WEEK-3 NOTES

▼ .NET: History & Architecture

The History of .NET

To understand what .NET is today, you have to understand the problem it was created to solve back in the late 1990s.

The "Before Times" (Late 1990s)

At the time, Microsoft was the king of the desktop with Windows, but software development was fragmented. You could use Visual Basic, C++, or other languages, but they didn't work well together. Meanwhile, Java was gaining massive traction with its "Write Once, Run Anywhere" philosophy. Microsoft needed a modern, unified platform to compete.

The Birth: The .NET Framework Era (2002 - 2019)

In 2002, Microsoft released the **.NET Framework**. It was a revolutionary managed execution environment for Windows.

Core Ideas of the Original .NET Framework:

- **Common Language Runtime (CLR):** The heart and soul of .NET. Think of it as a protective bubble and translator for your code. It manages memory (garbage collection), handles security, and deals with errors. It compiles your code into an **Intermediate Language (IL)** and then translates it *just-in-time* (JIT) into code the machine can execute.
- **C# (C-Sharp):** A brand new, modern, object-oriented language designed by Anders Hejlsberg. It took the power of C++ and the best ideas from Java to create something powerful yet productive.

- **Base Class Library (BCL):** A massive, comprehensive toolbox of pre-written code for common tasks like reading files, networking, and working with data.
- **ASP.NET:** A framework for building dynamic websites and web applications.

Evolution and Key Milestones:

- **.NET Framework 1.0 (2002):** The initial release with Windows Forms, ADO.NET, and ASP.NET.
- **.NET Framework 2.0 (2005):** Introduced **Generics**, a massive feature for creating type-safe, reusable code.
- **.NET Framework 3.5 (2007):** A game-changer that added **LINQ (Language-Integrated Query)**, allowing you to write SQL-like queries directly in C#.
- **.NET Framework 4.5 (2012):** Introduced the `async`/`await` keywords, dramatically simplifying asynchronous programming.

The Major Limitation: The .NET Framework was powerful, but it was **Windows-only**.

The Revolution: .NET Core (2016)

The world was changing—Linux, macOS, Docker, and the cloud were becoming dominant. To stay relevant, Microsoft rebuilt .NET from the ground up, creating **.NET Core**.

This was a complete reimagining with a new philosophy:

- **Cross-Platform:** It could run on Windows, macOS, and Linux. This was the single biggest change.
- **Open-Source:** The entire codebase was put on GitHub, allowing the community to contribute.
- **High-Performance:** It was engineered to be one of the fastest frameworks in the world.
- **Modular:** You only included the bits you needed, making applications lightweight.

The Unification: .NET 5 and Beyond (2020 - Present)

Having two separate platforms (.NET Framework and .NET Core) was confusing. So, starting in 2020, Microsoft merged them into a single, unified platform simply called **.NET**.

- **.NET 5 (2020):** The first unified release, dropping the "Core" brand.
- **.NET 6 (2021, LTS):** A Long-Term Support release that introduced **.NET MAUI** (Multi-platform App UI) for building cross-platform mobile and desktop apps from a single codebase.
- **.NET 7 (2022):** Focused on performance and enhancements for cloud-native applications.
- **.NET 8 (2023, LTS):** Another LTS release with major improvements to AI integration, Blazor (for web UI), and performance.

Today, there is just one **.NET**, with a new version released yearly.

An Overview of .NET Today

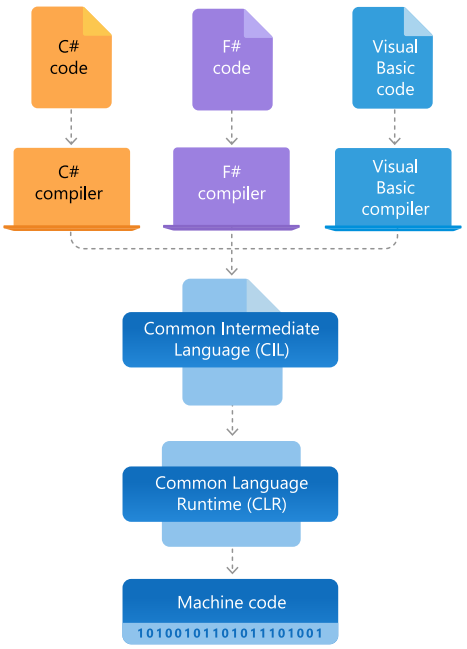
.NET is an ecosystem with several key parts that work together.

Core Components

1. **The Runtime (CLR):** The modern, fast, cross-platform engine that runs your code. Its main jobs are:
    - **Memory Management:** The Garbage Collector (GC) automatically frees up memory you're no longer using.
    - **JIT Compilation:** Compiles your Intermediate Language (IL) into native machine code on the fly for great performance.
    - **Type & Code Safety:** Ensures your code behaves correctly and safely.
2. **The SDK vs. The Runtime:** This is a crucial distinction:

- The **SDK (Software Development Kit)** is what you, the developer, install. It includes the compiler and tools (like the `dotnet` CLI) needed to **build** applications.
- The **Runtime** is all that's needed to **run** a .NET application. A server or user's machine only needs this.

3. **The Base Class Library (BCL):** The modern, massive toolbox of essential functions for collections, file I/O, networking, JSON handling, and more.
4. **The Languages:**
   - **C#:** The primary, modern, versatile language for .NET.
   - **F#:** A powerful, functional-first language.
   - **VB.NET:** A legacy language still supported on the platform.



What Can You Build with .NET?

You can build almost anything with the modern .NET platform:

- **Web Applications:** Using **ASP.NET Core**, build fast web APIs, microservices, and full-stack web UIs (with Razor Pages or Blazor).
- **Desktop Applications:** Build native Windows apps with **WPF** and **Windows Forms**, or cross-platform desktop apps with **.NET MAUI**.
- **Mobile Applications:** Use **.NET MAUI** to write code once and deploy native apps to Android, iOS, macOS, and Windows.
- **Cloud Services:** .NET is a first-class citizen for building cloud-native applications, microservices, and serverless functions, especially on Microsoft Azure.
- **Games:** The world's most popular game engine, **Unity**, uses C# as its scripting language.
- **AI & Machine Learning:** Use **ML.NET** to integrate custom machine learning models into your .NET applications.
- **Internet of Things (IoT):** Run .NET on small devices like a Raspberry Pi.

Getting Started

Your First Steps

1. **Install the .NET SDK:** Go to the official .NET website and download the latest **LTS version** of the SDK (e.g., .NET 8).
2. **Pick Your Editor:**
   - **Visual Studio (Windows/Mac):** A full-featured IDE with powerful debugging tools. Excellent for beginners.
   - **Visual Studio Code (Windows/Mac/Linux):** A lightweight, free, and highly extensible code editor.
3. Create Your First App (Using the Terminal):Bash

   Open a terminal and run these commands:

```
# Creates a new simple console application in a folder named "MyFi
 rstApp"
```

```
dotnet new console -o MyFirstApp

# Navigates into that folder
cd MyFirstApp

# Compiles and runs your program
dotnet run
```

▼ Console App Project Structure

The Solution (`.sln` file)

- **Purpose:** The **Solution** is the top-level container that holds one or more projects. Think of it as a **binder** that keeps related projects organized together.
- **Use:** You open the `.sln` file in Visual Studio to load your entire workspace.

The Project Folder & File (`.csproj`)

- **Purpose:** The project folder contains all the files for a single application. The `.csproj` file within it is the project's **blueprint**.
- **Use:** This XML file tells the compiler what to do. It defines:
  - The **.NET version** to target (e.g., `net8.0`).
  - The **output type** (e.g., `Exe` for an executable).
  - Any external libraries (**NuGet packages**) your project depends on.

Program.cs

- **Purpose:** This is the C# code file containing the **entry point** of your application.
- **Use:** Execution of your program begins here. In modern .NET, the code in this file runs from top to bottom. This is where you write your main logic.

`obj` Folder

- **Purpose:** This folder stores **intermediate files** generated by the compiler during the build process.
- **Use:** You should **completely ignore this folder**. It is managed automatically by the build tools. You can safely delete it, and it will be recreated the next time you build the project.

