

## 1 What is OOP? Why Use It?

### Definition:

OOP (**Object-Oriented Programming**) is a programming paradigm that organizes code into **objects**, which are instances of **classes**. It improves **code reusability, maintainability, and scalability**.

### Why Use OOP in .NET Core?

- ✓ **Code Reusability** – Use the same class in different parts of the application.
- ✓ **Modularity** – Divide an application into multiple classes/modules.
- ✓ **Scalability** – Easily extend applications without modifying existing code.
- ✓ **Security** – Encapsulation hides sensitive data from outside access.
- ✓ **Maintainability** – OOP promotes a cleaner, structured way to write code.

## 2 Core Principles of OOP (Pillars of OOP in .NET Core)

### 1. Encapsulation (Data Hiding with Properties)

Encapsulation **restricts direct access** to an object's data and allows controlled access using **getters & setters** (Properties).

#### public

- Accessible **anywhere** in the application.
- Can be accessed from **any class, assembly, or project**.

```
public class MyClass { public int Value; }
```

#### private

- Accessible **only within the same class**.
- Not visible to derived classes or other classes in the same assembly.

```
class MyClass { private int Value; }
```

#### protected

- Accessible **within the same class and derived (child) classes**.
- Not accessible outside the inheritance hierarchy.

```
class Parent { protected int Value; }
```

```
class Child : Parent { void Method() { Value = 10; } }
```

## internal

- Accessible **within the same assembly** (project).
- Not accessible from another project unless marked as **friend** in special cases.  
`internal class MyClass { int Value; }`

## Encapsulation in C# – Key Notes

1. Definition
  - Encapsulation is the process of wrapping data (variables) and methods together into a single unit (class).
  - It restricts direct access to certain details of an object and protects data from unintended modifications.
2. How to Achieve Encapsulation?
  - By using access modifiers (**private**, **public**, **protected**, **internal**).
  - By providing getter and setter methods (**properties** in C#).

### Benefits of Encapsulation

- ✓ **Data Protection** – Prevents direct modification of fields.
- ✓ **Code Maintainability** – Changes in implementation do not affect external code.
- ✓ **Flexibility** – Getters/setters allow validation and additional logic.
- ✓ **Better Security** – Restricts unauthorized access to sensitive data.

### Encapsulation vs Data Hiding

- **Encapsulation**: Bundling of data & methods, allowing controlled access.
- **Data Hiding**: Restricting access to internal data (achieved through private fields).

## 2. Abstraction (Hiding Implementation Details)

Abstraction **hides complex implementation details** and exposes only necessary functionalities.

### Abstraction in C# – Interview Notes

#### 1. Definition

- Abstraction is one of the **four pillars of Object-Oriented Programming (OOP)**.
- It **hides the implementation details** and only shows **essential features** to the user.
- Helps in reducing complexity and increasing code reusability.

#### 2. How to Achieve Abstraction?

- Using Abstract Classes (**abstract** keyword)
- Using Interfaces (**interface** keyword)

### 3. Abstraction Using Abstract Class

- An **abstract class** cannot be instantiated.
- It can have **abstract methods** (without implementation) and **concrete methods** (with implementation).

### 4. Abstraction Using Interface

- An **interface** contains only method declarations (no implementations).
- A class that implements an interface **must provide implementations for all its methods**.

## Key Differences: Abstract Class vs Interface

#### 1. Methods

- **Abstract Class:** Can have both **abstract (without implementation)** and **concrete (with implementation)** methods.
- **Interface:** Can only have **abstract methods** (C# 8+ allows default method implementations).

#### 2. Fields

- **Abstract Class:** Can have **fields (variables)**.
- **Interface:** Cannot have fields (only method declarations).

#### 3. Access Modifiers

- **Abstract Class:** Supports **access modifiers** (public, private, protected, etc.).
- **Interface:** Does **not** support access modifiers (all methods are **public** by default).

#### 4. Multiple Inheritance

- **Abstract Class:** Does **not** support multiple inheritance.
- **Interface:** A class can implement **multiple interfaces**.

#### 5. Use Case

- **Abstract Class:** Used when a class has **common functionality** that should be shared by subclasses.
- **Interface:** Used when defining **only method contracts** (ensuring implementation by multiple classes).

## 1. What is an Interface?

An **interface** in C# is a **contract** that defines a **set of methods, properties, events, or indexers** that a class must implement. It **only contains declarations**—no implementation.

