passWord.
// passWord must not be null and must be between 1 and 25 characters.
if ( ( null == passWord ) || ( 0 == passWord.Length ) || ( passWord.Length >25 ))
   {
      System.Diagnostics.Trace.WriteLine( "[ValidateUser] Input validation of passWord failed." );
returnfalse;
   }

try
   {
// Consult with your SQL Server administrator for an appropriate connection
// string to use to connect to your local SQL Server.
      conn = new SqlConnection( "server=localhost;Integrated Security=SSPI;database=pubs" );
      conn.Open();

// Create SqlCommand to select pwd field from users table given supplied userName.
      cmd = new SqlCommand( "Select pwd from users where uname=@userName", conn );
      cmd.Parameters.Add( "@userName", SqlDbType.VarChar, 25 );
      cmd.Parameters["@userName"].Value = userName;

// Execute command and fetch pwd field into lookupPassword string.
      lookupPassword = (string) cmd.ExecuteScalar();

// Cleanup command and connection objects.
      cmd.Dispose();
      conn.Dispose();
   }
   catch ( Exception ex )
   {
// Add error handling here for debugging.
// This error message should not be sent back to the caller.
      System.Diagnostics.Trace.WriteLine( "[ValidateUser] Exception " + ex.Message );
   }

// If no password found, return false.
if ( null == lookupPassword )
   {
// You could write failed login attempts here to event log for additional security.
returnfalse;
   }

// Compare lookupPassword and input passWord, using a case-sensitive comparison.
return ( 0 == string.Compare( lookupPassword, passWord, false ));
}
```


4. You can use one of two methods to generate the forms authentication cookie and redirect the user to an appropriate page in the `cmdLogin_ServerClick` event. Sample code is provided for both scenarios. Use either of them according to your requirement.

   - Call the `RedirectFromLoginPage` method to automatically generate the forms authentication cookie and redirect the user to an appropriate page in the `cmdLogin_ServerClick` event:

```
C#Copy
privatevoidcmdLogin_ServerClick(object sender, System.EventArgs e)
{
if (ValidateUser(txtUserName.Value,txtUserPass.Value))
      FormsAuthentication.RedirectFromLoginPage(txtUserName.Value,
chkPersistCookie.Checked);
else
      Response.Redirect("logon.aspx", true);
}
```

   - Generate the authentication ticket, encrypt it, create a cookie, add it to the response, and redirect the user. This operation gives you more control in how you create the cookie. You can also include custom data along with the `FormsAuthenticationTicket` in this case.

```
C#Copy
privatevoidcmdLogin_ServerClick(object sender, System.EventArgs e)
{
if (ValidateUser(txtUserName.Value,txtUserPass.Value))
   {
      FormsAuthenticationTicket tkt;
string cookiestr;
      HttpCookie ck;
      tkt = new FormsAuthenticationTicket(1, txtUserName.Value, DateTime.Now,
      DateTime.Now.AddMinutes(30), chkPersistCookie.Checked, "your custom data");
      cookiestr = FormsAuthentication.Encrypt(tkt);
      ck = new HttpCookie(FormsAuthentication.FormsCookieName, cookiestr);
if (chkPersistCookie.Checked)
         ck.Expires=tkt.Expiration;
      ck.Path = FormsAuthentication.FormsCookiePath;
      Response.Cookies.Add(ck);

string strRedirect;
      strRedirect = Request["ReturnUrl"];
if (strRedirect==null)
         strRedirect = "default.aspx";
      Response.Redirect(strRedirect, true);
   }
else
      Response.Redirect("logon.aspx", true);
```

```
}
```

Make sure that the following code is added to the `InitializeComponent` method in the code that the Web Form Designer generates:

```
C#Copy
this.cmdLogin.ServerClick += new System.EventHandler(this.cmdLogin_ServerClick);
```

Create a Default.aspx page

This section creates a test page to which users are redirected after they authenticate. If users browse to this page without first logging on to the application, they're redirected to the logon page.

1. Rename the existing *WebForm1.aspx* page as *Default.aspx*, and open it in the editor.
2. Switch to HTML view, and copy the following code between the `<form>` tags:

```
HTMLCopy
<inputtype="submit"Value="SignOut"runat="server"id="cmdSignOut">
```

This button is used to log off from the forms authentication session.

3. Switch to Design view, and save the page.
4. Import the required namespaces in the code-behind file:

```
C#Copy
using System.Web.Security;
```

5. Double-click **SignOut** to open the code-behind page (Default.aspx.cs), and copy the following code in the `cmdSignOut_ServerClick` event handler:

```
C#Copy
privatevoidcmdSignOut_ServerClick(object sender, System.EventArgs e)
{
  FormsAuthentication.SignOut();
  Response.Redirect("logon.aspx", true);
}
```

6. Make sure that the following code is added to the `InitializeComponent` method in the code that the Web Form Designer generates:

```
C#Copy
this.cmdSignOut.ServerClick += new System.EventHandler(this.cmdSignOut_ServerClick);
```

7. Save and compile the project. You can now use the application.

Additional notes

- You may want to store passwords securely in a database. You can use the `FormsAuthentication` class utility function named `HashPasswordForStoringInConfigFile` to encrypt the passwords before you store them in the database or configuration file.
- You may want to store the SQL connection information in the configuration file (*Web.config*) so that you can easily modify it if necessary.
- You may consider adding code to prevent hackers who try to use different combinations of passwords from logging on. For example, you can include logic that accepts only two or three logon attempts. If users can't log on in some attempts, you may want to set a flag in the database to not allow them to log on until the users re-enable their accounts by visiting a different page or by calling your support line. Also, you should add appropriate error handling wherever necessary.
- Because the user is identified based on the authentication cookie, you may want to use Secure Sockets Layer (SSL) on this application so that no one can deceive the authentication cookie and any other valuable information that is being transmitted.
- Forms-based authentication requires that your client accept or enable cookies on their browser.
- The timeout parameter of the `<authentication>` configuration section controls the interval at which the authentication cookie is regenerated. You can choose a value that provides better performance and security.
- Certain intermediary proxies and caches on the Internet may cache Web server responses that contain `Set-Cookie` headers, which are then returned to a different user. Because forms-based authentication uses a cookie to authenticate users, this behavior can cause users to accidentally (or intentionally) impersonate another user by receiving a cookie from an intermediary proxy or cache that wasn't originally intended for them.

What is Entity Framework Core?

However complex or simple your application, it is hard to not notice the vital part that a Database plays in it. I would call a Database / Datasource the heart of any application. So, we will need an efficient way to read and securely write clean data into our data source. You may think of traditional SQL Queries and ADO.NET approaches to achieve the task. While it is still a good option, You are facing the risk of maintainability and speed of development with the traditional SQL approaches.

Entity Framework Core is a lightweight, extensible, open-sourced version of the Entity Framework Data Access Technology built for .NET Core Applications. It is essentially a way to read and write data in a flexible and easier way. EFCore is an ORM (Object Relational Mapper) built over ADO.NET that enables developers to access & store data much more efficiently without compromising on the performance (Unless you write real poor code!). With this technology, you no longer will interact directly with the database, like you used to do with traditional SQL queries and SPs.

What can I do with Entity Framework Core?

These are a few of the features of Entity Framework Core

- Manage database connection
- Configure models
- Querying data
- Saving data
- Configure change tracking

Supported Database Providers

EFCore is intended for flexibility and rapid development. With that being said, EF supports a good like of database providers like SQL Server, MySQL, SQLite, In-memory, and PostgreSQL. For each of these providers, Microsoft has an extension to be installed from the NuGet package manager. The real flexibility this wide range of support provides is that you will not have to depend on the Database Provider ever again.

One fine day, your client wants to migrate to MySQL from SQL Server (I don't know why in the world anyone will do that, but taking as an example), It will be as easy to replace the connection string, installing the required package, and committing the migrations. **You would NOT have to change your code even a bit**, as EFCore takes care of data access. This is how you build decoupled applications the right way.

Various Development Approaches

There are 2 approaches while developing with Entity Framework Core in ASP.NET Core application namely. Code-First Approach and the Database-First Approach. It is quite obvious from the names.

With Code First Approach, you have the possibility to generate Database based on the Model classes and Rules applied while applying the migrations. In simple words, you could like a Student.cs class with the required properties, hook it up with a context class, and run some commands on your console BOOM! you have your database ready, based on Student.cs

As the name suggests, DB First approach is used when you already have a defined database with existing data. What EF Core does in this scenario is quite interesting. In the previous approach, we generated Tables from Model Classes, while using the DB First Approach we can generate Model classes from our existing Database Schema. Pretty cool, yeah?

When Should I use Entity Framework Core?

You can go ahead with Entity Framework Core

- if your application consists mostly of CRUD Operations only (Create, Read, Write). Trust me, this is the case 9 out of 10 times.
- if rapid development is necessary. With complex requirements, EFCore may have slower performance. But it is totally optimizable as well.
- if you need a cleaner and well maintainable data access code.

When to NOT use Entity Framework Core?

Entity Framework Core may not be a good option if

- if your application has to perform bulk operation on the database with Millions of records. Raw SQL Approach / Stored Procedure is recommended for such requirements.
- if your business logic is quite complicated to achieve with EF.

That being said, **DO NOT THINK** that you can use only one way to access data in an application. Practical systems often use both RAW SQL and Entity Framework Core within the application. This ensures that you have the best of both worlds. EFCore to query the data and perform quick operations, while Stored Procedures to perform complex, Bulk Updates.

The DB Context Class

DbContext class is the backbone of this entire concept. Treat DbContext class as a session with the database. An instance of this class allows us to do the data operations on the specified models. Here is how the DbContext Implementation would look like,

publicclass ApplicationContext : DbContext, IApplicationContext

