Asp.NET MVC Core

Trainer : Priyanka R Rangole

Email: priyanka.rangole@sunbeaminfo.com

What we will Going to cover

- **ASP.NET MVC using Razor Pages**
    - Build dynamic, interactive web applications
    - Learn MVC architecture & separation of concerns
    - Create forms, validations, and responsive UI
- **Git & GitHub**
    - Version control best practices
    - Track changes, collaborate, and push projects online
- **Database (SQL Server)**
    - Store, retrieve, and manage application data
    - Understand relationships, normalization, and queries
    - Integrate database with .NET applications

Core Technologies Overview

- **ASP.NET Core MVC**
    - Build structured web apps using **Model-View-Controller**
    - Clean separation of **business logic, UI, and data**
- **Razor Pages for UI**
    - Dynamic web pages with **C# + HTML**
    - Layout pages, sections, and form handling
    - Integrate **Bootstrap** for responsive design
- **Database (SQL Server)**
    - Store & manage data for applications
    - Create tables, relationships, and queries
    - Connect to .NET applications using **Entity Framework Core**
- **Basic Technologies**
- ASP.NET Core MVC +C#+ Razor Pages
- SQL Server
- Entity Framework Core
- Git / GitHub
- Visual Studio / SSMS

Asp .NET Core

+

Razor pages

+

Sql Server

github

What is ASP.NET Core?

- Modern, **cross-platform framework** for building web apps and APIs
- Supports **MVC (Model-View-Controller)** pattern
- Lightweight, fast, and modular
- Works with **Razor Pages** for dynamic UI
- Integrates with **SQL Server** and other databases
- Built-in **dependency injection** and middleware support
- Highly suitable for **enterprise-level web applications**

Introduction to C#

- **C# (pronounced “C-Sharp”)** is a simple, modern, and powerful **object-oriented programming language** developed by **Microsoft**.

- It is widely used for building different types of applications such as:
    - **Web applications**
    - **Windows desktop applications**
    - **Console applications**
    - **Games** (using Unity)
    - **APIs and cloud services**

- C# programs are usually developed using **Visual Studio**, which provides tools for writing, testing, and debugging code.

Introduction to C#

- **What is C#?**
  - **C# (C-Sharp)** is a **simple, modern, and powerful** programming language.
  - Developed by **Microsoft** as part of the **.NET Framework**.
  - Fully **object-oriented** (supports classes, objects, inheritance, encapsulation, etc.).

- **Why C#?**
  - Easy to learn and beginner-friendly
  - Clean and readable syntax
  - Strong memory management
  - High performance
  - Huge community support
  - Works seamlessly with Visual Studio

Introduction to Visual Studio

1. What is Visual Studio?

•**Visual Studio** is a **powerful Integrated Development Environment (IDE)** developed by **Microsoft**.
•Used to create **C#, .NET, ASP.NET Core, web, desktop, and console applications**.
•Provides a **complete environment** for writing, debugging, and running programs.

2. Why Use Visual Studio?

•Easy to **write and manage code**
•Built-in **IntelliSense** (code suggestions)
•**Debugging tools** to find and fix errors
•Supports **multiple project types** (web, desktop, console, mobile)
•Integrated **Git tools** for version control
•Works with **NuGet packages** to add libraries easily

4. Visual Studio Features

•**Project Templates**: Quickly start Console, Web, or Desktop apps
•**Code Navigation**: Jump between classes, methods, and files
•**Refactoring**: Easily rename or modify code structure
•**Debugging**: Set breakpoints, watch variables, step through code
•**Extensions**: Add productivity tools like ReSharper, GitHub, etc.

Introduction to Classes & Objects

- **Class:** A blueprint or template to create objects.
- **Object:** An instance of a class, represents real-world entities.

- **Why Classes & Objects?**
- Organize code
- Reuse functionality
- Model real-world entities

```
class Student
{
   public string Name;
   public int Age;
}
Student s1 = new Student();
s1.Name = “RIya";
s1.Age = 20;
```

1. What is a Class?

- A **class** is a **blueprint/template** that defines how an object should look and behave.
- It contains **attributes (data)** and **functionalities (methods)**.

Real-life example

Think about a **Car**:

- **Attributes:** 4 wheels, doors, steering, color
- **Functionalities:** start, stop, run

The **Car** is a *class*

Each **actual car you see on the road** is an *object* of that class.

. Class in C#

We create a class using the **class** keyword.

A class can contain:

- Fields
- Properties
- Methods
- Constructors
- Events
- Access Modifiers (public, private, protected, internal)

These together are called **class members**.

