Introduction to the .NET Framework

The .NET Framework is a software development platform created by Microsoft to build, run, and deploy applications on Windows. It provides a unified environment for application development, offering libraries, runtime, and tools.

Key Components of .NET Framework

Common Language Runtime (CLR):

-Acts as the execution engine.

-Handles code execution, memory management, garbage collection, and more.

Base Class Library (BCL):

-A collection of reusable classes and APIs.

-Includes functionalities for string manipulation, file handling, data access, etc.

Languages:

Supports multiple languages like C#, VB.NET, and F# via the Common Language Specification (CLS).

Application Models:

<u>Frameworks for building different types of applications:</u>

-Windows Forms for desktop apps.

-ASP.NET for web apps.

-ADO.NET for data access.

Intermediate Language (IL)

<u>Definition:</u> IL (Intermediate Language) is a low-level, platform-independent code generated by compilers (like C# and VB.NET) targeting the .NET Framework.

<u>Execution:</u> IL is further compiled to native machine code by the Just-In-Time (JIT) compiler.

<u>Advantages:</u>

Enables cross-language interoperability.

Platform-independent and optimized for execution on any system with CLR.

Assemblies and Their Structure

<u>Definition:</u> Assemblies are the building blocks of .NET applications and are used to package compiled code, resources, and metadata.

<u>Types:</u>

EXEs: Executable files for standalone applications.

DLLs: Dynamic Link Libraries for reusable code.

Structure of an Assembly:

<u>Manifest:</u>

Contains metadata about the assembly, like version and dependencies.

<u>IL Code:</u>

The compiled Intermediate Language code.

<u>Metadata:</u>

Provides information about types, members, and references.

<u>Resources:</u>

Embedded resources like images, icons, or strings.

Common Language Runtime (CLR)

CLR is the runtime environment of the .NET Framework that manages application execution.

Functions of CLR:

JIT Compilation:

Converts IL into native machine code at runtime for execution.

-Offers three types:

  -Pre-JIT: Compiles entire code at startup.

  -Econo-JIT: Compiles code on demand and discards it after execution.

  -Normal JIT: Compiles code on demand and caches it.

Memory Management:

Allocates and releases memory for applications.

Garbage Collection (GC):

-Automatically reclaims unused memory.

-Three generations for efficiency:

Gen 0: Short-lived objects.

Gen 1: Objects that survive Gen 0 collection.

Gen 2: Long-lived objects.

AppDomain Management:

Provides isolation for running multiple applications in a single process.

Security:

Offers mechanisms like Code Access Security (CAS) and Role-Based Security (RBS) to enforce application security.

Common Language Specification (CLS)

A subset of rules and constraints ensuring language interoperability.

Guarantees that code written in one .NET language can be used in another.

Common Type System (CTS)

Defines how types are declared, used, and managed in the .NET Framework.

Ensures consistency across .NET languages by unifying data types.

Garbage Collection

Automates memory management to prevent memory leaks.

Key Features:

Releases memory occupied by unreferenced objects.

Compact memory to reduce fragmentation.

Operates on managed heap organized into three generations.

Security in .NET Framework

Code Access Security (CAS):

Restricts the permissions of code based on evidence like origin and publisher.

Role-Based Security (RBS):

Manages access based on user roles.

Validation and Encryption:

Includes APIs for cryptographic operations and data validation.

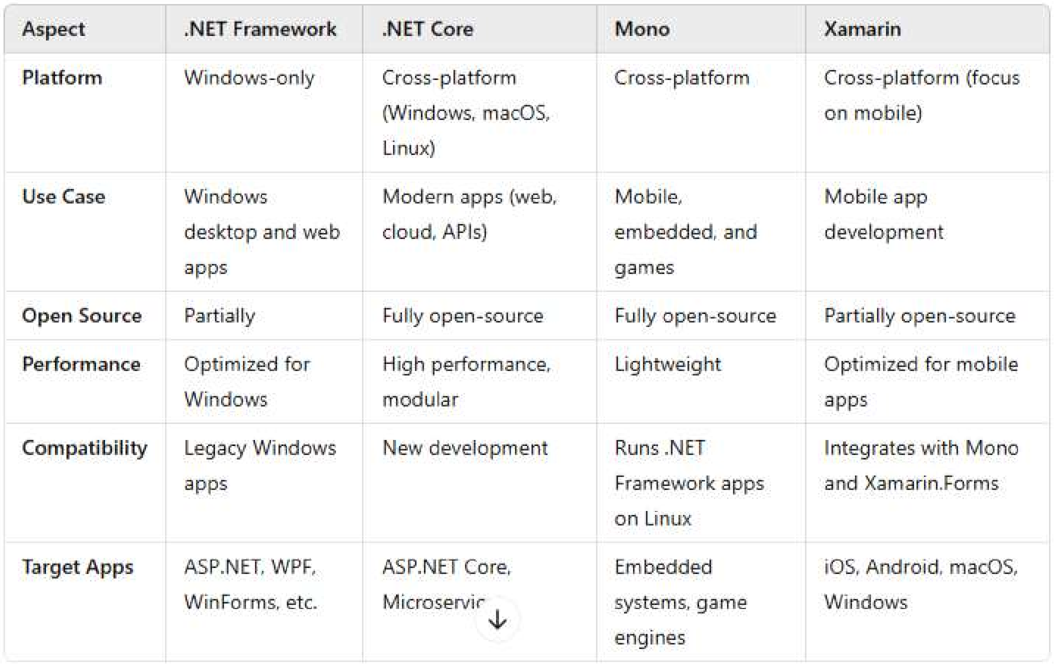
Sandboxing:

Executes untrusted code in a restricted environment.

Summary

The .NET Framework is a comprehensive and versatile platform for building applications. Its features like CLR, garbage collection, and assemblies make it a robust environment for developers. Understanding its components like IL, assemblies, memory management, and security mechanisms is crucial for leveraging the framework effectively.

.NET Framework, .NET Core, Mono, and Xamarin: Differences



Aspect	.NET Framework	.NET Core	Mono	Xamarin
Platform	Windows-only	Cross-platform (Windows, macOS, Linux)	Cross-platform	Cross-platform (focus on mobile)
Use Case	Windows desktop and web apps	Modern apps (web, cloud, APIs)	Mobile, embedded, and games	Mobile app development
Open Source	Partially	Fully open-source	Fully open-source	Partially open-source
Performance	Optimized for Windows	High performance, modular	Lightweight	Optimized for mobile apps
Compatibility	Legacy Windows apps	New development	Runs .NET Framework apps on Linux	Integrates with Mono and Xamarin.Forms
Target Apps	ASP.NET, WPF, WinForms, etc.	ASP.NET Core, Microservic	Embedded systems, game engines	iOS, Android, macOS, Windows

Versions of the .NET Framework
<u>.NET Framework:</u>
Initial Release: Version 1.0 (2002).
Latest Release: 4.8.1 (2023).
Supports Windows-based desktop and web applications.
<u>.NET Core:</u>
Released in 2016 as a modular, open-source, and cross-platform framework.
Unified with .NET Framework into .NET 5+.
<u>.NET 5+:</u>
Introduced in 2020 as a unified framework for all platforms.
Latest Version: .NET 7 (2022).
<u>Mono/Xamarin:</u>
Used for cross-platform mobile and embedded development.
Latest Xamarin updates are part of .NET 6 and beyond.
Managed and Unmanaged Code
<u>Managed Code:</u>
-Executed within the Common Language Runtime (CLR).
-Features:
  -Automatic memory management via Garbage Collection.
  -Enhanced security and exception handling.
Example: Code written in C#, VB.NET, or F#.
<u>Unmanaged Code:</u>
-Executed directly by the operating system.
-Features:

-Requires manual memory management.
-Less secure and prone to errors.
Example: Code written in C or C++.