```
{
publicApplicationContext(DbContextOptions<ApplicationContext> options)
: base(options)
{
}
public DbSet<Product> Products {get; set; }
}
```

Here, we can see that our DbContext Class has a Products entity defined as a property. This means that by using an instance of this DbContext class, we will be able to perform operations over the Product Entity.

Implementing Entity Framework Core in ASP.NET Core 3.1 WebApi – Code-First Approach

Let's see how to implement the Code-First Approach of Entity Framework Core in ASP.NET Core 3.1. We will create a new ASP.NET Core 3.1 Project with API Template using Visual Studio 2019 Community. Like always, I will be leaving the GitHub Repository link of the finished source code at the end of this post,

Installing the Required EFCore Packages

Firstly, after creating the API Project, let's installed the required packages for this tutorial. Run the following commands on your package manager console.

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.Tools
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Microsoft.EntityFrameworkCore.Design
Install-Package Swashbuckle.AspNetCore
Install-Package Swashbuckle.AspNetCore.Swagger
Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Design
```

For this demonstration, we will be using SQLServer as our database provider. That is why I am installing the SQLServer extension of EntityFrameworkCore.

What good is an API without Swagger? Swagger is the first thing that I install on any APIs that I work on. It makes documentation and testing so easy.

Creating a Model

Create a new Class and Name it Entity.cs

```
publicclass Entity
{
```



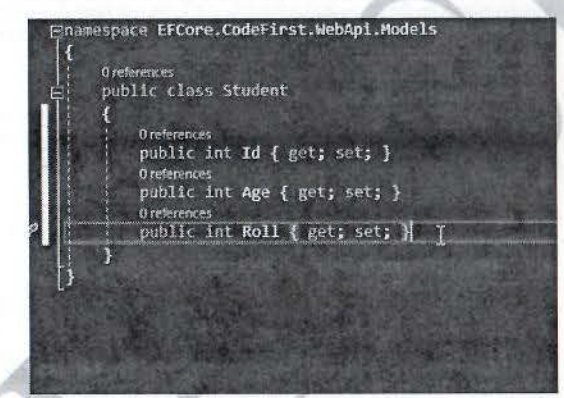
```
publicint Id {get; set; }
}
```

Why this Class? We will use this class as the base class for our future entities. Here we can define the common and secured properties like Id, CreatedDate, etc.

I will create a Student Model in the Models folder, and add a bunch of properties to it.

PRO TIP – Sometimes, writing lines of properties and adding the get and sets can be quite frustrating from a developer's point of view. We like everything automated right? Here is a tip to auto-generate properties. Just type in prop and hit Tab twice. All you have to enter is the data type and the name of the property.

These are called Code-Snippets. I will write about this in another article. **Efficient Coding!**




```
publicclass Student : Entity
{
publicint Age {get; set; }
publicint Roll {get; set; }
publicstring Name {get; set; }
publicint Class {get; set; }
publicstring Section {get; set; }
}
```

Adding the DataContext Class and Interface

Let's name our implementation of the DbContext Class as ApplicationDbContext.cs and IApplicationDbContext.cs respectively.

```
publicclass ApplicationDbContext : DbContext
{
publicApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
: base(options)
{
}
public DbSet<Student> Students {get; set; }
publicasync Task<int>SaveChanges()
{
returnawaitbase.SaveChangesAsync();
}
}
```

In this class, we have defined Student class as one of the entities that we intend to work with. Now, let's extract an interface for this class. Here is yet another PRO-TIP.

By definition, we now have the means to interact with Student Entity. But we haven't generated the database yet, have we.

Adding the Connection String.

Let's add the connection string pointing to the database we need to connect to. It can either be an existing database or a completely new one. I am going to use a fresh database. Open up appsettings.json and add this piece of line to the top.

```
"ConnectionStrings":{
"DefaultConnection":"Server=(localdb)\\mssqllocaldb;Database=developmentDb;Trusted_Connecti
on=True;MultipleActiveResultSets=true"
},
```

I am trying to connect to a localDb instance of the SQL Server. You can connect to a SQL Server Instance as well, by modifying the connection strings. You can refer here for connection string variants of SQL Server.

Configuring the Services

Now that we have finished setting up the bare minimum, let's configure our ASP.NET Core Application to support Entity Framework Core and set up Swagger as well. Go to your Startup class and modify the ConfigureServices method as below.

```
publicvoidConfigureServices(IServiceCollection services)
{
services.AddDbContext<ApplicationDbContext>(options =>
```



```
options.UseSqlServer(
Configuration.GetConnectionString("DefaultConnection"),
b => b.MigrationsAssembly(typeof(ApplicationDbContext).Assembly.FullName)));
services.AddScoped<IApplicationDbContext>(provider =>
provider.GetService<ApplicationDbContext>());
#region Swagger
services.AddSwaggerGen(c =>
{
c.IncludeXmlComments(string.Format(@"{0}\EFCore.CodeFirst.WebApi.xml",
System.AppDomain.CurrentDomain.BaseDirectory));
c.SwaggerDoc("v1", new OpenApiInfo
{
Version = "v1",
Title = "EFCore.CodeFirst.WebApi",
});
});
#endregion
services.AddControllers();
}
```

Line #3 defined the name of the context class to be added. In our cases it is ApplicationDbContext.
Line #4 states that we are using SQL Server as our Database Provider.
Line #5 mentions the Connection string name that we have already defined in appsettings.json.
Line #6 Binds the Concrete Class and the Interface into our Application Container.

