# ✅ 1. What is the .NET Framework?

**📌 Concept:**

* A **software framework developed by Microsoft**.
* It provides:
  + A **runtime environment** (CLR)
  + A **huge library** (FCL) for common functions (I/O, DB, web, etc.)
* Runs mainly on **Windows**.

### Intermediate Language (IL) / Microsoft Intermediate Language (MSIL) / Common Intermediate Language (CIL)

This is a very important concept for your MCQs.

* **What is it?** When you write C# code (or code in any other .NET language like VB.NET, F#), it's *not* directly compiled into machine code (the binary instructions your computer's processor understands). Instead, it's first compiled into an intermediate, CPU-independent set of instructions called **Intermediate Language (IL)**. It's also often referred to as **Microsoft Intermediate Language (MSIL)** or **Common Intermediate Language (CIL)**. They all mean the same thing.

**🧠 Real-world Analogy:**

Think of .NET like a **school system**:

* CLR is the **principal**, managing everything
* FCL is the **library** and **tools**
* C#/VB.NET are **teachers** giving lessons (your code)

**✅ 2. What is IL (Intermediate Language)?**

**📌 Concept:**

* C# is **not compiled directly to machine code**.
* It's compiled into **MSIL (Microsoft Intermediate Language)**.
* At runtime, **JIT** (Just-In-Time Compiler) converts IL → Machine Code.

**🔍 Why use IL?**

* IL allows **language interoperability**. 🡸
* All .NET languages (C#, VB.NET, F#) compile into the **same IL**.

**✅ 3. What are Assemblies?**

**📌 Concept:**

* Assemblies are the **compiled output** of your C# program.

**Types of Assemblies:**

* **Executable files (.exe):** These are applications that can be directly run (e.g., a desktop application or a console application). They always contain an entry point (like the Main method in a C# console app).
* **Dynamic Link Libraries (.dll):** These are libraries of code that other applications or assemblies can use. They don't have an entry point and cannot be run directly. Think of them as toolboxes that provide specific functionalities to other programs. For example, a DLL might contain code for handling database operations or complex calculations.

**📦 Each assembly contains:**

* **Metadata** (information about the code: classes, methods, version) This metadata is crucial for the CLR.
* **IL Code**
* **Manifest** A special part of the metadata that describes the assembly's identity, its files, and its referenced assemblies

**✅ 4. CLR – Common Language Runtime**

**📌 Role of CLR:**

It is the **execution engine** of .NET Framework.

It provides:

| **Feature** | **Role** |
| --- | --- |
| **JIT Compiler** | Converts IL to machine code |
| **Memory Management** | Allocates memory for objects |
| **Garbage Collection** | Frees unused memory |
| **Security** | Validates code safety |
| **Exception Handling** | Manages errors in programs |
| **AppDomain** | Isolates applications |

Unmanaged code (like C++ code compiled directly to machine code without a runtime like CLR) requires manual memory management and typically relies on the operating system for security, not an intrinsic runtime like CLR providing these services.

**✅ Let's Explore Each CLR Component**

**💡 4.1 JIT Compilation (Just-in-Time)**

* Converts **IL → Machine Code at runtime**
* Only converts **what is needed** → performance benefit
* Types of JIT:
  + **Pre-JIT**: Entire code at once (used rarely)
  + **Econo-JIT**: Used on resource-limited devices
  + **Normal JIT**: Most common, just what’s needed now

**💡 4.2 Memory Management**

* Done automatically by CLR
* Developers don’t use malloc / free like in C/C++

**💡 4.3 Garbage Collection**

**GC** automatically:

* Tracks **which objects are still in use**
* Frees **unreachable objects**

➡️ It uses a **Mark-and-Sweep** algorithm

🔁 Happens in **Generations**:

* Gen 0: Short-lived objects (local variables)
* Gen 1: Survived 1 GC
* Gen 2: Long-lived objects (like static fields)

🧠 **Best Practices**:

* Avoid unnecessary object creation
* Use IDisposable and using block for cleanup

**💡 4.4 AppDomain Management**

* A **lightweight process boundary** inside CLR
* Multiple .NET apps run in **isolated AppDomains**

**💡 4.5 CLS & CTS**

| **Term** | **Meaning** |
| --- | --- |
| **CLS** – Common Language Specification | Ensures **code written in one language** can be **used in another** |
| **CTS** – Common Type System | Defines the **data types** used in all .NET languages (e.g., int in C# = Int32) |

**💡 4.6 Security**

* CLR enforces **code access security**
* Prevents **unauthorized access** or memory leaks
* Uses **role-based security, permission sets**

 **i. The MSIL code includes instructions to load, initialize and invoke methods on objects.**

* **TRUE**. IL is a low-level, object-oriented instruction set. It contains all the necessary instructions to manage objects, call methods, perform operations, etc. It's the "blueprint" for your program's logic.

 **ii. The MSIL code is collected and assembled in the form of byte codes and is converted to a .NET assembly.**

* **TRUE**. When you compile your C# code, the MSIL is packaged along with metadata (information about your code) into an **assembly**. These assemblies are typically .ex

 **i) Managed code is the code that is executed directly by the CLR.**

* **TRUE**. Managed code is called "managed" because its execution is *managed* by the CLR. The CLR handles things like JIT compilation, garbage collection, and security.

 **ii) Managed code directly compiles to the machine code and runs on the machine where it has to be compiled.**

* **FALSE**. As we discussed, managed code first compiles to IL, then the CLR's JIT compiler converts it to machine code *at runtime*. It doesn't compile directly to machine code at the initial compilation step.

 **iii) Unmanaged code does not have services such as security or memory management.**

* **TRUE**. Unmanaged code (like C++ code compiled directly to machine code without a runtime like CLR) requires manual memory management and typically relies on the operating system for security, not an intrinsic runtime like CLR providing these services.

**🔁 Q36. DLR vs CLR**

✅ DLR = **Dynamic Language Runtime**, supports dynamic operations at runtime  
✅ CLR = Manages memory, execution  
➡️ So, correct answer is **DLR**

**🔁 Q38. C# compiler in .NET Core**

✅ It is **Roslyn**, a modern compiler as a service

**✅ Summary of Session 1 Key Concepts**

| **Topic** | **Purpose** |
| --- | --- |
| .NET Framework | Platform for running C# code |
| IL | Common compiled code |
| Assemblies | EXE/DLL containers |
| CLR | Execution engine |
| JIT | Converts IL to machine code |
| Garbage Collector | Frees unused memory |
| AppDomain | Isolates applications |
| CLS/CTS | Interoperability between languages |
| Security | Code access and execution safety |

# .NET Framework vs .NET Core vs Mono vs Xamarin

| **Feature** | **.NET Framework** | **.NET Core (.NET 5+)** | **Mono** | **Xamarin** |
| --- | --- | --- | --- | --- |
| Platform | Windows only | Cross-platform (Windows, Linux, macOS) | Cross-platform | Mobile apps |
| App Types | Desktop, Web | Web, Cloud, Console | Embedded devices | iOS & Android |
| Development Focus | Legacy enterprise apps | Modern, cloud-native apps | Light devices | Mobile UI |
| Open Source? | No | Yes | Yes | Yes |
| Runtime | CLR | CoreCLR | Mono Runtime | Mono Runtime with Xamarin libraries |

**💡 Important Notes:**

* .NET Core is now merged into **.NET 5, 6, 7...** (modern unified platform)
* Xamarin uses Mono under the hood but provides mobile-specific APIs
* Mono was developed for **Linux and embedded systems**

**🔁 Related MCQ:**

No direct MCQ in your list, but concepts may be asked like:  
**"Which platform is suitable for mobile cross-platform development?" → Xamarin**

**✅ 2. Versions of the .NET Framework**

| **Version** | **Key Features** |
| --- | --- |
| 1.0 – 2.0 | Initial release, Windows Forms |
| 3.0 – 3.5 | WPF, WCF, LINQ |
| 4.0 – 4.8 | TPL, ASP.NET enhancements |
| .NET Core 1.0–3.1 | Cross-platform, high performance |
| .NET 5+ | Unified platform (Windows, Linux, mobile, cloud, AI) |

**⚠️ .NET Framework is no longer being enhanced**

➡️ Future is with **.NET 6, 7, 8 (LTS versions)**

**✅ 3. Managed vs Unmanaged Code**

**📌 Managed Code:**

* Runs **under the control of CLR**
* Examples: C#, VB.NET
* Has features like:
  + Garbage Collection
  + Security checks
  + Type safety
  + Exception handling

**📌 Unmanaged Code:**

* Runs **outside CLR**
* Examples: C/C++, Win32 APIs
* Manual memory management
* Faster, but **no runtime protection**

**✅ 4. Introduction to Visual Studio**

**📌 Visual Studio:**

* The **main IDE** for .NET developers
* Features:
  + IntelliSense
  + Debugging
  + Code Refactoring
  + UI Designer
  + Git Integration
  + Extensions (Resharper, etc.)
  + Supports C#, VB.NET, F#, C++

**🧠 Quick Tip:**

Use **“.NET Core Console App”** for learning core concepts  
➡️ It is minimal and cross-platform

**📌 ILDASM – Intermediate Language Disassembler**

* A tool that **shows the IL code** of compiled assemblies
* It comes with **Visual Studio Developer Command Prompt**
* You can open your .exe or .dll and see:
  + IL code
  + Metadata
  + Manifest
  + Method structure

**🔍 Why Use ILDASM?**

* Useful for understanding:
  + How C# code translates to IL
  + What metadata exists in assemblies
  + Code analysis and debugging

**🔁 Related MCQs Recap:**

| **Question** | **Topic** | **Answer** |
| --- | --- | --- |
| Q10 | Managed Code vs Unmanaged | **i-True, ii-False, iii-True** |
| Q35 | MSIL Code | **i-True, ii-True** |
| Q36 | Runtime for dynamic operations | **DLR** |
| Q38 | Compiler for .NET Core | **Roslyn** |

**✅ Summary of Session 2 Key Concepts**

| **Concept** | **Summary** |
| --- | --- |
| .NET Framework vs Core | .NET Core is cross-platform and open source |
| Mono/Xamarin | Mono for embedded, Xamarin for mobile |
| Managed Code | Runs on CLR with memory/safety features |
| Unmanaged Code | C/C++ outside CLR |
| ILDASM | Tool to see IL code |
| Visual Studio | IDE to build, debug, deploy .NET apps |

# ✅ 1. Console Applications & Class Libraries (.NET Core)

**🔹 Console Application:**

* Small program that runs in terminal/command-line.
* Created using template:  
  dotnet new console

**🔹 Class Library:**

* A **.dll project** that contains reusable code (no Main() method).
* Class libraries do not have a Main method and cannot be run directly. They are "referenced" by other projects (like a console application or a web application) that then use the code within them.
* Used when you want to create:
  + Utility code
  + Shared business logic
  + Data models, etc.