Introduction to Visual Studio

Visual Studio is an Integrated Development Environment (IDE) by Microsoft used for .NET application development.
It supports multiple programming languages and offers a rich set of tools for coding, debugging, and deployment.

Features:

-Code Editor:
IntelliSense for autocompletion and code navigation.
-Debugger:
Integrated debugging for detecting and fixing errors.
-Project Templates:
Predefined templates for applications like ASP.NET, WPF, and more.
-Extensions:
Plugins to enhance functionality.
-Integration:
Built-in support for Git, Azure, and CI/CD pipelines.

Using ILDASM (Intermediate Language Disassembler)

ILDASM is a tool provided with the .NET Framework SDK to inspect the contents of an assembly (EXE/DLL).
It converts the compiled assembly code back into Intermediate Language (IL), showing metadata, manifest, and method-level details.

Steps to Use ILDASM:

-Open ILDASM:
Use the Developer Command Prompt or locate ildasm.exe in the .NET SDK folder.
-Load an Assembly:
Drag and drop the .exe or .dll file into the ILDASM interface.
-Inspect Assembly Contents:
-View sections like:
-Manifest: Assembly metadata.
-Classes and Methods: Displaying IL code for each method.
Resources: Embedded files in the assembly.
-Export IL Code:
Save the IL code to a file using File → Dump.

Example Usage:

Open MyApp.dll and analyze its methods and metadata to debug or understand the compiled code.

Summary

1).NET Framework, Core, Mono, and Xamarin cater to different platforms and use cases.
2)Understanding managed vs. unmanaged code is essential for memory and resource management.
3)Visual Studio provides a rich development environment for .NET projects.
4)ILDASM is a powerful tool for inspecting assemblies, aiding in debugging and analysis.

Console Applications and Class Libraries in .NET Core

Console Applications:

.NET Core allows you to create cross-platform console applications that can run on Windows, macOS, and Linux.

You can build console applications using C# in Visual Studio or Visual Studio Code, and they are usually started with a Main method.
Example of a simple Console Application in .NET Core:

```
using System;

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

Class Libraries:
.NET Core allows creating Class Libraries that can be referenced in other applications (console, web, or desktop).
These libraries contain reusable logic and can be packaged as DLLs.
Example of a simple Class Library in .NET Core:

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

C# Basics
Variables: You can declare variables using the syntax type variableName;. Example:

```
int age = 30;
string name = "John";
```

Comments:
Single-line comment: // comment
Multi-line comment: /* comment */

Project References and Using

Project References:
-To reference another project or class library, you can add a reference in your project's .csproj file or use the dotnet add reference command.
-Example: Adding a project reference in the .csproj file:

```
<ProjectReference Include="..\Library\Library.csproj" />
```

Using:
The using keyword is used to include namespaces and also to manage resources through the IDisposable pattern.
Example:

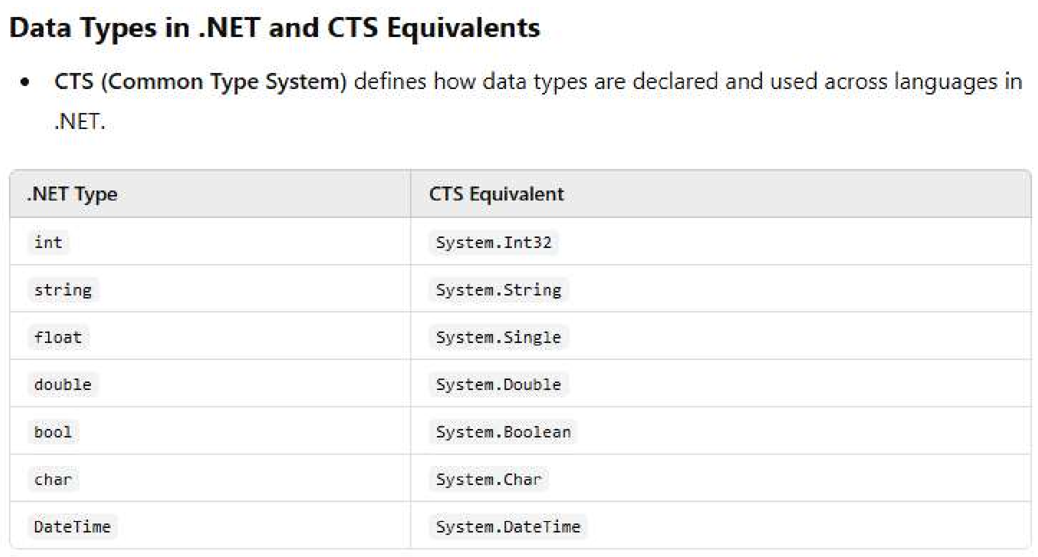
```
using System; // Including the System namespace
```

Classes

Definition: A class is a blueprint for objects, containing fields, properties, methods, and events.
Example:

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

```
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
}
```



Data Types in .NET and CTS Equivalents

- CTS (Common Type System) defines how data types are declared and used across languages in .NET.

.NET Type	CTS Equivalent
`int`	`System.Int32`
`string`	`System.String`
`float`	`System.Single`
`double`	`System.Double`
`bool`	`System.Boolean`
`char`	`System.Char`
`DateTime`	`System.DateTime`

Methods

<u>Definition:</u> Methods define the functionality within a class. They can take parameters and return values.

Example:

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

Method Overloading

<u>Definition:</u> Method overloading allows you to define multiple methods with the same name but different parameter types or numbers.

Example:

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

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

Optional Parameters

<u>Definition:</u> Optional parameters allow you to call a method without explicitly providing a value for each parameter.

Example:

```
public void PrintMessage(string message, int times = 1)
```

```
{
   for (int i = 0; i < times; i++)
   {
      Console.WriteLine(message);
   }
}
```

Named Parameters and Positional Parameters

Positional Parameters: These are the parameters passed based on their position.

Example: PrintMessage("Hello", 3);

Named Parameters: These are parameters passed by explicitly specifying their names.

Example:

```
PrintMessage(message: "Hello", times: 3);
```

Using params

Definition: The params keyword allows a method to accept a variable number of arguments as an array.

Example:

```
public void DisplayNumbers(params int[] numbers)
{
   foreach (var number in numbers)
   {
      Console.WriteLine(number);
   }
}
```

Local Functions

Definition: Local functions are methods defined inside other methods, typically for encapsulating functionality.

Example:

```
public void OuterMethod()
{
   int AddNumbers(int x, int y) => x + y; // Local function

   Console.WriteLine(AddNumbers(3, 4)); // Calling the local function
}
```

Properties

Definition: Properties are members of a class that provide a mechanism to read, write, or compute values.

get, set:

Example:

```
public int Age { get; set; }
```

Readonly Properties:

Example:

```
public int Id { get; } = 100;
```

Using Property Accessors to Create Readonly Property:

Example:

```
private int _age;
public int Age
{
   get { return _age; }
   set { _age = value; }
}
```

Constructors
Definition: Constructors are special methods that are invoked when an object of a class is created. They are used to initialize the object's state.
Example:
```
class Person
{
   public string Name { get; set; }

   public Person(string name)
   {
      Name = name;
   }
}
```
Object Initializer
Definition: Object initializers allow initializing an object without explicitly calling the constructor.
Example:
```
Person person = new Person { Name = "John" };
```
Destructors
Definition: Destructors are used for cleanup activities before an object is destroyed.
Example:
```
class Person
{
   ~Person() // Destructor
   {
      Console.WriteLine("Object is being destroyed.");
   }
}
```
IDisposable and Disposal
Definition: IDisposable is an interface that defines the Dispose method for releasing unmanaged resources.
Example:
```
public class Resource : IDisposable
{
   private bool disposed = false;