`bin` Folder

- **Purpose:** This folder contains the **final, compiled output** of your application—the files you can actually run. It's short for "Binaries."
- **Use:** Inside, you will find two main subfolders:
  - `Debug`: Contains the build used for development. It includes extra debugging information (`.pdb` files) that allows you to step through your code in Visual Studio.
  - `Release`: Contains an optimized version of your app, ready for distribution. It is smaller and faster because it lacks the extra debugging information.

▼ Compilation Model & More

C# Compilation Model

The process of turning your C# code into a running application involves two main steps, which is what makes .NET applications portable.

Step 1: Source Code to Intermediate Language (IL)

First, the C# compiler (csc.exe) takes your human-readable source code (.cs files) and compiles it into a low-level, platform-independent language called Intermediate Language (IL), also known as Common Intermediate Language (CIL).

Step 2: Intermediate Language (IL) to Native Code

When you run the application, the Common Language Runtime (CLR) takes over. Its Just-In-Time (JIT) compiler translates the IL code into native machine code (the binary

instructions that the computer's CPU can execute directly). This translation happens "just in time" as the code is needed during execution.

This two-step process allows the same compiled IL to be run on any operating system that has a compatible .NET runtime installed.

Assemblies

An **assembly** is the fundamental building block of a .NET application. It's the compiled output of your code, containing the IL and supporting metadata.

- **Definition:** An assembly is a packaged unit for deployment, versioning, and security.
- **Formats:** Assemblies come in two forms:
  - **Executable (`.exe`):** The primary output for an application that can be run directly. It contains an entry point to start execution.
  - **Library (`.dll`):** A Dynamic Link Library that contains reusable code (classes, methods, etc.) intended to be used by other assemblies. It has no entry point and cannot be run on its own.
- **Assembly Manifest:** Every assembly includes a **manifest**. This is metadata that describes the assembly's identity, including its name, version, culture, and a list of all files and other assemblies it depends on.

Managed vs. Unmanaged Code

This distinction is central to understanding the .NET environment.

Managed Code

Managed code is code that is executed and managed by the **Common Language Runtime (CLR)**.

- **What it is:** The C# code you write is managed code by default.

- **Benefits:** The CLR provides several services automatically, including:
    - **Automatic Memory Management:** The Garbage Collector (GC) allocates and releases memory, preventing common memory leaks.
    - **Type Safety:** The CLR ensures that objects are used in a type-safe way, preventing errors.
    - **Security:** The CLR can enforce security restrictions on the code.
    - **Exception Handling:** Provides a structured way to handle errors.

Unmanaged Code

Unmanaged code is code that executes outside the control of the CLR.

- **What it is:** This typically involves interacting directly with the operating system (e.g., calling the Windows API) or using code from older C/C++ libraries.
- **Characteristics:**
    - **Manual Memory Management:** The developer is responsible for allocating and freeing memory.
    - **Less Safe:** It bypasses the safety and security features of the CLR.
    - **Interop:** C# can interact with unmanaged code through a feature called "Interop" (Interoperability), but this is an advanced topic.

▼ Datatypes In C#

In C#, a **type** is a blueprint that defines the kind of data a variable can hold and what operations can be performed on it. C# is a **strongly-typed** language, meaning every variable and constant must have a type.

The C# type system is divided into two main categories: **Value Types** and **Reference Types**.

Value Types

Value types are variables that **directly contain their data**. When you copy a value type variable, you are creating a completely new and independent copy of the data. They are generally stored in a fast memory location called the **stack**.

- **Analogy:** A value type is like a **physical notebook**. The information is written directly on the page. If you copy the page, you have a new, separate page with the same information; changing one doesn't affect the other.

Common Value Types:

Structs

This is the main keyword for creating a custom value type.

All of the simple, built-in types in C# are technically structs.

- **Integer types:** `int`, `byte`, `short`, `long`
- **Floating-point types:** `float`, `double`, `decimal` (for high-precision financial calculations)
- `bool`: Represents `true` or `false`
- `char`: Represents a single character

```
// 'x' directly contains the value 10.
int x = 10;

// 'y' is a new copy of x's value.
int y = x;

// Changing 'y' does NOT affect 'x'.
y = 20;

Console.WriteLine(x); // Output: 10
Console.WriteLine(y); // Output: 20
```

Enums

An enumeration is a special value type that represents a set of named constants. It makes your code more readable.

```
public enum Season
{
   Spring, // 0
   Summer, // 1
   Autumn, // 2
   Winter  // 3
}

Season currentSeason = Season.Summer;
Console.WriteLine(currentSeason); // Output: Summer
```

Reference Types

Reference types are variables that do not store the data directly. Instead, they store a **reference** (like a memory address) to where the data is actually located. The data itself is stored in a more flexible memory area called the **heap**.

- **Analogy:** A reference type is like a **sticky note with a library book's shelf number**. The sticky note itself isn't the book; it just tells you where to find it. If you copy the sticky note, both notes point to the exact same book.

When you copy a reference type variable, you are only copying the reference, not the actual object. Both variables will point to the **same object in memory**.

Common Reference Types:

1. Classes

   This is the most common way to define a reference type. It's the blueprint for creating objects.

```
public class Person
 {
```

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

// 'person1' holds a reference to a new Person object on the heap.
var person1 = new Person { Name = "Alice" };

// 'person2' is a new variable that gets a copy of person1's referenc
e.
// Both now point to the SAME Person object.
var person2 = person1;

// Changing the object through 'person2' WILL affect 'person1'.
person2.Name = "Bob";

Console.WriteLine(person1.Name); // Output: Bob
Console.WriteLine(person2.Name); // Output: Bob
```

2. Arrays

   A collection of elements of the same type.

```
int[] numbers = new int[3]; // An array is a reference type.
```

3. String (string)

   A sequence of characters. string is a special reference type. It's immutable, meaning its value cannot be changed after it's created. When you "modify" a string, you are actually creating a new string object in memory. This makes it behave much like a value type in practice, which is a common point of confusion.

4. Interfaces (interface)

   A contract that defines a set of properties and methods that a class must implement.

5. Delegates (delegate)

   A type that can hold a reference to a method.

The Unified Type System

C# has a **unified type system**. This means that all types, both value and reference, ultimately inherit from a single base class: `System.Object` (or `object`). Because of this, you can call methods like `.ToString()` on any variable, even simple numbers.

- **Boxing:** The process of converting a value type (like `int`) into a reference type (`object`).
- **Unboxing:** The process of converting the boxed object back into a value type.

▼ Access Modifiers

public

The `public` modifier is the most permissive and least restrictive access level. When a member is `public`, it can be accessed from anywhere without restrictions.

- **Analogy:** A **public park bench**. Anyone, from any project or assembly, can access and use it.
- **Use Case:** Exposing the primary functionality of your code to other developers or projects (e.g., creating a public API).

Example

Imagine you have a class library project (`ProjectA`) and a console application (`ProjectB`) that references it.

```
// In Project A (e.g., a class library)
namesace ProjectA
{
    public class Customer
    {
    // This property is accessible from anywhere
    public string Name { get; set; }

    // This method is accessible from anywhere
    public void PrintCustomerInfo()
```

```
    {
        Console.WriteLine($"Customer Name: {Name}");
    }
    }
}
```

```
// In Project B (e.g., a console app that references Project A)
using ProjectA;

var customer = new Customer();
customer.Name = "Nisharg";
customer.PrintCustomerInfo();
```

private

The `private` modifier is the most restrictive. A `private` member can only be accessed by code within the **same class or struct** in which it is declared.

- **Analogy:** A **personal diary**. Only the owner (the class itself) can read or write in it.
- **Use Case:** Hiding implementation details that are internal to a class and should not be modified from the outside.
- **Default:** If you don't specify any access modifier for a member inside a class or struct, it is `private` by default.

Example

A `Stopwatch` class needs to manage its start and end times internally to prevent invalid states.