### Key Features:

- ✓ **No Implementation** – Only method signatures (implementation is provided by the class that implements it).
- ✓ **Multiple Inheritance** – A class can implement multiple interfaces (unlike class inheritance).
- ✓ **Loose Coupling** – Helps in achieving better abstraction and dependency injection.
- ✓ **Default Members (C# 8.0+)** – Allows default method implementations in interfaces.

## 8. When to Use Interfaces?

Use **interfaces** when:

- ✓ You need **multiple inheritance** (C# does not support multiple class inheritance).
- ✓ You want to **define a contract** that multiple classes must follow.
- ✓ You are working with **loosely coupled** code (e.g., Dependency Injection).
- ✓ You need **better testability** (mocking dependencies in unit tests).

## 3. Inheritance (Code Reusability)

Inheritance allows a **child class to inherit the properties and methods** of a **parent class**, reducing code duplication.

- ◆ **Example: Inheritance in .NET Core (Base & Derived Classes)**

### Inheritance in C# – Key Notes

1. Definition
  - Inheritance is an OOP concept where a child class (derived class) inherits properties and methods from a parent class (base class).
  - It promotes code reusability by allowing shared functionality among multiple classes.
2. How to Achieve Inheritance?
  - Use the **: symbol to inherit from a base class.**
  - The derived class gets access to public and protected members of the base class.

### Types of Inheritance in C#

- Single Inheritance – One child class inherits from one parent class.
- Multilevel Inheritance – A class inherits from another derived class.
- Hierarchical Inheritance – Multiple child classes inherit from one parent class.
- Multiple Inheritance (via Interfaces) – A class implements multiple interfaces (since C# does not support multiple class inheritance).

### Advantages of Inheritance

- ✓ Code Reusability – No need to rewrite the same code multiple times.
- ✓ Extensibility – Allows modification and extension of existing classes.
- ✓ Scalability – Helps in designing large applications efficiently.
- ✓ Reduces Redundancy – Avoids duplication of code.

## 4. Polymorphism (Overloading & Overriding)

Polymorphism allows **multiple methods with the same name** but different behaviors.

### 1 Method Overloading (Compile-time Polymorphism)

- Same method name, different parameters

### 2 Method Overriding (Runtime Polymorphism)

- Same method name, different behavior in the derived class

## Difference Between Virtual Method and Child Method

When dealing with inheritance and polymorphism, understanding virtual methods and child methods is important.

### 1 Virtual Method

#### ◆ Definition:

A virtual method is a method in the base (parent) class that can be overridden by a child (derived) class. It allows runtime polymorphism by ensuring that the correct method is called based on the object type at runtime.

#### ◆ Key Characteristics:

- Defined in the parent class using the **virtual** keyword (in languages like C++) or implicitly virtual in Java.
- Can be overridden by a derived class.
- Supports dynamic binding (runtime polymorphism).

### 2 Child Method

#### ◆ Definition:

A child method is a method that is defined in a derived (child) class, whether it overrides a parent method or not.

#### ◆ Key Characteristics:

- Exists only in the child class.
- Can override a virtual method from the parent class (method overriding).
- Can be completely new and unrelated to any parent method.

- If it overrides a non-virtual method, static binding occurs (parent class method gets called when using parent reference).

## How to Decide Whether It Is Runtime Polymorphism or Compile-Time Polymorphism?

To determine whether a given polymorphism is compile-time or runtime, ask yourself these two questions:

1] Is the method/function call decided at compile-time or runtime?

- If it's determined at compile-time, it's Compile-Time Polymorphism.
- If it's determined at runtime (dynamically), it's Runtime Polymorphism.

2] Is it method overloading or method overriding?

- Method Overloading → Compile-Time Polymorphism
- Method Overriding → Runtime Polymorphism

### ♦ Identifying Compile-Time Polymorphism

✓ It is Compile-Time Polymorphism when:

- The method call is resolved at compile-time.
- It uses Method Overloading (same method name, different parameters).
- It uses Operator Overloading (in languages like C++ where operators can be overloaded).
- The method signature (parameters) determines which method gets called.

### ♦ Identifying Runtime Polymorphism

✓ It is Runtime Polymorphism when:

- The **method call is resolved at runtime**.
- It uses **Method Overriding** (child class provides a new definition for a parent class method).
- It involves **dynamic method dispatch** (calling an overridden method using a parent reference to a child object).
- It allows **flexibility** where the behavior can change at runtime.

🔍 Why is it Runtime Polymorphism?

