Pre-Joining Topics

Week 3: C# Fundamentals

Introduction

1. Introduction to C#

C# (pronounced “C-sharp”) is a modern, object-oriented programming language developed by Microsoft. It is designed for building a wide range of applications, from desktop and web apps to mobile and cloud-based services.

Key Features of C#:

- **Simple & Modern:** Easy to learn and expressive.
- **Object-Oriented:** Supports encapsulation, inheritance, and polymorphism.
- **Type-Safe:** Prevents type errors.
- **Scalable & Updateable:** Ideal for large-scale applications.
- **Rich Library Support:** Access to .NET libraries for various functions.
- **Interoperability:** Can work with COM and other languages like C++ and VB.NET.

---

2. History of C#

- **1999:** Development of C# started at Microsoft as part of a project called *“Cool”* under Anders Hejlsberg.
- **2000:** Announced publicly at the Professional Developers Conference (PDC).
- **2002:** Officially released with **.NET Framework 1.0**.
- **Key Milestones:**
    - **C# 1.0 (2002):** Basic OOP features, Windows Forms, Web Forms.
    - **C# 2.0 (2005):** Generics, partial classes, nullable types.
    - **C# 3.0 (2007):** LINQ (Language Integrated Query), anonymous types, lambda expressions.
    - **C# 4.0 (2010):** Dynamic binding, named & optional arguments.
    - **C# 5.0 (2012):** Async and await for asynchronous programming.
    - **C# 6.0+ (2015+):** Expression-bodied members, null-conditional operators.

- **C# 7.0 - 11.0 (2017-2022):** Pattern matching, records, global usings, nullable reference types, source generators.

Takeaway: C# evolved to support modern programming paradigms and cross-platform development.

---

3. Overview of .NET Framework

C# runs primarily on the **.NET platform**, which is a software framework developed by Microsoft.

3.1 What is .NET?

- A **software development platform** used to build and run applications on Windows, Linux, macOS.
- Provides a **runtime environment** (CLR) and a **class library** (FCL – Framework Class Library).

3.2 Components of .NET Framework

1. **Common Language Runtime (CLR):**
   - Acts as the execution engine for .NET programs.
   - Handles **memory management** (garbage collection), **type safety**, **exception handling**, and **security**.
   - Converts **C# code** (compiled into **Intermediate Language – IL**) into **machine code**.
2. **Framework Class Library (FCL):**
   - Pre-built libraries of classes, interfaces, and value types.
   - Examples:
     - System.Collections – data structures
     - System.IO – file operations
     - System.Net – networking
     - System.Data – database access
   - Saves developers from writing common functionalities from scratch.
3. **ASP.NET / Windows Forms / WPF / Xamarin:**

- **ASP.NET** → Web applications
- **Windows Forms / WPF** → Desktop apps
- **Xamarin / MAUI** → Mobile apps

4. **Language Interoperability:**
   - .NET supports multiple languages (C#, VB.NET, F#).
   - All languages compile to **IL** and run on CLR.

3.3 .NET Framework vs .NET Core vs .NET 5/6/7/8

Feature	.NET Framework	.NET Core	.NET 5+
OS Support	Windows only	Cross-platform	Cross-platform
Open Source	No	Yes	Yes
App Types	Desktop, Web	Web, Cloud, Console	All types
Performance	Moderate	High	Higher

Note: Modern C# development focuses on **.NET Core / .NET 7/8** for cross-platform apps.

---

4. How C# and .NET Work Together

1. Developer writes **C# code** → .cs files.
2. C# code is compiled by **C# compiler (csc.exe)** → **Intermediate Language (IL) code**.
3. IL is packaged into **assemblies (.exe / .dll)**.
4. CLR **loads the assembly**, performs **JIT compilation** → machine code.
5. Program executes on the computer.

---

5. Advantages of Using C# with .NET

- Cross-platform development (via .NET Core / .NET 5+)
- Rich standard libraries reduce coding effort.
- Automatic memory management (garbage collection)
- Strongly typed language reduces errors.

- Modern language features like async/await, LINQ, lambda expressions.