   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }
   protected virtual void Dispose(bool disposing)
   {
      if (!disposed)
      {
         if (disposing)
         {
            // Free managed resources here.
         }
         // Free unmanaged resources here.
```

```
            disposed = true;
        }
    }

    ~Resource()
    {
        Dispose(false);
    }
}
```
Summary
-C# offers a variety of features for building robust and flexible applications, including methods, overloading, properties, and constructors.
-Key concepts like params, optional parameters, and IDisposable help manage resources and improve code flexibility.
-Understanding object-oriented principles, such as constructors, destructors, and method overloading, is essential in C# development.

Static Members of a Class
Static Members:
-Belong to the class rather than an instance of the class.
-Shared across all instances of the class.
-Accessed using the class name.
Static Fields
-Fields declared with the static keyword.
-Shared across all instances of the class.
Example:
```
class MyClass
{
    public static int Counter = 0;
}
```
Static Methods
Methods that don’t require an instance to be invoked.
Example:
```
class MyClass
{
    public static void DisplayMessage()
    {
        Console.WriteLine("Static Method Called");
    }
}
```
Static Properties
Properties that provide access to static fields.
Example:
```
class MyClass
{
    private static int _value;
    public static int Value
    {
        get { return _value; }
        set { _value = value; }
    }
}
```

Static Constructors

A special constructor used to initialize static fields or perform one-time actions.
Runs automatically when the class is accessed for the first time.

Example:

```
class MyClass
{
    static MyClass()
    {
        Console.WriteLine("Static Constructor Called");
    }
}
```

Static Classes

A class that can only contain static members.
Cannot be instantiated.

Example:

```
static class Utilities
{
    public static void PrintMessage(string message)
    {
        Console.WriteLine(message);
    }
}
```

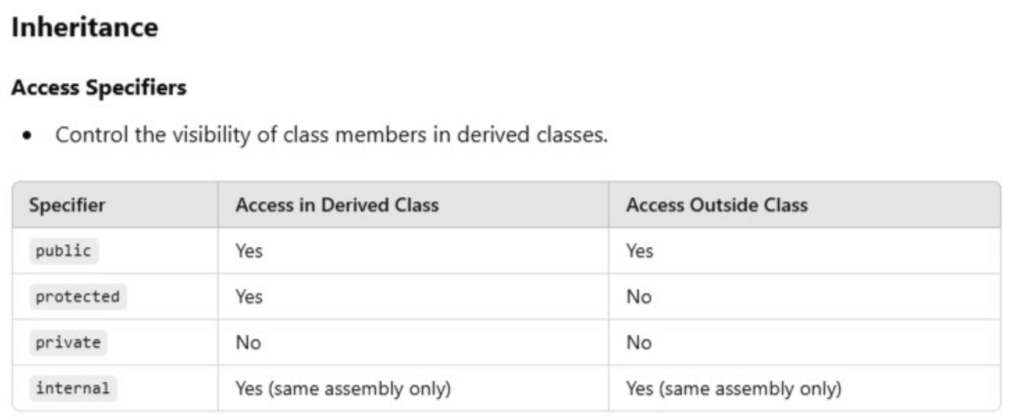
Static Local Functions

Functions defined within a method and declared static.
Cannot access instance members of the containing class.

Example:

```
class MyClass
{
    public void OuterMethod()
    {
        static void LocalFunction()
        {
            Console.WriteLine("Static Local Function");
        }

        LocalFunction();
    }
}
```



Inheritance

Access Specifiers

- Control the visibility of class members in derived classes.

Specifier	Access in Derived Class	Access Outside Class
`public`	Yes	Yes
`protected`	Yes	No
`private`	No	No
`internal`	Yes (same assembly only)	Yes (same assembly only)

Constructors in a Hierarchy
The base class constructor is always called before the derived class constructor.
Use the base keyword to explicitly call a base class constructor.
Example:

```
class BaseClass
{
   public BaseClass()
   {
      Console.WriteLine("Base Constructor");
   }
}
class DerivedClass : BaseClass
{
   public DerivedClass() : base()
   {
      Console.WriteLine("Derived Constructor");
   }
}
```

Overloading in a Derived Class
Derived classes can overload base class methods by providing methods with the same name but different parameters.
Example:

```
class BaseClass
{
   public void Display(string message)
   {
      Console.WriteLine(message);
   }
}

class DerivedClass : BaseClass
{
   public void Display(string message, int times)
   {
      for (int i = 0; i < times; i++)
         Console.WriteLine(message);
   }
}
```

Hiding Members Using new

Hides a base class member in the derived class.

Example:

```
class BaseClass
{
   public void Display()
   {
      Console.WriteLine("Base Class Display");
   }
}
class DerivedClass : BaseClass
{
   public new void Display()
   {
      Console.WriteLine("Derived Class Display");
   }
}
```

Overriding

-Allows a derived class to provide a specific implementation for a virtual method in the base class.

-The base class method must be marked as virtual.

Example:

```
class BaseClass
{
   public virtual void Display()
   {
      Console.WriteLine("Base Class Display");
   }
}

class DerivedClass : BaseClass
{
   public override void Display()
   {
      Console.WriteLine("Derived Class Display");
   }
}
```

Sealed Methods

Prevents further overriding of a method in derived classes.

Marked with the sealed keyword.

Example:

```
class BaseClass
{
   public virtual void Display()
   {
      Console.WriteLine("Base Class Display");
   }
}
class IntermediateClass : BaseClass
```

```
{
   public sealed override void Display()
   {
      Console.WriteLine("Intermediate Class Display");
   }
}
class FinalClass : IntermediateClass
{
   // Cannot override Display method here
}
```

Abstract Classes

-Classes that cannot be instantiated and must be inherited.

-Can contain abstract methods (without implementation) and concrete methods.

Example:

```
abstract class Shape
{
   public abstract void Draw();
   public void DisplayInfo()
   {
      Console.WriteLine("Shape Information");
   }
}
class Circle : Shape
{
   public override void Draw()
   {
      Console.WriteLine("Drawing Circle");
   }
}
```

Abstract Methods

-Methods declared in an abstract class without a body.

-Must be overridden in derived classes.

Example:

```
abstract class Animal
{
   public abstract void Speak();
}
class Dog : Animal
{
   public override void Speak()
   {
      Console.WriteLine("Woof!");
   }
}
```

Sealed Classes

-Classes that cannot be inherited.

-Marked with the sealed keyword.

Example:

```
sealed class Utility
{
    public static void PrintMessage(string message)
    {
        Console.WriteLine(message);
    }
}
```
Summary
-Static members and classes are used for shared functionality.
-Inheritance allows reusability and polymorphism with control via access specifiers.
-Virtual, override, new, and sealed keywords manage method overriding and visibility.
-Abstract classes and methods enforce the implementation of specific functionality in derived classes.

Interfaces
An interface in C# defines a contract that classes or structs can implement. It can contain methods, properties, events, or indexers, but without any implementation.
Implementing an Interface
A class or struct implements an interface by providing definitions for all its members.
Example:
```
public interface IAnimal
{
    void Speak();
    string Name { get; set; }
}
public class Dog : IAnimal
{
    public string Name { get; set; }

    public void Speak()
    {
        Console.WriteLine("Woof!");
    }
}
```
Explicitly Implementing an Interface
Explicit implementation allows you to provide different implementations for the same member name when implementing multiple interfaces. These members can only be accessed via the interface, not directly through the class.
Example:
```
public interface IAnimal
{
    void Speak();
}
public interface IRobot
{
    void Speak();
}
public class RoboDog : IAnimal, IRobot
{
    void IAnimal.Speak()
    {
        Console.WriteLine("Animal Speak: Woof!");
    }
    void IRobot.Speak()
    {
        Console.WriteLine("Robot Speak: Beep!");
    }
}
// Usage:
RoboDog roboDog = new RoboDog();
((IAnimal)roboDog).Speak(); // Outputs: Animal Speak: Woof!
((IRobot)roboDog).Speak(); // Outputs: Robot Speak: Beep!
```

Inheritance in Interfaces

Interfaces can inherit other interfaces. A derived interface must include all members of the base interfaces.

<u>Example:</u>