Now add the below code to the Configure method. This is to let the application know that we intent to use Swagger.

```
#region Swagger
/ Enable middleware to serve generated Swagger as a JSON endpoint.
app.UseSwagger();
// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
// specifying the Swagger JSON endpoint.
app.UseSwaggerUI(c =>
{
c.SwaggerEndpoint("/swagger/v1/swagger.json", "EFCore.CodeFirst.WebApi");
});
#endregion
```

The 2 Most Important Commands

add-migrations

```
publicclass ApplicationDbContext : DbContext
{
publicApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
: base(options)
{
}
public DbSet<Student> Students {get; set; }
publicasync Task<int>SaveChanges()
{
returnawaitbase.SaveChangesAsync();
}
}
```

In this class, we have defined Student class as one of the entities that we intend to work with. Now, let's extract an interface for this class. Here is yet another PRO-TIP.

By definition, we now have the means to interact with Student Entity. But we haven't generated the database yet, have we.

Adding the Connection String.

Let's add the connection string pointing to the database we need to connect to. It can either be an existing database or a completely new one. I am going to use a fresh database. Open up appsettings.json and add this piece of line to the top.

```
"ConnectionStrings":{
"DefaultConnection":"Server=(localdb)\\mssqllocaldb;Database=developmentDb;Trusted_Connecti
on=True;MultipleActiveResultSets=true"
},
```

I am trying to connect to a localDb instance of the SQL Server. You can connect to a SQL Server Instance as well, by modifying the connection strings. You can refer here for connection string variants of SQL Server.

Configuring the Services

Now that we have finished setting up the bare minimum, let's configure our ASP.NET Core Application to support Entity Framework Core and set up Swagger as well. Go to your Startup class and modify the ConfigureServices method as below.

```
publicvoidConfigureServices(IServiceCollection services)
{
services.AddDbContext<ApplicationDbContext>(options =>
```



```
options.UseSqlServer(
Configuration.GetConnectionString("DefaultConnection"),
b => b.MigrationsAssembly(typeof(ApplicationDbContext).Assembly.FullName)));
services.AddScoped<IApplicationDbContext>(provider =>
provider.GetService<ApplicationDbContext>());
#region Swagger
services.AddSwaggerGen(c =>
{
c.IncludeXmlComments(string.Format(@"{0}\EFCore.CodeFirst.WebApi.xml",
System.AppDomain.CurrentDomain.BaseDirectory));
c.SwaggerDoc("v1", new OpenApiInfo
{
Version = "v1",
Title = "EFCore.CodeFirst.WebApi",
});
});
#endregion
services.AddControllers();
}
```

Line #3 defined the name of the context class to be added. In our cases it is ApplicationDbContext.
Line #4 states that we are using SQL Server as our Database Provider.
Line #5 mentions the Connection string name that we have already defined in appsettings.json.
Line #6 Binds the Concrete Class and the Interface into our Application Container.

Now add the below code to the Configure method. This is to let the application know that we intent to use Swagger.

```
#region Swagger
// Enable middleware to serve generated Swagger as a JSON endpoint.
app.UseSwagger();
// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
// specifying the Swagger JSON endpoint.
app.UseSwaggerUI(c =>
{
c.SwaggerEndpoint("/swagger/v1/swagger.json", "EFCore.CodeFirst.WebApi");
});
#endregion
```

The 2 Most Important Commands

add-migrations

update-database

Remember these 2 commands always. This is what you will need every single time to make changes to your database schema.Open your package manager console and type in the following.

add-migration "added student entity"
update-database

Line #1 will add a new Migrations folder to your Project. This contains the schema data of your models. EFCore uses this data to generate the Tables. The Comment inside quotes is essentially the description you would give for this particular migration. In our case, we have added the student entity and hence this description.

Line #2 , as the command says, updates / creates the database.

You will receive a done message, indicating that the database has been generated properly. Let's check it out. Go to View -> SQL Server Object Explorer. Here you can see various database and server available.

We have successfully created the Student Database using the Code First Approach of Entity Framework Core in ASP.NET Core! Now let's move to perform CRUD Operations for the Student Entity. **CRUD – Create, Read, Update and Delete.**

For this, I will create a new Empty API Controller under the Controllers folder. I will call it Student Controller. On this controller, we will implement CRUD Operations