---

6. Summary

- **C#** is a modern, object-oriented language designed by Microsoft.
- Initially Windows-only, now cross-platform via .NET Core / .NET 5+.
- **.NET Framework** provides runtime (CLR) + libraries (FCL).
- History shows continuous improvement: from basic OOP features to async programming and pattern matching.

Scope And Accessibility

1. Scope vs Accessibility

Scope and **Accessibility** are related but different concepts in C#:

- **Scope** → Refers to **where a variable or method is visible inside the code** (block, class, method, namespace).
  Example: A variable declared inside a method cannot be accessed outside the method.
- **Accessibility** → Refers to **who can access a class, method, or variable** from **other classes or assemblies**. Controlled by **access modifiers** like public, private, protected, internal.

Think of **scope** as "**local visibility**" and **accessibility** as "**global visibility rules**."

1. Types of Scope in C#

1.1 Local Scope

- **Definition:** Variable declared **inside a method, constructor, or block**.
- **Visibility:** Only **inside that method or block**.
- **Lifetime:** Exists **only while the method/block is executing**.

Example:

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

```
    int x = 10; // local variable
    Console.WriteLine(x); // ✅ Accessible here

    {
        int y = 20; // block-level local variable
        Console.WriteLine(y); // ✅ Accessible here
    }

    // Console.WriteLine(y); // ❌ Error: y is out of scope
  }
}
```

1.2 Class/Instance Scope

- **Definition:** Variables declared **inside a class but outside any method** (also called **fields**).
- **Visibility:** Accessible **inside all methods of the class**, depending on **access modifiers** (private, protected, etc.).
- **Lifetime:** Exists **as long as the object exists**.

Example:

```
class Employee
{
  private string name; // class-level scope (field)

  public void SetName(string n)
  {
    name = n; // ✅ Accessible here
  }
```

```
    public void ShowName()
    {
        Console.WriteLine(name); // ✅ Accessible here
    }
}
```

1.3 Static/Class Scope

- **Definition:** Static variables declared **inside a class using static keyword**.
- **Visibility:** Accessible **inside the class or through class name**, depending on access modifier.
- **Lifetime:** Exists **for the lifetime of the program**.

Example:

```
class Employee
{
    public static int employeeCount = 0;

    public Employee()
    {
        employeeCount++; // ✅ Accessible inside class
    }
}

class Program
{
    static void Main()
    {
        Employee e1 = new Employee();
        Employee e2 = new Employee();
```

```
        Console.WriteLine(Employee.employeeCount); // ✅ Accessible via class name
    }
}
```

1.4 Namespace/Global Scope

- **Definition:** Classes, interfaces, or enums declared **inside a namespace but outside any class**.
- **Visibility:** Accessible **through the namespace**; may require using directive in other files.

Example:

```
namespace Company
{
    public class Employee
    {
        public string Name;
    }
}


class Program
{
    static void Main()
    {
        Company.Employee emp = new Company.Employee(); // ✅ Accessible via namespace
    }
}
```

---

2. Access Modifiers in C#

C# has **Six main access modifiers**:

Modifier	Description	Scope/Visibility
public	Accessible from **anywhere**, inside or outside the class, even from other assemblies.	Global
private	Accessible **only within the same class** or struct.	Class-local
protected	Accessible **within the class and its derived (child) classes**.	Inheritance-based
internal	Accessible **within the same assembly/project**.	Assembly-local
protected internal	Accessible **within the same assembly or from derived classes in other assemblies**.	Combination
private protected	Accessible **within the containing class or derived classes in the same assembly**.	Restricted combination

---

2.1 public Modifier

- **Definition**: Member is accessible **from anywhere**, no restriction.
- **Use Case**: Methods, properties, or classes that need to be accessed globally.
- **Example**:

```
public class Employee
{
   public string Name; // accessible everywhere
}


class Program
{
   static void Main()
   {
      Employee emp = new Employee();
```

```
        emp.Name = "Heer"; // ✅ Allowed
        Console.WriteLine(emp.Name);
    }
}
```

- **Note**: Making everything public can break encapsulation. Use wisely.