```
public interface IAnimal
{
    void Speak();
}
public interface IPet : IAnimal
{
    void Play();
}
public class Cat : IPet
{
    public void Speak()
    {
        Console.WriteLine("Meow!");
    }
    public void Play()
    {
        Console.WriteLine("Cat is playing.");
    }
}
```

Default Interface Methods

Starting with C# 8.0, interfaces can include default implementations for members. Classes implementing the interface can use or override the default implementation.

<u>Example:</u>

```
public interface IAnimal
{
    void Speak();
    // Default method implementation
    void Eat()
    {
        Console.WriteLine("Eating...");
    }
}
public class Dog : IAnimal
{
    public void Speak()
    {
        Console.WriteLine("Woof!");
    }
    // Optionally override the default method
    public void Eat()
    {
        Console.WriteLine("Dog is eating.");
    }
}
// Usage:
IAnimal dog = new Dog();
dog.Speak(); // Outputs: Woof!
dog.Eat();   // Outputs: Dog is eating.
```

Operator Overloading

-Operator overloading allows a class or struct to define its behavior for specific operators like +, -, *, etc.

Rules for Operator Overloading:

-Operators are defined as public and static.
-At least one operand must be a class or struct.
-Not all operators can be overloaded (e.g., &&, ||, =).

Example:

```
public class Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }

    public Complex(double real, double imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }
    // Overload + operator
    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);
    }
    // Override ToString for display
    public override string ToString()
    {
        return $"{Real} + {Imaginary}i";
    }
}
// Usage:
Complex c1 = new Complex(1, 2);
Complex c2 = new Complex(3, 4);
Complex result = c1 + c2;
Console.WriteLine(result); // Outputs: 4 + 6i
```

Summary

-Interfaces define a contract that classes or structs must implement.
-Explicit implementation provides member isolation for specific interface usage.
-Interfaces support inheritance and can have default methods starting with C# 8.0.
-Operator overloading enables custom behavior for operators in classes or structs.

Reference and Value Types
Value Types: Stored on the stack; contain the actual data.
Examples: int, float, bool, struct, enum
Reference Types: Stored on the heap; contain a reference (address) to the actual data.
Examples: class, interface, delegate, string, object
Value Types
Struct
-A value type that groups variables.
-Useful for small data structures.
Example:
```
public struct Point
{
   public int X { get; set; }
   public int Y { get; set; }
   public Point(int x, int y)
   {
      X = x;
      Y = y;
   }
}
```
Enum
A value type that defines named constants.
Example:
```
public enum Days
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
}
```
out and ref
ref: Passes arguments by reference, requiring initialization before passing.
out: Passes arguments by reference but does not require initialization before passing; must be assigned within the method.
Example:
```
void UpdateValue(ref int x, out int y)
{
   x += 10;
   y = 20;
}
int a = 5, b;
UpdateValue(ref a, out b);
Console.WriteLine($"a: {a}, b: {b}"); // Outputs: a: 15, b: 20
```
Nullable Types
Represent value types that can hold null.
Declared with a ? suffix.

Example:

```
int? num = null;
if (num.HasValue)
   Console.WriteLine(num.Value);
else
   Console.WriteLine("Value is null");
```

Nullable Reference Types

Introduced in C# 8.0 to avoid null reference errors.
Controlled via nullable context settings.

Example:

```
string? nullableString = null; // Nullable reference type
string nonNullableString = "Hello"; // Non-nullable reference type
```

Null-Coalescing Operators

??

Returns the left-hand operand if not null; otherwise, returns the right-hand operand.

Example:

```
string? name = null;
string displayName = name ?? "Default Name";
Console.WriteLine(displayName); // Outputs: Default Name
```

??=

Assigns the right-hand value to the left-hand variable only if the left-hand variable is null.

Example:

```
string? name = null;
name ??= "Default Name";
Console.WriteLine(name); // Outputs: Default Name
```

Working with Arrays

Single-Dimensional Array

```
int[] numbers = { 1, 2, 3, 4, 5 };
```

Multidimensional Array

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

Jagged Array

An array of arrays.

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

Array Class Members

-Array.Sort(): Sorts elements.
-Array.Reverse(): Reverses elements.
-Array.IndexOf(): Finds the index of an element.

Example:

```
int[] numbers = { 5, 3, 8, 1 };
Array.Sort(numbers);
Console.WriteLine(string.Join(", ", numbers)); // Outputs: 1, 3, 5, 8
```

Indices and Ranges

Indices

Use the ^ operator to specify elements from the end of the array.

Example:

```
int[] numbers = { 10, 20, 30, 40 };
Console.WriteLine(numbers[^1]); // Outputs: 40
```

Ranges

Use .. to define ranges of elements.

Example:

```
int[] numbers = { 10, 20, 30, 40, 50 };
int[] subset = numbers[1..3];
Console.WriteLine(string.Join(", ", subset)); // Outputs: 20, 30
```

Indexers

-Allow objects to be indexed like arrays.

-Defined using this keyword.

Example:

```
class Library
{
   private string[] books = { "Book1", "Book2", "Book3" };

   public string this[int index]
   {
      get { return books[index]; }
      set { books[index] = value; }
   }
}
// Usage:
Library library = new Library();
Console.WriteLine(library[1]); // Outputs: Book2
library[1] = "New Book2";
Console.WriteLine(library[1]); // Outputs: New Book2
```

Summary

-Value types are stored on the stack, while reference types are stored on the heap.

-Nullable types add flexibility to handle null for value types.

-Arrays support single, multidimensional, and jagged structures with utility methods via the Array class.

-Indexers enable array-like access to custom objects.

-Null-coalescing operators simplify null-checks and assignments.

1. Generic Classes

Generic classes provide type safety and reusability by allowing you to define a class with a placeholder for the type.

Example:

```
public class GenericClass<T>
{
   private T data;

   public void SetData(T value)
   {
      data = value;
   }

   public T GetData()
   {
      return data;
   }
}
// Usage
var genericInt = new GenericClass<int>();
genericInt.SetData(10);
Console.WriteLine(genericInt.GetData()); // Output: 10
```
2. Generic Methods

Generic methods allow you to define a method with a type parameter.

Example:
```
public class GenericMethods
{
   public void Print<T>(T value)
   {
      Console.WriteLine(value);
   }
}
// Usage
var gm = new GenericMethods();
gm.Print(123); // Output: 123
gm.Print("Hello"); // Output: Hello
```
3. Generic Constraints

Constraints restrict the types that can be used as arguments for a type parameter.

Example:
```
public class GenericWithConstraint<T> where T : class
{
   public T CreateInstance()
   {
      return Activator.CreateInstance<T>();
   }
}
```
4. Collections – Generic and Non-Generic

Generic Collections: Provide type safety (e.g., List<T>, Dictionary<TKey, TValue>).

Non-Generic Collections: Can store objects of any type, but lack type safety (e.g., ArrayList, Hashtable).

5. Collection Examples

ICollection (Generic and Non-Generic)
```
// Generic ICollection
ICollection<int> genericCollection = new List<int> { 1, 2, 3 };
genericCollection.Add(4);

// Non-Generic ICollection
System.Collections.ICollection nonGenericCollection = new System.Collections.ArrayList { 1,
"Two", 3.0 };
nonGenericCollection.Add(4);
```
IList (Generic and Non-Generic)
```
// Generic IList
IList<string> genericList = new List<string> { "A", "B", "C" };
genericList.Add("D");

