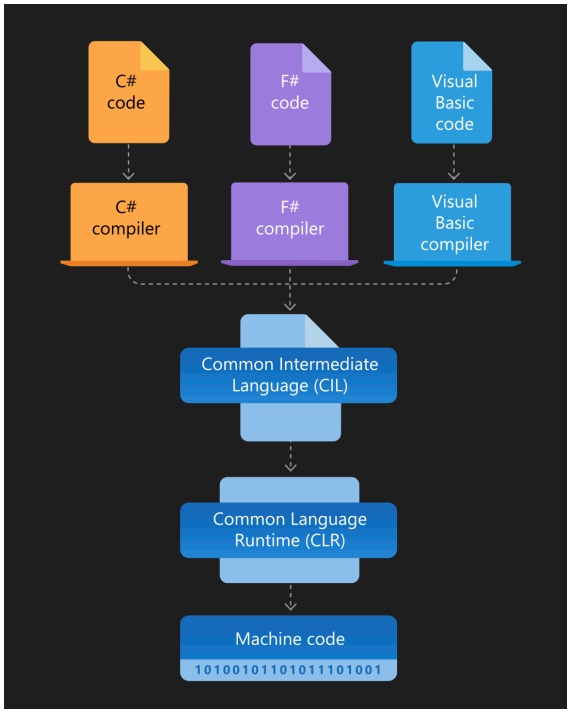
Week 3

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.

Access Modifiers

Access modifiers in C# define the scope and visibility of classes, methods, fields, constructors and other members. They determine where and how a member can be accessed in a program.

To control the visibility of class members (the security level of each individual class and class member).To achieve "**Encapsulation**" - which is the process of making sure that "sensitive" data is hidden from users.

A) public - Can be accessed from **anywhere**, even from another project. Eg public api

B) private - Can be accessed **only inside the class**., ensuring encapsulation. Eg account balance

C) protected - Accessible **inside the class** AND **derived classes, ensuring inheritance. Eg for derived classes**

D) internal - Accessible **only inside the same assembly/project**.

Modifiers	Entire program	Containing Class	Current assembly	Derived Types	Derived Types within Current Assembly
public	Yes	Yes	Yes	Yes	Yes
private	No	Yes	No	No	No
protected	No	Yes	No	Yes	Yes
internal	No	Yes	Yes	No	Yes

Difference between protected and internal is that protected needs a derived class whereas internal does not need a derived class and it can access from a third class altogether.

Namespace

In C#, a namespace is a way to organize and group related classes, interfaces, structs and other types. It helps avoid name conflicts and makes code easier to manage, especially in large projects.

The members of a namespace are accessed by using dot(.) operator. A class in C# is fully known by its respective namespace.

- Two classes with the same name can be created inside 2 different namespaces in a single program.
- Inside a namespace, no two classes can have the same name.
- In C#, the full name of the class starts from its namespace name followed by dot(.) operator and the class name, which is termed as the fully qualified name of the class.

Nested Namespace

You can also define a namespace into another namespace which is termed as the nested namespace. To access the members of nested namespace user has to use the dot(.) operator.

Using Keyword

In C#, calling a class or function with its **fully qualified name** (namespace + class) every time is not practical. To avoid this, C# provides the using keyword. By adding using at the top of the program, you can access the classes and methods of that namespace directly, without writing the full name each time.

ENUMS

Enumeration (enum) in C# is a special value type that defines a set of named constants. It improves code readability by giving meaningful names to numeric values.

- **Value type:** Stored as integers by default (can use other integral types like byte, short, long).
- **Default underlying type:** int (starting from 0 unless specified).
- **Custom values:** You can assign explicit numeric values to members.
- **Improves readability:** Use enum Days.Monday instead of 0.
- Useful in switch statements for handling predefined options.

By default, the first member starts at 0, but you can assign custom values.

Best practices while using enums:

1. Enum **names** and **values** should follow PascalCase.
2. use when the variable can have a fixed and set of values that does not change
3. instead of using numbers to declare states we can use words. Eg, 1 for success or 0 for failure
4. assign explicit indexing only when required to
5. maintain an enum only for related values. Eg not for {red, right, south, fail}

Changing the data type of enum

By default, enum values are of type int, but you can change them to byte, short, long, etc.

By default, **C# enums are of type int**, meaning each enum value is stored as a **4-byte integer** (32 bits).

- : byte → **enum values are stored as bytes** (1 byte = 8 bits).
- Saves memory if you have **lots of enum instances** and values are small (0–255).
- Valid underlying types: byte, sbyte, short, ushort, int (default), uint, long, ulong

```
public enum Priority : byte
{
    Low = 1,
    Medium = 2,
    High = 3,
    Critical = 4
}
```

Datatable

DataTable, in C#, is like a mini-database living in your computer's memory, representing a single table of in-memory data. DataTable is part of ADO.NET, the data access model from

Microsoft used in .NET framework. It represents a single table of data with a collection of columns and rows. ADO/NET is a part of the .NET Framework that helps your program talk to a database.