```
public class Stopwatch
{
    // These fields are hidden from the outside world.
    private DateTime _startTime;
    private bool _isRunning;

    public void Start()
```

```
    {
        if (_isRunning)
            throw new InvalidOperationException("Stopwatch is already runn
ing.");

        _startTime = DateTime.Now;
        _isRunning = true;
    }
}
```

```
// In another file
var watch = new Stopwatch();
watch.Start();
// watch._startTime = DateTime.Now; // ERROR! Inaccessible due to its p
rotection level.
```

Corner Cases/Key Points:

- A nested class (a class inside another class) can access the `private` members of its containing (outer) class.

protected

The **`protected`** modifier is related to inheritance. A `protected` member can be accessed from within the same class and by any class that **derives (inherits) from it**.

- **Analogy:** A **secret family recipe**. Only family members (the base class and its descendants) can use it. To the outside world, it's hidden.
- **Use Case:** Sharing implementation details with child classes to allow them to extend or modify base class behavior, without exposing those details publicly.

Example

A base `Vehicle` class has a `StartEngine()` method that is used by more specific vehicle types.

```
public class Vehicle
{
    // Child classes can access this, but the public cannot.
    protected void StartEngine()
    {
        Console.WriteLine("Engine started.");
    }
}

public class Car : Vehicle
{
    public void Drive()
    {
        StartEngine();
        Console.WriteLine("Car is driving.");
    }
}
```

```
// In another file
var myCar = new Car();
myCar.Drive();
// myCar.StartEngine(); // ERROR! Inaccessible. It's protected.
```

Corner Cases/Key Points:

- **The Crucial Rule:** A derived class can only access a `protected` member of the base class **through an instance of its own type or a more derived type**. It cannot access it through an instance of the base class.

  ```
  public class Car : Vehicle
  {
      public void CompareEngines(Vehicle otherVehicle)
      {
          this.StartEngine();      // OK: Accessing through 'this' (type Car).
          // otherVehicle.StartEngine(); // ERROR: Cannot access protecte
  d member
                                  // on an instance of the base class.
  ```

```
    }
}
```

internal

The `internal` modifier limits access to the **current assembly**. An assembly is a compiled project (a `.dll` or `.exe`).

- **Analogy:** An **internal company-wide email**. Any employee within the company can see it, but it's not meant for the public.
- **Use Case:** Creating helper classes, utilities, or services that are used throughout your project but should not be exposed as part of its public API.
- **Default:** If you don't specify an access modifier for a top-level type (like `class MyClass`), it is `internal` by default.

Example

A project has a configuration reader that should be accessible to all classes in that project, but not to any external projects.

```
// In Project A (MyApplication.dll)
internal class ConfigurationReader
{
    public string GetConnectionString() => "Server=...";
}

public class DataService
{
    public void Connect()
    {
        var reader = new ConfigurationReader();
        var connection = reader.GetConnectionString();
    }
}
```

```
// In Project B (AnotherApp.exe that references MyApplication.dll)
// var reader = new ConfigurationReader(); // ERROR! Inaccessible due t
o its
                                        // protection level.
```

Corner Cases/Key Points:

- You can grant another specific assembly access to your `internal` members by using the `[InternalsVisibleTo("AssemblyName")]` attribute in your project's `AssemblyInfo.cs` or `.csproj` file. This creates a "friend" assembly.

`file` (New in C# 11)

The `file` modifier is the newest and most restrictive access modifier. A `file`-local type is only visible within the **source file (`.cs`) in which it is declared**.

- **Analogy: Scribbled notes on a single page of a notebook.** These notes are only for your reference while writing on that page; they aren't even meant for other pages in the same notebook.
- **Use Case:**
  - **Source Generators:** To generate helper code that won't cause naming conflicts.
  - **Avoiding Namespace Pollution:** To create a small, single-purpose helper class for one file without making it visible to the rest of the project.

Example

Imagine you have a complex method in `DataProcessor.cs` that needs a small helper struct, but you don't want this struct to be usable anywhere else.

```
// In DataProcessor.cs
public class DataProcessor
{
   public void Process()
```

```
    {
        var helper = new ProcessingState(10);
    }
}

// Only visible within DataProcessor.cs
file class ProcessingState
{
    public int State { get; private set; }
    public ProcessingState(int initialState) => State = initialState;
}
```

```
// In AnotherFile.cs (within the same project)
// var state = new ProcessingState(0); // ERROR! 'ProcessingState' is no
t found.
                                    // It's as if it doesn't exist.
```

Combined Modifiers

C# provides two modifiers that combine the logic of the ones above.

protected internal

This provides access if a condition is met in an **OR** relationship. Access is granted if the code is:

1. In the same assembly (internal).

   OR

2. In a **derived class** in another assembly (`protected`).

- It's the more permissive of the two combined modifiers.

private protected

This provides access if a condition is met in an **AND** relationship. Access is granted only if the code is:

1. In a derived class (protected).

   AND

2. In the **same assembly** (`internal`).

- It's the more restrictive of the two combined modifiers. It's for sharing implementation details with child classes, but only within the project you are building.

Caller's location	public	protected internal	protected	internal	private protected	private	file
Within the file	✓	✓	✓	✓	✓	✓	✓
Within the class	✓	✓	✓	✓	✓	✓	✗
Derived class (same assembly)	✓	✓	✓	✓	✓	✗	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗	✗

▼ Namespaces

The Problem: Why Do We Need Namespaces?

Imagine you are writing a large application. You create a class called `Client`. Later, you add a third-party library for handling online payments, and it also has a class named `Client`. Now, when you write `Client`, how does the compiler know which one you mean? This is called a **naming collision**, and it can cause errors in your project.

Namespaces solve this problem by providing a "full address" for your types.

- **Analogy 1 (Full Names):** If you have two people named "John" in a room, you need a last name to tell them apart. `John Smith` is different from `John Doe`. In C#, the full name of a class is `Namespace.ClassName`. For example:
  - `MyCompany.Crm.Client`
  - `Stripe.Api.Client`
- **Analogy 2 (Folders):** A namespace is like a folder on your computer. You can have a file named `Report.docx` in `C:\Work` and another `Report.docx` in `C:\Vacation`. They are unique because of their full paths.

How to Declare a Namespace

You declare a namespace using the `namespace` keyword. All code inside the curly braces belongs to that namespace.

```
// Customer.cs
namespace MyAwesomeApp.BusinessLogic
{
    public class Customer
    {
        public string Name { get; set; }
    }
}

// Order.cs
namespace MyAwesomeApp.DataAccess
{
    public class Order
    {
        public int OrderId { get; set; }
    }
}
```

Here, `Customer` and `Order` are kept separate by being in different namespaces.

Best Practice: Use a hierarchy like `CompanyName.ProductName.Feature`.

Using a Namespace (`using` directive)

Declaring namespaces gives your classes full names, but typing them fully is tedious. The `using` directive is a shortcut.

```
// Program.cs
using System; // Lets us write 'Console' instead of 'System.Console'
using MyAwesomeApp.BusinessLogic; // Lets us use 'Customer' directly

class Program
{
    static void Main()
    {
        Customer customer = new Customer();
        customer.Name = "Jane Doe";
```

```
        Console.WriteLine(customer.Name);
    }
}
```

Without the `using` directive, you would need to write:

```
MyAwesomeApp.BusinessLogic.Customer customer = new MyAwesome
App.BusinessLogic.Customer();
```

Common Scenarios

1. **Organizing Projects:**

   Separate logic into namespaces like `YourApp.UI`, `YourApp.BusinessLogic`, `YourApp.Data`.

2. **Using Third-Party Libraries:**

   For example, Newtonsoft.Json uses the namespace `Newtonsoft.Json`.

   ```
   using Newtonsoft.Json;
    var json = JsonConvert.SerializeObject(myObject);
   ```

3. **Resolving Ambiguity:**

   If two libraries have classes with the same name, use aliases.

   ```
   using FileSystemData = MyProject.FileSystem.DataHandler;
    using WebData = MyProject.Web.DataHandler;

    class DataProcessor
    {
       public void ProcessFiles()
       {
          FileSystemData fileHandler = new FileSystemData();
       }

       public void ProcessWeb()
       {
   ```

```
        WebData webHandler = new WebData();
    }
}
```

4. **Global Usings (C# 10+):**

   For namespaces you use everywhere, declare them once globally.

```
// GlobalUsings.cs
global using System;
global using MyAwesomeApp.Models;
```

Advanced Details

- **Nested Namespaces:**

  You can nest namespaces inside others.

```
namespace Company.Product
{
    namespace Features
    {
        public class Tool { }
    }
}
```

- **File-Scoped Namespaces (C# 10+):**

  Cleaner syntax for one namespace per file.