- The method call (`show()`) is resolved at runtime, not compile-time.
- Upcasting (`Parent obj = new Child();`) enables dynamic method dispatch.
- The child class overrides the parent method, and the correct method is called based on the actual object type.

If the method is overloaded (same name, different parameters) → Compile-Time Polymorphism

✓ If the method is overridden (same name, same parameters, but different behavior in child class)  
→ Runtime Polymorphism

## Class and Objects from Programming Language Point of View.

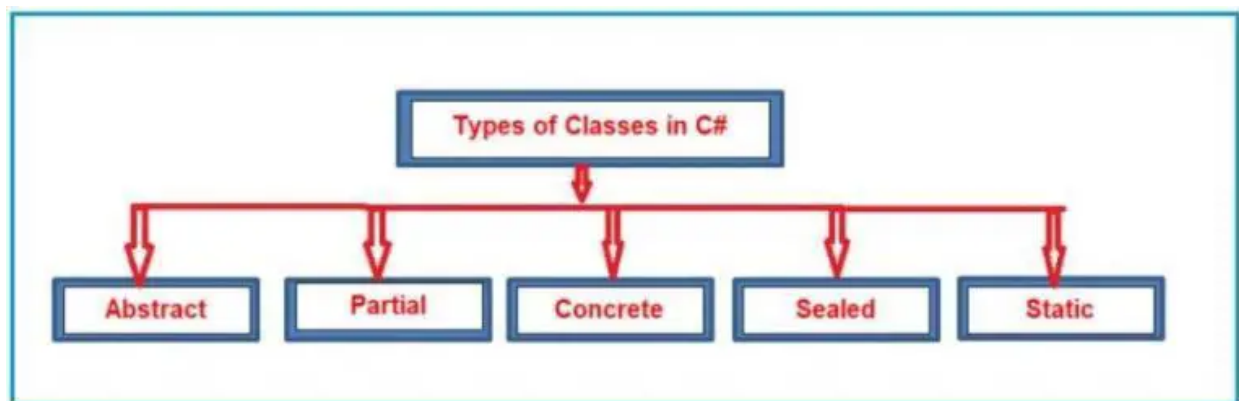
### Class:

A class is simply a user-defined data type that represents both state and behavior. The state represents the properties and **behavior** is the action that objects can perform. In other words, we can say that a class is the blueprint/plan/template that describes the details of an object. A class is a blueprint from which the individual objects are created. In C#, a Class is composed of three things i.e. a name, attributes, and operations.

### Objects:

It is an instance of a class. A class is brought live by creating objects. An object can be considered as a thing that can perform activities. The set of activities that the object performs defines the object's behavior. All the members of a class can be accessed through the object. To access the class members, we need to use the dot (.) operator. The dot operator links the name of an object with the name of a member of a class.

**Types of Classes in C#:** Please have a look at the following image.



### Abstract Class

1. A method that does not have a body is called an abstract method, and the class that is declared using the keyword `abstract` is called an abstract class. If a class contains an abstract method, it must be declared `abstract`.
2. An abstract class can contain both abstract and non-abstract methods. If a child class of an abstract class wants to consume any non-abstract methods of its parent, it should implement all abstract methods.
3. An abstract class is never usable in itself because we cannot create the object of an abstract class. The members of an abstract class can be consumed only by the child class of the abstract class.
4. You can't directly create the object of the abstract class first; you inherit the method in the child class then create the object of the child in the main method and call the abstract method.
5. The abstract class contains both the abstract method as well as non-abstract method; it's not necessary to have only abstract method in class.

## ❏ Can an abstract class have a constructor?

 **Answer:**

Yes, an abstract class **can have a constructor**. Although we cannot instantiate an abstract class directly, **its constructor is called when a subclass object is created** to initialize common properties.

## ❏ Can an abstract class have concrete (non-abstract) methods?

 **Answer:**

Yes, an abstract class **can have both abstract and concrete methods**. Concrete methods provide default implementations that child classes can inherit or override.


## ❏ Can an abstract class implement an interface?

 **Answer:**

Yes, an abstract class **can implement an interface** and **provide partial or full implementation** of its methods.

## ❏ Can an abstract class be final?

 **Answer:**

 **No, an abstract class cannot be `final`** because the purpose of an abstract class is to be extended by subclasses. If a class is marked as `final`, it **cannot be inherited**, making it useless as an abstract class.

## ❏ Why use an abstract class instead of an interface?



### Answer:

Use an **abstract class** when:

- ✓ You need to provide **default behavior** that child classes can inherit.
- ✓ You expect **some methods to have implementations** and some to be abstract.
- ✓ You want to use **access modifiers** (private, protected, etc.).