➡️ In Visual Studio:

Class Library = DLL (no output window)  
Console App = EXE (with Main())

**✅ 2. C# Basics – Class, Methods, Fields**

csharp

class Student {

public string Name;

public int RollNo;

public void Display() {

Console.WriteLine($"Name: {Name}, RollNo: {RollNo}");

}

}

**✅ 3. using and Project References**

* using System; → Imports namespace.
* To use code from another project, you add a **project reference**.



**✅ 4. Data Types & CTS Equivalents**

| **C# Type** | **CTS Type** |
| --- | --- |
| int | System.Int32 |
| float | System.Single |
| double | System.Double |
| string | System.String |
| bool | System.Boolean |

📌 Why it matters?  
To ensure **cross-language compatibility** in .NET (C#, VB.NET, F#), all types are mapped to **CTS**. insures that all .NET languages use a common set of data types at the CLR level. This allows for seamless interaction between code written in different .NET languages.

**✅ 5. Methods Overloading**

* Same method name, different parameter count/type.

csharp

public void Print(string msg) { }

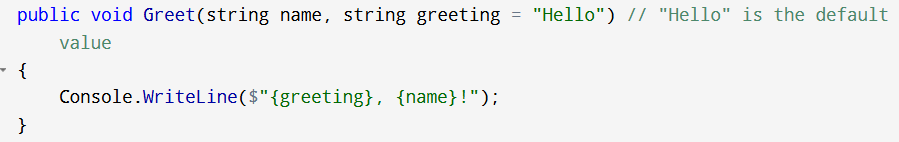
public void Print(int x) { }

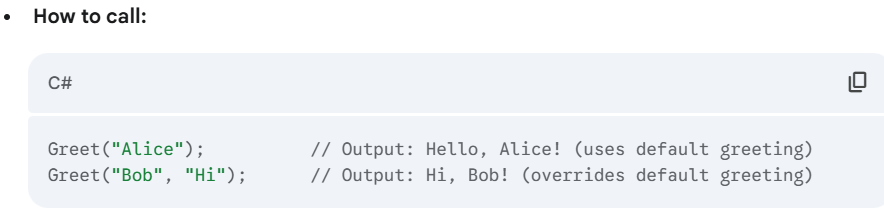
public void Print(string msg, int x) { }

**✅ 6. Optional, Named, Positional Parameters**

**🔸 Optional Parameters:**

Parameters that have a default value assigned in the method signature. This means you don't *have* to provide a value for them when calling the method; if you don't, the default value will be used.

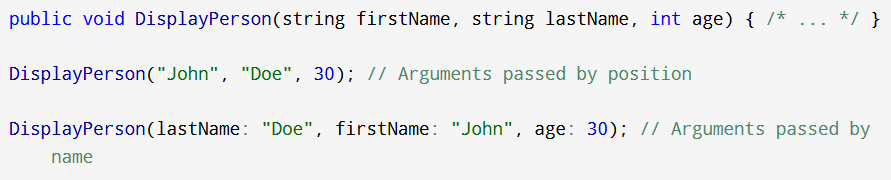




**🔸 Named Parameters:**

you can specify the parameter name when you pass the argument.

**🔸 Positional Parameters (Normal Way):** You pass arguments to a method in the order they are defined in the method's signature.

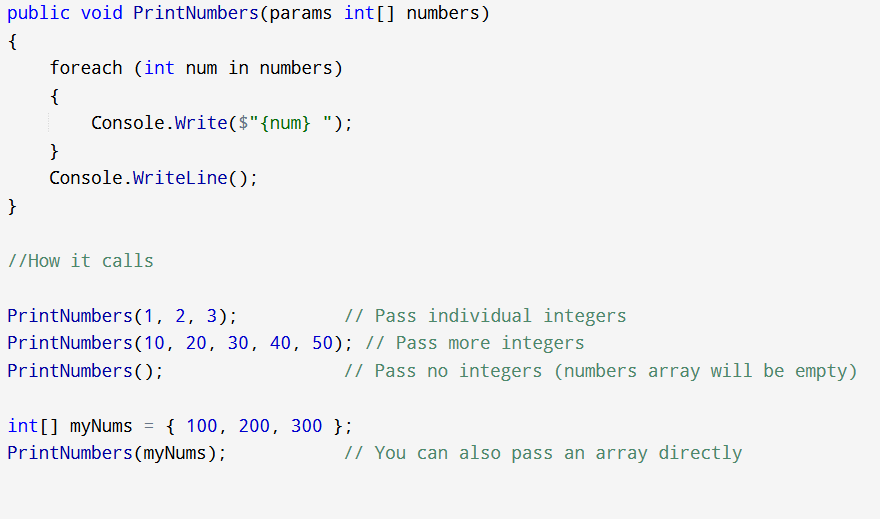


**✅ 7. params Keyword**

Allows **variable number of arguments**:

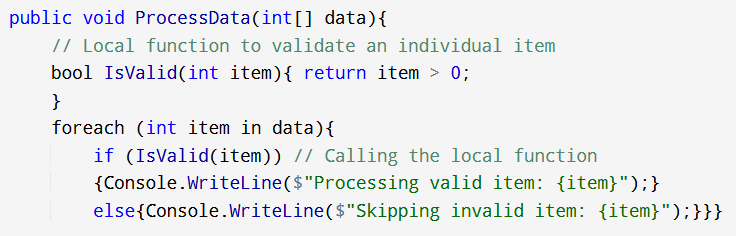
It essentially lets you pass an array of arguments, but you can pass them as individual items.

**Syntax:** The **params** keyword is used with an array parameter, and it must be the last parameter in the method signature.



**✅ 8. Local Functions**

Functions defined **inside other methods**:



**✅ 9. Properties (get, set, readonly)**

public class Student {

private string name;

public string Name {

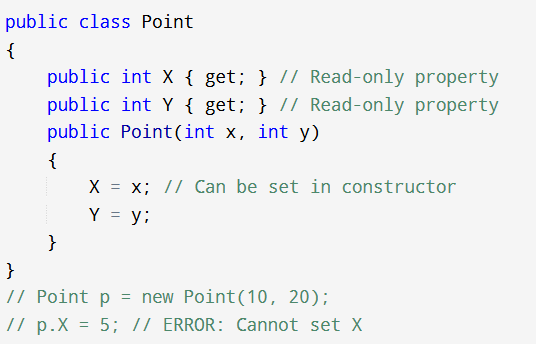
get { return name; }

set { name = value; }

}

}

**🔸 Readonly Properties:**



**✅ 10. Constructors**

* Automatically called when object is created.
* Can be **overloaded**.

public Student(string name) {

this.Name = name;

}

**✅ 11. Object Initializer**

****

**✅ 12. Destructors (~)**

Used to **clean up resources** before object is destroyed. And called by the Garbage Collector (GC) just before an object is destroyed and its memory is reclaimed.

~Student() {

Console.WriteLine("Destructor called");

}

⚠️ Runs **non-deterministically** by GC. Don’t depend on it.

**✅ 13. IDisposable (Discussed here, implemented in Session 5)**

Used when your class **uses unmanaged resources** (like files, DB).

You implement Dispose() to release them.

public void Dispose() {

// cleanup code

}

**🔁 Q28 – Destructor Operator**

➡️ Answer: **~** (used before class name like ~Student())

**🔁 Q33 – Constructor Return Type**

public int Employee() { Empld = 1001; }

❌ Constructor must **not have return type**  
✅ Answer: **Return type is not possible in constructor**

**🔁 Q34 – List<T>.Contains() Return Type**

✅ It's used to check presence of item → returns bool

✅ Answer: **Boolean**

**🧠 Summary – Session 3 Key Concepts**

| **Concept** | **Summary** |
| --- | --- |
| Console/Class Lib | Console → EXE, Class Lib → DLL |
| C# Basics | Classes, fields, methods |
| Method Overloading | Same name, diff. params |
| Optional/Named Params | Clean method calls |
| params | Variable args |
| Local Functions | Nest methods inside methods |
| Properties | Controlled field access |
| Constructors | Auto run on object creation |
| Destructors | Called by GC |
| ref, out | Pass variables by reference |
| IDisposable | Manual resource cleanup |

# Static Members, Inheritance, Abstract, Sealed

**✅ 1. Static Members**

**🔸 Static Field / Method / Property:**

* Belongs to the **class**, not to instance.
* **Key Idea:** Static members are shared among all instances of a class. There's only one copy of a static member, regardless of how many objects (or zero objects) of that class you create.
* **Accessed using class name**, not object.

class Counter {

public static int count = 0;

public static void ShowCount() {

Console.WriteLine("Count = " + count);

}

}

➡️ Usage:

Counter.count = 5;

Counter.ShowCount();

**🔸 Static Constructor:**

* Runs **once per type**, before any object is created.
* No parameters, no access modifier.
* you cannot explicitly call them. The CLR calls them automatically.

static Counter() {

count = 1;

}

**✅ 2. Static Classes**

* Contains only **static members**.
* Cannot be instantiated or inherited.

static class MathUtils {

public static int Square(int x) => x \* x;

}

➡️ Usage:

int sq = MathUtils.Square(5); // 25

**✅ 3. Inheritance in C#**

class Animal {

public void Eat() {

Console.WriteLine("Eating...");

}

}

class Dog : Animal {

public void Bark() {

Console.WriteLine("Barking...");

}

}

➡️ Dog inherits from Animal

The reason that C# does not support multiple inheritances is because of Revisit

Choose the best option

Method collision

Name collision

Function collision

Interface

**Explanation:** The primary reason C# (and many other object-oriented languages like Java) does not support multiple inheritance of classes is the **"Diamond Problem"** or **Method/Name Collision**.