```
namespace MyApp.Models;

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

- **Default Namespaces:**

If you don’t declare a namespace, the class belongs to the global namespace. Avoid this in real projects for clarity.

- **Fully Qualified Names:**

  You can always bypass `using` and write the full path:

```
System.Console.WriteLine("Hello");
```

▼ Important Libraries (BCL)

Overview of .NET Libraries

.NET libraries are organized into **namespaces**.

At the top level, you will often see `System.*`, `Microsoft.*`, and sometimes third-party ones (like `Newtonsoft.*`).

1. `System` (The Root Namespace)

This is the foundation

Common Classes:

- `System.Console` → Read and write from the console.
- `System.String` → Represents text.
- `System.DateTime` → Date and time handling.
- `System.Math` → Math operations (sqrt, pow, etc.).
- `System.Random` → Random number generator.
- `System.Exception` → Base class for exceptions.

Example:

```
using System;

class Program
{
   static void Main()
```

```
    {
        Console.WriteLine("Today is " + DateTime.Now);
        Random rnd = new Random();
        Console.WriteLine("Random number: " + rnd.Next(1, 100));
    }
}
```

2. `System.Collections` and `System.Collections.Generic`

These provide **data structures** like lists, dictionaries, and queues.

- `System.Collections` → Non-generic (older, less type-safe).
- `System.Collections.Generic` → Generic, modern, and type-safe.

Useful Types:

- `List<T>` → Dynamic array.
- `Dictionary<TKey, TValue>` → Key-value store.
- `Queue<T>` → First In, First Out.
- `Stack<T>` → Last In, First Out.
- `HashSet<T>` → Unique elements.

Example:

```
using System.Collections.Generic;

class Example
{
    static void Main()
    {
        List<string> names = new List<string> { "Alice", "Bob" };
        names.Add("Charlie");

        Dictionary<int, string> students = new Dictionary<int, string>
        {
            {1, "John"},
```

```
        {2, "Jane"}
    };

    Console.WriteLine(students[1]); // John
  }
}
```

3. `System.IO` (File & Stream Handling)

For reading and writing files, directories, and streams.

Useful Classes:

- `File` → Static methods for file operations.
- `FileInfo` → Instance methods for file metadata.
- `Directory`, `DirectoryInfo` → Work with folders.
- `StreamReader`, `StreamWriter` → Reading/writing text.
- `BinaryReader`, `BinaryWriter` → Binary data.

Example:

```
using System.IO;

class FileDemo
{
  static void Main()
  {
    File.WriteAllText("demo.txt", "Hello .NET!");
    string content = File.ReadAllText("demo.txt");
    Console.WriteLine(content);
  }
}
```

4. `System.Linq` (Language Integrated Query)

LINQ (Language Integrated Query) is one of the most powerful .NET features.

It lets you query collections like databases.

Example:

```
using System;
using System.Linq;
using System.Collections.Generic;

class LinqDemo
{
   static void Main()
   {
      List<int> numbers = new() { 1, 2, 3, 4, 5, 6 };
      var evens = numbers.Where(n ⇒ n % 2 == 0).ToList();
      Console.WriteLine(string.Join(", ", evens)); // 2, 4, 6
   }
}
```

5. System.Threading & System.Threading.Tasks

For concurrency, parallelism, and async programming.

- `Thread` → Low-level threading.
- `Task` → Modern asynchronous programming.
- `async/await` keywords use `Task`.

Example:

```
using System;
using System.Threading.Tasks;

class AsyncDemo
{
   static async Task Main()
   {
      await Task.Delay(1000); // Wait 1 second
      Console.WriteLine("Task finished!");
```

```
    }
}
```

6. System.Net & System.Net.Http

For network programming, making HTTP requests, sockets, etc.

Common Classes:

- `HttpClient` → Send HTTP requests.
- `WebClient` (older).
- `TcpClient`, `UdpClient` → Low-level sockets.

Example:

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class HttpDemo
{
    static async Task Main()
    {
        HttpClient client = new HttpClient();
        string html = await client.GetStringAsync("https://dotnet.microsoft.
com");
        Console.WriteLine(html.Substring(0, 100)); // Print first 100 chars
    }
}
```

7. System.Text

For working with text encodings, string building, and JSON.

Common Classes:

- `StringBuilder` → Efficient string manipulation.
- `Encoding` → Text encoding (UTF-8, ASCII).

- `JsonSerializer` (from `System.Text.Json`) → JSON handling.

Example:

```
using System.Text.Json;

class JsonDemo
{
    static void Main()
    {
        var person = new { Name = "Alice", Age = 25 };
        string json = JsonSerializer.Serialize(person);
        Console.WriteLine(json); // {"Name":"Alice","Age":25}
    }
}
```

8. System.Data

For database access (SQL Server, ADO.NET).

- `SqlConnection`, `SqlCommand`, `SqlDataReader`.
- Higher-level ORM: **Entity Framework Core** (separate NuGet package).

9. Microsoft.Extensions.*

These are modern utility libraries introduced in .NET Core for dependency injection, logging, configuration, etc.

- `Microsoft.Extensions.Logging` → Structured logging.
- `Microsoft.Extensions.Configuration` → JSON/config file handling.
- `Microsoft.Extensions.DependencyInjection` → Dependency Injection container.

10. Popular External Libraries (via NuGet)

Not part of core .NET, but very commonly used:

- **Newtonsoft.Json** → Powerful JSON library (before `System.Text.Json`).
- **Entity Framework Core** → ORM for databases.
- **AutoMapper** → Maps between objects.
- **Serilog** → Advanced logging.
- **NUnit / xUnit / MSTest** → Unit testing frameworks.

▼ Enum

What is an Enum?

An **enum** (short for **enumeration**) is a **special value type** in C# that defines a **set of named constants**. It improves code readability and maintainability because you use meaningful names instead of magic numbers or strings.

Think of an enum as a way to give **friendly names to a set of integral values**.

Why Use Enums?

Problems solved by enums:

- Avoid **magic numbers** in your code.
- Make **intent clear**: `Status = 1` is unclear, but `Status = OrderStatus.Pending` is readable.
- **Compile-time safety**: The compiler checks that you only assign allowed enum values.
- Easy **switch statements**: Enums pair naturally with `switch/case`.

Example problem without enum:

```
int orderStatus = 1; // What does 1 mean? Pending? Shipped? Cancelle
d?
if(orderStatus == 2) { /* shipped */ }
```

Using enum instead:

```
enum OrderStatus { Pending = 1, Shipped = 2, Cancelled = 3 }

OrderStatus status = OrderStatus.Pending;
if(status == OrderStatus.Shipped) { /* shipped */ }
```

Basic Enum Declaration

```
enum WeekDay
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
}
```

- By default, underlying type = `int`.
- First value = 0, then 1, 2, … unless explicitly assigned.

```
Console.WriteLine((int)WeekDay.Monday); // Output: 1
```

Custom Underlying Type

You can specify the underlying numeric type: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`.

```
enum WeekDay : byte
{
   Sunday = 0,
   Monday = 1,
   Tuesday = 2
}
```

- Using smaller types can **save memory** if you have large arrays of enums.
- **Default type** is `int`.

Explicit Value Assignment

You can assign values explicitly:

```
enum OrderStatus
{
   Pending = 1,
   Shipped = 5,
   Delivered = 10,
   Cancelled = 20
}
```

- If you assign one value, subsequent members **increment by 1** unless overridden:

```
enum Level
{
   Low = 1,
   Medium,   // 2
   High     // 3
}
```

Casting Enums

Enums can be **cast to/from their underlying type**.

```
OrderStatus status = (OrderStatus)5;
Console.WriteLine(status); // Shipped

int value = (int)OrderStatus.Delivered;
Console.WriteLine(value); // 10
```

Enums and Strings

Convert enum to string:

```
OrderStatus status = OrderStatus.Pending;
string name = status.ToString(); // "Pending"
```

Flags Enum (Bitwise Enums)

- Some enums represent **combinations of values**, e.g., file permissions.
- Use `[Flags]` attribute to enable **bitwise operations**.