CRUD Operations

Below is the code for the Student Controller. I will explain it in a while.

```
namespace EFCore.CodeFirst.WebApi.Controllers
{
[Route("api/[controller]")]
[ApiController]
publicclass StudentController : ControllerBase
{
private IApplicationDbContext _context;
publicStudentController(IApplicationDbContext context)
{
_context = context;
}
[HttpPost]
```



```
publicasync Task<IActionResult>Create(Student student)
{
_context.Students.Add(student);
await _context.SaveChanges();
returnOk(student.Id);
}
[HttpGet]
publicasync Task<IActionResult>GetAll()
{
var students = await _context.Students.ToListAsync();
if(students == null)returnNotFound();
returnOk(students);
}
[HttpGet("{id}")]
publicasync Task<IActionResult>GetById(int id)
{
var student = await _context.Students.Where(a => a.Id == id).FirstOrDefaultAsync();
if(student == null)returnNotFound();
returnOk(student);
}
[HttpDelete("{id}")]
publicasync Task<IActionResult>Delete(int id)
{
var student = await _context.Students.Where(a => a.Id == id).FirstOrDefaultAsync();
if(student == null)returnNotFound();
_context.Students.Remove(student);
await _context.SaveChanges();
returnOk(student.Id);
}
[HttpPut("{id}")]
publicasync Task<IActionResult>Update(int id, Student studentUpdate)
{
var student = _context.Students.Where(a => a.Id == id).FirstOrDefault();
if(student == null)returnNotFound();
else
{
student.Age = studentUpdate.Age;
student.Name = studentUpdate.Name;
await _context.SaveChanges();
```

```
returnOk(student.Id);
}
}
}
}
```

Line #6 and #7 are for injecting our Instance of ApplicationDbContext to the Constructor of the Student Controller so that we have access to the instance readily.

The different CRUD Action Methods.

GetAll – Here we use the context instance to access the dataset of Student Entity to retrieve the list of all students.
GetById – Using LINQ, we access the Student Dataset and retrieve the record with respective Id. If the record does not exist, the API throws a NotFound Exception.
Create – Saves the passes record to the database and returns the Id.
Delete – Deletes the record with specific Id. Return a NotFound Exception if invalid
Update – Updates the database with the record by Id. For keeping it simple, we are only allowing the Age and Name Property to be updated.

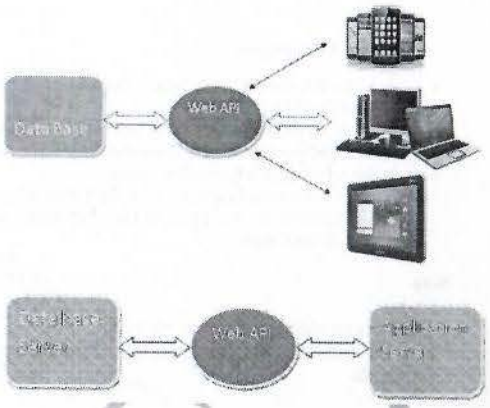
Web APIs

Web API Definition

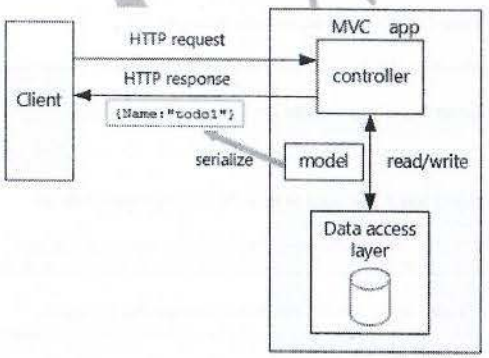
Web API is an application programming interface (API) that is used to enable communication or interaction with software components with each other. ASP.NET Web API is a framework that makes it easy to build HTTP Service that reaches a broad range of clients, including browsers and mobile devices. Using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



The following diagram shows the design of the app.



Create a web project

- From the **File** menu, select **New** > **Project**.
- Enter *Web API* in the search box.
- Select the **ASP.NET Core Web API** template and select **Next**.
- In the **Configure your new project dialog**, name the project *TodoApi* and select **Next**.
- In the **Additional information** dialog:
  - Confirm the **Framework** is **.NET 7.0** (or later).
  - Confirm the checkbox for **Use controllers(uncheck to use minimal APIs)** is checked.
  - Select **Create**.

Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at **Package consumption workflow (NuGet documentation)**. Confirm correct package versions at **NuGet.org**.

Test the project

The project template creates a `WeatherForecast` API with support for Swagger.

Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:

Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:

Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio launches the default browser and navigates to `https://localhost:<port>/swagger/index.html`, where `<port>` is a randomly chosen port number.

The Swagger page `/swagger/index.html` is displayed. Select **GET** > **Try it out** > **Execute**. The page displays:

- The Curl command to test the WeatherForecast API.
- The URL to test the WeatherForecast API.
- The response code, body, and headers.
- A drop-down list box with media types and the example value and schema.

Swagger is used to generate useful documentation and help pages for web APIs. This tutorial focuses on creating a web API. For more information on Swagger, see ASP.NET Core web API documentation with Swagger / OpenAPI.