Use an **interface** when:

- ✓ You want to **enforce a contract** (all methods must be implemented).
- ✓ You need **multiple inheritance** (Java supports multiple interfaces).

## 8 What happens if an abstract class does not have an abstract method?

### Answer:

An **abstract class is not required to have abstract methods**. It can still be declared abstract to prevent instantiation.

## 9 Can an abstract class have **main()** method?

### Answer:

Yes, an abstract class **can have a **main()** method** and be executed like a normal Java class.

## 10 What if a subclass does not override all abstract methods?

### Answer:

If a child class **does not override all abstract methods**, it **must also be declared abstract**; otherwise, a compilation error will occur.

## Sealed Class in C#

A class from which it is not possible to create/derive a new class is known as a sealed class. In simple words, we can say that when we define the class using the sealed modifier, then it is known as a **sealed class and a sealed class cannot be inherited by any other classes**.

To make any class a sealed class we need to use the keyword sealed. The keyword sealed tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class. For a better understanding, please have a look at the below code.

### Sealed Class in C#:

1. A class from which it is not possible to derive a new class is known as a sealed class.
2. The sealed class can contain non-abstract methods; it cannot contain abstract and virtual methods.
3. It is not possible to create a new class from a sealed class.
4. We should create an object for a sealed class to consume its members.
5. We need to use the keyword sealed to make any class sealed.
6. The sealed class should be the bottom-most class within the inheritance hierarchy.

### ❏ What is a sealed class? Why do we use it?

#### 📌 Answer:

A **sealed class** is a class that **restricts which other classes can inherit from it**. Unlike abstract classes or interfaces, where any class can extend them, a **sealed class allows only specific subclasses**.

#### ♦ Why use it?

- **Controlled Inheritance** → Prevents unwanted subclassing.
- **Better Code Safety** → Compiler knows all possible subclasses.
- **Useful in Pattern Matching** → Especially in Kotlin & C# for **when-expressions** or **switch-case expressions**.

### ❏ Can a sealed class have abstract methods?

#### 📌 Answer:

Yes, a **sealed class can have abstract methods**. The subclasses **must override them**.

- ♦ A sealed class cannot be inherited, but it **can still interact with other classes** via **interfaces**, **method overriding**, and **return types**.
- ♦ **Sealing methods** prevents **further overriding** in derived classes.

## Generics in C# – Key Notes

### 1. Definition

- Generics allow you to **define classes, methods, interfaces, and delegates** with a **placeholder for data types**.
- They provide **type safety** and **code reusability** without compromising performance.

### 2. Why Use Generics?

- ✓ **Type Safety** – Prevents runtime type errors by ensuring type correctness at compile-time.
- ✓ **Code Reusability** – One generic class/method works with different data types.
- ✓ **Performance** – Avoids unnecessary boxing/unboxing, improving performance.

```
class Box<T> // T is a type parameter

{

    private T value;

    public void SetValue(T val) { value = val; }

    public T GetValue() { return value; }

}

class Program

{

    static void Main() {

        Box<int> intBox = new Box<int>();

        intBox.SetValue(100);

        Console.WriteLine(intBox.GetValue()); // Output: 100

        Box<string> strBox = new Box<string>();

        strBox.SetValue("Hello");

        Console.WriteLine(strBox.GetValue()); // Output: Hello

    }

}
```

## Partial Class in C# – Interview & Development Perspective

### ♦ What is a Partial Class?

A **partial class** in C# allows a **single class definition** to be **split across multiple files**. At compile time, C# combines all parts into a single class.

- ♦ Defined using the **partial** keyword
- ♦ All parts must be in the same namespace and assembly
- ♦ Used mainly for better code organization and maintainability

### ♦ Interview Perspective



#### ✓ Common Interview Questions on Partial Class

1. **What is a partial class in C#?**
  - A **partial class** is a class that is split across multiple files using the **partial** keyword.
2. **Why do we use partial classes in C#?**
  - To **divide large classes** into smaller, manageable parts.
  - To **separate auto-generated code** from custom code (e.g., in **WinForms**, **ASP.NET**).
3. **What are the rules for using partial classes?**
  - Must use the **partial** keyword.
  - All parts must be in the **same namespace**.
  - The **same class name** must be used in each part.
4. **Can a partial class have different access modifiers in different files?**
  - **✗ No**, all parts must have the **same access modifier**.
5. **Can a partial class have different base classes in different files?**
  - **✗ No**, a class can only inherit from one base class.
  - However, **different parts** can implement **different interfaces**.

