🕰️ 1. History of C# and .NET Framework

📘 What is .NET?

- **.NET Framework** is a **software development platform** created by **Microsoft** in early 2000s.
- It provides:
    - A **runtime environment** (called **CLR** – Common Language Runtime)
    - A rich **class library** (called **.NET Framework Class Library**)
- It allows you to build:
    - Console apps
    - Windows desktop apps
    - Web apps (ASP.NET)
    - Web services, APIs, etc.

⚙️ What is CLR?

- The **Common Language Runtime** is like a virtual machine.
- It handles:
    - Code execution
    - Memory management
    - Garbage collection
    - Exception handling
    - Security

C# code → compiled into **Intermediate Language (IL)** → executed by **CLR** on any Windows machine.

💡 C# Language Overview

- **C# (C Sharp)** is a **modern, object-oriented language** developed by **Microsoft** in 2000 under **Anders Hejlsberg**.
- It's inspired by **C++** and **Java**.
- Designed for simplicity, readability, and productivity.

🔄 Evolution

Version	Key Features
C# 1.0	Basic OOP, classes, structs, enums
C# 2.0	Generics, Nullable types
C# 3.0	LINQ, Lambda expressions
C# 5.0	Async/Await
C# 7.0	Tuples, Pattern Matching
C# 9.0	Records
C# 10+	Global usings, file-scoped namespaces

Today, C# runs on:

- **.NET Framework** (Windows-only, older)
- **.NET Core / .NET 5+** (cross-platform, modern)

🧱 2. Access Modifiers: `public`, `private`, `protected`, `internal`

Access modifiers define **who can access a class, method, or variable**.

Modifier	Accessibility	Description
`public`	Everywhere	Can be accessed from anywhere in your program or other assemblies.
`private`	Inside the same class only	Most restrictive. Members are hidden from outside code.

`protected`	Same class or derived classes	Allows access only within the class and its subclasses.
`internal`	Within same assembly/project	Accessible only within the same **.dll** or **.exe** file.
`protected internal`	Same assembly + subclasses	Accessible to derived classes or within the same project.
`private protected`	Derived + same assembly only	Accessible within derived classes that are in the same assembly.

🔹 Example:

```
public class Car
{
   private int speed;            // accessible only inside Car
   protected string model;       // accessible in Car + derived classes
   internal string brand;        // accessible within same project
   public void Start() {}        // accessible everywhere
}
```

Best practice:

- **Start restrictive** → use `private` by default.
- Open access only when required.

🧩 3. Using Namespaces and .NET Libraries

📦 What is a Namespace?

- A **namespace** organizes classes logically.
- Prevents name conflicts between classes.

Example:

```
namespace MyApp.Utilities
{
   public class MathHelper
   {
      public static int Add(int a, int b) => a + b;
```

```
    }
}
```

You can use it like:

```
using MyApp.Utilities;

Console.WriteLine(MathHelper.Add(5, 3)); // Output: 8
```

⚙️ System Namespaces (Built-in .NET Libraries)

The .NET Framework comes with rich **base class libraries** (BCL).

Common namespaces:

Namespace	Description
`System`	Core classes (Console, Math, String, etc.)
`System.Collections.Generic`	Generic lists, dictionaries, queues
`System.IO`	File input/output operations
`System.Linq`	Query operations (LINQ)
`System.Threading`	Threading & Tasks
`System.Net`	Networking (HTTP requests, sockets)

📘 Using `using` keyword

`using` tells the compiler **which namespace** to look in:

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

class Program
{
    static void Main()
    {
        List<int> numbers = new List<int> { 1, 2, 3 };
        Console.WriteLine(numbers.Count);
    }
}
```

Without `using`, you'd have to write full path:

System.Collections.Generic.List<int> numbers = new System.Collections.Generic.List<int>();

🔢 4. Enum Usage and Best Practices

📘 What is an Enum?

- `enum` is a special type that lets you define a set of **named constants**.
- It improves **readability** and avoids **magic numbers**.

Example:

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

Usage:

```
Days today = Days.Monday;
Console.WriteLine(today);         // Output: Monday
Console.WriteLine((int)today);    // Output: 1
```

🧠 Default Behavior

- By default, the first item = 0, and then increments by 1.
- You can assign custom values:

```
enum ErrorCode
{
    None = 0,
```

```
    NotFound = 404,
    ServerError = 500
}
```

✅ Best Practices

1. **Use meaningful names** – Make sure enum names clearly represent values.
2. **Avoid changing existing values** – It can break existing code.

Use explicit underlying type if needed:

```
enum Status : byte { Started = 1, Completed = 2 }
```

3.
4. **Use Enums for related constants only**, not arbitrary values.

Combine with **switch statements**:

```
switch (today)
{
    case Days.Sunday:
        Console.WriteLine("Holiday!");
        break;
    default:
        Console.WriteLine("Working day.");
        break;
}
```

🧠 What is a DataTable?

A **DataTable** is a class in the **System.Data** namespace that represents an **in-memory table** — just like a table in a database, but stored temporarily in your application.

It contains:

- **Columns** (schema)
- **Rows** (data)

- Can be part of a **DataSet**

You can think of it like an Excel sheet in memory.

🧱 Basic Structure

Namespace:

using System;
using System.Data;

To create a DataTable:

DataTable table = new DataTable("Students");

Now, you can define **columns** and **rows**.

🏗️ Step 1: Creating Columns

You can define columns by:

table.Columns.Add("ID", typeof(int));
table.Columns.Add("Name", typeof(string));
table.Columns.Add("Age", typeof(int));

Explanation:

- `"ID"`, `"Name"`, `"Age"` → column names
- `typeof(int)` or `typeof(string)` → column data type

So now your DataTable looks like this (in memory):

ID	Name	Age

🧩 Step 2: Adding Rows

You can add rows in multiple ways.

✅ Option 1: Using `Rows.Add()`

```
table.Rows.Add(1, "John", 20);
table.Rows.Add(2, "Sara", 22);
table.Rows.Add(3, "Mike", 19);
```

✅ Option 2: Creating a Row Manually

```
DataRow newRow = table.NewRow();
newRow["ID"] = 4;
newRow["Name"] = "Emma";
newRow["Age"] = 21;
table.Rows.Add(newRow);
```

👀 Step 3: Displaying Data

You can loop through all rows:

```
foreach (DataRow row in table.Rows)
{
    Console.WriteLine($"{row["ID"]} - {row["Name"]} - {row["Age"]}");
}
```

Output:

```
1 - John - 20
2 - Sara - 22
3 - Mike - 19
4 - Emma - 21
```

🔍 Step 4: Accessing Data

You can access individual values:

```
int age = (int)table.Rows[1]["Age"];
Console.WriteLine("Sara's Age: " + age);
```

Output:

```
Sara's Age: 22
```

✏️ Step 5: Updating Data

To modify a value:

```
table.Rows[0]["Age"] = 25;
Console.WriteLine("Updated John's age: " + table.Rows[0]["Age"]);
```

Output:

```
Updated John's age: 25
```

❌ Step 6: Deleting a Row

```
table.Rows[2].Delete();  // deletes row with index 2 (Mike)
```

To confirm:

```
foreach (DataRow row in table.Rows)
{
    if (row.RowState != DataRowState.Deleted)
        Console.WriteLine($"{row["Name"]}");
}
```

🔎 Step 7: Filtering and Selecting Rows

You can **query** data using `Select()`:

```
DataRow[] result = table.Select("Age > 21");

foreach (DataRow r in result)
{
    Console.WriteLine($"{r["Name"]} ({r["Age"]})");
}
```

Output:

```
Sara (22)
John (25)
```

✅ You can use conditions like `Age = 20`, `Name = 'John'`, `Age < 25`, etc.

🧮 Step 8: Sorting Data

```
DataRow[] sortedRows = table.Select("", "Age DESC");
foreach (DataRow r in sortedRows)
{
    Console.WriteLine($"{r["Name"]} - {r["Age"]}");
}
```

Output:

```
John - 25
Sara - 22
Emma - 21
```

🧰 Step 9: Cloning & Copying Tables

- **Clone()** → Copies structure only (no data)
- **Copy()** → Copies both structure + data

```
DataTable newTable = table.Copy();
Console.WriteLine("Copied rows: " + newTable.Rows.Count);
```

🗂️ Step 10: Using Primary Keys

You can set a column as a **Primary Key**:

```
table.PrimaryKey = new DataColumn[] { table.Columns["ID"] };
```

Now you can **find** rows quickly:

```
DataRow foundRow = table.Rows.Find(2);
if (foundRow != null)
    Console.WriteLine("Found: " + foundRow["Name"]);
```

💡 When to Use DataTable?

Use Case	Why DataTable?
Working with database data	You can fill it from SQL queries
Need table-like structure in memory	It behaves like an in-memory table
No need for full ORM like EF	Lightweight and simple
Temporary data storage	Good for calculations, reports

🧱 Example: Complete Program

```
using System;
using System.Data;

class Program
{
    static void Main()
    {
        // Create DataTable
        DataTable table = new DataTable("Students");

        // Add columns
        table.Columns.Add("ID", typeof(int));
        table.Columns.Add("Name", typeof(string));
        table.Columns.Add("Age", typeof(int));

        // Add rows
        table.Rows.Add(1, "John", 20);
        table.Rows.Add(2, "Sara", 22);
        table.Rows.Add(3, "Mike", 19);

        // Update data
        table.Rows[0]["Age"] = 25;

        // Filter and display
        DataRow[] result = table.Select("Age > 20");
        Console.WriteLine("Students older than 20:");
        foreach (DataRow row in result)
        {
            Console.WriteLine($"{row["ID"]}: {row["Name"]} ({row["Age"]})");
        }
    }
```

```
}
```

✅ Output:

```
Students older than 20:
1: John (25)
2: Sara (22)
```

📘 Summary

Concept	Description
DataTable	In-memory table to store structured data
Columns.Add()	Defines schema
Rows.Add()	Adds data rows
Select()	Filters rows
Copy() / Clone()	Duplicate structure/data
PrimaryKey	Sets unique identifier
RowState	Tracks changes (Added, Deleted, Modified)

Classes & methods for manipulating DataTable

1) The core classes you'll work with

- `DataTable` — an in-memory table (columns + rows).

- `DataRow` — represents a single row in a `DataTable`.
- `DataColumn` — describes a column (name, type, expression, read-only, etc.).
- `DataRowCollection` / `DataColumnCollection` — `table.Rows`, `table.Columns`.
- `DataSet` — container for multiple `DataTable`s and `DataRelation`s.
- `DataView` — a *live* view (filter + sort) of a `DataTable`.
- `DataAdapter` (e.g., `SqlDataAdapter`) — fills a `DataTable` from DB and can persist changes back.
- `DataTableReader` — read-only `IDataReader`-like cursor over `DataTable`.

---

2) Create table / columns / primary key / constraints

```
var table = new DataTable("Students");


// add columns
table.Columns.Add("ID", typeof(int));
table.Columns.Add("Name", typeof(string));
table.Columns.Add("Age", typeof(int));


// set primary key for fast lookup
table.PrimaryKey = new DataColumn[] { table.Columns["ID"] };


// add a unique constraint on Name (example)
table.Constraints.Add(new UniqueConstraint(table.Columns["Name"]));
```

Why: Primary keys make `Rows.Find()` fast and are essential for `Merge` behavior.

---

3) Add rows — `NewRow()`, `Rows.Add()`, `LoadDataRow()` and `ItemArray`

```
// option 1: Add with values

table.Rows.Add(1, "John", 20);


// option 2: NewRow then add (useful if you want to set selectively)

DataRow r = table.NewRow();

r["ID"] = 2;

r["Name"] = "Sara";

r["Age"] = 22;

table.Rows.Add(r);


// option 3: LoadDataRow (updates existing row if PK found, otherwise inserts)

table.LoadDataRow(new object[] { 2, "Sara Updated", 23 }, LoadOption.PreserveChanges);


// access many columns via ItemArray

object[] rowValues = r.ItemArray;
```

---

4) Read/iterate rows

```
// simple iteration
```

```
foreach (DataRow row in table.Rows)
    Console.WriteLine($"{row["ID"]}: {row["Name"]} ({row["Age"]})");

// index access
var firstName = table.Rows[0]["Name"];

// safer strongly-typed access using Field<T>()
int? ageNullable = table.Rows[0].Field<int?>("Age"); // handles DBNull
int age = table.Rows[0].Field<int>("Age");
```

Use `Field<T>` over direct casts — it handles `DBNull` nicely and supports nullable types.

---

5) Update & delete rows — `RowState`, `AcceptChanges`, `Delete` vs `Remove`

```
// update a value
table.Rows[0]["Age"] = 25; // RowState becomes Modified

// mark a row for deletion (keeps information until AcceptChanges or Update)
table.Rows[1].Delete();   // RowState = Deleted