// Non-Generic IList
System.Collections.IList nonGenericList = new System.Collections.ArrayList { "A", 1, 2.5 };
```

```
nonGenericList.Add("D");
```

IDictionary (Generic and Non-Generic)

```
// Generic IDictionary
IDictionary<int, string> genericDict = new Dictionary<int, string>
{
   {1, "One"},
   {2, "Two"}
};

// Non-Generic IDictionary
System.Collections.IDictionary nonGenericDict = new System.Collections.Hashtable();
nonGenericDict.Add(1, "One");
nonGenericDict.Add(2, "Two");
```

6. Iterating Collections Using foreach

```
var list = new List<int> { 1, 2, 3, 4 };

foreach (var item in list)
{
   Console.WriteLine(item); // Output: 1, 2, 3, 4
}
```

7. Using Tuples to Pass Multiple Values to a Function

Tuples allow you to return or pass multiple values.

Example:

```
public (int, string) GetData()
{
   return (1, "Hello");
}
// Usage
var result = GetData();
Console.WriteLine(result.Item1); // Output: 1
Console.WriteLine(result.Item2); // Output: Hello
```

1. Delegates

A delegate is a type-safe function pointer, allowing you to encapsulate a reference to a method.

Example:

```
public delegate void MyDelegate(string message);
public class Example
{
   public void ShowMessage(string message)
   {
      Console.WriteLine(message);
   }
}
// Usage
var example = new Example();
MyDelegate del = example.ShowMessage;
del("Hello, Delegates!"); // Output: Hello, Delegates!
```

2. Calling Methods Using Delegates

You invoke a method through a delegate instance.

Example:

```
MyDelegate del = example.ShowMessage;
del("This is a delegate call.");
```

3. Uses of Delegates

Event handling: Delegates are the foundation of events.
Callback mechanisms: Pass methods as parameters.
Multithreading: Execute methods asynchronously.
Encapsulation: Encapsulate method references.

4. Multicast Delegates

A delegate that can hold references to multiple methods.
Example:

```
public delegate void MultiDelegate(string message);
public class MulticastExample
{
   public void Method1(string message)
   {
      Console.WriteLine("Method1: " + message);
   }

   public void Method2(string message)
   {
      Console.WriteLine("Method2: " + message);
   }
}
// Usage
var multicast = new MulticastExample();
MultiDelegate del = multicast.Method1;
del += multicast.Method2;

del("Hello Multicast!");

// Output:
// Method1: Hello Multicast!
// Method2: Hello Multicast!
```

5. Action, Func, and Predicate Delegates

Action: Encapsulates methods with no return value.
Func: Encapsulates methods with a return value.
Predicate: Encapsulates methods returning a boolean.
Examples:

```
// Action
Action<string> action = message => Console.WriteLine("Action: " + message);
action("Hello Action");

// Func
Func<int, int, int> add = (x, y) => x + y;
Console.WriteLine("Func: " + add(3, 4)); // Output: 7

// Predicate
Predicate<int> isEven = num => num % 2 == 0;
Console.WriteLine("Predicate: " + isEven(4)); // Output: True
```

6. Anonymous Methods
An inline method with no name, assigned to a delegate.
Example:
```
Action<string> anonymous = delegate (string message)
{
   Console.WriteLine("Anonymous Method: " + message);
};
anonymous("Hello Anonymous!");
```
7. Lambda Expressions
Short syntax for writing inline methods or anonymous functions.
Example:
```
// Lambda with Action
Action<string> lambda = msg => Console.WriteLine("Lambda: " + msg);
lambda("Hello Lambda!");

// Lambda with Func
Func<int, int, int> multiply = (a, b) => a * b;
Console.WriteLine("Lambda Func: " + multiply(3, 5)); // Output: 15
```
By mastering these concepts, you can leverage delegates and lambdas effectively for event-driven programming, functional-style coding, and simplifying callback mechanisms.

Error Handling (Exception Handling)
Exception handling ensures your program can gracefully recover from errors or unexpected situations.
1. Checked and Unchecked Statements
<u>Checked Statements:</u> Explicitly enable overflow checking during arithmetic operations.
<u>Unchecked Statements:</u> Ignore overflow checking.
Example:
```
// Checked
checked
{
   int max = int.MaxValue;
   // This will throw an OverflowException
   int result = max + 1;
}

// Unchecked
unchecked
{
   int max = int.MaxValue;
   // Overflow occurs but no exception is thrown
   int result = max + 1;
   Console.WriteLine(result); // Wraps around to negative value
}
```
2. The try, catch, finally
Use try for code that might throw exceptions, catch to handle them, and finally for cleanup.
Example:
```
try
{
   int a = 10;
```

```
    int b = 0;
    int result = a / b; // Will throw DivideByZeroException
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Error: Division by zero is not allowed.");
}
finally
{
    Console.WriteLine("Cleanup code runs here.");
}
```

3. Dos & Don’ts of Exception Handling

Dos:

Catch specific exceptions.
Use finally for releasing resources.
Log exceptions for debugging.
Rethrow exceptions if necessary using throw;.

Don’ts:

Avoid catching System.Exception unless necessary.
Do not suppress exceptions without handling.
Avoid throwing exceptions in performance-critical code.
Do not use exceptions for control flow.

4. User-Defined Exception Classes

You can create custom exception classes by inheriting from Exception.
Example:

```
public class CustomException : Exception
{
    public CustomException(string message) : base(message)
    {
    }
}
// Usage
try
{
    throw new CustomException("This is a custom exception.");
}
catch (CustomException ex)
{
    Console.WriteLine(ex.Message);
}
```

5. Declaring and Raising Events

Events are based on delegates and are used to notify when something occurs.
Example:

```
public class Publisher
{
    public delegate void NotifyEventHandler(string message);
    public event NotifyEventHandler Notify;

    public void RaiseEvent(string message)
    {
        Notify?.Invoke(message); // Raise the event
    }
}
public class Subscriber
{
    public void OnNotify(string message)
    {
        Console.WriteLine("Event received: " + message);
    }
}
// Usage
var publisher = new Publisher();
var subscriber = new Subscriber();

// Subscribe to event
publisher.Notify += subscriber.OnNotify;

// Raise event
publisher.RaiseEvent("Hello Event!");
```

6. Handling Events

Events are handled by subscribing methods to the event.

Key Steps:

-Define a delegate for the event.
-Declare the event using the delegate.
-Raise the event using Invoke.
-Handle the event by subscribing methods.

Example of Multiple Handlers:

```
publisher.Notify += msg => Console.WriteLine("Handler 1: " + msg);
publisher.Notify += msg => Console.WriteLine("Handler 2: " + msg);

publisher.RaiseEvent("Event with multiple handlers!");
```

By mastering these aspects of error handling and events, you can write robust, maintainable, and responsive code.

1. Anonymous Types

Anonymous types allow you to create objects without explicitly defining their type.

Example:

```
var person = new { Name = "Alice", Age = 25 };
Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");
// Output: Name: Alice, Age: 25
```

2. Extension Methods

Extension methods enable you to add methods to existing types without modifying them.

Example:

```
public static class StringExtensions
{
    public static string ReverseString(this string str)
    {
        return new string(str.Reverse().ToArray());
    }
}
```

```
// Usage
string original = "hello";
Console.WriteLine(original.ReverseString()); // Output: olleh
```

3. Partial Classes

Partial classes allow a single class to be split across multiple files.

File 1:

```
public partial class Example
{
   public void Method1()
   {
      Console.WriteLine("Method1");
   }
}
```

File 2:

```
public partial class Example
{
   public void Method2()
   {
      Console.WriteLine("Method2");
   }
}
```

Usage:

```
var obj = new Example();
obj.Method1();
obj.Method2();
```

4. Partial Methods

Partial methods allow you to declare methods in one part of a partial class and implement them in another.

Example:

```
public partial class Example
{
   partial void OnStart();
   public void Start()
   {
      OnStart();
   }
}
public partial class Example
{
   partial void OnStart()
   {
      Console.WriteLine("OnStart called");
   }
}
```

5. LINQ to Objects

LINQ to Objects lets you query collections like arrays, lists, etc., using LINQ.

Example:

```
int[] numbers = { 1, 2, 3, 4, 5 };
var evenNumbers = from num in numbers
         where num % 2 == 0
```

```
              select num;
foreach (var num in evenNumbers)
{
   Console.WriteLine(num); // Output: 2, 4
}
```
6. Writing LINQ Queries
LINQ queries can be written in two ways:
Query Syntax:
```
var query = from num in numbers
          where num > 2
          select num;
```
Method Syntax:
```
var query = numbers.Where(num => num > 2);
```
7. Deferred Execution
Deferred execution means a query is not executed until you enumerate it.
Example:
```
var query = numbers.Where(num => num > 2); // Query is not executed here
foreach (var num in query)             // Query is executed here
{
   Console.WriteLine(num);
}
```
8. LINQ Methods
Common LINQ methods include:
Filtering: Where()
Projection: Select()
Sorting: OrderBy(), OrderByDescending()
Aggregates: Sum(), Count(), Max(), Min()
Joining: Join()
Example:
```
var result = numbers.Where(n => n % 2 == 0).Select(n => n * n);
```
9. PLINQ (Parallel LINQ)
PLINQ enables parallel execution of LINQ queries for better performance on multi-core systems.
Example:
```
var numbers = Enumerable.Range(1, 1000);
var evenNumbers = numbers.AsParallel().Where(n => n % 2 == 0).ToArray();

Console.WriteLine(string.Join(", ", evenNumbers.Take(10))); // Parallel processing
```
These features collectively help write efficient, readable, and maintainable C# code.

1. Creating a Shared Assembly
A shared assembly is a reusable .NET assembly that can be shared across multiple applications.
Steps to Create a Shared Assembly:
1)Create the Assembly:
Write your class library and compile it.
```
public class SharedLibrary
{
   public static void SayHello()
   {
        Console.WriteLine("Hello from Shared Library!");
    }
}
```
Compile as a .dll.

2)Strong Name the Assembly:

-Generate a strong name key file using the sn.exe tool:
```
sn -k keyfile.snk
```
-Add the strong name to the assembly:
```
[assembly: AssemblyKeyFile("keyfile.snk")]
```
3)Add to Global Assembly Cache (GAC):

-Use the gacutil.exe tool to add the assembly to GAC:
```
gacutil -i YourAssembly.dll
```

2. Creating Custom Attributes

Custom attributes allow you to attach metadata to program elements.

Example:
```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class CustomAttribute : Attribute
{
    public string Description { get; }
    public CustomAttribute(string description)
    {
        Description = description;
    }
}
// Usage
[Custom("This is a custom attribute")]
public class ExampleClass
{
}
```
3. Using Reflection to Explore an Assembly

Reflection allows you to inspect metadata and explore an assembly at runtime.

Example:
```
using System.Reflection;
Assembly assembly = Assembly.GetExecutingAssembly();
Console.WriteLine("Assembly: " + assembly.FullName);
foreach (Type type in assembly.GetTypes())
{
    Console.WriteLine("Type: " + type.Name);
    foreach (MethodInfo method in type.GetMethods())
    {
        Console.WriteLine("  Method: " + method.Name);
    }
}
```
4. Using Reflection to Load an Assembly Dynamically

You can load an assembly at runtime and invoke its methods.

Example:
```
Assembly assembly = Assembly.LoadFrom("YourLibrary.dll");
Type type = assembly.GetType("Namespace.ClassName");
MethodInfo method = type.GetMethod("MethodName");
```

```
object instance = Activator.CreateInstance(type);
method.Invoke(instance, null);
```

5. Files I/O and Streams

Working with Drives, Directories, and Files

Example:

```
// List drives
foreach (var drive in DriveInfo.GetDrives())
{
   Console.WriteLine($"Drive: {drive.Name}, Type: {drive.DriveType}");
}
// Create a directory
string path = @"C:\ExampleDirectory";
if (!Directory.Exists(path))
{
   Directory.CreateDirectory(path);
}
// Create a file
string filePath = Path.Combine(path, "example.txt");
File.WriteAllText(filePath, "Hello, File I/O!");
```

Reading and Writing Files

Reading a File:

```
string content = File.ReadAllText(filePath);
Console.WriteLine("File Content: " + content);
```

Writing to a File:

```
File.WriteAllText(filePath, "Updated content!");
```

Using Streams:

Writing to a File with Stream:

```
using (FileStream fs = new FileStream(filePath, FileMode.Create))
using (StreamWriter writer = new StreamWriter(fs))
{
   writer.WriteLine("Hello using StreamWriter!");
}
```

Reading a File with Stream:

```
using (FileStream fs = new FileStream(filePath, FileMode.Open))
using (StreamReader reader = new StreamReader(fs))
{
   string line;
   while ((line = reader.ReadLine()) != null)
   {
      Console.WriteLine(line);
   }
}
```

These tools and techniques give you a solid foundation for managing assemblies, working with attributes, exploring reflection, and handling file I/O operations in .NET applications.

1. Threading

ThreadStart and ParameterizedThreadStart

Threads are created using the Thread class with ThreadStart or ParameterizedThreadStart delegates.

Example: ThreadStart

```
using System.Threading;
void PrintMessage()
{
   Console.WriteLine("Hello from Thread!");
}
Thread thread = new Thread(new ThreadStart(PrintMessage));
thread.Start();
```

Example: ParameterizedThreadStart

```
void PrintMessage(object message)
{
   Console.WriteLine($"Message: {message}");
}
Thread thread = new Thread(new ParameterizedThreadStart(PrintMessage));
thread.Start("Hello with Parameter!");
```

ThreadPool

The ThreadPool manages a pool of worker threads to efficiently execute tasks.

Example:

```
using System.Threading;
ThreadPool.QueueUserWorkItem(state =>
{
   Console.WriteLine($"Task executed on thread pool. State: {state}");
}, "ExampleState");
```

Synchronizing Critical Data

Using lock: Ensures only one thread can access a critical section at a time.

```
object lockObj = new object();
int counter = 0;

void Increment()
{
   lock (lockObj)
   {
      counter++;
      Console.WriteLine($"Counter: {counter}");
   }
}
```

Using Monitor: Provides more control over synchronization.

```
Monitor.Enter(lockObj);
try
{
   counter++;
   Console.WriteLine($"Counter: {counter}");
}
finally
{
   Monitor.Exit(lockObj);
}
```

Using Interlocked: For atomic operations like incrementing or decrementing.

```
Interlocked.Increment(ref counter);
Console.WriteLine($"Counter: {counter}");
```

2. Working with Tasks

Calling Functions with and without Return Values

Without Return Value:

```
using System.Threading.Tasks;
Task.Run(() =>
{
   Console.WriteLine("Task without return value");
});
```

With Return Value:

```
Task<int> task = Task.Run(() =>
{
   return 42;
});
Console.WriteLine($"Task result: {task.Result}");
```

Using async and await

async and await enable asynchronous programming for better performance and responsiveness.

Example:

```
async Task<int> FetchDataAsync()
{
   await Task.Delay(1000); // Simulates delay
   return 42;
}
async Task RunAsync()
{
   int result = await FetchDataAsync();
   Console.WriteLine($"Result: {result}");
}
RunAsync().Wait();
```

3. Using the Task Parallel Library (TPL)

The TPL provides high-level APIs for parallel programming.

Parallel.For and Parallel.ForEach:

```
using System.Threading.Tasks;
// Parallel.For
Parallel.For(0, 10, i =>
{
   Console.WriteLine($"Processing {i}");
});
// Parallel.ForEach
string[] items = { "Item1", "Item2", "Item3" };
Parallel.ForEach(items, item =>
{
   Console.WriteLine($"Processing {item}");
});
```

Task.WhenAll: Waits for multiple tasks to complete.