2.2 private Modifier

- **Definition**: Member is accessible **only inside the same class**.
- **Use Case**: Hide sensitive details; encapsulate data.
- **Example**:

```
class Employee
{
    private double salary; // only Employee class can access

    public void SetSalary(double s)
    {
        salary = s; // ✅ Allowed
    }

    public double GetSalary()
    {
        return salary; // ✅ Allowed
    }
}

class Program
{
    static void Main()
```

```
    {
        Employee emp = new Employee();
        // emp.salary = 5000; // ❌ Error: salary is private
        emp.SetSalary(5000); // ✅ Correct
        Console.WriteLine(emp.GetSalary());
    }
}
```

- **Note**: Encourages **Encapsulation** in OOP.

2.3 protected Modifier

- **Definition**: Member is accessible **inside the class and its subclasses**.
- **Use Case**: Share data/methods with child classes but not outside world.
- **Example**:

```
class Employee
{
    protected string company = "RKIT";
}

class Manager : Employee
{
    public void ShowCompany()
    {
        Console.WriteLine(company); // ✅ Accessible in derived class
    }
}

class Program
{
```

```
    static void Main()
    {
        Manager m = new Manager();
        // Console.WriteLine(m.company); // ❌ Error: protected not accessible here
        m.ShowCompany(); // ✅ Allowed
    }
}
```

- **Note**: Useful for **inheritance-based OOP design**.

2.4 internal Modifier

- **Definition**: Member is accessible **only within the same assembly/project**.
- **Use Case**: Hide implementation from external assemblies but allow usage within project.
- **Example**:

```
internal class Employee
{
    internal string Name = "Heer"; // accessible within same assembly
}


class Program
{
    static void Main()
    {
        Employee emp = new Employee(); // ✅ Allowed
        Console.WriteLine(emp.Name); // ✅ Allowed
    }
}
```

```
// From another project referencing this assembly:
// Employee emp = new Employee(); // ❌ Not allowed
```

- **Note**: Common in **libraries or internal tools**.

2.5 protected internal Modifier

- **Definition**: Member is accessible **either from derived classes or within the same assembly**.
- **Use Case**: Flexible sharing between inheritance and assembly scope.
- **Example**:

```
class Employee
{
    protected internal string Name = "Heer";
}

class Manager : Employee
{
    public void ShowName()
    {
        Console.WriteLine(Name); // ✅ Accessible in derived class
    }
}

class Program
{
    static void Main()
    {
        Employee emp = new Employee();
        Console.WriteLine(emp.Name); // ✅ Accessible in same assembly
```

```
    }
}
```

2.6 private protected Modifier (C# 7.2+)

- **Definition**: Member is accessible **only within the containing class or derived classes in the same assembly**.
- **Use Case**: Restrictive encapsulation with inheritance.
- **Example**:

```
class Employee
{
    private protected string Dept = "IT";
}

class Manager : Employee
{
    public void ShowDept()
    {
        Console.WriteLine(Dept); // ✅ Accessible
    }
}

class Program
{
    static void Main()
    {
        Manager m = new Manager();
        // Console.WriteLine(m.Dept); // ❌ Not accessible outside class hierarchy
    }
```

}

Access Modifiers from High to Low Visibility

1. **public** – Highest visibility
    - Accessible **from anywhere**, any class, any assembly.
2. **protected internal**
    - Accessible **anywhere in the same assembly** or **from derived classes in other assemblies**.
3. **internal**
    - Accessible **only within the same assembly/project**.
4. **protected**
    - Accessible **only in the class itself and its derived classes** (any assembly).
5. **private protected**
    - Accessible **only in the class itself and derived classes in the same assembly**.
6. **private** – Lowest visibility
    - Accessible **only within the class itself**.

Namespace & Libraries

What is a Namespace in .NET?

A **namespace** is a way to organize **classes, interfaces, structs, and other types** in .NET into a logical group. It helps prevent **naming conflicts** and makes code easier to read and maintain.

Think of a namespace like a **folder in a computer**. Multiple files (classes) can exist in different folders with the same name, but the folder (namespace) keeps them unique.

Why Use Namespaces?

1. **Avoid Naming Conflicts**
    - Two developers may create a class called Student.
    - Using namespaces, you can distinguish them:
    - SchoolManagement.Student
    - UniversityManagement.Student

2. **Organize Code**
    - Large applications have many classes. Grouping them in namespaces keeps the codebase clean.
3. **Easier Maintenance**
    - You can logically separate functionalities (e.g., DataAccess, UI, BusinessLogic).
4. **Enable Reusability**
    - Namespaces allow you to reuse classes from .NET libraries or your own libraries without worrying about conflicts.