// remove immediately from collection (no RowState tracking)
table.Rows.Remove(table.Rows[0]); // permanently removed

// Accept or revert changes
table.AcceptChanges();     // All rows -> RowState.Unchanged
```

table.RejectChanges(); // Undo changes to Last AcceptChanges boundary

Important:

- `Delete()` marks a row as deleted (so DataAdapter can send DELETE).
- `Remove()` physically removes it from `Rows` and **does not** mark it in `RowState` for later DB update.
- `AcceptChanges()` resets `RowState` → DataAdapter will consider no changes to send.

`DataRowState` values: `Detached`, `Added`, `Unchanged`, `Modified`, `Deleted`.

---

6) Find / Select / Filter / Sort

`Rows.Find()` (fast when PK set)

DataRow found = table.Rows.Find(2);

if (found != null) Console.WriteLine(found["Name"]);

`Select(filter, sort)`

DataRow[] rows = table.Select("Age > 20", "Age DESC");

`DataView` (live, mutable view)

var dv = new DataView(table) { RowFilter = "Name LIKE 'A%'", Sort = "Age DESC" };

DataTable filtered = dv.ToTable(); // snapshot from the view

---

7) Aggregates — `Compute()`

object sumObj = table.Compute("SUM(Age)", "Age > 20");

int sum = sumObj == DBNull.Value ? 0 : Convert.ToInt32(sumObj);

object avg = table.Compute("AVG(Age)", "");

`Compute` supports aggregate functions like `SUM`, `AVG`, `COUNT`, `MIN`, `MAX`.

---

8) Computed columns (DataColumn.Expression)

table.Columns.Add("Quantity", typeof(int));

table.Columns.Add("UnitPrice", typeof(decimal));

table.Columns.Add("TotalPrice", typeof(decimal), "Quantity * UnitPrice"); // expression column

This evaluates automatically for each row.

---

9) Clone / Copy / Import / Merge

- `Clone()` — copies schema only (columns, constraints) — no data.
- `Copy()` — copies schema + data.
- `ImportRow(row)` — imports a `DataRow` keeping its `RowState` (useful when merging while retaining state).
- `Merge(source, preserveChanges, missingSchemaAction)` — merges another table into this one (matches by PK).

DataTable schemaOnly = table.Clone();

DataTable copy = table.Copy();

var other = new DataTable(); // assume filled

table.Merge(other, preserveChanges: true, missingSchemaAction:
MissingSchemaAction.Add);

`Merge` is useful for combining server data or syncing changes.

Basic file reading & writing in C# — a complete beginner-friendly guide

Working with files is one of the most common tasks in real apps: config files, logs, CSVs, images, binary blobs, etc. Below I'll explain the important classes and patterns, show clear code examples (text + binary + async + safe writes), and finish with practical tips and gotchas.

---

1) Key namespaces & types

using System;

using System.IO;

using System.Text;

using System.Threading.Tasks;

Common types:

- `File` / `FileInfo` — quick static helpers (ReadAllText/WriteAllText, Exists, Copy, Delete).
- `FileStream` — low-level stream for reading/writing bytes.
- `StreamReader` / `StreamWriter` — high-level text reading/writing over a stream.
- `BinaryReader` / `BinaryWriter` — read/write binary primitives (int, string, etc.).
- `Path` — manipulate file paths (`Path.Combine`, `GetExtension`, etc.).
- `Directory` / `DirectoryInfo` — work with directories.
- `FileSystemWatcher` — watch filesystem changes (optional).

---

2) Quick text-file examples

Read whole file (small files)

```
string contents = File.ReadAllText("notes.txt", Encoding.UTF8);
```

Write whole file (overwrite)

```
File.WriteAllText("notes.txt", "Hello, world!", Encoding.UTF8);
```

Read all lines into an array

```
string[] lines = File.ReadAllLines("notes.txt", Encoding.UTF8);
```

Iterate lazily over lines (large files)

```
foreach (string line in File.ReadLines("bigfile.txt", Encoding.UTF8))
{
    Console.WriteLine(line);
```

```
}
```

`ReadLines` **is lazy** — it yields lines one-by-one and does not load the whole file into memory.

---

3) StreamReader / StreamWriter — recommended for controlled read/write

Read line-by-line (sync)