```
Task task1 = Task.Run(() => Console.WriteLine("Task 1"));
Task task2 = Task.Run(() => Console.WriteLine("Task 2"));

Task.WhenAll(task1, task2).Wait();
```

Example of Complex Task Execution:

```
Task<int> task1 = Task.Run(() => 10);
Task<int> task2 = Task.Run(() => 20);

Task.WhenAll(task1, task2).ContinueWith(t =>
{
   int sum = task1.Result + task2.Result;
   Console.WriteLine($"Sum: {sum}");
}).Wait();
```

These threading and task management tools provide powerful capabilities for concurrent programming, ensuring efficient execution and synchronization.

Introduction to ASP.NET MVC Core

<u>1. Architecture of an ASP.NET MVC Application</u>

ASP.NET MVC follows the Model-View-Controller pattern:
Model: Represents application data and business logic.
View: Displays data to the user and collects user input.
Controller: Handles user input, manipulates the model, and returns a response.

<u>2. Understanding Folder Structures and Configuration Files</u>

<u>Folder Structure:</u>

Controllers: Contains controller classes that handle requests.
Views: Holds Razor view files for rendering UI.
Models: Contains model classes representing application data.
wwwroot: Stores static files like CSS, JS, and images.
Startup.cs: Configures the application services and middleware.

<u>Key Configuration Files:</u>

appsettings.json: Stores application configuration settings.
Program.cs and Startup.cs: Initialize and configure the application.

Understanding Controllers and Actions

<u>1. Creating a Controller</u>

Controllers handle incoming requests and return responses.
Example:

```
public class HomeController : Controller
{
   public IActionResult Index()
   {
      return View();
   }
}
```

<u>2. How Actions Are Invoked</u>

An action method in a controller is invoked via URL routing.
Example: /Home/Index calls the Index method in the HomeController.

<u>3. Attributes for Actions</u>

HttpGet: Marks a method to handle GET requests.
HttpPost: Marks a method to handle POST requests.
NoAction: Prevents a method from being invoked as an action.

<u>Example:</u>

```
[HttpPost]
public IActionResult SubmitForm() { ... }

[HttpGet]
```

```
public IActionResult GetForm() { ... }

[NonAction]
public void HelperMethod() { ... }
```

Understanding Views and Models

1. Creating Models and ViewModels

Model:

```
public class Product
{
   public int Id { get; set; }
   public string Name { get; set; }
   public decimal Price { get; set; }
}
```

ViewModel:

```
public class ProductViewModel
{
   public Product Product { get; set; }
   public List<Category> Categories { get; set; }
}
```

2. Creating Razor Views

Razor views use .cshtml files to render UI:

```
@model Product
<h1>@Model.Name</h1>
<p>Price: @Model.Price</p>
```

3. Using ViewBag

Dynamic object for passing data from controller to view.

```
ViewBag.Message = "Hello, ViewBag!";
return View();
```

In View:

```
<p>@ViewBag.Message</p>
```

4. Validations

Using Data Annotations:

```
public class Product
{
   [Required]
   public string Name { get; set; }

   [Range(0.01, 1000)]
   public decimal Price { get; set; }
}
```

Client-Side and Server-Side Validation:

1)Client-Side: Automatic when using Razor and jQuery validation libraries.
2)Server-Side: Validates using model binding in the controller.

MVC State Management

ViewBag: Temporary, lives during the request.
TempData: Temporary data stored between requests.
Session: Stores data for a user session.
Cookies: Stores data in the client browser.
QueryString: Passes data in the URL.

MVC Module

1. Partial Views

Reusable UI components.

```
@Html.Partial("_PartialViewName")
```

Data Management with ADO.NET

Key Objects:

Connection: Manages database connection.
Command: Executes SQL queries.
DataReader: Reads data from the database.
DataAdapter/DataSet/DataTable: Used for disconnected data.

Example:

```
using (var connection = new SqlConnection("connectionString"))
{
   connection.Open();
   SqlCommand command = new SqlCommand("SELECT * FROM Products", connection);
   SqlDataReader reader = command.ExecuteReader();
   while (reader.Read())
   {
      Console.WriteLine(reader["Name"]);
   }
}
```

Understanding Routing and Request Lifecycle

1. Routing

RouteConfig: Configures routing patterns.

```
app.UseEndpoints(endpoints =>
{
   endpoints.MapControllerRoute(
      name: "default",
      pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

Layouts, Bundling, and Minification

1. Layouts

Defines a common structure for pages.

```
<!DOCTYPE html>
<html>
<head>
   <title>@ViewData["Title"]</title>
</head>
<body>
   @RenderBody()
</body>
</html>
```

2. Bundling and Minification

Improves performance by combining and minifying CSS/JS.

BundleConfig:

```
BundleTable.Bundles.Add(new ScriptBundle("~/bundles/js").Include("~/Scripts/*.js"));
```

MVC Security

Using Authorize and AllowAnonymous

Authorize: Restricts access to authenticated users.
AllowAnonymous: Overrides Authorize to allow public access.

Example:

```
[Authorize]
public IActionResult SecureAction() { ... }

[AllowAnonymous]
public IActionResult PublicAction() { ... }
```

Authentication and Security

1)Forms-Based Authentication

A technique to manage user authentication in web applications.

Steps:

-Users log in via a form that submits credentials to the server.
-If authenticated, an authentication ticket is issued and stored in a cookie.
-Subsequent requests use this ticket to validate the user session.

Configuration in web.config:

```
<authentication mode="Forms">
  <forms loginUrl="Login.aspx" timeout="30" />
</authentication>
```

Ensure secure storage of passwords using hashing (e.g., SHA-256).

2)Preventing Forgery Attacks using AntiForgeryToken

-Cross-Site Request Forgery (CSRF): An attacker tricks a user into performing unintended actions on a website they're authenticated on.
-Solution:
Use @Html.AntiForgeryToken() in forms.
Validate the token in the controller using [ValidateAntiForgeryToken].

-Example:

```
@using (Html.BeginForm()) {
  @Html.AntiForgeryToken()
  <input type="text" name="Name" />
  <button type="submit">Submit</button>
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Submit(FormModel model) {
  // Process form
}
```

3)Preventing Cross-Site Scripting (XSS) Attacks

-XSS occurs when attackers inject malicious scripts into a webpage.

-Prevention:

 -Use @Html.Encode() or the @ Razor syntax (e.g., @Model.Property) to automatically encode output.
 -Avoid rendering untrusted HTML directly; if necessary, sanitize it.
 -Use Content Security Policy (CSP) headers to restrict script execution.

Entity Framework (EF)

Introduction to EF

-EF is an Object-Relational Mapping (ORM) tool for .NET that simplifies database interactions.
-It allows developers to interact with a database using strongly typed .NET objects instead of raw SQL.

Different Approaches

1)Database First: Start with an existing database. EF generates models based on the database schema.
2)Model First: Define the database schema using a visual designer, then generate the database.
3)Code First: Write .NET classes as models and let EF generate the database schema dynamically.

Code First Approach

Define entities as POCO (Plain Old CLR Object) classes.

Example:

```
public class Product {
   public int ProductId { get; set; }
   public string Name { get; set; }
   public decimal Price { get; set; }
}
```

Using Data Annotations

-Apply attributes to model properties for validation and constraints.