Syntax

```
namespace NamespaceName
{
   class ClassName
   {
      // Class members here
   }
}
```

Example

```
using System;  // Importing System namespace
using MyApp.Utilities; // Importing custom namespace

namespace MyApp
{
   class Program
   {
      static void Main()
      {
         Console.WriteLine("Hello from MyApp!");
```

```
        Helper.SayHello(); // Using class from MyApp.Utilities namespace
    }
  }
}

// Another namespace for utility classes
namespace MyApp.Utilities
{
  class Helper
  {
    public static void SayHello()
    {
      Console.WriteLine("Hello from Helper class!");
    }
  }
}
```

Explanation:

1. System is a **built-in namespace**. We use it to access Console.
2. MyApp.Utilities is a **custom namespace** containing the Helper class.
3. Namespaces allow us to have Program and Helper classes with clear organization.

Key Points

- You can have **nested namespaces**:
- namespace MyApp.DataAccess
- {
- class Database { }
- }

- To use classes from a namespace, either:
    1. Use using NamespaceName; at the top, or
    2. Fully qualify the class name: NamespaceName.ClassName.

What is a .NET Library?

A **.NET library** is a collection of pre-written code, classes, interfaces, and functions provided by Microsoft (or third parties) that developers can use to build applications quickly. Instead of writing everything from scratch, you can leverage these libraries to perform common tasks efficiently.

In .NET, these libraries come in the form of **assemblies** (.dll files).

Purpose / Uses of .NET Libraries

1. **Code Reusability**
    - You can use pre-built classes and functions without writing them yourself.
    - Example: Instead of writing your own method to sort a list, you can use System.Collections.Generic.List<T>.Sort().
2. **Faster Development**
    - Libraries provide ready-to-use functionality like database access, file handling, network communication, etc.
    - Example: Using System.IO classes to read/write files instead of manually handling file streams.
3. **Standardization**
    - Ensures that developers use consistent methods to perform common tasks.
    - Example: Using DateTime class for all date/time operations.
4. **Reliability and Security**
    - .NET libraries are tested and optimized, reducing bugs and security risks in your code.
    - Example: Using System.Security.Cryptography for encryption instead of writing your own algorithm.
5. **Cross-Platform Support**
    - Libraries like System.Net.Http or System.Linq work across Windows, Linux, and macOS when using .NET Core or .NET 5/6/7+.

Commonly Used .NET Libraries

1. **System** – Basic classes like Console, Math, String, DateTime.
2. **System.Collections** – Classes for collections like List, Dictionary, Queue.
3. **System.IO** – File and stream handling.
4. **System.Data** – Working with databases (ADO.NET).
5. **System.Net** – Networking tasks (HTTP requests, sockets).
6. **System.Threading** – Multithreading and parallel programming.
7. **System.Linq** – Querying collections using LINQ.

Example Usage

```
using System;
using System.Collections.Generic; // Using a library for collections

class Program
{
   static void Main()
   {
      List<int> numbers = new List<int>() { 5, 2, 9, 1 };

      numbers.Sort(); // Using library function to sort
      foreach (int num in numbers)
      {
         Console.WriteLine(num); // Output: 1, 2, 5, 9
      }

      Console.WriteLine("Current Date and Time: " + DateTime.Now); // Using System library
   }
```

}

✅ Here, System and System.Collections.Generic libraries provide ready-made classes and methods so we don't have to implement sorting or date-time functionality ourselves.

Enumerations

What is an Enum?

An **Enum (short for Enumeration)** in C# is a distinct type that defines a set of **named constants** (integral values).
It is used when you have a collection of related values that make your code more **readable, maintainable, and less error-prone**.