Imagine Class C inherits from A and B. Both A and B inherit from Root. If Root has a method DoSomething(), and both A and B override it in different ways, how does C know which version of DoSomething() to inherit? This ambiguity is the "method collision" or "name collision" problem.

Therefore, the correct answer for **Question #15** is: **Method collision**

**🡺Inheritance is transitive in nature.**

**✅ 4. Access Specifiers Recap**

| **Modifier** | **Access Scope** |
| --- | --- |
| public | Everywhere |
| private | Within the same class only |
| protected | In same class and derived classes |
| internal | Accessible within the same assembly (.exe or .dll). |
| protected internal | Same assembly + derived classes in other assemblies. |

**private protected:** Accessible within the same assembly AND from derived classes in the same assembly.

**✅ 5. Constructors in Inheritance**

* Base class constructor runs **before** derived class. 🡺b**ase class's constructor is always called first**, implicitly or explicitly.
* You can call base constructor using base(...)

class Base {

public Base(int x) { }

}

class Derived : Base {

public Derived() : base(10) { }

}

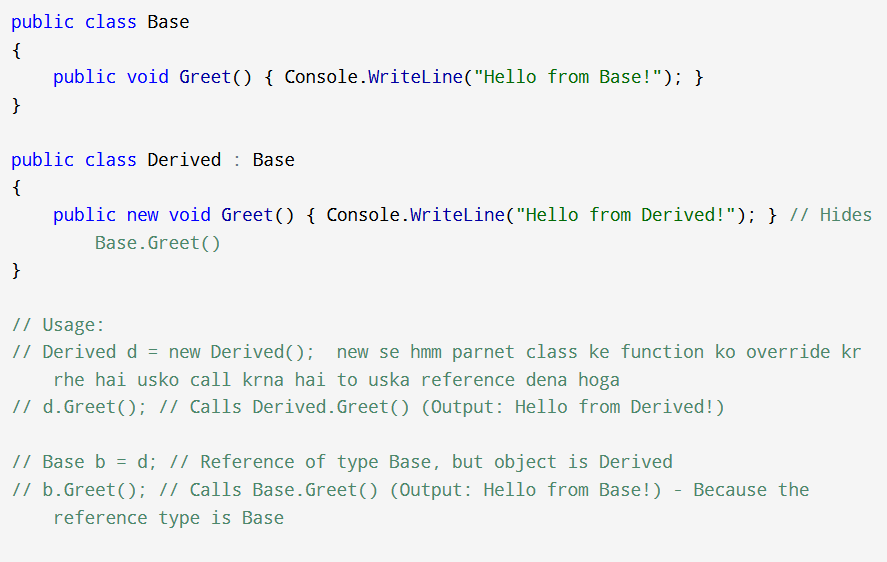
**✅ 6. Overloading vs Hiding**

**🔹 Overloading:**

* Same name, different parameters (same class or derived)

**🔹 Hiding with new:**

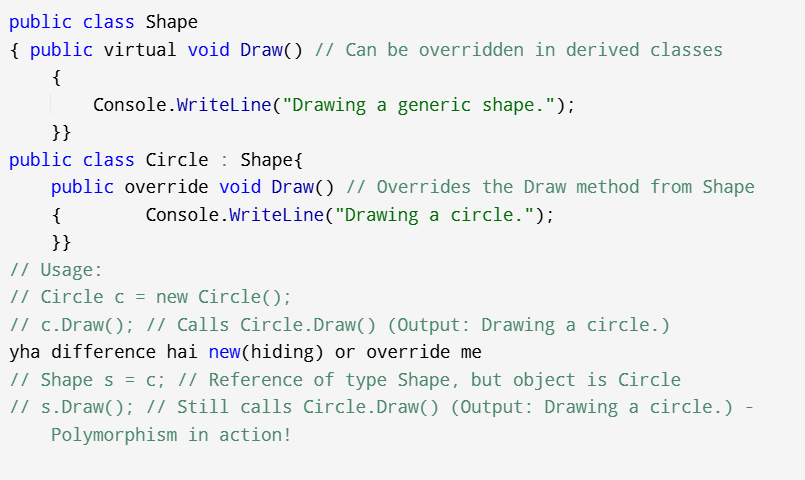
* If base and derived class have same method signature
*  **Compiler Warning:** The compiler will give a warning if you hide a base class member without explicitly using the new keyword.
*  **new keyword:** You use the new keyword explicitly to tell the compiler you *intend* to hide the base class member and suppress the warning.



**✅ 7. Method Overriding**

**Requirements:**

* Method must be virtual or abstract in base
* Derived class uses override
* The method signature (name, parameters, return type) in the derived class must exactly match the base class method.



Agar hmm virtula ke place pr new kr ke banate to hmm Shape s = c //reference dene ke baad s.Draw() krte to parent class ka draw method call hota pr

Override me ye paraent class ko override kr deta hai refence dene ke baad bhi hm usko call nhi kr shakte hai

**✅ 8. Sealed Methods and Classes**

**🔹 sealed Method:**

* Prevents further overriding.
* To stop the inheritance chain for a particular virtual method

public sealed override void Show() { }

**🔹 sealed Class:**

* Cannot be inherited.
* sealed class FinalClass { }

**✅ 9. Abstract Classes and Methods**

* **Cannot be instantiated**
* Must be inherited
* Can contain:
  + Abstract methods (no body)
  + Concrete methods (with body)

 Declared with the abstract keyword.

 Cannot be instantiated (new AbstractClass() is an error).

 Can contain both abstract members and non-abstract (concrete) members.

 If a class contains any abstract members, the class itself must be declared abstract.

 Derived classes *must* implement all abstract members of their abstract base class (unless the derived class is also abstract).

abstract class Shape {

public abstract void Draw();

public void Move() => Console.WriteLine("Move");

}

class Circle : Shape {

public override void Draw() => Console.WriteLine("Drawing Circle");

}

**🔁 ✅ Related MCQs from Your List**

**🔁 Q23 – Inheritance is \_\_\_\_ in nature**

✅ Answer: **Transitive**  
Explanation: If A → B → C, then C inherits from both A and B indirectly → **transitivity**

**🔁 Q37 – Which modifier forces derived class to implement method?**

✅ Answer: **Abstract**  
Explanation: Abstract method must be overridden in derived class

**🔁 Q29 – How can external users raise your class event?**

✅ Answer: **Add a public method that raises the event**  
➡️ event keyword prevents external raising — solution: wrap event trigger in a public method

**🧠 Summary – Session 4 Key Concepts**

| **Concept** | **Summary** |
| --- | --- |
| Static Member | Belongs to class |
| Static Class | Cannot be instantiated |
| Inheritance | Reuse of base members |
| Access | public, private, protected etc. |
| Constructors | Base → Derived order |
| new | Hides base method |
| override | Overrides base virtual method |
| sealed | Prevents overriding |
| abstract | Must be overridden in derived class |

# Interfaces, IDisposable, Operator Overloading

**✅ 1. What is an Interface?**

An **interface**:

* Defines a **contract** (set of method/property signatures)
* **Does NOT contain implementation** (prior to C# 8)
* Cannot contain fields
* Supports **multiple inheritance**
* Cannot be instantiated directly (new IMyInterface() is an error).
* Do not have constructors.
* All members declared in an interface are implicitly public and abstract (cannot have access modifiers like public, private, protected).
* it's a way to specify "what a class can do" without specifying "how it does it."

**⚠️ A class must implement all members of an interface.**

**// This is our interface (the contract)**

**// It starts with 'I' by convention (like IPlayable)**

**public interface IPlayable**

**{**

**void PlaySound(); // Every playable thing must have a PlaySound method**

**void Move(); // Every playable thing must have a Move method**

**}**

**✅ 2. Implementing an Interface**

interface IDevice {

void Start();

void Stop();

}

class Printer : IDevice {

public void Start() => Console.WriteLine("Printer started");

public void Stop() => Console.WriteLine("Printer stopped");

}

➡️ Printer is now said to **implement** the IDevice interface.

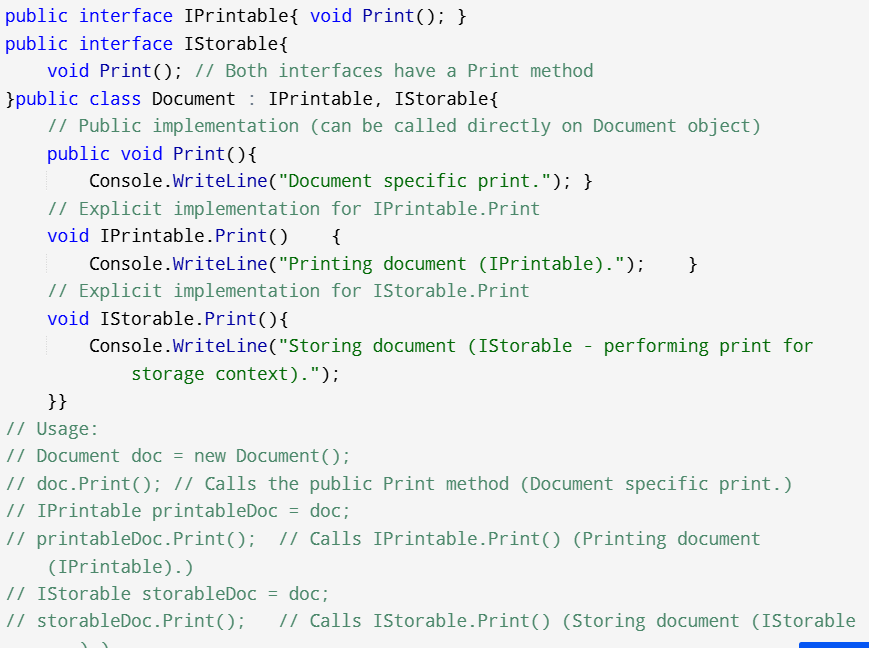
**✅ 3. Explicit Interface Implementation**

Sometimes, a class might implement two interfaces that both declare a member with the same name and signature. Or, you might want to hide an interface member from direct public access on the class instance, only exposing it when the instance is cast to the interface type. In these cases, you can use explicit interface implementation.