-Example:

```
public class Product {
   [Key]
   public int ProductId { get; set; }

   [Required]
   [StringLength(100)]
   public string Name { get; set; }

   [Range(0.01, 1000.00)]
   public decimal Price { get; set; }
}
```

Using Fluent APIs

-Configure EF behavior programmatically in the OnModelCreating method.

-Example:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder) {
   modelBuilder.Entity<Product>()
      .HasKey(p => p.ProductId)
      .Property(p => p.Name)
      .HasMaxLength(100)
      .IsRequired();
}
```

Database Migrations

-Track and apply changes to the database schema over time.

-Commands:

   -Add migration: Add-Migration MigrationName
   -Apply migration: Update-Database

-Example:

```
Add-Migration AddProductsTable
Update-Database
```

CRUD Operations using EF

Examples:

```
// Create
var product = new Product { Name = "Laptop", Price = 1000 };
```

```
context.Products.Add(product);
context.SaveChanges();

// Read
var products = context.Products.ToList();

// Update
var productToUpdate = context.Products.First();
productToUpdate.Price = 1200;
context.SaveChanges();

// Delete
var productToDelete = context.Products.First();
context.Products.Remove(productToDelete);
context.SaveChanges();
```

Developing MVC Application Using EF Code First Approach

Steps:

1)Create a new ASP.NET MVC project.
2)Add EF models and configure the database context.
3)Use EF migrations to set up the database.
4)Implement controllers and views for CRUD operations.
5)Use scaffolding to generate views and controllers automatically.

Introduction to Razor Pages

-Razor Pages is a feature in ASP.NET Core for building page-focused web apps.
-Benefits:
 -Simpler than MVC for scenarios where each page has its logic.
 -Supports Page Model classes for cleaner separation of UI and logic.
-Example:

```
public class IndexModel : PageModel {
   public string Message { get; set; }

   public void OnGet() {
      Message = "Hello, Razor Pages!";
   }
}
@page
@model IndexModel
<h1>@Model.Message</h1>
```

Localization in MVC (Demo)

Localization in ASP.NET MVC allows applications to support multiple languages and regions. Here's how to implement it:

Steps for Localization in MVC

1)Add Resource Files
 -Create .resx files for different languages:
  -Resources.resx (Default language, e.g., English)
  -Resources.fr.resx (French)
 -Store them in the App_GlobalResources folder or another location.
2)Define Resource Keys and Values

-Example:
-Key: Greeting
-Value (English): Hello!
-Value (French): Bonjour!

3)Access Resources in Views

-Use @Resources.ResourceName in Razor views.
-Example:

```
<p>@Resources.Greeting</p>
```

4)Set the Culture in the Controller

-Set the thread culture based on user preferences or browser settings.
-Example in Global.asax:

```
protected void Application_BeginRequest() {
   var culture = HttpContext.Current.Request.UserLanguages?[0] ?? "en-US";
   Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
   Thread.CurrentThread.CurrentUICulture = new CultureInfo(culture);
}
```

5)Change Culture Dynamically

Provide a dropdown or links for users to switch languages.
Example:

```
public ActionResult ChangeLanguage(string lang) {
   Thread.CurrentThread.CurrentCulture = new CultureInfo(lang);
   Thread.CurrentThread.CurrentUICulture = new CultureInfo(lang);
   return RedirectToAction("Index");
}
```

Deploying ASP.NET MVC Application (Demo)

Deploying an ASP.NET MVC application involves publishing the application to a hosting environment. Here's a quick guide:

Steps for Deployment

-Publish the Application

-In Visual Studio:
1)Right-click on the project and select Publish.
2)Choose a target (Azure, IIS, Folder, etc.).
3)Configure settings (e.g., server, credentials, destination path).
4)Click Publish to generate the deployment package.

-Deploy to IIS (Internet Information Services)

-Install IIS:
-Enable it via "Turn Windows Features On or Off".
-Set Up a Website in IIS:
-Open IIS Manager.
-Add a new website.
-Specify the site name, physical path (where the published files are stored), and port.
-Configure .NET Framework:
Ensure the correct version of .NET is installed and linked to the application pool.

-Database Deployment

-Deploy the database using SQL Server tools or scripts.
-Update the connection string in Web.config.

-Test the Application

-Access the application in a browser using the deployed URL.
-Verify all features work as expected.

-Optional: Deploy to Azure

-Use Visual Studio’s Publish to Azure option.
-Set up an Azure App Service and deploy directly.

1. Creating an ASP.NET MVC Web API

Steps to Create a Web API:

1.Create a New Web API Project
In Visual Studio, select ASP.NET Web Application and choose the Web API template.

2.Add a Controller
Create a new Web API controller.
Example:

```
public class ProductsController : ApiController {
   private static List<string> products = new List<string> { "Laptop", "Phone", "Tablet" };

   // GET: api/products
   public IEnumerable<string> Get() {
      return products;
   }

   // GET: api/products/{id}
   public string Get(int id) {
      return products[id];
   }

   // POST: api/products
   public void Post([FromBody] string product) {
      products.Add(product);
   }

   // DELETE: api/products/{id}
   public void Delete(int id) {
      products.RemoveAt(id);
   }
}
```

3.Test the API
Use tools like Postman or Swagger to test API endpoints.

2. Configuring for CORS

What is CORS?

Cross-Origin Resource Sharing (CORS) allows a server to specify which domains can access its resources.

Enable CORS in Web API:

-Install the Microsoft.AspNet.WebApi.Cors package:

```
Install-Package Microsoft.AspNet.WebApi.Cors
```

-Enable CORS globally in WebApiConfig.cs:

```
public static class WebApiConfig {
   public static void Register(HttpConfiguration config) {
      config.EnableCors();
      config.MapHttpAttributeRoutes();
   }
}
```

-Apply CORS to controllers or actions:

```
[EnableCors(origins: "*", headers: "*", methods: "*")]
public class ProductsController : ApiController {
   // Actions
}
```
3. Different HTTP Verbs

Common Verbs and Their Usage:

GET: Retrieve data.
POST: Create new resources.
PUT: Update existing resources.
DELETE: Remove resources.
PATCH: Partially update resources.

Example Endpoints:
```
[HttpGet]
public string Get(int id) { return "data"; }

[HttpPost]
public void Post([FromBody] string value) { }

[HttpPut]
public void Put(int id, [FromBody] string value) { }

[HttpDelete]
public void Delete(int id) { }
```
4. Consuming Web API Using a Client

Using HttpClient (C#)

1.Install System.Net.Http.
2.Example:
```
using System.Net.Http;
using System.Threading.Tasks;
class Program {
   static async Task Main(string[] args) {
      HttpClient client = new HttpClient();
      var response = await client.GetStringAsync("https://api.example.com/products");
      Console.WriteLine(response);
   }
}
```
Using JavaScript (Fetch API)
```
fetch('https://api.example.com/products')
   .then(response => response.json())
   .then(data => console.log(data));
```
Using Postman

Create a request, specify the HTTP method, and test the API.

5. Using Newtonsoft.Json APIs

Purpose of Newtonsoft.Json

A popular library for JSON serialization and deserialization in .NET.

Install Newtonsoft.Json
```
Install-Package Newtonsoft.Json
```
-Usage Examples

-Serialize Objects to JSON:

```
using Newtonsoft.Json;
var product = new { Name = "Laptop", Price = 1000 };
string json = JsonConvert.SerializeObject(product);
Console.WriteLine(json);
```

-Deserialize JSON to Objects:

```
string json = "{ \"Name\": \"Laptop\", \"Price\": 1000 }";
var product = JsonConvert.DeserializeObject<dynamic>(json);
Console.WriteLine(product.Name);
```

-Formatting Options:

```
string formattedJson = JsonConvert.SerializeObject(product, Formatting.Indented);
```