```
[Flags]
enum FileAccess
{
   Read = 1,    // 001
   Write = 2,   // 010
   Execute = 4  // 100
}

FileAccess permissions = FileAccess.Read | FileAccess.Write;
Console.WriteLine(permissions); // Output: Read, Write

//Without [Flags]:
FileAccess p = FileAccess.Read | FileAccess.Write;
Console.WriteLine(p); // Output: 3
```

- Check if a flag is set:

```
if((permissions & FileAccess.Write) == FileAccess.Write)
{
   Console.WriteLine("Write access granted");
}
```

Enum Utilities

- **Get all names or values**

```
string[] names = Enum.GetNames(typeof(OrderStatus));
Array values = Enum.GetValues(typeof(OrderStatus));
```

- `Iterate:`

```
foreach(OrderStatus status in Enum.GetValues(typeof(OrderStatus)))
{
   Console.WriteLine($"{status} = {(int)status}");
}
```

- **`Check if value is defined`**

```
bool valid = Enum.IsDefined(typeof(OrderStatus), 5); // true
```

Enum Comparison and Switch

- `Enums can be compared using ==, <, >, etc., because they are essentially integers.`
- `Perfect for switch statements:`

```
switch(status)
{
   case OrderStatus.Pending:
      Console.WriteLine("Pending");
      break;
   case OrderStatus.Shipped:
      Console.WriteLine("Shipped");
      break;
   case OrderStatus.Delivered:
      Console.WriteLine("Delivered");
      break;
   default:
      Console.WriteLine("Unknown status");
      break;
}
```

Enum Limitations

- Cannot inherit from another enum.
- Cannot inherit from classes (only `System.Enum`).
- Can only use integral underlying types.
- Cannot have methods or properties directly.

▼ ADO.NET Introduction

ADO.NET (ActiveX Data Objects for .NET) is a core part of the .NET framework that provides a bridge between your application and a data source, which is typically a relational database like SQL Server, Oracle, or MySQL. It's the foundational, low-level technology for all data access in .NET.

Think of ADO.NET as the **plumbing** for your application's data. It gives you the essential tools—the pipes and valves—to connect to a database, send commands, and retrieve data.

The Two Core Models of ADO.NET

ADO.NET operates in two primary ways: a connected model for high-performance, short-lived operations, and a disconnected model for working with data offline.

1. The Connected Model

This model requires a constant, open connection to the database to read or write data. It's fast, efficient, and provides forward-only, read-only access to data.

- **Analogy:** This is like a **live phone call** with the database. You open the connection (dial the number), ask your questions (send commands), get answers back one by one, and then hang up. The line is tied up for the entire conversation.

Key Objects:

- `Connection` (e.g., `SqlConnection`): Represents the actual connection to the database. You open it to start and close it when you're finished.

- `Command` (e.g., `SqlCommand`): Represents the SQL query or stored procedure you want to execute.
- `DataReader` (e.g., `SqlDataReader`): A high-performance, forward-only stream of data. It reads results directly from the database one row at a time, making it extremely memory-efficient.

When to Use It:

Perfect for quickly fetching data to display, like populating a list on a webpage. It's the fastest way to get read-only data from the database.

```
string connectionString = "connectionstring";
using (var connection = new SqlConnection(connectionString))
{
    var command = new SqlCommand("SELECT FirstName, LastName FR
OM Employees", connection);
    connection.Open();

    using (SqlDataReader reader = command.ExecuteReader())
    {
        while (reader.Read()) // Read one row at a time
        {
            Console.WriteLine($"{reader["FirstName"]} {reader["LastNam
e"]}");
        }
    }
} // Connection is automatically closed here
```

2. The Disconnected Model

This model connects to the database, fetches a complete set of data, copies it into an in-memory cache, and then immediately closes the connection. You can then work with this local copy of the data for as long as you need.

- **Analogy:** This is like **sending a fax or email** to the database. You send your request, it sends back a full copy of the requested documents, and you can then

disconnect. You work with your local copies and can later send back any changes in a single batch.

Key Objects:

- `DataSet`: The heart of the disconnected model. It's an **in-memory database**, complete with tables, columns, relationships, and constraints.
- `DataTable`: Represents a single table within the `DataSet`.
- `DataAdapter` (e.g., `SqlDataAdapter`): Acts as the bridge. It connects to the database, executes a command, and then populates a `DataSet` or `DataTable` with the results. It's also used to send changes from your `DataSet` back to the database.

When to Use It:

Ideal for applications where you need to hold onto data, perform complex operations like sorting and filtering, or make multiple edits before updating the database. It's great for desktop applications (like Windows Forms) where a user might edit a grid of data.

```
string connectionString = "Your_Connection_String_Here";
var adapter = new SqlDataAdapter("SELECT * FROM Products", connect
ionString);

var productDataSet = new DataSet();
adapter.Fill(productDataSet, "Products"); // Fills the DataSet and closes
the connection

// Now you can work with the data offline
DataTable productsTable = productDataSet.Tables["Products"];
Console.WriteLine($"Fetched {productsTable.Rows.Count} products.");
```

▼ DataTable

A `DataTable` is a core component of ADO.NET that represents a single, in-memory table of data. It's a versatile object

that can be created and manipulated manually, or it can be filled with data from a source like a database.

- **Analogy:** The best way to think of a `DataTable` is as a **spreadsheet (like in Excel)** within your application's memory. It has a defined set of columns (the schema) and a collection of rows that hold the data.

`DataTable` without a Database (In-Memory Usage) C#

You can use a `DataTable` as a powerful, standalone data structure without ever connecting to a database. This is useful for storing and manipulating structured data temporarily.

Here's the step-by-step process:

1. **Create the `DataTable`:** You start by creating an instance of the table.
2. **Define Columns:** You must define the table's schema by adding `DataColumn` objects. You specify the column name and data type for each.
3. **Add Rows:** You populate the table by creating `DataRow` objects, filling them with data, and adding them to the table's `Rows` collection.
4. **Access and Manipulate Data:** Once populated, you can iterate, filter, sort, and modify the data.

Complete Example

This example creates a simple "Products" table in memory, adds some data, and then displays it.

```
using System.Data;

// Create the DataTable
var productsTable = new DataTable("Products");

// Define Columns
productsTable.Columns.Add("ProductID", typeof(int));
productsTable.Columns.Add("ProductName", typeof(string));
```

```
productsTable.Columns.Add("Price", typeof(decimal));

// Set a primary key
productsTable.PrimaryKey = new DataColumn[] { productsTable.Column
s["ProductID"] };

// Add Rows with data
// Method 1: Create a new row, populate it, and then add it
DataRow newRow = productsTable.NewRow();
newRow["ProductID"] = 101;
newRow["ProductName"] = "Laptop";
newRow["Price"] = 1200.50m;
productsTable.Rows.Add(newRow);

// Method 2: Use the simpler Rows.Add() overload with an array of value
s
productsTable.Rows.Add(102, "Mouse", 25.00m);
productsTable.Rows.Add(103, "Keyboard", 75.75m);

//  Access and Manipulate Data
Console.WriteLine("All Products");
foreach (DataRow row in productsTable.Rows)
{
    Console.WriteLine($"ID: {row["ProductID"]}, Name: {row["ProductNa
me"]}, Price: {row["Price"]:C}");
}

// Find a specific row by its primary key
DataRow foundRow = productsTable.Rows.Find(102);
if (foundRow != null)
{
    Console.WriteLine("\nFound Product");
    Console.WriteLine($"Found by key: {foundRow["ProductName"]}");
}

// Filter rows using the Select() method
Console.WriteLine("\nExpensive Products (Price > $100)");
DataRow[] expensiveProducts = productsTable.Select("Price > 100");
```

```
foreach (DataRow row in expensiveProducts)
{
    Console.WriteLine($"Expensive: {row["ProductName"]} costs {row["P
rice"]:C}");
}
```

`DataTable` with a Database Connection

This is the traditional use case for a `DataTable`. Here, it acts as a container for data fetched from a database. The key object that bridges the database and your `DataTable` is the **`DataAdapter`**.

The process is much simpler:

1. Define your connection string and SQL query.
2. Create a `DataAdapter` (e.g., `SqlDataAdapter` for SQL Server).
3. Create an empty `DataTable`.
4. Use the `DataAdapter`'s `Fill()` method to execute the query and automatically populate the `DataTable`. The `Fill()` method handles opening the connection, fetching all the data, and closing the connection for you.

Complete Example

This example assumes you have a SQL Server database with a `Products` table.