Copy and paste the **Request URL** in the browser: `https://localhost:<port>/weatherforecast`

JSON similar to the following example is returned:

```
JSONCopy
[
  {
"date": "2019-07-16T19:04:05.7257911-06:00",
"temperatureC": 52,
"temperatureF": 125,
"summary": "Mild"
  },
  {
"date": "2019-07-17T19:04:05.7258461-06:00",
"temperatureC": 36,
"temperatureF": 96,
"summary": "Warm"
  },
  {
"date": "2019-07-18T19:04:05.7258467-06:00",
"temperatureC": 39,
"temperatureF": 102,
"summary": "Cool"
  },
  {
"date": "2019-07-19T19:04:05.7258471-06:00",
"temperatureC": 10,
"temperatureF": 49,
"summary": "Bracing"
  },
  {
"date": "2019-07-20T19:04:05.7258474-06:00",
```

```
"temperatureC": -1,
"temperatureF": 31,
"summary": "Chilly"
  }
]
```

Add a model class

A *model* is a set of classes that represent the data that the app manages. The model for this app is the `TodoItem` class.

- In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder `Models`.
- Right-click the `Models` folder and select **Add** > **Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following:

```
#Copy
namespaceTodoApi.Models;

publicclassTodoItem
{
publiclong Id { get; set; }
publicstring? Name { get; set; }
publicbool IsComplete { get; set; }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the `Models` folder is used by convention.

Add a database context

The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the Microsoft.EntityFrameworkCore.DbContext class.

Add NuGet packages

- From the **Tools** menu, select **NuGet Package Manager > Manage NuGet Packages for Solution**.
- Select the **Browse** tab, and then enter `Microsoft.EntityFrameworkCore.InMemory` in the search box.
- Select `Microsoft.EntityFrameworkCore.InMemory` in the left pane.
- Select the **Project** checkbox in the right pane and then select **Install**.

Add the TodoContext database context

- Right-click the `Models` folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.
- Enter the following code:

```
C#Copy
using Microsoft.EntityFrameworkCore;

namespaceTodoApi.Models;

publicclassTodoContext : DbContext
{
publicTodoContext(DbContextOptions<TodoContext> options)
    : base(options)
  {
  }

public DbSet<TodoItem> TodoItems { get; set; } = null!;
}
```

Register the database context

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers.

Update `Program.cs` with the following highlighted code:

```
C#Copy
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddDbContext<TodoContext>(opt =>
  opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
```

```
app.UseSwagger();
app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The preceding code:

- Adds `using` directives.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

- Right-click the *Controllers* folder.
- Select **Add** > **New Scaffolded Item**.
- Select **API Controller with actions, using Entity Framework**, and then select **Add**.
- In the **Add API Controller with actions, using Entity Framework** dialog:
  - Select **TodoItem (TodoApi.Models)** in the **Model class**.
  - Select **TodoContext (TodoApi.Models)** in the **Data context class**.
  - Select **Add**.

If the scaffolding operation fails, select **Add** to try scaffolding a second time.

The generated code:

- Marks the class with the `[ApiController]` attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

The ASP.NET Core templates for:

- Controllers with views include `[action]` in the route template.
- API controllers don't include `[action]` in the route template.

When the `[action]` token isn't in the route template, the action name (method name) isn't included in the endpoint. That is, the action's associated method name isn't used in the matching route.

Update the PostTodoItem create method

Update the return statement in the `PostTodoItem` to use the nameof operator:

```
C#Copy
[HttpPost]
publicasync Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
  _context.TodoItems.Add(todoItem);
await _context.SaveChangesAsync();

//  return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem);
return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id }, todoItem);
}
```

The preceding code is an `HTTP POST` method, as indicated by the `[HttpPost]` attribute. The method gets the value of the `TodoItem`from the body of the HTTP request.

The CreatedAtAction method:

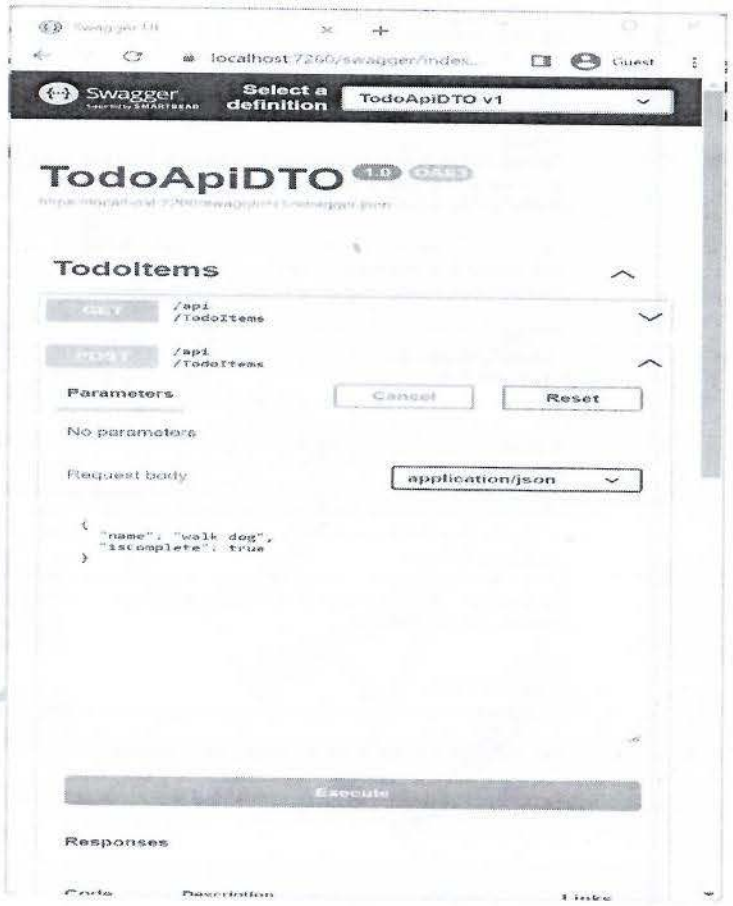
- Returns an HTTP 201 status code if successful. `HTTP 201` is the standard response for an `HTTP POST` method that creates a new resource on the server.
- Adds a Location header to the response. The `Location` header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