🔹 **Declaration of Enum**

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

- By default, the underlying type is int.
- The first value is 0 unless specified otherwise.
- You can explicitly set values:

```
enum StatusCode
{
    Success = 200,
    BadRequest = 400,
```

```
    Unauthorized = 401,
    NotFound = 404
}
```

- **Why Use Enum?**

1. **Readability** – Instead of using magic numbers or strings:
2. if (status == 200)   // less readable
3. if (status == StatusCode.Success) // more meaningful
4. **Maintainability** – If values change, update only in enum instead of everywhere in code.
5. **Type Safety** – You can only assign predefined values (reduces errors).
6. **Performance** – Enums are value types stored as integers by default, so they are faster than strings.

Strings: Comparisons are slower (character by character). **Enums**: Stored as integers internally → much faster for comparisons and memory usage.

- **Accessing Enum Values**

```
StatusCode code = StatusCode.Success;

// Convert enum to int
int numValue = (int)StatusCode.NotFound;  // 404

// Convert int to enum
StatusCode newCode = (StatusCode)200; // Success

// Convert enum to string
string name = StatusCode.Unauthorized.ToString();  // "Unauthorized"
```

```
// Parse string to enum
StatusCode parsed = (StatusCode)Enum.Parse(typeof(StatusCode), "BadRequest");
```

🔹 **Best Practices with Enums in C#**

✅ **1. Use enums for fixed sets of related constants**
Example: OrderStatus, UserRole, DaysOfWeek.
❌ Don't use enums for values that are dynamic (e.g., fetched from DB).

✅ **2. Explicitly assign values if required**
Especially useful when enum values represent external codes (HTTP status codes, DB constants).

✅ **3. Use Flags Attribute for bit fields**
If multiple values can be combined, use [Flags]:

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

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

✅ **4. Validate Enum Values**
Use Enum.IsDefined or Enum.TryParse to ensure valid values:

```
if (Enum.IsDefined(typeof(StatusCode), 200))
    Console.WriteLine("Valid Enum");
```

✅ **5. Prefer switch or if with enums**
Makes decision-making clear and compiler can warn for missing cases.

```
switch (code)
```

```
{
    case StatusCode.Success:
        Console.WriteLine("Request successful");
        break;
    case StatusCode.NotFound:
        Console.WriteLine("Resource not found");
        break;
    default:
        Console.WriteLine("Unknown status");
        break;
}
```

✅ **6. Don’t overuse enums**

If values are **dynamic or need behavior**, consider using classes or records instead.

✅ **7. Keep enums small & cohesive**

Avoid mixing unrelated constants in one enum.

✅ **8. Naming convention**

- Use **PascalCase** for enum type name and members.
- Don’t use prefix/suffix like Enum or E_.

🔹 **Example: Enum in Real-world Use**

```
public enum OrderStatus
{
    Pending,
    Processing,
    Shipped,
    Delivered,
    Cancelled
}
```

```
class Program
{
   static void Main()
   {
      OrderStatus status = OrderStatus.Processing;


      switch (status)
      {
         case OrderStatus.Pending:
            Console.WriteLine("Order placed, awaiting confirmation.");
            break;
         case OrderStatus.Processing:
            Console.WriteLine("Order is being processed.");
            break;
         case OrderStatus.Shipped:
            Console.WriteLine("Order has been shipped.");
            break;
         case OrderStatus.Delivered:
            Console.WriteLine("Order delivered successfully.");
            break;
         case OrderStatus.Cancelled:
            Console.WriteLine("Order was cancelled.");
            break;
      }
   }
}
```

✅ **Summary of Enum Best Practices**

- Use for fixed sets of related constants.
- Assign explicit values when mapping to external systems.
- Use [Flags] for bitwise combinations.
- Validate enum inputs.
- Use switch statements for better clarity.
- Follow PascalCase naming.
- Don't use for dynamic values.

Data-Table in C#

A **DataTable** in C# is an in-memory representation of a single table of data.
It is part of **ADO.NET** (System.Data namespace).

👉 You can think of it as a **mini database table** stored in RAM.

🔹 **When to Use DataTable?**

- When you want to **hold tabular data in memory** (rows/columns).
- When fetching data from a **database** (e.g., SqlDataAdapter.Fill).
- When you need to **manipulate data** (filter, sort, compute) before displaying.
- When you want to pass tabular data around (e.g., between layers in an app).