**Key point:** Explicitly implemented members are implicitly private and can only be accessed by casting the object to the interface type.

Used when:

* You want to **hide** interface methods from outside access.
* Or when implementing **multiple interfaces with same method name**



**✅ 4. Interface Inheritance**

Just like classes, **interfaces can inherit from other interfaces**:

any class implementing the derived interface must implement *all* members from both the derived interface and its base interfaces.



**✅ 5. Default Interface Methods (C# 8+)**



Usage:

// EmailNotifier email = new EmailNotifier();

// email.SendNotification("Meeting reminder!");

// email.LogNotification("Meeting reminder sent by email."); // Uses default implementation

// SmsNotifier sms = new SmsNotifier();

// sms.SendNotification("Appointment at 3 PM.");

// sms.LogNotification("Appointment SMS sent."); // Uses overridden implementation

**✅ 6. Operator Overloading**

You can **overload operators** to work with user-defined types.

class Complex {

public int real, imag;

public static Complex operator +(Complex a, Complex b) {

return new Complex { real = a.real + b.real, imag = a.imag + b.imag };

}

}

Which of the following keyword is used to overload user defined types? Choose the best option

Op

Opoverload

Operator

Operatoroverload

Based on our explanation, the correct answer is: **O Operator**

* **Example: Overloading + operator for a Vector class:**

**⚠️ Only allowed on:**

* Binary operators (+, -, \*, etc.)
* Unary (++, --, !, etc.)
* Comparison (==, !=)

🚫 Not allowed for logical (&&, ||), conditional (?:)

**🔁 MCQ Match:**

**Q6: Keyword to overload user-defined types**  
✅ Answer: operator  
Explanation: Use operator keyword to define custom behavior for operators in a class.

**✅ 7. Implementing IDisposable(From System namespace)**

Used to **release unmanaged resources** (like files, DB connections).

class FileManager : IDisposable {

public void Dispose() {

Console.WriteLine("Releasing resources...");

}

}

➡️ Best used with using block:

using (var fm = new FileManager()) {

// do something

} // Dispose called automatically

**Why IDisposable over destructors?** Destructors are non-deterministic; you don't know exactly when the GC will run and call them. IDisposable allows you to explicitly and immediately release resources.

**✅ 8. Implementing IComparable (From System namespace)**

Used for **sorting objects** in collections.

Allows objects of a class to be compared with other objects of the same type, enabling sorting.

**Method:** It defines a single method: int CompareTo(object? obj).

* Returns a negative value if the current instance is less than obj.
* Returns zero if the current instance is equal to obj.
* Returns a positive value if the current instance is greater than obj.

class Student : IComparable<Student> {

public int Marks;

public int CompareTo(Student other) {

return this.Marks.CompareTo(other.Marks);

}

}

➡️ Now you can sort:

List<Student> students = new List<Student>();

students.Sort(); // uses CompareTo logic

**🔁 ✅ MCQs Linked to Session 5**

**🔁 Q6 – Keyword to overload user-defined types**

✅ Answer: operator  
You must use the operator keyword when overloading in C#

**🔁 Q17 – Choose incorrect statement about delegates**

✔ This is **related**, because operator overloading and interface implementations are both used in advanced method handling.

❌ Incorrect option: **Delegates are not type-safe**  
✅ Delegates *are* type-safe

**🔁 Q25 – Definition of Extension Methods**

(This is from Session 10, but linked to interface + IDisposable usage.)

Correct:

“Extension methods are static methods that extend functionality **without modifying original type**.”

➡️ Often used with IDisposable or interface-based helper methods.

**✅ Real-Life Example: Interface + IDisposable + Overload**

interface IShape {

double Area();

}

class Circle : IShape, IDisposable {

public double Radius;

public double Area() => 3.14 \* Radius \* Radius;

public void Dispose() {

Console.WriteLine("Circle disposed");

}

public static Circle operator +(Circle a, Circle b) =>

new Circle { Radius = a.Radius + b.Radius };

}

**🧠 Summary – Session 5 Key Concepts**

| **Concept** | **Summary** |
| --- | --- |
| Interface | Contract-only, no implementation |
| Explicit Impl. | Interface method hidden from object |
| Interface Inheritance | Interface extends other interface |
| Operator Overload | Use operator keyword |
| IDisposable | Cleanup unmanaged resources |
| IComparable | Sorting custom objects |

# ✅ Session 6 – Value Types, Arrays, Nullable, Indexers

**Memory in a Nutshell:**

* **Stack:** A region of memory used for small, temporary data like method parameters, local variables, and return addresses. It's very fast. Data is added and removed in a Last-In, First-Out (LIFO) manner.
* **Heap:** A region of memory used for larger, dynamically allocated objects. It's slower than the stack, but it allows objects to live longer (beyond the scope of the method where they were created). The Garbage Collector manages the heap.

**✅ 1. Value Types vs Reference Types**

**Value Types**

* What they are: Types whose variables directly contain their data. When you assign a value type variable to another, a copy of the actual data is made**.**

**Reference Types**

* What they are: Types whose variables do not directly contain their data. Instead, they contain a reference (like a memory address or pointer) to where the actual data (the object) is stored on the heap.

| **Value Type** | **Reference Type** |
| --- | --- |
| Stores **data directly** | Stores **reference (address)** |
| Stack-allocated | Heap-allocated |
| int, char, bool, struct, enum | **class, interface, delegate, array, string** | |

**❗ MCQ Hint:**

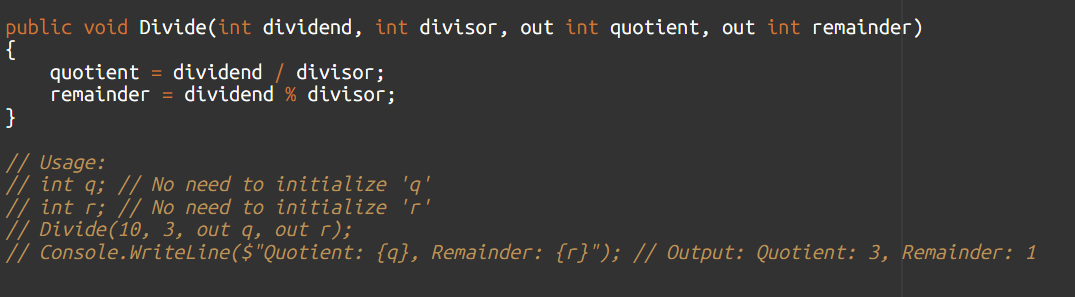
You had a question based on parameter passing (ref/out) — value types are copied unless explicitly passed by reference.

**✅ 2. ref, out, and in Parameters**

| **Keyword** | **Use Case** | **Initialization** |
| --- | --- | --- |
| ref | Pass value **by reference**, method can read/write | Must be initialized before call |
| out | Method returns **more than one** value | Must be initialized **inside** the method |
| in | Read-only reference (C# 7.2+) | Must be initialized before call, cannot modify |

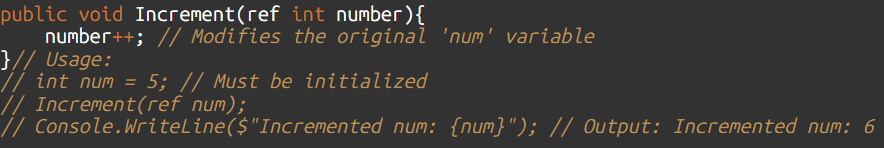
**out keyword:**

* Purpose: Used when a method needs to return multiple values or when it needs to initialize a variable that you pass into it.



**ref keyword:**

* **Purpose:** Used when a method needs to modify the original value of a variable passed into it.
* **Rule:** The variable *must be initialized* in the calling method before it's passed to the ref parameter. The method can then read and modify the original variable.



**Key Difference (for MCQs):**

* out: Variable **doesn't** need to be initialized before passing, method **must** assign a value.
* ref: Variable **must** be initialized before passing, method **can** modify it.

**✅ 3. struct – Value Type**

**What they are:** User-defined value types. Like classes, they can have fields, methods, and properties

* Similar to class but **value type**
* Cannot inherit from another struct or class (implicitly sealed.)
* Can implement interfaces
* Good for small, lightweight objects

public struct Point // 'Point' is now a value type

{

public int X;

public int Y;

public Point(int x, int y)

{

X = x;

Y = y;

}

public void Move(int dx, int dy)

{

X += dx;

Y += dy;

}

}

// Usage:

Point p1 = new Point(10, 20); // p1 holds the actual Point data

Point p2 = p1; // p2 gets a COPY of p1's data

p2.X = 30; // Change p2's X coordinate

Console.WriteLine($"p1: ({p1.X},{p1.Y}), p2: ({p2.X},{p2.Y})"); // Output: p1: (10,20),

//p2: (30,20)

🡺 // p1 was not affected.