DataTables are often used as intermediate storage before connecting to MySQL (or any database).

Why Use DataTable?

1. Organize data in a **table-like structure**.
2. Can **manipulate data in memory** before saving to database.
3. Useful for **reporting, data binding** in UI (like DataGridView).
4. Supports **filtering, sorting, and expressions**.

```
using System;
using System.Data;

namespace DataTableDemo
{
   class Program
   {
      static void Main()
      {
         // Create a DataTable
         DataTable studentsTable = new DataTable("Students");

         // Add Columns (define type)
         studentsTable.Columns.Add("ID", typeof(int));
         studentsTable.Columns.Add("Name", typeof(string));
         studentsTable.Columns.Add("Age", typeof(int));
         studentsTable.Columns.Add("Grade", typeof(string));

         // Add Rows
         studentsTable.Rows.Add(1, "Saksham", 20, "A");
         studentsTable.Rows.Add(2, "Ruchika", 21, "B");
         studentsTable.Rows.Add(3, "Barkha", 19, "A");
         studentsTable.Rows.Add(4, "Nisarg", 22, "C");
         studentsTable.Rows.Add(5, "Aryan", 20, "B");

         // Access single cell
         Console.WriteLine($"\nFirst student name: {studentsTable.Rows[0]["Name"]}");

         // Filter rows
         Console.WriteLine("\nStudents with Grade A:");
         DataRow[] gradeAStudents = studentsTable.Select("Grade = 'A'");
         foreach (DataRow row in gradeAStudents)
```

```
        {
            Console.WriteLine($"{row["Name"]} - {row["Grade"]}");
        }

        // Modify a value
        Console.WriteLine("\nUpdating saksham's Grade to B...");
        DataRow sakshamRow = studentsTable.Select("Name = 'Saksham'")[0];
        sakshamRow["Grade"] = "B";
        DisplayTable(studentsTable);

        // Remove a row
        Console.WriteLine("\nRemoving aryan from table...");
        DataRow aryanRow = studentsTable.Select("Name = 'Aryan'")[0];
        studentsTable.Rows.Remove(aryanRow);
        DisplayTable(studentsTable);
    }
  }
}
```

Why [0]?

- Because even if you expect **only one row**, Select() still returns an **array**.
- To access the actual row, you must index into the array.

Date, String, Math Classes & Methods

1. Dates :

To set dates in C#, use DateTime class. The DateTime value is between 12:00:00 midnight, January 1, 0001 to 11:59:59 P.M., December 31, 9999 A.D.

KEY METHODS/ PROPERTIES -

- DateTime.Now → current date & time
- DateTime.Today → current date only
- AddDays(n), AddMonths(n), AddYears(n)
- Subtract → gives a TimeSpan (difference)
- ToString("format") → custom formatting

```
using System;

class Program
{
  public static void Main()
  {
    // current date and time
```

```
        DateTime now = DateTime.Now;
        Console.WriteLine($"Roght now, the date and time is {now}");

        // current date
        Console.WriteLine($"Date {DateTime.Today.ToShortDateString()}");
        // .ToShortTimeString()

        // Create specific date
        DateTime birthday = new DateTime(2004, 5, 12);
        Console.WriteLine("Birthday: " + birthday.ToString("dd/MM/yyyy"));

        // Add days/months/years
        Console.WriteLine("10 days later: " + now.AddDays(10));
        Console.WriteLine("2 months earlier: " + now.AddMonths(-2));

        // Difference between dates
        TimeSpan age = now - birthday;
        Console.WriteLine("Days lived: " + age.Days);
    }
}
```

2. Strings :

A string is represented by a class System.String. Strings are **immutable objects** in C#. You manipulate them using methods.

CHARACTERISTICS -

- The System.String class is immutable, i.e., once created, its state cannot be altered.
- Provides the properties like length used to get a total number of characters present in the given string.
- String objects can include a null character, which counts as part of the string's length.
- It provides the position of the characters in the given string called as indexes.
- It allows empty strings. Empty strings are valid instances of String objects that contain zero characters.
- It also supports searching strings, comparison of strings, testing of equality, modifying the string, normalization of string, copying of strings, etc.
- It also provides several ways to create strings like using a constructor, using concatenation, etc.

```
using System;

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

```
    {
        string name = "  Hello World  ";

        // Length
        Console.WriteLine("Length: " + name.Length);

        // Trim spaces
        Console.WriteLine("Trimmed: '" + name.Trim() + "'");

        // Upper & Lower case
        Console.WriteLine("Upper: " + name.ToUpper());
        Console.WriteLine("Lower: " + name.ToLower());

        // Substring
        Console.WriteLine("Substring(2,5): " + name.Substring(2, 5));