🔹 **Creating a DataTable**

1. Basic Creation

```
using System;
using System.Data;


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

```
        // Create DataTable object
        DataTable dt = new DataTable("Employee");

        // Add columns
        dt.Columns.Add("ID", typeof(int));
        dt.Columns.Add("Name", typeof(string));
        dt.Columns.Add("Salary", typeof(double));
        dt.Columns.Add("JoiningDate", typeof(DateTime));

        // Add rows
        dt.Rows.Add(1, "John", 50000, DateTime.Now.AddYears(-2));
        dt.Rows.Add(2, "Alice", 60000, DateTime.Now.AddYears(-1));
        dt.Rows.Add(3, "Bob", 55000, DateTime.Now);

        // Display data
        foreach (DataRow row in dt.Rows)
        {
            Console.WriteLine($"{row["ID"]}, {row["Name"]}, {row["Salary"]},
{row["JoiningDate"]}");
        }
    }
}
```

✅ Output (example):

1, John, 50000, 30-09-2023 10:45:00

2, Alice, 60000, 30-09-2024 10:45:00

3, Bob, 55000, 30-09-2025 10:45:00

Accessing Data

```
// Access by column name
```

```
Console.WriteLine(dt.Rows[0]["Name"]);


// Access by column index
Console.WriteLine(dt.Rows[0][1]);


// Loop through rows
foreach (DataRow r in dt.Rows)
{
   Console.WriteLine($"{r["ID"]} - {r["Name"]}");
}
```

🔹 **Updating and Deleting Rows**

```
// Update
dt.Rows[0]["Name"] = "John Smith";


// Delete
dt.Rows[1].Delete();
```

⚡ Note: Deletion marks the row as "Deleted". Use AcceptChanges() to permanently remove it.

```
dt.AcceptChanges(); // Confirms update/delete changes
```

How DataTable Differs from Other Data Structures

Feature	Core Collections (List, Dictionary, etc.)	DataTable
Purpose	General-purpose data storage & manipulation	Relational/tabular data (like a DB table)
Organization	One-dimensional or key-value	Rows & Columns (tabular)
Type Safety	Strongly typed (generic List<int>)	Columns can have different types

Feature	Core Collections (List, Dictionary, etc.)	DataTable
Constraints	None (you code logic)	Can define Primary Keys, Unique, Foreign Keys
Data Relation	Not directly supported	Supports relations via DataSet
Use Case	Algorithms, collections, in-memory data	In-memory database operations, disconnected data from SQL

A **DataTable** in C# is an **in-memory data structure**.

- It stores data **temporarily** while the program is running.
- Once the program stops, the data inside the DataTable is lost (unless you explicitly save/export it to a **database, file, or XML/JSON**).

Why use DataTable if it's temporary?

1. **In-memory representation of database data**
    - Most of the time, we fetch data from a **database** into memory (using SqlDataAdapter, SqlCommand, etc.).
    - Instead of hitting the database again and again, we keep it in a **DataTable** so we can work with it quickly.
    - Example: Show employee records in a grid → store in DataTable → filter, sort, modify without DB round trips.
2. **Disconnected architecture**
    - In **ADO.NET**, once you fill a DataTable from the database, you can **disconnect** from the DB.
    - This improves performance and reduces database load, because you don't keep an open connection.
3. **Structured data handling in memory**
    - DataTable has **rows, columns, constraints, relations** just like a real database table.
    - You can apply **filtering, sorting, searching, relationships** between multiple DataTables (inside a DataSet) without needing SQL queries every time.

4. **Temporary staging area**
    - Often we do calculations, transformations, or validations on data **before saving it back** to a permanent storage.
    - Example: Import a CSV → load into DataTable → validate → if correct, save to database.
5. **Integration with UI controls**
    - Many UI frameworks (like WinForms DataGridView, WPF DataGrid, ASP.NET GridView) bind **directly** to a DataTable.
    - Makes displaying and editing data easy.
6. **Flexibility with data sources**

- You can build a DataTable manually (without a database) and still use it as a table-like structure in your program.
- Example: Temporary reports, caching, test data, offline processing.

Date/String/Math Classes and methods for manipulation

1. Date and Time in C# (System.DateTime, System.TimeSpan)

The DateTime structure is used to work with dates and times.

✅ **Common properties**

- DateTime.Now → Current date & time
- DateTime.Today → Current date (time = 00:00:00)
- DateTime.UtcNow → Current UTC date & time
- dt.Year, dt.Month, dt.Day, dt.Hour, dt.Minute, dt.Second

✅ **Common methods**

