🧱 Abstract Classes

💡 Concept:

An **abstract class** is a **base class** that cannot be instantiated directly —
you can **only inherit** from it.

It is meant to define a **common structure or behavior** for its child classes.

It can contain:

- **Abstract methods** (no implementation — must be overridden)
- **Concrete methods** (with implementation)
- **Fields, properties, and constructors**

🧠 Why use abstract classes?

- To define a **template** or **blueprint** for subclasses
- To **enforce** that child classes must provide some methods
- To share **common code** across derived classes

💡 Example:
You might have a base `Vehicle` class defining `Start()` and `Stop()` methods,
and each specific vehicle (Car, Bike, Truck) must implement its own behavior.

🔒 Sealed Classes

💡 Concept:

A **sealed class** is the **opposite** of abstract class.
It **cannot be inherited**.

This is used when you want to **prevent inheritance** — to make the class **final**.

🧠 Why use sealed classes?

- To **prevent modification** through inheritance
- For **security** or **performance** (compiler optimizations)
- In frameworks, some classes are sealed intentionally (like `String`, `Math`, etc.)

💡 Example:
`System.String` is sealed — you can't inherit and modify how strings work.

🔗 Interfaces

💡 Concept:

An **interface** is a **pure contract** — it contains only **method signatures**, **properties**, **events**, or **indexers**, but **no implementation**.

Any class that **implements** an interface **must define all members**.

👉 Think of an interface as a **"promise"**:
"If you implement me, you must provide these behaviors."

🧠 Why use interfaces?

✅ **Multiple inheritance** (C# supports only single class inheritance, but multiple interfaces)
✅ Helps in **loose coupling** (you depend on interface, not concrete class)
✅ Makes **unit testing & mocking** easier
✅ Encourages **code contracts**

Feature	Abstract Class	Interface
Can be instantiated	❌ No	❌ No
Can contain implementation	✅ Yes	✅ Default methods (C# 8+)
Can have fields	✅ Yes	❌ No
Supports multiple inheritance	❌ No	✅ Yes
Use case	Shared code + enforce rules	Enforce contract only

🧩 What Are Generics?

Generics allow you to **define classes, methods, and collections** that can **work with any data type** — while maintaining **type safety** and **performance**.

In simple terms:

> Instead of writing different versions of a class or method for `int`, `string`, `double`, etc., you write one **generic** version that works with any type.

🧱 Generic Classes

A **generic class** lets you define a type placeholder (like `T`) that is decided when you **create an object**.

⚙️ Generic Methods

A **generic method** allows defining a type parameter **inside a method** (not for the whole class).

📦 Generic Collections (From `System.Collections.Generic`)

Instead of using old non-generic collections (`ArrayList`, `Hashtable`), use **type-safe generic ones**.

🔹 List<T>

Dynamic array storing only a specific type.

🔹 Dictionary<TKey, TValue>

Stores key-value pairs (like maps).

🔹 Queue<T>

FIFO (First In First Out)

🔹 Stack<T>

LIFO (Last In First Out)

🔹 HashSet<T>

Stores unique items (no duplicates)

🧱 File Operation In Depth

Class	Purpose / Use	Notes
`File`	Static class with many file-related utility methods (create, delete, copy, read, write)	Good for simple tasks
`FileInfo`	Represents a file; has instance methods & properties for more control	Useful if you do multiple operations on same file
`Directory`	Static helper for directory (folder) operations (create, delete, list, move)	Similar to `File` but for directories
`DirectoryInfo`	Represents a directory; instance-based API	More object-oriented control
`Path`	Static class for manipulating path strings (combine, get extension, get filename)	Helps avoid mistakes with path separators
Streams & Readers/Writers: `FileStream`, `StreamReader`, `StreamWriter`, `BinaryReader`, `BinaryWriter`	For lower-level or streaming I/O (reading in chunks, reading line by line, binary data)	Useful when file is large or you need fine control

Key Namespaces / Classes

- `System.IO` — primary namespace
- `File`, `FileInfo`
- `Directory`, `DirectoryInfo`

- `Path`
- Streams: `FileStream`, `StreamReader`, `StreamWriter`, `BinaryReader`, `BinaryWriter`

Path Helpers

- `Path.Combine(...)`
- `Path.GetExtension()`, `GetFileName()`, `GetDirectoryName()`

File Static Methods (via `File`)

Method	Purpose
`WriteAllText(path, str)`	Create/write text (overwrite)
`WriteAllLines(path, string[])`	Write many lines
`ReadAllText(path)`	Read entire file as string
`ReadAllLines(path)`	Read all lines into string[]
`ReadLines(path)`	Lazy enumerate lines
`AppendAllText(path, str)`	Append text
`AppendAllLines(path, lines)`	Append lines
`Exists(path)`	Check if file exists
`Copy(src, dest, overwrite)`	Copy file
`Move(src, dest)`	Move or rename
`Delete(path)`	Delete file
`Open(path, mode, access)`	Get FileStream

`Replace(source, destination, backup)`	Replace with backup

FileInfo & Metadata (instance)

- `fi.Name`, `fi.FullName`
- `fi.Length` (size in bytes)
- `fi.CreationTime`, `fi.LastAccessTime`, `fi.LastWriteTime`
- `fi.Exists`
- `fi.MoveTo(dest)`
- `fi.Delete()`
- `fi.Open(...)`, `fi.OpenRead()`, `fi.OpenWrite()`