**✅ 4. enum(Enumerations) – Named Constants**

What they are: A special kind of value type that defines a set of named integer constants. They make your code more readable and less error-prone by replacing "magic numbers" with descriptive names.

public enum DayOfWeek // Defines a set of named constants

{

Sunday, // Default value 0

Monday, // Default value 1

Tuesday, // Default value 2

Wednesday,

Thursday,

Friday,

Saturday

}

public enum StatusCode : int // You can specify the underlying integer type

{

Success = 200,

NotFound = 404,

Error = 500

}

// Usage:

DayOfWeek today = DayOfWeek.Wednesday;

Console.WriteLine($"Today is {today}"); // Output: Today is Wednesday

Console.WriteLine($"The integer value of today is {(int)today}"); // Output: The integer value of today is 3

StatusCode result = StatusCode.Success;

Console.WriteLine($"Status: {result} (Code: {(int)result})"); // Output: Status: Success (Code: 200)

* Internally stored as integers (Red = 0, Green = 1…)
* You can set explicit values: enum Status { Open = 1, Close = 2 }

**✅ 5. Nullable types (Value Types)**

* What they are: By default, value types (like int, bool, struct) cannot be null. They always hold a value. Nullable types allow you to assign null to value type variables.
* Syntax: Add a ? after the value type name.

int? x = null;

* **Purpose:** Useful for database fields that can be null, or optional parameters.

**Nullable reference types (C# 8.0+)**

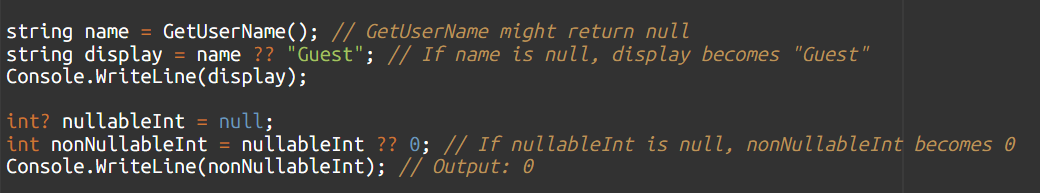
* What they are: By default, reference types *can* be null. Nullable reference types (a feature introduced in C# 8.0) provide a way for the compiler to warn you about potential null reference exceptions *before* you run your code. It's a compile-time feature to improve code safety.
* Syntax: Add a ? after the reference type name to *allow* it to be null, or mark it without ? to indicate it's *non-nullable* (and the compiler will warn you if you assign null to it without proper checks).

string? firstName = null; // Compiler knows firstName MIGHT be null, no warning here. string lastName = "Doe"; // Compiler assumes lastName will NOT be null. // Console.WriteLine(firstName.Length); // Compiler would warn you here about //potential null dereference! // (Unless you add null checks: if (firstName != null) ...)

Purpose: To reduce NullReferenceException errors at runtime by shifting the responsibility of null-checking to the compiler**.**

**🔹 Null-Coalescing Operator ??**

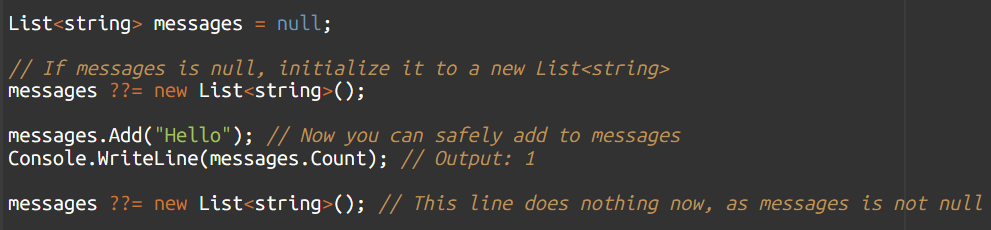
**Provides a default value for an expression if the expression is null.**

**🔹 Null-Coalescing Assignment ??=**

 **Purpose:** Assigns the right-hand operand to the left-hand operand *only if* the left-hand operand is null.

 **Syntax:** variable ??= value

 **Meaning:** If variable is null, then variable is assigned value. Otherwise, variable remains unchanged.



**✅ 6. Arrays in C#**

**🔹 Single-Dimensional Array**

int[] nums = new int[5];

**🔹 Multi-Dimensional Array**

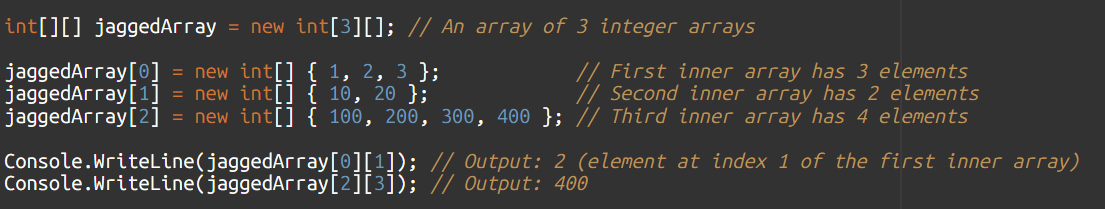
int[,] matrix = new int[3,3];

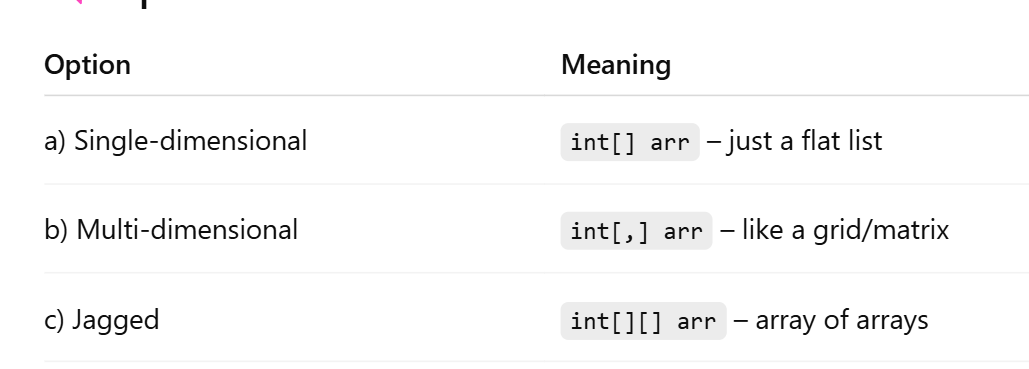
➡️ Access: matrix[0,1]

**🔹 Jagged Array (array of arrays)**

An array whose elements are *arrays*. Each inner array can have a different size.

Syntax: Use multiple sets of square brackets.

****

****

**✅ 7. Array Class Members**

Useful methods:

 Array.Sort(): Sorts the elements of an array.

 Array.Reverse(): Reverses the order of elements.

 Array.IndexOf(): Finds the index of the first occurrence of a value.

 Array.Clear(): Sets a range of elements to their default value (0 for numbers, null for reference types).

 Array.Copy(): Copies a range of elements from one array to another.

 Array.GetLength(dimension): Gets the number of elements in a specified dimension of the array.

 array.Length: Property to get the total number of elements in all dimensions for rectangular arrays, or the number of inner arrays for jagged arrays.

**✅ 8. Indices and Ranges (C# 8+)**

**🡺^ (Hat)** **operator for "from end" index:**

int[] nums = { 1, 2, 3, 4, 5 };

Console.WriteLine(nums[^1]); // Last element

var sub = nums[1..3]; // From index 1 to 2 (3 is excluded)

**🡺 .. (Range) operator**

start..end (start is inclusive, end is exclusive)

start.. : from start to end

..end : from beginning to end

.. : entire array

**✅ 9. Indexers**

**Indexers**

* **What they are:** Allow objects of your class or struct to be accessed like an array. You can use the square bracket notation ([]) to get or set values within your object.
* **Why use them?** To provide a natural, array-like interface for accessing internal data within your custom class, especially if your class represents a collection or a list of items.
* **Syntax:** Declared like a property, but with the this keyword and parameters inside square brackets.

**🧠 Summary – Session 6 Key Concepts**

| **Concept** | **Summary** |
| --- | --- |
| ref, out, in | Parameter passing by reference |
| struct | Value type, lightweight |
| enum | Constant values |
| Nullable | int?, ??, ??= |
| Arrays | Single, multi, jagged |
| Indexers | Like array syntax for objects |
| Array class | Helper functions like Sort() |

# ✅ Generics and Collections

**✅ 1. What Are Generics?**

Generics allow you to **write flexible, type-safe code**.

List<int> numbers = new List<int>();

➡️ No need to cast, no boxing/unboxing, and ensures **compile-time type safety**.

**✅ 2. Generic Classes**

A generic class is a blueprint for objects that can work with different data types. It uses a placeholder for the type, usually written as <T>.

* T stands for "Type." It's a temporary name for a data type that you will choose **later.**

class Box<T> {

public T Value;

}

**Usage:**

Box<int> intBox = new Box<int> { Value = 42 };

Box<string> strBox = new Box<string> { Value = "Hello" };

**✅ 3. Generic Methods**

**You can also make individual methods generic. This allows one method to work with different types of data.**

* **Syntax: The <T> goes *after* the method name.**

void Swap<T>(ref T a, ref T b) {

T temp = a;

a = b;

b = temp;

}

Works for any type:

Swap<int>(ref x, ref y);

Swap<string>(ref a, ref b);

**✅ 4**. Most Common Constraints (for MCQs):

* where T : class: Means T *must be a reference type*. (e.g., string, object, any custom class). This allows you to check if T is null.
* where T : struct: Means T *must be a value type*. (e.g., int, bool, double, any struct). This ensures T can never be null.
* where T : new(): Means T *must have a parameterless constructor* (a constructor that takes no inputs). This allows you to create new objects of type T using new T().
* where T : IComparable<T>: Means T *must be a type that can be compared* (like numbers or strings). This allows you to use the .CompareTo() method.