Fields

- A **field** is a variable declared inside a class.
- Mostly kept **private** for security.

```
class Student
{
    public int id;
}
```

Properties

- Properties give **controlled access** to fields.
- They use **get** to read and **set** to assign value

```
class Student
{
    private int id;

    public int StudentId
    {
        get { return id; }
        set { id = value; }
    }
}
```

Auto-Implemented Properties

```
class Student
{
   public string FirstName { get; set; }
   public string LastName  { get; set; }
}
```

- Saves time
- Cleaner code
- Compiler creates the private backing field automatically

Methods

- A method performs an **action** or **operation**.
- It may or may not return a value.

```
return-type MethodName(parameters)
{
   // statements
}
```

Example:with return type

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

Example:with no return type

```
public void Greet()
{
   Console.WriteLine("Hello World!");
}
```

Constructors

- A constructor is a **special method** that runs automatically when an object is created.
- Name must be **same as class name**.
- Cannot have a return type.

```
class Student
{
    public Student()
    {
        // initialize values
    }
}
```

- Can be **public, private, or protected**
- You can have **multiple constructors** (overloading)
- Only **one** constructor can be parameterless
- If you don’t create one, C# will automatically create a **default constructor**

Creating Objects

To create an object, use the **new keyword**.

```
Student s1 = new Student();
```

Accessing Members Using Objects

```
Student mystudent = new Student();
mystudent.FirstName = "Steve";
mystudent.LastName = "Jobs";

mystudent.GetFullName();
```

Multiple Objects Example

```
Student s1 = new Student();
s1.FirstName = "Steve";
s1.LastName = "Jobs";

Student s2 = new Student();
s2.FirstName = "Bill";
s2.LastName = "Gates";
```

- Every object has its **own values**, but shares the **same class structure**.

C# Namespace

What is a Namespace?

A **namespace** is a container used to organize related **classes, interfaces, structs, enums, and other namespaces**.

Why do we use namespaces?

- To **group related classes together**
- To **avoid name conflicts**
- To **improve code organization**
- To allow same class names in different namespaces

- **Declaring a Namespace**

```
namespace School
{
   // classes, interfaces, etc.
}
```

- **Namespace with Multiple Classes**

```
namespace School
{
 class Student { }
 class Course { }
 }
```

- Access using **namespace.classname**:

```
School.Student std = new School.Student();
School.Course cs = new School.Course();
```

- **using Keyword (Importing a Namespace**)

```
using School;

class Program
{
    static void Main(string[] args)
    {
        Student std = new Student();   // No need for School.
    }
}
```

Nested Namespaces (Inner Namespaces)

C# allows namespaces inside other namespaces using dot notation:

```
namespace School.Education
{
    class Student
    {
    }
}
```

Fully qualified name:

School.Education.Student

C# 10 Feature: File-Scoped Namespace

No need for braces { }.

Applies to entire file.

```
namespace School.Education;

class Student
{
}
```

What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is a software development approach that uses **real-world concepts** to build applications.
It models the software using **classes** and **objects** that interact with each other.

Benefits of OOP

- ✓ **Flexible** — easy to change or add features
- ✓ **Reusable** — components/classes can be reused
- ✓ **Organized** — code is well-structured
- ✓ **Maintainable** — easier debugging and testing

- ## Basic Building Blocks of OOP

- **Classes**

A **class** defines the blueprint for an object (structure + behavior).
It contains:

•**Fields/Properties** -> data

•**Methods** -> behavior

- **Methods**

A **method** represents an action/behavior of a class.
A method can:

•Perform operations

•Update object data

•Return results

- **Properties**

Properties store data for an object **during program execution**.
They provide **controlled access** to fields using get/set.

- **Objects**

An **object** is an **instance of a class**.
Different objects of the same class hold **different data**.

Major Pillars of OOP

Abstraction

- getting only essential things and hiding unnecessary details is called as abstraction.
- Abstraction always describe outer behavior of object.
- In console application when we give call to function in to the main function , it represents the abstraction. • By Creating object and calling public member function on it we can achieve abstraction.

Encapsulation

- binding of data and code together is called as encapsulation. By defining class we can achieve encapsulation.
- Implementation of abstraction is called encapsulation.
- Encapsulation always describe inner behavior of object
- Function call is abstraction and Function definition is encapsulation.
- Information hiding :- Hiding information from user is called information hiding.
- In c# we used access Specifier to provide information hiding.

Modularity

- Dividing programs into small modules for the purpose of simplicity is called modularity.

Hierarchy

- Hierarchy is ranking or ordering of abstractions.
- Main purpose of hierarchy is to achieve re-usability.
- Types –> 1: Inheritance [is-a] , 2: Association [has-a]

Minor pillars of oops

Polymorphism (Typing)

- One interface having multiple forms is called as polymorphism.
- Polymorphism have two types

1. **Compile time polymorphism**:

when the call to the function resolved at compile time it is called as compile time polymorphism. And it is achieved by using
function overloading

2. **Runtime polymorphism** :-

when the call to the function resolved at run time it is called as run time polymorphism. And it is achieved by using function overriding.

- **What is Concurrency?**
  - **Concurrency** means performing **multiple tasks simultaneously**.
  - It helps applications to run faster and utilize **CPU cores efficiently.**

How to achieve Concurrency in C#?

- Using Multithreading
- Perform multiple operations at the same time.
- Efficient use of hardware resources.

Tools in C#:

- Thread class
- Task Parallel Library (TPL)
- async/await
- Parallel.For, Parallel.ForEach

Persistence in C#

✓ What is Persistence?

- Persistence means **preserving the state of an object across time and storage**.
- Even after program ends, data is stored and can be retrieved later.

Ways to achieve Persistence

✓ **File Handling**
✓ **Serialization** (Convert object → stream of bytes)
✓ **Deserialization** (stream → object)
✓ **Databases (SQL Server, etc.)**
✓ **Socket Programming** (for transferring object data across network)

Serialization Methods in C#:

- **Binary serialization**
- **XML serialization**
- **JSON serialization (System.Text.Json / Newtonsoft.Json)**