Test PostTodoItem

- Press Ctrl+F5 to run the app.
- In the Swagger browser window, select **POST /api/TodoItems**, and then select **Try it out**.
- In the **Request body** input window, update the JSON. For example,

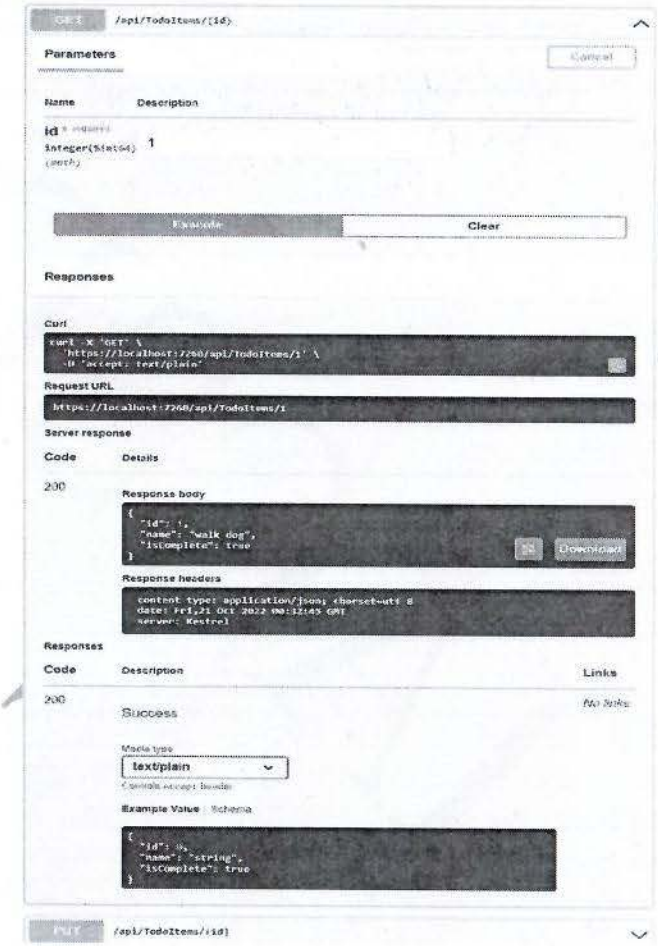
```
JSONCopy
{
"name": "walk dog",
"isComplete": true
}
```

- Select **Execute**



Test the location header URI
In the preceding POST, the Swagger UI shows the location header under Response headers. For

example, `location: https://localhost:7260/api/TodoItems/1`. The location header shows the URI to the created resource.
To test the location header:
In the Swagger browser window, select GET /api/TodoItems/{id}, and then select Try it out.
Enter `1` in the `id` input box, and then select Execute.



Examine the GET methods

Two GET endpoints are implemented:

```
GET /api/todoitems
GET /api/todoitems/{id}
```

The previous section showed an example of the `/api/todoitems/{id}` route.
Follow the POST instructions to add another todo item, and then test the `/api/todoitems` route using Swagger.
This app uses an in-memory database. If the app is stopped and started, the preceding GET request will not return any data. If no data is returned, POST data to the app.
Routing and URL paths

The `[HttpGet]` attribute denotes a method that responds to an `HTTP GET` request. The URL path for each method is constructed as follows:
Start with the template string in the controller's `Route` attribute:

C#Copy

```
[Route("api/[controller]")]
[ApiController]
publicclassTodoItemsController : ControllerBase
```

Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is TodoItemsController, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.
If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.
In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

C#Copy

```
[HttpGet("{id}")]
publicasync Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
var todoItem = await _context.TodoItems.FindAsync(id);

if (todoItem == null)
    {
return NotFound();
    }

return todoItem;
}
```

Return values
The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response

message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.
`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:
If no item matches the requested ID, the method returns a 404 status NotFound error code.
Otherwise, the method returns 200 with a JSON response body. Returning `Item` results in an `HTTP 200` response.