You can **restrict** what T can be using where:

| **Constraint** | **Meaning** |
| --- | --- |
| where T : class | T must be a reference type |
| where T : struct | T must be a value type |
| where T : new() | T must have parameterless constructor |
| where T : SomeBase | T must inherit SomeBase |

 "Which constraint allows you to use new T()?" -> where T : new()

 "Which constraint prevents T from being a class?" (Implied by forcing it to be a struct) -> where T : struct

**✅ 5. Collections – Generic vs Non-Generic**

We learned that when you declare a List<string>, it *only* accepts strings. You cannot add an int to it.

Since Non-Generic object can represent *any* type in C# (an int, a string, a bool, etc.), a Non-Generic collection *can* indeed store different data types all mixed within the same collection.

| **Non-Generic** | **Generic** |
| --- | --- |
| ArrayList, Hashtable | List<T>, Dictionary<K,V> |
| Not type-safe | Type-safe |
| Boxing/unboxing needed | No boxing/unboxing |

Which type of collection can store data of different data types in C# Revisit

Choose the best option O Generic Collection O **Non-Generic Collection** O Template Collection O None of the above

**✅ 6. Important Interfaces in Collections**

**using System.Collections.Generic; at the top of your file).**

**🔹 ICollection<T>**

* Basic methods: Add, Remove, Count, Clear

**🔹 IList<T>**

* Extends ICollection<T>
* A Flexible, Resizable Array
* Provides index-based access: list[0]

**🔹 IDictionary<TKey, TValue>**

* Key-Value storage
* E.g., Dictionary<string, int>

**✅ 7. Iterating Collections using foreach**

csharp

Copy code

foreach (int n in list) {

Console.WriteLine(n);

}

**✅ 8. Using Tuples to Return Multiple Values**

csharp

Copy code

(string, int) GetData() {

return ("Shivam", 100);

}

var result = GetData();

Console.WriteLine(result.Item1); // "Shivam"

➡️ Or use named tuple:

(string Name, int Score) GetInfo() => ("Shivam", 90);

var data = GetInfo();

Console.WriteLine(data.Name); // Shivam

**🔁 MCQ Matches for Session 7**

**✅ Q34: List<T>.Contains(T) Return Type**

List<string> names = new List<string> { "A", "B" };

bool exists = names.Contains("A");

✅ Return Type: \*\*bool\*\*

**✅ Q40: Collection for FIFO, Strongly Typed, Index-Based**

You have to create a collection of Order objects. The collection must meet the following:

* Strongly typed ✅
* FIFO ✅
* Index-based ✅
* Key-value storage ❌

➡️ The correct collection is:

**Queue<T>** for FIFO  
But it doesn't have index-based access.

➡️ **List<T>** satisfies:

* Strongly typed
* Index-based
* Can simulate FIFO if you only add at end and remove from start

✅ Correct answer: **List<T>** *(as per full requirement)*

**✅ Summary – Key Takeaways**

| **Topic** | **Summary** |
| --- | --- |
| Generics | Type-safe, reusable code |
| Generic Methods | Use <T> to work with any type |
| Constraints | Restrict T using where |
| Collections | Prefer Generic collections |
| IList | Index-based access |
| IDictionary | Key-value pairs |
| Tuple | Return multiple values from a method |

# Delegates, Action, Func, Predicate, Anonymous Methods & Lambdas

**🔷 What is a Delegate?**

A **delegate** is like a pointer to a function in C#, but **type-safe**.

It defines a **method signature**, and can point to any method with **matching signature**.

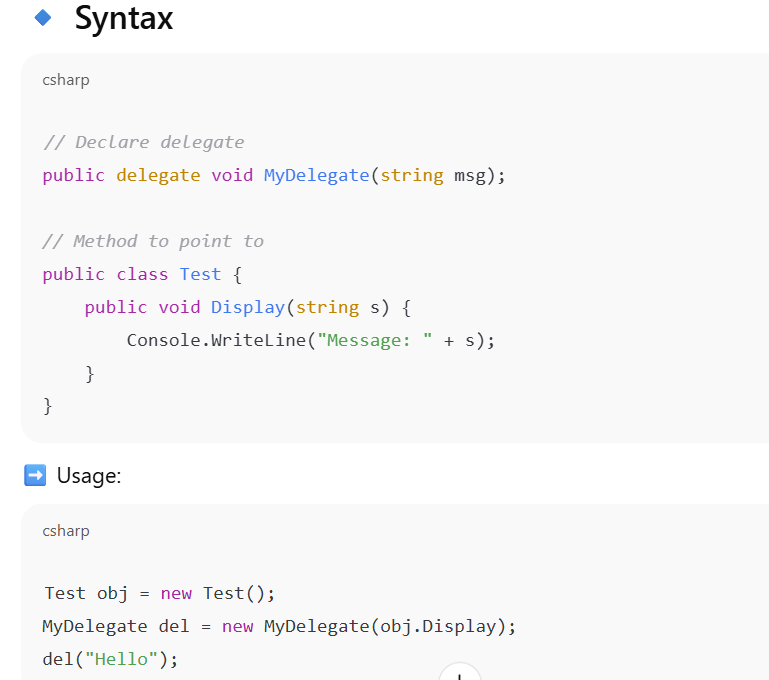
"Type-safe" means: If your delegate's "contact card" says it can call methods that take one int and return nothing (void), then you can *only* put methods with that exact signature on the card. The compiler enforces this.

**Why use it?**

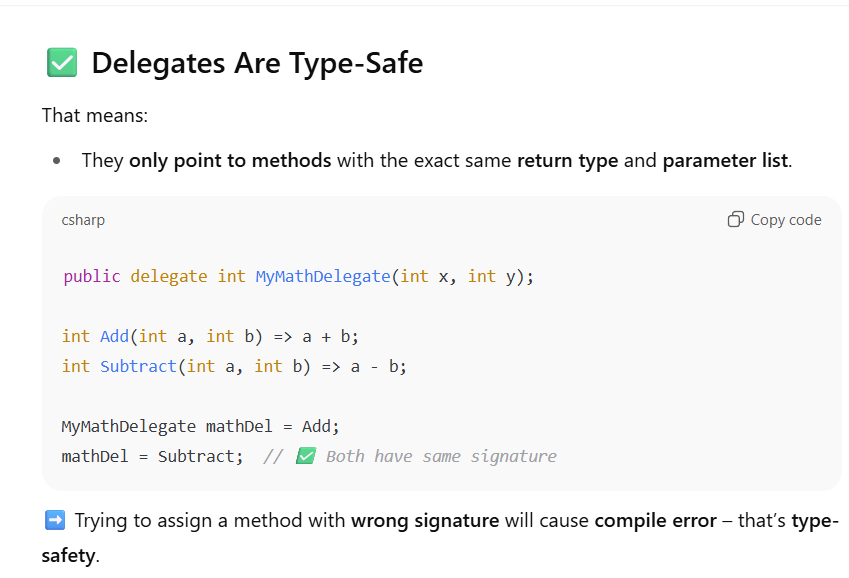
* To pass methods as arguments to other methods.
* To create "callback" mechanisms (like our download example).
* They are the foundation for Events (

**🔹 Why Use Delegates?**

* To **pass methods as arguments**
* To implement **event handlers**
* For **callback** functionality
* For **multicasting (invoking multiple methods)**
*  delegates can be used to implement callback notification ✅
*  delegate is a user-defined type ✅
*  delegates permit execution of a method in an async manner ✅







**Multicast Delegates: A "Contact List" for Methods**

Delegates can hold references to **multiple methods** simultaneously. When you invoke such a delegate, it calls all the methods it points to, one after another. This is called **multicasting**.

Think of it like sending an email to a group list: you send one email, and everyone on the list receives it.

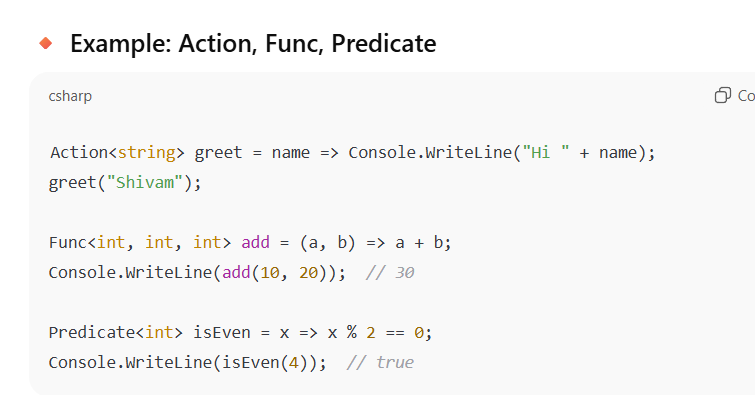
* **Operators:** You use + or += to add a method to a delegate, and - or -= to remove one.



**✅ Built-in Delegates in .NET**

To avoid writing custom delegate types every time:

| **Delegate** | **Description** |
| --- | --- |
| Action<> | No return value |
| Func<> | Returns a value |
| Predicate<> | Returns bool, used for filtering |



**🔄 Quick Recap**

| **Feature** | **Description** |
| --- | --- |
| Delegate | Type-safe function pointer |
| Use | Callbacks, Events, Multicast |
| Multicast | Can call multiple methods |
| Type Safety | Only allows exact matching signature |
| Built-in | Action, Func, Predicate |

**✅ 4. Lambda Expressions**

A **shorter way** to write anonymous methods using =>

Action greet = () => Console.WriteLine("Hi!");

Func<int, int, int> multiply = (a, b) => a \* b;

**🧠 Recap Table – Delegates & Built-in Forms**

| **Concept** | **Example** | **Return** |
| --- | --- | --- |
| Delegate | delegate void Show(string msg); | Custom |
| Multicast | del += Method2; | Calls both |
| Anonymous | delegate(string msg) { } | Inline method |
| Lambda | msg => Console.WriteLine(msg) | Short form |
| Action | Action<int> | No return |
| Func | Func<int, int, int> | Returns value |
| Predicate | Predicate<string> | Returns bool |

**. Events: The "Doorbell" for Notifications**

Events are a special kind of delegate, designed specifically for the **publisher-subscriber model**.