```
using (var sr = new StreamReader("notes.txt", Encoding.UTF8))
{
    string line;
    while ((line = sr.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
}
```

Write with overwrite (sync)

```
using (var sw = new StreamWriter("log.txt", append: false, encoding: Encoding.UTF8))
{
    sw.WriteLine("Log started: " + DateTime.Now);
}
```

Append to file

```
using (var sw = new StreamWriter("log.txt", append: true))
```

```
{
    sw.WriteLine("New log entry");
}
```

> `using` ensures the stream is closed and disposed even on exceptions. In C# 8+ you can use `using var sw = new StreamWriter(...);` as a short form.

---

4) Async text I/O (modern & non-blocking)

```
static async Task AsyncExample()
{
    string path = "async.txt";
    await File.WriteAllTextAsync(path, "Hello async!", Encoding.UTF8);

    string contents = await File.ReadAllTextAsync(path, Encoding.UTF8);
    Console.WriteLine(contents);

    // Async streaming
    using var sr = new StreamReader(path, Encoding.UTF8);
    while (!sr.EndOfStream)
    {
        string line = await sr.ReadLineAsync();
        Console.WriteLine(line);
    }
}
```

Use async for UI apps or servers to avoid blocking threads.

---

5) Binary read/write (FileStream + BinaryReader/BinaryWriter)

Write some binary data

```
using (var fs = new FileStream("data.bin", FileMode.Create, FileAccess.Write))
using (var bw = new BinaryWriter(fs, Encoding.UTF8))
{
    bw.Write(123);          // Int32
    bw.Write(3.14);         // Double
    bw.Write("Hello binary");  // String
}
```

Read it back

```
using (var fs = new FileStream("data.bin", FileMode.Open, FileAccess.Read))
using (var br = new BinaryReader(fs, Encoding.UTF8))
{
    int a = br.ReadInt32();
    double b = br.ReadDouble();
    string s = br.ReadString();
}
```

Binary formats are compact and fast — good for numbers, images, serialized objects. Make sure reader/writer agree on the layout.

---

6) Low-level buffered read (good for very large files)

```
using (var fs = new FileStream("huge.bin", FileMode.Open, FileAccess.Read,
FileShare.Read))

{

    byte[] buffer = new byte[81920]; // 80 KB buffer (System default)

    int bytesRead;

    while ((bytesRead = fs.Read(buffer, 0, buffer.Length)) > 0)

    {

        // Process bytesRead bytes in buffer

    }

}
```

Or async:

```
int bytesRead = await fs.ReadAsync(buffer, 0, buffer.Length);
```

---

7) File modes, access & sharing — why they matter

When constructing `FileStream`, you control behavior:

```
new FileStream(path, FileMode.Create, FileAccess.Write, FileShare.None);
```

Common `FileMode` values:

- `Create` — create or overwrite.
- `CreateNew` — create, throw if exists.
- `Open` — open existing, throw if missing.
- `OpenOrCreate` — open if exists, else create.
- `Append` — open to append (writes go to end).
- `Truncate` — open and truncate to zero length.

`FileAccess` = `Read` / `Write` / `ReadWrite`
`FileShare` controls other processes' access (e.g., `FileShare.Read` allows others to read while you have it open).

---

8) Path & directory helpers

string combined = Path.Combine("data", "reports", "2025-10-02.csv");

string dir = Path.GetDirectoryName(path);

string ext = Path.GetExtension(path);

string name = Path.GetFileName(path);

string temp = Path.GetTempPath();

Create directories before writing:

Directory.CreateDirectory("data/reports");

---

9) File existence & common operations

if (File.Exists(path))

```
{
    File.Copy(path, "backup.txt", overwrite: true);
    File.Delete("oldfile.txt");
    File.Move("a.txt", "b.txt"); // move / rename
}
```

---

10) Error handling — what to catch

Common exceptions:

- `IOException` — general I/O errors (file locked, disk full).
- `UnauthorizedAccessException` — no permission.
- `DirectoryNotFoundException` / `FileNotFoundException`
- `PathTooLongException` (older Windows)
  Wrap I/O in try/catch and handle or log errors gracefully.

```
try
{
    File.WriteAllText(path, "hello");
}
catch (UnauthorizedAccessException ex)
{
    Console.WriteLine("Permission denied: " + ex.Message);
}
catch (IOException ex)
{
```

```
    Console.WriteLine("I/O error: " + ex.Message);
}
```