Directory Methods / DirectoryInfo

- `Directory.CreateDirectory(path)`
- `Directory.GetFiles(path)`, `Directory.GetDirectories(path)`
- `Directory.Delete(path, recursive)`

- `Directory.Exists(path)`
- `Directory.Move(src, dest)`
- `DirectoryInfo` instance methods: `Create()`, `Delete()`, `GetFiles()`, `GetDirectories()`
- `Directory.GetLogicalDrives()` lists drives

Stream / Reader / Writer

- `FileStream` for byte-level I/O
- `StreamWriter` / `StreamReader` for text
- `BinaryWriter` / `BinaryReader` for binary formats

JSON Serialization (C# - modern)

- Namespace: `System.Text.Json`
- Key Types / Methods:
  - `JsonSerializer.Serialize(object, options)`
  - `JsonSerializer.Deserialize<T>(jsonString, options)`

- Options you often set: pretty-print (indented), ignore nulls, custom property names (`[JsonPropertyName]`), custom converters for special types (dates etc.)
- Behavior: extra properties in JSON ignored by default, missing properties get default values unless you enforce required properties.

XML Serialization (C#)

- Namespace: `System.Xml.Serialization`
- Key Type: `XmlSerializer`
- Methods: `Serialize(stream/writer, object)`, `Deserialize(stream/reader)`
- Requirements: class must be public, have public parameterless constructor; only public properties/fields get serialized
- Attributes:
  - `[XmlIgnore]` – skip
  - `[XmlAttribute]` – attribute vs element
  - `[XmlRoot]`, `[XmlElement]`, etc. for naming / namespaces

Pros & Cons JSON vs XML

JSON	XML
+ Lightweight, more compact, faster parse for simple data	+ Stronger schema support (XSD), namespaces, rich metadata
+ More common in modern APIs and web	+ Easier for document models, more established tools (XPath, XSLT)
– Less good if you need built in comment support, or complex validation	– More verbose, more overhead, can be slower to parse/serialize for large object graphs

🧱 Important .NET libraries

Namespace	Purpose	Common Classes / Methods
`System`	Core classes	Console, Math, String, DateTime, Random
`System.Collections`	Collections (non-generic)	ArrayList, Hashtable
`System.Collections.Generic`	Collections (generic)	List, Dictionary<TKey,TValue>, Stack
`System.IO`	File & directory operations	File, Directory, StreamReader, StreamWriter
`System.Linq`	Querying collections	Where, Select, Sum, OrderBy
`System.Threading`	Threading	Thread, Mutex
`System.Threading.Tasks`	Async programming	Task, async, await
`System.Net`	Networking	HttpClient, WebRequest
`System.Text`	Encoding & string manipulation	StringBuilder, Encoding

🧱 Lambda Syntax and Usage

Concept	Syntax / Example	Purpose / Use
Expression Lambda	`x => x * x`	Short, single expression
Statement Lambda	`(a, b) => { int s = a + b; return s; }`	Multiple statements
No parameters	`() => 42` or `() => { ... }`	Lambda with no input

Assign to Func / Action / Predicate	`Func<int, int> f = x => x + 2Action<string> a = s => Console.WriteLine(s)Predicate<int> p = n => n % 2 == 0`	Map lambdas to suitable delegate types
LINQ usage	`list.Where(x => x > 10)list.Select(x => x * 2)`	Filtering, mapping, etc.
Explicit types	`(int x, int y) => x + y`	Use when inference is ambiguous or you want clarity
Discards / unused params	`(_, _) => 42`	Ignore parameters you don't need
Expression tree lambda	`Expression<Func<T, bool>> expr = x => x > 5`	Used in IQueryable / LINQ-to-SQL / EF

🧱 Delegates

Concept	Example	Return Type	Common Use
Custom Delegate	`delegate void Notify(string msg);`	Custom (defined by you)	Events, notifications
Action<T>	`Action<Employee> print = e => ...;`	void	Printing, logging
Func<T, TResult>	`Func<Employee, decimal> calc = e => e.Salary * 0.1m;`	TResult	Transformations, computations
Predicate<T>	`Predicate<Employee> isHighEarner = e => e.Salary > 50000;`	bool	Testing, filtering

🧱 What is an Extension Method?

- An extension method is really just a **static method** defined in a **static class**, but thanks to syntax, you can call it as if it were an instance method of the type being extended.
- The trick is: the **first parameter** of that static method is prefixed with `this`, which tells the compiler: "this method extends that type."
- When calling, you don't pass that first parameter explicitly — it's the object on which you call the method.

Rules / Requirements & Important Points

1. **Static class**
   You must define extension methods inside a static (top-level) class. You can't put them in a non-static class.
2. **Static method**
   The method itself must be static.
3. **First parameter with `this`**
   The first parameter indicates which type you are "extending." For example: `this string s` means you're adding a method to `string`
4. **Cannot access private members**
   The extension method cannot reach into private fields or private methods of the extended type—only its public (or internal, etc.) surface.
5. **Instance method preference**
   If the type already has an instance method with the same name & signature, that instance method is preferred over your extension. Your extension is used only if no matching instance method exists.
6. **Namespace & using**
   To use extension methods, you need to include (via `using`) the namespace in which your static class is defined. If you forget, the extension method won't show up in IntelliSense.

7. **Don’t overuse them**
   While handy, too many extensions, especially on very general types (like `object`), can clutter IDE suggestions (IntelliSense) and make code harder to navigate.