* **What it is:** An event is a mechanism that allows an object (**publisher**) to notify other objects (**subscribers**) that something interesting has happened.
* **Analogy:** Think of an event like a **doorbell button**.
  + The **publisher** is the house that *has* the doorbell button.
  + **Subscribers** are people who are *listening* for the doorbell sound (e.g., you in the living room, your dog, a security camera).
  + When someone presses the button (the event is **raised**), all the listeners get notified.
* **Key Idea: Publisher-Subscriber Model**
  + **Publisher (or Event Source):** The object that *declares* the event and *raises* it when something happens. It doesn't care *who* is listening or *what* they do; it just announces.
  + **Subscriber (or Event Listener/Handler):** The object that *registers* (subscribes) to an event and provides a method (an event handler) to be called when the event is raised.
* **The event keyword:**
  + Events are declared using the event keyword, which restricts how a delegate can be used.
  + **Crucially:** Only the publisher (the class that *declares* the event) can *raise* (trigger) the event. Outside the publisher class, subscribers can *only add or remove* methods from the event's list (+= or -=), but they cannot directly invoke the event themselves. This protects the event from being triggered by unauthorized code.

**Key Points about Events for MCQs:**

* They use the event keyword.
* They follow the **publisher-subscriber model**.
* The event keyword limits who can **raise** the event (only the declaring class/publisher).
* Subscribers += to register their methods (event handlers) and -= to unregister.
* The standard pattern for events uses EventHandler (or a custom delegate) and EventArgs (or a custom class derived from EventArgs to pass custom data).

**Question:** To add multiple methods to a single delegate instance, which operator is commonly used?

* O ==
* O ->
* O +=
* O .
* **Explanation:** We saw that += is the operator for adding methods to a delegate's invocation list, creating a multicast delegate. == is for comparison, -> is for pointers in C++, . is for member access.

**Question:** In the C# event model, the object that notifies other objects that something has happened is called the:

* O Listener
* O Subscriber
* O Publisher
* O Handler
* **Explanation:** The "publisher" is the one who "raises" or "notifies" others. Listeners, Subscribers, and Handlers are all terms for the objects *receiving* the notification.
* **Correct Answer:** O Publisher

**Question:** What is the primary purpose of the event keyword when declaring a delegate in C#?

* O To allow any external code to trigger the delegate directly.
* O To ensure the delegate can only hold one method reference.
* O To restrict direct invocation of the delegate from outside the declaring class.
* O To specify that the delegate must return a value.
* **Explanation:** The event keyword is a crucial safety mechanism. It restricts calling the delegate directly from outside the class where it's declared, ensuring only the "publisher" can raise its own events.
* **Correct Answer:** O To restrict direct invocation of the delegate from outside the declaring class.

**✅ Session 9 – Exception Handling & Events (with Exam-Style Insights)**

**✅ 1. Exception Handling in C# 🡺runtime error**

An exception in C# is literally an event that disrupts the normal flow of a program. When a runtime error occurs, C# "throws" an exception.

The try-catch-finally Block: Your Error Handling "Safety Net"

try {

// risky code

}

catch (Exception ex) {

Console.WriteLine(ex.Message);

}

finally {

// always runs

}

* try – write code that might throw error
* catch – handle specific or general exception
* finally – runs **regardless** of exception (for cleanup, etc.)

**🔸 Common Exceptions in C#:**

| **Exception** | **Cause** |
| --- | --- |
| DivideByZeroException | Divide by zero |
| IndexOutOfRangeException | Invalid array index |
| NullReferenceException | Accessing null object |
| FormatException | Invalid format, e.g., parsing "abc" to int |

🡺The throw Keyword: "I Can't Handle This, Someone Else Deal With It!"

**Key Points about throw for MCQs:**

* Used to explicitly generate an exception.
* Stops the current code execution and signals an error.
* The exception then "travels up" the call stack until a catch block handles it.

you don't need to memorize all exceptions, but recognizing common ones helps:

* System.DivideByZeroException: Occurs when you try to divide a number by zero.
* System.FormatException: Occurs when a value is not in the expected format (e.g., trying to convert "hello" to an int).
* System.NullReferenceException: Occurs when you try to use an object variable that is null (doesn't point to any object). This is extremely common!
* System.IndexOutOfRangeException: Occurs when you try to access an array element (or List<T> element) using an index that is outside its valid range (e.g., myArray[10] when the array only has 5 elements).
* System.ArgumentNullException: Occurs when a method receives a null argument that it doesn't expect to be null.
* System.ArgumentOutOfRangeException: Occurs when a method receives an argument whose value is outside the expected range (like our negative age example).
* System.IO.FileNotFoundException: Occurs when a program tries to open a file that doesn't exist.

**✅ 2. checked and unchecked**

Used for **overflow checking** in numeric calculations.

int x = int.MaxValue;

int y;

checked {

y = x + 1; // throws OverflowException

}

unchecked {

y = x + 1; // wraps around without error

}

**✅ 3. Best Practices (Dos & Don’ts)**

✅ **Do**:

* Catch specific exceptions
* Use finally for cleanup
* Log errors

❌ **Don't**:

* Swallow exceptions (catch {} without handling)
* Use exceptions for flow control

**✅ 4. Custom Exception Class**

You can create your own error types by extending **System.Exception.**

class MyCustomException : Exception {

public MyCustomException(string msg) : base(msg) {}

}

Usage:

throw new MyCustomException("Something went wrong");

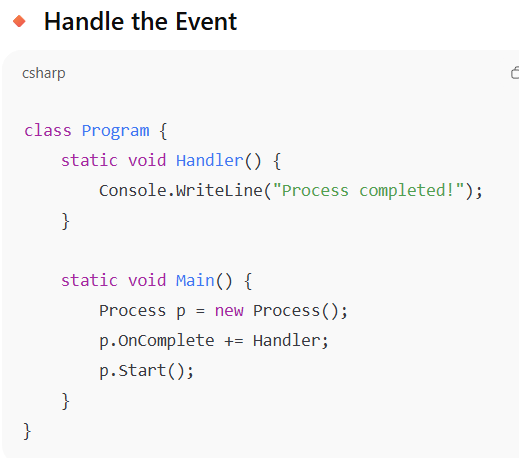
**✅ 5. Events – Declaring, Raising, Handling**

Events allow communication between objects using the **publisher-subscriber** model.

**🔸 Declare & Raise Event**



**🔸 Handle the Event**



**🔁 MCQ Match Recap**

| **MCQ** | **Concept** |
| --- | --- |
| Q4 | DivideByZeroException |
| Q17 | Delegate/Event Callback |
| Q29 | Raise event via public method |
| Q9 | SqlConnection (not related here, skip) |

**🧠 Quick Summary Table**

| **Feature** | **Purpose** |
| --- | --- |
| try-catch-finally | Structured error handling |
| checked / unchecked | Numeric overflow behavior |
| Custom exception | Define user-defined errors |
| event keyword | Trigger custom notifications |
| ?.Invoke() | Safe event raising |
| Delegate | Required to define event signature |

# ✅ Session 10 – LINQ, Extension Methods, Anonymous Types, Partial Classes

**🔷 Topics Covered:**

1. Anonymous Types
2. Extension Methods
3. Partial Classes & Partial Methods
4. LINQ to Objects
5. LINQ Queries (Syntax & Methods)
6. Deferred Execution
7. PLINQ
8. MCQ Mapping

**✅ 1. Anonymous Types**

A quick way to create a **read-only object** with properties but **no class definition**.

var student = new { Name = "Shivam", Marks = 90 };

Console.WriteLine(student.Name); // Output: Shivam

* Auto-generated class behind the scenes
* Properties are **read-only**
* student.Marks = 95; ❌ → Not allowed

**✅ 2. Extension Methods**

Add new methods to existing classes **without modifying the original class**.

**Syntax:**

public static class StringExtensions {

public static int WordCount(this string str) {

return str.Split(' ').Length;

}

}

Usage:

string msg = "Hello from Shivam";

Console.WriteLine(msg.WordCount()); // Output: 3

➡️ this string str — tells compiler it’s an extension of string.

**🔁 MCQ Match: Q13 – Not Supported for Dynamic Types**

❌ **Extension methods are NOT supported** for dynamic types.

✅ Correct answer: **Extension**

**✅ 3. Partial Classes & Partial Methods**

**Partial Class**

Split a class into **multiple files**.

// File1.cs

public partial class Student {

public int Id;

}

// File2.cs

public partial class Student {

public string Name;

}

**Partial Method**

Used in partial classes for **optional method implementation**.

partial class Student {

partial void OnEnroll(); // Declared

}

partial class Student {

partial void OnEnroll() {

Console.WriteLine("Enrolled");

}

}

**✅ 4. LINQ (Language Integrated Query)**

Allows **SQL-like querying** on collections.

**Example:**

int[] nums = { 1, 2, 3, 4, 5 };

var even = from n in nums

where n % 2 == 0

select n;

foreach (int i in even)

Console.WriteLine(i);

**LINQ Method Syntax**

var result = nums.Where(n => n % 2 == 0).ToList();

**✅ 5. Deferred Execution**

LINQ queries are **not executed immediately** — they execute **only when iterated**.

var query = from n in nums where n > 2 select n;

// Changes to nums here will affect the query result

foreach (var n in query)

Console.WriteLine(n); // Only now it's executed

**✅ 6. PLINQ (Parallel LINQ)**

Enables **parallel execution** of LINQ queries to improve performance.

var parallelQuery = nums.AsParallel().Where(n => n > 2);

➡️ Great for **large datasets** and **multi-core performance**.