```
using System.Data;
using MySql.Data.MySqlClient;


string connectionString = "server=myServerName;database=myDataba
se;user=myUsername;password=myPassword;";
string queryString = "SELECT ProductID, ProductName, Price FROM Pro
ducts WHERE IsAvailable = 1";


var productsFromDb = new DataTable();
```

```
Console.WriteLine("Fetching data from the MySQL database...");

try
{

    using (var adapter = new MySqlDataAdapter(queryString, connection
String))
    {
        adapter.Fill(productsFromDb);
    }

    Console.WriteLine($"Successfully fetched {productsFromDb.Rows.Co
unt} products.");

    Console.WriteLine("\nProducts from Database");
    foreach (DataRow row in productsFromDb.Rows)
    {
        Console.WriteLine($"ID: {row["ProductID"]}, Name: {row["Product
Name"]}, Price: {row["Price"]:C}");
    }
}
catch (Exception ex)
{
    Console.WriteLine("An error occurred: " + ex.Message);
}
```

▼ DateTime and TimeSpan

1. DateTime in C#

`DateTime` represents **a specific point in time**, including date and time.

- Namespace: `System`
- Structure: `struct DateTime`
- Stores: year, month, day, hour, minute, second, millisecond

Creating DateTime objects

Using constructors

```
using System;

class Program
{
   static void Main()
   {
      // Year, Month, Day
      DateTime dt1 = new DateTime(2025, 9, 23);

      // Year, Month, Day, Hour, Minute, Second
      DateTime dt2 = new DateTime(2025, 9, 23, 18, 30, 45);

      Console.WriteLine(dt1); // 9/23/2025 12:00:00 AM
      Console.WriteLine(dt2); // 9/23/2025 6:30:45 PM
   }
}
```

Using static properties

```
DateTime now = DateTime.Now;        // Current local date and time
DateTime utcNow = DateTime.UtcNow;  // Current UTC time
DateTime today = DateTime.Today;    // Current date, time = 00:00:00
```

Using Parse / TryParse

```
DateTime dt = DateTime.Parse("2025-09-23 18:30:00");
DateTime dt2;
bool success = DateTime.TryParse("2025-09-23", out dt2);
```

DateTime properties

Property	Description
`Year`	Gets the year (e.g., 2025)
`Month`	Gets the month (1-12)
`Day`	Gets the day of the month
`DayOfWeek`	Gets the day of the week (Sunday=0)
`DayOfYear`	Day number in the year (1-365/366)
`Hour`, `Minute`, `Second`	Gets hour, minute, second
`Date`	Date part only, time = 00:00:00
`Ticks`	Number of 100-nanosecond intervals since 1 Jan 0001
`Kind`	Local, UTC, or Unspecified

DateTime methods

Method	Description
`AddDays(double)`	Add days to the DateTime
`AddHours(double)`	Add hours
`AddMinutes(double)`	Add minutes
`AddSeconds(double)`	Add seconds
`AddMilliseconds(double)`	Add milliseconds
`AddMonths(int)`	Add months
`AddYears(int)`	Add years
`ToShortDateString()`	Converts to “MM/dd/yyyy”
`ToLongDateString()`	Converts to “Day, Month dd, yyyy”
`ToString(format)`	Custom formatting using format strings

Example:

```
DateTime now = DateTime.Now;
Console.WriteLine(now.AddDays(5)); // 5 days later
Console.WriteLine(now.AddMonths(-2)); // 2 months back
Console.WriteLine(now.ToString("yyyy-MM-dd HH:mm:ss")); // Custom format
```

2. TimeSpan in C#

`TimeSpan` represents a **duration or interval** — the difference between two DateTime objects.

- Namespace: `System`
- Structure: `struct TimeSpan`
- Stores: days, hours, minutes, seconds, milliseconds

Creating TimeSpan

```
// Using constructor: Days, Hours, Minutes, Seconds
TimeSpan ts1 = new TimeSpan(1, 2, 30, 0); // 1 day, 2 hours, 30 minutes

// Using hours, minutes, seconds
TimeSpan ts2 = new TimeSpan(5, 45, 0); // 5 hours, 45 minutes

// Using From methods
TimeSpan ts3 = TimeSpan.FromDays(1.5);   // 1 day 12 hours
TimeSpan ts4 = TimeSpan.FromHours(10);   // 10 hours
TimeSpan ts5 = TimeSpan.FromMinutes(90); // 1 hour 30 minutes
```

TimeSpan properties

Property	Description
`Days`	Total days part
`Hours`	Hours part
`Minutes`	Minutes part
`Seconds`	Seconds part
`Milliseconds`	Milliseconds part
`Ticks`	100-nanosecond intervals
`TotalDays`	Total duration in days (double)
`TotalHours`	Total duration in hours
`TotalMinutes`	Total duration in minutes
`TotalSeconds`	Total duration in seconds

Property	Description
TotalMilliseconds	Total duration in ms

Operations with TimeSpan

Subtract DateTime objects to get TimeSpan

```
DateTime start = new DateTime(2025, 9, 23, 8, 0, 0);
DateTime end = new DateTime(2025, 9, 23, 17, 30, 0);

TimeSpan duration = end - start;
Console.WriteLine(duration); // 09:30:00
Console.WriteLine(duration.TotalHours); // 9.5 hours
```

Add or subtract TimeSpan to DateTime

```
DateTime now = DateTime.Now;
TimeSpan ts = new TimeSpan(2, 0, 0); // 2 hours
DateTime later = now + ts;          // 2 hours later
DateTime earlier = now - ts;        // 2 hours earlier
```

Arithmetic with TimeSpan

```
TimeSpan t1 = TimeSpan.FromHours(5);
TimeSpan t2 = TimeSpan.FromMinutes(30);

TimeSpan sum = t1 + t2;
TimeSpan diff = t1 - t2;

Console.WriteLine(sum);  // 05:30:00
Console.WriteLine(diff); // 04:30:00
```

Formatting DateTime and TimeSpan

```
DateTime dt = DateTime.Now;
Console.WriteLine(dt.ToString("yyyy-MM-dd"));      // 2025-09-23
Console.WriteLine(dt.ToString("dd/MM/yyyy HH:mm")); // 23/09/2025 1
```

```
8:30

TimeSpan ts = new TimeSpan(1, 12, 30, 15);
Console.WriteLine(ts.ToString());                    // 1.12:30:15 (1 day, 12h, 30
m, 15s)
```

▼ String

What is a String in C#

- In C#, `string` **is an alias for** `System.String` in .NET.
- It represents a **sequence of characters** (Unicode).
- Strings are **immutable**, meaning once created, they **cannot be changed**. Any operation that modifies a string creates a **new string**.

```
string s1 = "Hello";
string s2 = "World";

string s3 = s1 + " " + s2; // "Hello World"
```

Creating Strings

```
// Literal
string str1 = "Hello";

// Using constructor
string str2 = new string('a', 5); // "aaaaa"

// From char array
char[] chars = { 'C', '#', 'R', 'o', 'c', 'k', 's' };
string str3 = new string(chars); // "C#Rocks"
```

String Properties

Property	Description
`Length`	Number of characters in the string
`Chars[index]`	Access character at given index (0-based)
`Empty`	Represents an empty string (`""`)
`IsNullOrEmpty`	Static method to check if string is null or empty
`IsNullOrWhiteSpace`	Static method to check if string is null, empty, or whitespace

Example:

```
string str = "Hello World";
Console.WriteLine(str.Length);     // 11
Console.WriteLine(str[0]);         // H
Console.WriteLine(string.Empty);   // ""
Console.WriteLine(string.IsNullOrEmpty(str)); // False
Console.WriteLine(string.IsNullOrWhiteSpace(" ")); // True
```

Common String Methods

Basic Methods

Method	Description	Example
`ToUpper()`	Convert to uppercase	`"hello".ToUpper()` → `"HELLO"`
`ToLower()`	Convert to lowercase	`"HELLO".ToLower()` → `"hello"`
`Trim()`	Remove leading/trailing whitespaces	`" hello ".Trim()` → `"hello"`
`TrimStart()`	Remove leading whitespaces	`" hello".TrimStart()` → `"hello"`
`TrimEnd()`	Remove trailing whitespaces	`"hello ".TrimEnd()` → `"hello"`

Searching

Method	Description	Example
`Contains(string)`	Check if string exists	`"Hello".Contains("ll")` → `True`
`StartsWith(string)`	Check beginning	`"Hello".StartsWith("He")` → `True`
`EndsWith(string)`	Check ending	`"Hello".EndsWith("lo")` → `True`
`IndexOf(string)`	First index of substring	`"Hello".IndexOf("l")` → `2`
`LastIndexOf(string)`	Last index of substring	`"Hello".LastIndexOf("l")` → `3`

Substrings & Splitting

Method	Description	Example
`Substring(startIndex)`	Get substring from index	`"Hello".Substring(2)` → `"llo"`
`Substring(start, length)`	Get substring with length	`"Hello".Substring(1,3)` → `"ell"`
`Split(char[])`	Split string into array	`"a,b,c".Split(',')` → `["a","b","c"]`
`Replace(old, new)`	Replace substring	`"Hello".Replace("l","x")` → `"Hexxo"`

Joining & Formatting

Method	Description	Example
`string.Join(separator, array)`	Join array into string	`string.Join("-", new[] {"A","B","C"})` → `"A-B-C"`
`string.Format(template, args)`	Format string	`string.Format("Hello {0}", "World")` → `"Hello World"`
`$"..."` (interpolation)	Embedded expressions	`$"Hello {name}"`

Comparison

Method	Description	Example
`Equals()`	Compare strings	`"abc".Equals("ABC")` → `False`
`Compare(a,b)`	Lexicographical comparison	`string.Compare("a","b")` → `-1`
`Compare(a,b,true)`	Ignore case	`string.Compare("a","A",true)` → `0`

Other Useful Methods

Method	Description	Example
`PadLeft(totalWidth)`	Pad left with spaces	`"abc".PadLeft(5)` → `" abc"`
`PadRight(totalWidth)`	Pad right with spaces	`"abc".PadRight(5)` → `"abc "`
`Insert(index, string)`	Insert string at index	`"Hello".Insert(2,"yy")` → `"Heyyllo"`
`Remove(startIndex)`	Remove from start index	`"Hello".Remove(2)` → `"He"`
`Remove(startIndex, count)`	Remove specific chars	`"Hello".Remove(1,3)` → `"Ho"`

Immutability of String

- Strings **cannot be changed in memory**.
- Any operation like `Replace`, `Trim`, `Concat` **creates a new string**.
- Example:

```
string s = "Hello";
s.Replace("H", "J");
Console.WriteLine(s); // Still "Hello"
s = s.Replace("H", "J"); // Assign new string
Console.WriteLine(s); // "Jello"
```

StringBuilder vs String

- If you need **frequent modifications**, use `StringBuilder` (from `System.Text`) for performance.
- `StringBuilder` is mutable, whereas `string` is immutable.

```
using System.Text;

StringBuilder sb = new StringBuilder("Hello");
sb.Append(" World");
```

```
sb.Replace("Hello", "Hi");
Console.WriteLine(sb); // "Hi World"
```

Converting Strings

Conversion	Example
To int	`int.Parse("123")` or `Convert.ToInt32("123")`
To double	`double.Parse("12.5")`
To bool	`bool.Parse("true")`
From other types	`123.ToString()`, `12.5.ToString()`

▼ Math

What is Math class?

- The `Math` class provides **mathematical constants and static methods**.
- You **do not create an instance** — all methods are `static`.
- Common uses: **rounding, powers, roots, trigonometry, logarithms, etc.**

```
using System;

class Program
{
   static void Main()
   {
      double result = Math.Sqrt(25); // 5
      Console.WriteLine(result);
   }
}
```

Math Constants

Constant	Value	Description
`Math.PI`	3.14159265358979323846	Pi
`Math.E`	2.71828182845904523536	Euler's number

Basic Arithmetic Methods

Method	Description	Example
`Math.Abs(x)`	Absolute value	`Math.Abs(-5)` → 5
`Math.Sign(x)`	Returns -1,0,1 for negative,0,positive	`Math.Sign(-10)` → -1
`Math.Max(a,b)`	Maximum of two numbers	`Math.Max(5,10)` → 10
`Math.Min(a,b)`	Minimum of two numbers	`Math.Min(5,10)` → 5
`Math.Clamp(x, min, max)`	Restrict value to min/max	`Math.Clamp(15,0,10)` → 10

Rounding and Truncation

Method	Description	Example
`Math.Ceiling(x)`	Round up to nearest integer	`Math.Ceiling(3.2)` → 4
`Math.Floor(x)`	Round down to nearest integer	`Math.Floor(3.8)` → 3
`Math.Round(x)`	Round to nearest integer	`Math.Round(3.5)` → 4
`Math.Round(x, n)`	Round to `n` decimal places	`Math.Round(3.14159, 2)` → 3.14
`Math.Truncate(x)`	Remove fractional part	`Math.Truncate(3.9)` → 3

Powers and Roots

Method	Description	Example
`Math.Pow(x, y)`	x^y	`Math.Pow(2,3)` → 8
`Math.Sqrt(x)`	Square root	`Math.Sqrt(16)` → 4
`Math.Cbrt(x)`	Cube root (from .NET 7)	`Math.Cbrt(27)` → 3
`Math.Exp(x)`	e^x	`Math.Exp(1)` → 2.718281828…

Logarithms

Method	Description	Example
`Math.Log(x)`	Natural log (ln)	`Math.Log(Math.E)` → 1
`Math.Log10(x)`	Base-10 log	`Math.Log10(100)` → 2
`Math.Log(x, base)`	Logarithm with custom base	`Math.Log(8,2)` → 3

Trigonometric Functions

- All angles in **radians**, not degrees.

Method	Description	Example
`Math.Sin(x)`	Sine	`Math.Sin(Math.PI/2)` → 1
`Math.Cos(x)`	Cosine	`Math.Cos(0)` → 1
`Math.Tan(x)`	Tangent	`Math.Tan(Math.PI/4)` → 1
`Math.Asin(x)`	Inverse sine (returns radians)	`Math.Asin(1)` → π/2
`Math.Acos(x)`	Inverse cosine	`Math.Acos(0)` → π/2
`Math.Atan(x)`	Inverse tangent	`Math.Atan(1)` → π/4
`Math.Atan2(y,x)`	Angle from (0,0) to (x,y)	`Math.Atan2(1,1)` → π/4

Hyperbolic Functions

Method	Description
`Math.Sinh(x)`	Hyperbolic sine
`Math.Cosh(x)`	Hyperbolic cosine
`Math.Tanh(x)`	Hyperbolic tangent

Remainder & Modulus

Method	Description	Example
`Math.IEEERemainder(x,y)`	Returns remainder using IEEE rules	`Math.IEEERemainder(7,3)` → 1
`%` operator	Standard modulus	`7 % 3` → 1

12. Example Program Using Multiple Math Methods

```
using System;

class Program
{
    static void Main()
    {
        double a = -7.5;
        double b = 16;

        Console.WriteLine("Abs: " + Math.Abs(a));         // 7.5
        Console.WriteLine("Max: " + Math.Max(a, b));      // 16
        Console.WriteLine("Pow: " + Math.Pow(2, 3));      // 8
        Console.WriteLine("Sqrt: " + Math.Sqrt(b));       // 4
        Console.WriteLine("Round: " + Math.Round(a));     // -8
        Console.WriteLine("Ceiling: " + Math.Ceiling(a));// -7
        Console.WriteLine("Floor: " + Math.Floor(a));     // -8
        Console.WriteLine("Sin(π/2): " + Math.Sin(Math.PI/2)); // 1
        Console.WriteLine("Log(100): " + Math.Log10(100));      // 2
    }
}
```

▼ File I/O

- **File I/O** (Input/Output) is used to **read from or write to files**.
- Namespace: `System.IO`
- Supports **text**, **binary**, **JSON**, **CSV**, etc.
- Key Classes:

Class	Purpose
`File`	Static methods for creating, copying, deleting, reading, writing files
`FileInfo`	Instance-based file operations, detailed file info

Class	Purpose
`Directory`	Static methods for directories
`DirectoryInfo`	Instance-based directory operations
`Stream`	Abstract base class for byte streams
`FileStream`	Read/write bytes to files
`StreamReader`	Read text from files
`StreamWriter`	Write text to files
`BinaryReader`	Read binary data
`BinaryWriter`	Write binary data
`Path`	

1. File Class

- **Namespace:** `System.IO`
- **Type:** Static class → cannot create instances
- **Purpose:** Simplest way to perform file operations like **create, read, write, copy, delete**.

Common Methods

Method	Description
`File.Create(path)`	Creates a new file (overwrites if exists). Returns a `FileStream`.
`File.Exists(path)`	Checks if file exists (true/false).
`File.Delete(path)`	Deletes the specified file.
`File.Copy(source, dest, overwrite)`	Copies file; optional overwrite.
`File.Move(source, dest)`	Moves/renames file.
`File.ReadAllText(path)`	Reads entire text file as a string.
`File.WriteAllText(path, text)`	Writes text to a file (overwrites).
`File.AppendAllText(path, text)`	Appends text to an existing file.
`File.ReadAllLines(path)`	Reads all lines into a string array.

Method	Description
`File.WriteAllLines(path, string[])`	Writes all lines from an array.
`File.ReadAllBytes(path)`	Reads all bytes of a file.
`File.WriteAllBytes(path, byte[])`	Writes byte array to file.

Example:

```
File.WriteAllText("example.txt", "Hello File");
string text = File.ReadAllText("example.txt");
Console.WriteLine(text);
```

2. FileInfo Class

- **Namespace:** `System.IO`
- **Type:** Instance-based → must create an object.
- **Purpose:** Works on a specific file, provides **metadata and instance operations**.

Properties

Property	Description
`FullName`	Complete path of the file
`Name`	File name with extension
`Extension`	File extension
`DirectoryName`	Directory containing the file
`Length`	File size in bytes
`Exists`	Checks if file exists
`CreationTime`	File creation date/time
`LastAccessTime`	Last accessed date/time
`Attributes`	File attributes (ReadOnly, Hidden, etc.)

Methods

Method	Description
`Create()`	Creates a file (returns FileStream).
`Delete()`	Deletes the file.
`CopyTo(dest)`	Copies file to destination.
`MoveTo(dest)`	Moves/renames file.
`OpenRead()`	Returns FileStream for reading.
`OpenWrite()`	Returns FileStream for writing.
`AppendText()`	Returns StreamWriter for appending text.
`OpenText()`	Returns StreamReader for reading text.

Example:

```
FileInfo file = new FileInfo("example.txt");
if (!file.Exists) file.Create().Close();

Console.WriteLine(file.Length);
file.CopyTo("copy.txt", true);
```

3. Directory Class

- **Namespace:** `System.IO`
- **Type:** Static class
- **Purpose:** Manage directories: create, delete, move, get files/subdirectories.

Methods

Method	Description
`CreateDirectory(path)`	Creates a directory
`Exists(path)`	Checks if directory exists
`Delete(path, recursive)`	Deletes directory, optionally recursive
`GetFiles(path, pattern)`	Gets files matching pattern
`GetDirectories(path)`	Gets all subdirectories
`Move(source, dest)`	Moves/renames directory

Example:

```
Directory.CreateDirectory("MyFolder");
string[] files = Directory.GetFiles(".");
foreach(var f in files) Console.WriteLine(f);
Directory.Delete("MyFolder", true);
```

4. DirectoryInfo Class

- **Namespace:** `System.IO`
- **Type:** Instance-based → object for specific directory
- **Purpose:** Provides **metadata and operations** for directories.

Properties

Property	Description
`FullName`	Full path
`Name`	Directory name
`Parent`	Parent directory
`Exists`	Check if exists
`Attributes`	Directory attributes
`CreationTime`	Creation date/time
`LastAccessTime`	Last accessed date/time

Methods

Method	Description
`Create()`	Create the directory
`Delete(recursive)`	Delete directory
`GetFiles()`	Get files in directory
`GetDirectories()`	Get subdirectories
`MoveTo(dest)`	Move/Rename directory

Example:

```
DirectoryInfo dir = new DirectoryInfo("MyFolder");
dir.Create();
Console.WriteLine(dir.Exists);
dir.Delete(true);
```

5. FileStream Class

- **Namespace:** `System.IO`
- **Type:** Stream-based → works with **bytes**
- **Purpose:** Read/write files **byte by byte or block by block**
- **Useful:** Binary files, large files, performance-sensitive I/O.

Constructors

```
FileStream fs = new FileStream("file.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);
```

- **FileMode:** Create, Open, OpenOrCreate, Append, Truncate
- **FileAccess:** Read, Write, ReadWrite
- **FileShare:** None, Read, Write, ReadWrite

Methods

Method	Description
`Read(byte[], int, int)`	Reads bytes into buffer
`Write(byte[], int, int)`	Writes bytes from buffer
`Seek(offset, origin)`	Move pointer
`Close()` / `Dispose()`	Close file stream

Example:

```
byte[] data = System.Text.Encoding.UTF8.GetBytes("Hello FileStream");
using (FileStream fs = new FileStream("fs.txt", FileMode.Create))
{
```

```
    fs.Write(data, 0, data.Length);
}
```

6. StreamReader & StreamWriter

- **Namespace:** `System.IO`
- **Purpose:** High-level text file reading/writing

StreamReader Methods

Method	Description
`ReadLine()`	Reads one line
`ReadToEnd()`	Reads entire file
`Peek()`	Checks next char without reading
`Close()`	Close stream

StreamWriter Methods

Method	Description
`Write()`	Write text without newline
`WriteLine()`	Write text with newline
`Flush()`	Flush buffer to file
`Close()`	Close stream

Example:

```
using(StreamWriter sw = new StreamWriter("text.txt"))
{
    sw.WriteLine("Hello World");
}

using(StreamReader sr = new StreamReader("text.txt"))
{
    string line = sr.ReadLine();
    Console.WriteLine(line);
}
```

7. BinaryReader & BinaryWriter

- **Purpose:** Work with **binary files** (images, audio, serialized objects)

BinaryWriter Methods

Method	Description
`Write(int/double/string/...)`	Write primitive types
`Close()`	Close stream

BinaryReader Methods

Method	Description
`ReadInt32()`	Reads int
`ReadDouble()`	Reads double
`ReadString()`	Reads string
`Close()`	Close stream

Example:

```
using(BinaryWriter bw = new BinaryWriter(File.Open("data.bin", FileMode.Create)))
{
    bw.Write(123);
    bw.Write(45.6);
    bw.Write("Hello");
}

using(BinaryReader br = new BinaryReader(File.Open("data.bin", FileMode.Open)))
{
    Console.WriteLine(br.ReadInt32());   // 123
    Console.WriteLine(br.ReadDouble());  // 45.6
    Console.WriteLine(br.ReadString());  // Hello
}
```

8. Path Class

- **Namespace:** `System.IO`
- **Purpose:** Manipulate file and directory paths

Method	Description
`Combine(a,b)`	Combine paths
`GetFileName(path)`	Get file name
`GetExtension(path)`	Get file extension
`GetDirectoryName(path)`	Get directory part
`GetTempPath()`	Get system temp folder

Example:

```
string path = Path.Combine("C:\\Temp", "file.txt");
Console.WriteLine(Path.GetFileName(path)); // file.txt
Console.WriteLine(Path.GetExtension(path)); // .txt
```

Advanced Scenarios: JSON & CSV

CSV

- Use `StreamReader` + `Split(',')` for simple files.
- Use **`CsvHelper`** (NuGet) for advanced parsing, headers, quotes, etc.

```
// Basic CSV
using (StreamReader sr = new StreamReader("data.csv"))
{
   while(!sr.EndOfStream)
   {
      string[] row = sr.ReadLine().Split(',');
      Console.WriteLine($"{row[0]} - {row[1]}");
   }
}
```

JSON

- Use `System.Text.Json` or `Newtonsoft.Json`
- Convert objects → JSON → file, and read JSON → objects.

```
using System.Text.Json;

var emp = new { ID=1, Name="Alice", Salary=50000 };
string json = JsonSerializer.Serialize(emp);
File.WriteAllText("emp.json", json);

string readJson = File.ReadAllText("emp.json");
var empObj = JsonSerializer.Deserialize<Dictionary<string, object>>(re
adJson);
```