The PutTodoItem method
Examine the `PutTodoItem` method:
C#Copy

```
[HttpPut("{id}")]
publicasync Task<IActionResult>PutTodoItem(long id, TodoItem todoItem)
{
if (id != todoItem.Id)
    {
return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;

try
    {
await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
if (!TodoItemExists(id))
        {
return NotFound();
        }
else
        {
throw;
        }
    }

return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses `HTTP PUT`. The response is 204 (No Content). According to the HTTP specification, a `PUT` request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.
Test the PutTodoItem method

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.
Using the Swagger UI, use the PUT button to update the `TodoItem` that has Id = 1 and set its name

to `"feed fish"`. Note the response is `HTTP 204 No Content`.
The DeleteTodoItem method
Examine the `DeleteTodoItem` method:

C#Copy

```
[HttpDelete("{id}")]
publicasync Task<IActionResult>DeleteTodoItem(long id)
{
var todoItem = await _context.TodoItems.FindAsync(id);
if (todoItem == null)
    {
return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
await _context.SaveChangesAsync();

return NoContent();
}
```

Test the DeleteTodoItem method
Use the Swagger UI to delete the `TodoItem`that has Id = 1. Note the response is `HTTP 204 No Content`.
Test with http-repl, Postman, or curl
http-repl, Postman, and curl are often used to test API's. Swagger uses `curl` and shows the `curl` command it submitted.

For instructions on these tools, see the following links:
Test APIs with Postman
Install and test APIs with `http-repl`
For more information on `http-repl`, see Test web APIs with the HttpRepl.
Prevent over-posting
Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this, and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. DTO is used in this tutorial.
A DTO may be used to:
Prevent over-posting.
Hide properties that clients are not supposed to view.
Omit some properties in order to reduce payload size.
Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.
To demonstrate the DTO approach, update the `TodoItem` class to include a secret field:
C#Copy

```
namespaceTodoApi.Models
{
```

```
publicclassTodoItem
    {
publiclong Id { get; set; }
publicstring? Name { get; set; }
publicbool IsComplete { get; set; }
publicstring? Secret { get; set; }
    }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.
Verify you can post and get the secret field.
Create a DTO model:
C#Copy

```
namespaceTodoApi.Models;

publicclassTodoItemDTO
{
publiclong Id { get; set; }
publicstring? Name { get; set; }
publicbool IsComplete { get; set; }
}
```

Update the `TodoItemsController` to use `TodoItemDTO`:
C#Copy

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespaceTodoApi.Controllers;

[Route("api/[controller]")]
[ApiController]
publicclassTodoItemsController : ControllerBase
{
privatereadonly TodoContext _context;

publicTodoItemsController(TodoContext context)
    {
        _context = context;
    }

// GET: api/TodoItems
    [HttpGet]
publicasync Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
    {
returnawait _context.TodoItems
            .Select(x => ItemToDTO(x))
            .ToListAsync();
    }

// GET: api/TodoItems/5
```



```
// <snippet_GetByID>
    [HttpGet("{id}")]
publicasync Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
    {
var todoItem = await _context.TodoItems.FindAsync(id);

if (todoItem == null)
        {
return NotFound();
        }

return ItemToDTO(todoItem);
    }
// </snippet_GetByID>

// PUT: api/TodoItems/5
// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
// <snippet_Update>
    [HttpPut("{id}")]
publicasync Task<IActionResult>PutTodoItem(long id, TodoItemDTO todoDTO)
    {
if (id != todoDTO.Id)
        {
return BadRequest();
        }

var todoItem = await _context.TodoItems.FindAsync(id);
if (todoItem == null)
        {
return NotFound();
        }

        todoItem.Name = todoDTO.Name;
        todoItem.IsComplete = todoDTO.IsComplete;

try
        {
await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
        {
return NotFound();
        }

return NoContent();
    }
// </snippet_Update>

// POST: api/TodoItems
// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
```

```
// <snippet_Create>
    [HttpPost]
publicasync Task<ActionResult<TodoItemDTO>> PostTodoItem(TodoItemDTO todoDTO)
    {
var todoItem = new TodoItem
        {
            IsComplete = todoDTO.IsComplete,
            Name = todoDTO.Name
        };

        _context.TodoItems.Add(todoItem);
await _context.SaveChangesAsync();

return CreatedAtAction(
nameof(GetTodoItem),
new { id = todoItem.Id },
            ItemToDTO(todoItem));
    }
// </snippet_Create>

// DELETE: api/TodoItems/5
    [HttpDelete("{id}")]
publicasync Task<IActionResult>DeleteTodoItem(long id)
    {
var todoItem = await _context.TodoItems.FindAsync(id);
if (todoItem == null)
        {
return NotFound();
        }

        _context.TodoItems.Remove(todoItem);
await _context.SaveChangesAsync();

return NoContent();
    }

privateboolTodoItemExists(long id)
    {
return _context.TodoItems.Any(e => e.Id == id);
    }

privatestatic TodoItemDTO ItemToDTO(TodoItem todoItem) =>
new TodoItemDTO
        {
            Id = todoItem.Id,
            Name = todoItem.Name,
            IsComplete = todoItem.IsComplete
        };
}
```

Verify you can't post or get the secret field