**🔁 MCQ Match: Q30 – LINQ Character Sorting**

This question was based on:

var chrs = from str in strs

let chrArray = str.ToCharArray()

from ch in chrArray

orderby ch

select ch;

➡️ Used:

* let
* orderby
* LINQ query on string[]
* ToCharArray()

✅ Concept tested: **LINQ + character selection + sorting**

**🧠 Summary Table**

| **Concept** | **Example** | **Purpose** |
| --- | --- | --- |
| Anonymous type | var p = new { Name = "A" } | Temp object |
| Extension method | this string | Add method |
| Partial class | partial class A | Split file |
| LINQ query | from x in list | Declarative |
| Method syntax | .Where(x => ...) | Functional |
| Deferred execution | query executed on use | Lazy eval |
| PLINQ | AsParallel() | Speed up |

# ✅ Session 11 – Reflection, Custom Attributes, File I/O in .NET Core

**✅ 1. Shared Assemblies (Concept Recap)**

An **assembly** is a compiled .DLL or .EXE. A **shared assembly** is:

* Placed in **Global Assembly Cache (GAC)**
* Identified by **strong name** (includes name, version, public key, etc.)

Used when **multiple apps** use the same library across the system.

✅ No MCQ directly, but understand terms like:

* Assembly
* Metadata
* GAC
* Strong Naming

**✅ 2. Reflection in C#**

**Reflection** lets you inspect **metadata** (like types, methods, properties) of assemblies at **runtime**.

**Basic Example:**

using System;

using System.Reflection;

Type type = typeof(string);

Console.WriteLine(type.FullName); // Output: System.String

MethodInfo[] methods = type.GetMethods();

foreach (var m in methods)

Console.WriteLine(m.Name);

**🔁 MCQ Match: Q12**

object Invoke(object obj, object[] parameters)

✅ This is about **Reflection** — using **MethodInfo.Invoke(...)** to call methods dynamically.

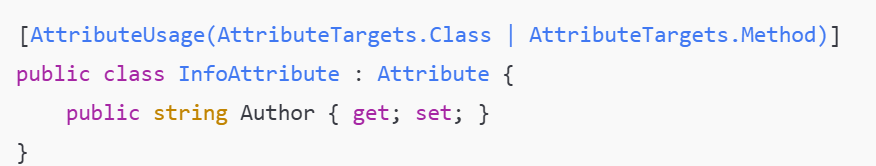
Correct MCQ Answer:  
✅ **All of the mentioned** — because:

* It **invokes a method**
* **Parameters are passed as object[]**
* **Return value is captured**

**✅ 3. Custom Attributes**

Add **custom metadata** to classes/methods/properties using **attributes**.

**Step 1: Define Custom Attribute**



**Step 2: Use the Attribute**

[Info(Author = "Shivam")]

public class MyClass {}

**Step 3: Read with Reflection**

Type t = typeof(MyClass);

var attrs = t.GetCustomAttributes(false);

foreach (var attr in attrs)

Console.WriteLine(attr.GetType().Name); // InfoAttribute

✅ **Important in questions** like:  
“What metadata is retrieved at runtime?” → This uses Reflection.

**✅ 4. File I/O – System.IO Namespace**

C# provides powerful file APIs:

**🔸 Working with Files and Directories**



**🔸 Using FileStream and StreamReader/Writer**



**🔁 Possible Exam Concepts from File I/O:**

| **Concept** | **API Used** |
| --- | --- |
| Check file exists | File.Exists() |
| Create file | File.Create() / File.WriteAllText() |
| Read file | File.ReadAllText() / StreamReader |
| Write file | File.WriteAllText() / StreamWriter |
| List files | Directory.GetFiles() |
| Delete file | File.Delete() |

**🧠 Summary Table**

| **Feature** | **API / Class** | **Notes** |
| --- | --- | --- |
| Reflection | Type, MethodInfo, PropertyInfo | Inspect metadata |
| Invoke method | MethodInfo.Invoke() | Call at runtime |
| Custom Attribute | [MyAttribute(...)] | Extend metadata |
| File write/read | StreamWriter / StreamReader | Safer than File.\* for big data |
| File short ops | File.WriteAllText(), ReadAllText() | For small files |
| Directory ops | Directory.CreateDirectory(), GetFiles() | Folder operations |

# Z✅ Session 12 – Threading, Tasks, async/await, and Parallel Programming

**✅ 1. Threading Basics**

A **Thread** is a lightweight process that runs code concurrently.

**Create a new thread:**

**With parameters:**



**✅ 2. Thread States (🧠 MCQ #7)**

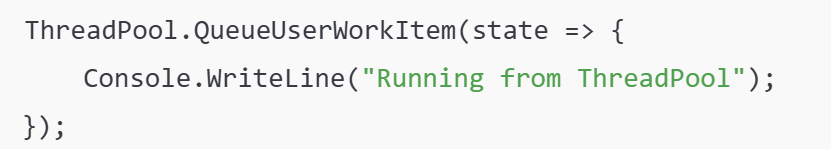
Valid thread states:

✅ Unstarted, Running, NotRunnable, Dead (Terminated), Runnable (Ready to run)

✅ **Answer:** 1, 2, 3, 4, 5

**✅ 3. ThreadPool (Optimized)**

**ThreadPool manages a pool of worker threads that can be reused.**

****

* Efficient for short, repeated background tasks
* Avoids creating new threads manually

**✅ 4. Synchronizing Data (Avoid Race Conditions)**

**🔸 lock Keyword**

object locker = new object();

lock (locker) {

// critical section

}

➡️ Ensures only one thread enters the block at a time.

**🔸 Monitor & Interlocked**

Monitor.Enter(locker);

// critical code

Monitor.Exit(locker);

Interlocked.Increment(ref count); // Safe increment

**✅ 5. Tasks and TPL (Task Parallel Library)**

Task t = Task.Run(() => {

Console.WriteLine("Task running");

});

t.Wait(); // Blocks until task finishes

➡️ Preferred over Thread for modern async work.

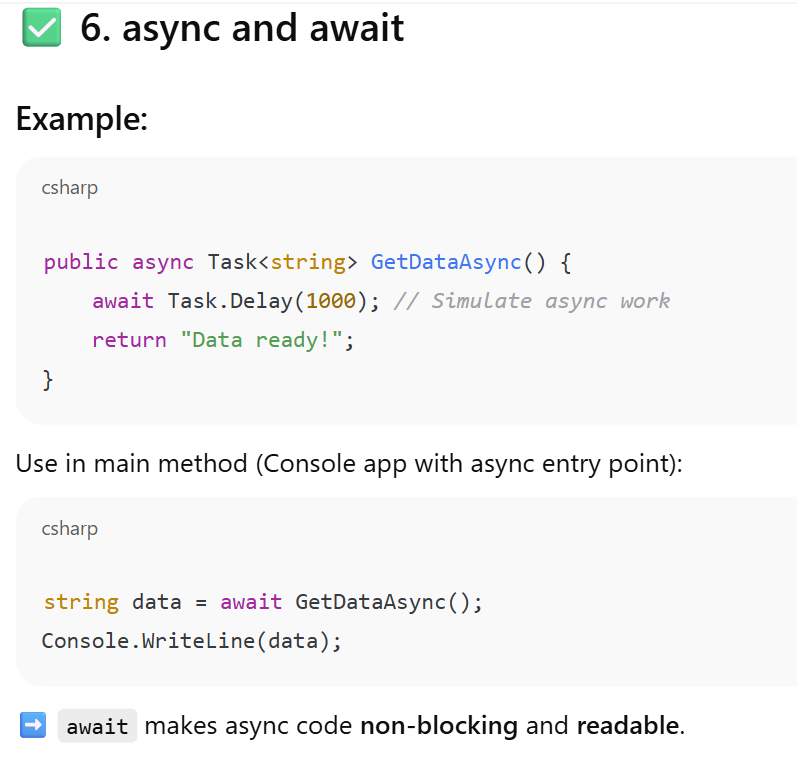
**With return value:**

csharp

CopyEdit

Task<int> t = Task.Run(() => 5 + 5);

int result = t.Result; // 10



**🔁 MCQ Match Reference:**

| **MCQ** | **Concept** | **Answer** |
| --- | --- | --- |
| Q7 | Thread states | 1,2,3,4,5 ✅ |
| Q18 | dynamic keyword | Introduced in C# 4.0 ✅ |
| Q20 | Controller from RouteData | ✅ RouteData.Values["Controller"].ToString() |
| Q29 | Raise event from outside | ✅ Use public method to raise event |

**🧠 Summary Table**

| **Feature** | **API** | **Notes** |
| --- | --- | --- |
| Thread creation | Thread, ThreadStart | Manual threading |
| Thread reuse | ThreadPool.QueueUserWorkItem | Efficient |
| Synchronization | lock, Monitor, Interlocked | Prevent data race |
| Parallelism | Task, Task.Run() | Modern way |
| Async | async, await | Background without blocking |

**✅ Quick Recap on MCQ Styles (based on what we just covered):**

🧠 Example:

**Q:** Which one ensures that only one thread can enter a block of code?  
✅ **Option:** lock

**Q:** Which of the following supports background threading efficiently in .NET?  
✅ **Option:** ThreadPool

**Q:** What is the return type of an async method returning a value?  
✅ **Option:** Task<T>

dynamic is a **runtime-typed** variable. The type is resolved **at runtime**, not at compile-time.



**🧠 How is it different from var?**

| **Feature** | **var** | **dynamic** |
| --- | --- | --- |
| Type resolution | **Compile-time** | **Runtime** |
| Can change type later | ❌ No | ✅ Yes |
| IntelliSense | ✅ Available | ❌ Not available (until runtime) |
| Performance | ⚡ Faster | 🐢 Slower (due to runtime lookup) |

The keyword is new to C# 4.0 and is used to tell the compiler that a variable’s type can change or that it is not known until runtime.

**Choose the best option:**

* Covariance
* **Dynamic** ✅
* Contravariance
* Object

✅ Correct: **dynamic**