```
DateTime dt = DateTime.Now;


dt.AddDays(5);        // Adds 5 days
dt.AddMonths(2);      // Adds 2 months
dt.AddYears(-1);      // Subtracts 1 year
dt.AddHours(3);       // Adds 3 hours
```

```
dt.ToString("dd-MM-yyyy"); // Custom formatting

dt.ToString("MMMM dd, yyyy"); // e.g., September 30, 2025


DateTime.Parse("2025-09-30"); // Convert string to DateTime

DateTime.TryParse("2025/09/30", out DateTime result); // Safe parse


TimeSpan diff = DateTime.Now - new DateTime(2025, 01, 01);

Console.WriteLine(diff.Days); // Number of days difference
```

🔹 2. String Class (System.String)

Strings in C# are **immutable**, meaning once created, they cannot be changed.

✅ Common properties

- str.Length → Number of characters
- string.IsNullOrEmpty(str) → Checks null or empty
- string.IsNullOrWhiteSpace(str) → Checks null, empty, or whitespace

✅ Common methods

```
string s = "  Hello World  ";


s.ToUpper();          // "  HELLO WORLD  "

s.ToLower();          // "  hello world  "

s.Trim();            // "Hello World"

s.TrimStart();        // "Hello World  "

s.TrimEnd();          // "  Hello World"


s.Substring(2, 5);     // "Hello" from index 2

s.Replace("World", "C#"); // "  Hello C#  "

s.Contains("Hello");    // true

s.StartsWith("He");     // true
```

```
s.EndsWith("ld");      // true


string[] parts = s.Split(' ');  // Split into words
string joined = string.Join("-", parts); // Join with -


"Hello".Equals("HELLO", StringComparison.OrdinalIgnoreCase); // Case-insensitive comparison
```

✅ **StringBuilder (for mutable strings)**

```
using System.Text;


StringBuilder sb = new StringBuilder("Hello");
sb.Append(" World");
sb.Insert(0, "Say ");
sb.Replace("World", "C#");
Console.WriteLine(sb.ToString()); // Say Hello C#
```

🔹 **3. Math Class (System.Math)**

Provides common mathematical functions.

✅ **Common methods**

```
Math.Abs(-10);     // 10
Math.Max(5, 10);   // 10
Math.Min(5, 10);   // 5


Math.Sqrt(16);     // 4
Math.Pow(2, 3);    // 8 (2^3)
Math.Round(3.75);  // 4
Math.Floor(3.75);  // 3
Math.Ceiling(3.75);// 4
```

```
Math.Sin(0);      // 0
Math.Cos(0);      // 1
Math.Tan(Math.PI/4); // 1
Math.PI;          // 3.141592...
Math.E;           // 2.718...
```

File Operations

What are File Operations?

File operations are tasks that deal with creating, reading, writing, appending, deleting, and managing files on disk.

In C#, these are done using classes in the System.IO namespace.

- Important Classes in System.IO

Class	Purpose
File	Static methods for creating, copying, deleting, moving, and opening files.
FileInfo	Provides instance methods for file operations.
Directory	Static methods for creating, moving, and deleting directories.
DirectoryInfo	Instance methods for directories.
FileStream	Used to read/write bytes to a file.
StreamReader	For reading text files (character by character / line by line).
StreamWriter	For writing text files.
BinaryReader / BinaryWriter	For binary file operations.
Path	For handling file paths (combine, get extension, etc.).

- Basic File Operations

1. Creating a File

```
using System;
using System.IO;
```

```
class Program
{
   static void Main()
   {
      string path = "testfile.txt";

      // Using File class
      if (!File.Exists(path))
      {
         File.Create(path).Close(); // Create file and close it immediately
         Console.WriteLine("File created.");
      }

      // Using FileInfo class
      FileInfo fileInfo = new FileInfo("infofile.txt");
      if (!fileInfo.Exists)
      {
         fileInfo.Create().Close();
         Console.WriteLine("File created using FileInfo.");
      }
   }
}
```

2. Writing to a File