        // Replace
        Console.WriteLine("Replace World with C#: " + name.Replace("World", "C#"));

        // Contains
        Console.WriteLine("Contains 'Hello'? " + name.Contains("Hello"));

        // Split into array
        string csv = "red,green,blue";
        string[] colors = csv.Split(',');
        Console.WriteLine("First color: " + colors[0]);
    }
}
```

KEY METHODS -

- .Length → number of characters
- .Trim() → removes extra spaces
- .ToUpper() / ToLower()
- .Substring(start, length)
- .Replace(old, new)
- .Contains(), StartsWith(), EndsWith()
- .Split() & string.Join()

3. Math :

In C#, **Math class** comes under the System namespace. It is used to provide static methods and constants for logarithmic, trigonometric, and other useful mathematical functions. It is a static class and inherits the object class.

KEY METHODS -

- Math.Abs(x) → absolute value
- Math.Max(a, b) / Math.Min(a, b)
- Math.Sqrt(x) → square root
- Math.Pow(a, b) → power
- Math.Round(x) → nearest integer
- Math.Ceiling(x) → next higher integer
- Math.Floor(x) → next lower integer
- Constants: Math.PI, Math.E
- Trigonometry: Math.Sin, Math.Cos, Math.Tan

```
using System;

class Program
{
    public static void Main()
    {
        Console.WriteLine("Absolute: " + Math.Abs(-25));       // 25
        Console.WriteLine("Max: " + Math.Max(10, 20));         // 20
        Console.WriteLine("Min: " + Math.Min(10, 20));         // 10

        Console.WriteLine("Square root: " + Math.Sqrt(16));    // 4
        Console.WriteLine("Power: " + Math.Pow(2, 3));         // 8

        Console.WriteLine("Round 3.6: " + Math.Round(3.6));    // 4
        Console.WriteLine("Ceiling 3.2: " + Math.Ceiling(3.2));// 4
        Console.WriteLine("Floor 3.8: " + Math.Floor(3.8));    // 3

        Console.WriteLine("PI: " + Math.PI);                   // 3.141...
        Console.WriteLine("E: " + Math.E);                     // 2.71...

        Console.WriteLine("Cos(0): " + Math.Cos(0));           // 1
        Console.WriteLine("Sin(0): " + Math.Sin(0));           // 0
    }
}
```

File Operations

Termination of a program leads to the deletion of all data related to it. Therefore, we need to store the data somewhere. Files are used for permanently storing and sharing data. C# can be used to retrieve and manipulate data stored in text files.

Reading a Text file:

- The File.ReadAllText() reads the entire file at once and returns a string. We need to store this string in a variable and use it to display the contents onto the screen.
- The File.ReadAllLines() reads a file one line at a time and returns that line in string format. We need an array of strings to store each line. We display the contents of the file using the same string array
- There is another way to read a file and that is by using a StreamReader object. The StreamReader also reads one line at a time and returns a string.

Writing a Text File:

- The File.WriteAllText() writes the entire file at once. It takes two arguments, the path of the file and the text that has to be written.
- The File.WriteAllLines() writes a file one line at a time. It takes two arguments, the path of the file and the text that has to be written, which is a string array.
- There is another way to write to a file and that is by using a StreamWriter object. The StreamWriter also writes one line at a time. **All of the three writing methods create a new file if the file doesn't exist, but if the file is already present in that specified location then it is overwritten.**

  In case you want to add more text to an existing file without overwriting the data already stored in it, you can use the append methods File.AppendAllText(location, text);

SUMMARY OF FILE METHODS :

Method / Class	Purpose
File.WriteAllText()	Write text, overwrite if exists
File.AppendAllText()	Write text, append if exists
File.ReadAllText()	Read entire file as string
File.ReadAllLines()	Read file into string array
StreamWriter	Write line by line, append/overwrite
StreamReader	Read line by line or all at once
File.Exists()	Check if file exists
File.Delete()	Delete file

Why do we need StreamWriter and StreamReader?

- The File class (File.WriteAllText, File.ReadAllText, etc.) is **quick and simple** for small files. It loads the whole file at once into memory.
- StreamWriter and StreamReader are for **working with streams** (data flowing bit by bit). They are **better for large files** and give you **more control** (write line by line, read line by line, append, buffer, etc.).
- They are used to read and write gradually, append instead of overwriting, format output.

ANALOGY -

- **File.ReadAllText() / WriteAllText()** → Like opening a notebook and reading/writing **all pages at once**.

- **StreamReader / StreamWriter** → Like opening the notebook and reading/writing **page by page** (or line by line).

WHEN TO USE WHAT -

- Small file, simple use → File.ReadAllText, File.WriteAllText.
- Large file / line-by-line control → StreamReader, StreamWriter.
- Logging system → Always StreamWriter.
- Processing CSV or data line by line → StreamReader.