```
using System.IO;

class Program
```

```
{
    static void Main()
    {
        string path = "testfile.txt";

        // WriteAllText (overwrites existing content)
        File.WriteAllText(path, "Hello, this is the first line!\n");

        // AppendAllText (adds content to existing file)
        File.AppendAllText(path, "This line is appended.\n");

        // Using StreamWriter
        using (StreamWriter writer = new StreamWriter(path, true)) // true = append
        {
            writer.WriteLine("Another line using StreamWriter.");
        }
    }
}
```

3. Reading from a File

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string path = "testfile.txt";
```

```
        // Read entire content
        string content = File.ReadAllText(path);
        Console.WriteLine("File Content:\n" + content);

        // Read all lines
        string[] lines = File.ReadAllLines(path);
        foreach (string line in lines)
        {
            Console.WriteLine("Line: " + line);
        }

        // Using StreamReader
        using (StreamReader reader = new StreamReader(path))
        {
            string line;
            while ((line = reader.ReadLine()) != null)
            {
                Console.WriteLine("Read with StreamReader: " + line);
            }
        }
    }
}
```

4. Copying, Moving, and Deleting Files

```
using System.IO;

class Program
```

```
{
    static void Main()
    {
        string path = "testfile.txt";
        string copyPath = "copyfile.txt";
        string movePath = "movedfile.txt";

        // Copy
        File.Copy(path, copyPath, true); // overwrite = true
        Console.WriteLine("File copied.");

        // Move
        File.Move(copyPath, movePath);
        Console.WriteLine("File moved.");

        // Delete
        if (File.Exists(movePath))
        {
            File.Delete(movePath);
            Console.WriteLine("File deleted.");
        }
    }
}
```

5. Working with FileStream

```
using System;
using System.IO;
```

```
class Program
{
   static void Main()
   {
      string path = "filestream.txt";

      // Write using FileStream
      using (FileStream fs = new FileStream(path, FileMode.Create, FileAccess.Write))
      {
         byte[] data = System.Text.Encoding.UTF8.GetBytes("Hello FileStream!");
         fs.Write(data, 0, data.Length);
      }
// Read using FileStream
      using (FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read))
      {
         byte[] buffer = new byte[fs.Length];
         fs.Read(buffer, 0, buffer.Length);
         string text = System.Text.Encoding.UTF8.GetString(buffer);
         Console.WriteLine("FileStream Read: " + text);
      }
   }
}
```

6. Binary File Operations

```
using System;
using System.IO;

class Program
```

```
{
    static void Main()
    {
        string path = "binaryfile.dat";

        // Write binary data
        using (BinaryWriter writer = new BinaryWriter(File.Open(path, FileMode.Create)))
        {
            writer.Write(25);       // int
            writer.Write("Hello");  // string
            writer.Write(3.14);     // double
        }

        // Read binary data
        using (BinaryReader reader = new BinaryReader(File.Open(path, FileMode.Open)))
        {
            int number = reader.ReadInt32();
            string text = reader.ReadString();
            double pi = reader.ReadDouble();

            Console.WriteLine($"Read from binary: {number}, {text}, {pi}");
        }
    }
}
```

7. Directory Operations

```
using System;
using System.IO;
```

```
class Program
{
   static void Main()
   {
      string dirPath = "MyFolder";

      // Create directory
      if (!Directory.Exists(dirPath))
      {
         Directory.CreateDirectory(dirPath);
         Console.WriteLine("Directory created.");
      }

      // Get files and subdirectories
      string[] files = Directory.GetFiles(dirPath);
      string[] dirs = Directory.GetDirectories(dirPath);

      // Delete directory
      Directory.Delete(dirPath, true); // true = delete recursively
      Console.WriteLine("Directory deleted.");
   }
}
```

- File Modes in FileStream

Mode Description

Create Creates a new file. If file exists, overwrites it.

CreateNew Creates a new file. Throws error if file exists.

Open	Opens an existing file. Throws error if not found.
OpenOrCreate	Opens file if exists, otherwise creates new.
Append	Opens file if exists, and writes at the end. If not exists, creates it.
Truncate	Opens file and clears contents.