## Development Perspective

#### ✓ Why Use Partial Classes in Real-World Projects?

1. 💡 **Code Maintainability:**
  - Helps in **splitting large classes** into smaller, readable files.
2. ⚡ **Auto-Generated Code Separation:**

- Used in **WinForms, ASP.NET, Entity Framework** for separating auto-generated and custom code.
- 3.  **Team Collaboration:**
  - Allows multiple developers to **work on different parts** of the same class.
- 4.  **Better Code Organization:**
  - Related functionalities can be grouped in **different files**.

## Concrete Class in C#

A **concrete class** in C# is a class that has a **complete implementation** and **can be instantiated directly**. Unlike **abstract classes** or **interfaces**, which only define structure, a concrete class provides full functionality.

### ◆ Key Characteristics of a Concrete Class

- ✓ **Can be instantiated** directly using the **new** keyword.
- ✓ **Implements all methods** if inheriting from an abstract class or interface.
- ✓ **Has complete functionality** with properties, methods, and constructors.
- ✓ **Can inherit from another class** but is not mandatory.

## Conclusion

- **Every concrete class is a normal class**, but **not every normal class is concrete**.
- If a class has **abstract methods**, it is **not concrete**.
- Use **concrete classes** when you need **fully implemented and usable objects**.

### ◆ Understanding Constructors & Destructors in C#

A **constructor** is a special method in a class that **initializes objects** when they are created. A **destructor** is used to **clean up resources** before the object is destroyed.

### ◆ Types of Constructors in C#

#### ① Default Constructor (Parameterless Constructor) // not need any parameter

- A **constructor with no parameters**.
- Automatically provided by C# if no constructor is defined.
- Initializes object with **default values**.

#### ② Parameterized Constructor //needs the parameter

- A constructor that **accepts parameters** to initialize an object with specific values.

### ③ Copy Constructor accepts the another constructor as org

- A constructor that **creates a new object by copying an existing object**.
- Useful when **duplicating objects**.

### ④ Static Constructor runs only once when the class called

- Called **only once** when the class is first loaded.
- Used to **initialize static data members**.
- Does **not take parameters** and has **no access modifiers**.

## This and Base Keyword in C#

this keyword is used to refer to the current instance of the class. It is used to access members from the constructors, instance methods, and instance accessors. this keyword is also used to track the instance which is invoked to perform some calculation or further processing related to that instance. Following are the different ways to use 'this' keyword in C#

base keyword serves as a vital tool for working with inheritance, allowing derived classes to interact with and extend the functionality of their base classes in various ways. In this section, we'll take a closer look at the syntax, usage, and significance of the base keyword in C#.

One of the primary uses of the base keyword is to access members(methods, properties, fields) defined in the Base class from within the derived class. This allows derived classes to reuse and build upon the functionality provided by the base class.

## ◆ Destructor and Garbage Collection

### ① Destructor

- Used to **clean up resources** before the object is destroyed.
- Defined using `~ClassName()`.
- Called **automatically** by the **Garbage Collector**.

### ② Garbage Collection

- **Automatic memory management** in C#.

- Uses `GC.Collect()` to force garbage collection.

## **C# | Namespaces**

Namespaces are used to organize the classes. It helps to control the scope of methods and classes in larger .Net programming projects. In simpler words you can say that it provides a way to keep one set of names (like class names) different from other sets of names. The biggest advantage of using namespace is that the class names which are declared in one namespace will not clash with the same class names declared in another namespace. It is also referred as named group of classes having common features. The members of a namespace can be namespaces, interfaces, structures, and delegates.

## **C# Call By Value**

In C#, value-type parameters are that pass a copy of original value to the function rather than reference. It does not modify the original value. A change made in passed value does not alter the actual value. In the following example, we have pass value during function call.

## **C# Call By Reference**

C# provides a `ref` keyword to pass argument as reference-type. It passes reference of arguments to the function rather than copy of original value. The changes in passed values are permanent and modify the original variable value.

## **C# Structs**

In C#, classes and structs are blueprints that are used to create instance of a class. Structs are used for lightweight objects such as `Color`, `Rectangle`, `Point` etc. Unlike class, structs in C# are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

## **C# Exception Handling**

Exception Handling in C# is a process to handle runtime errors. We perform exception handling so that normal flow of the application can be maintained even after runtime errors.

In C#, exception is an event or object which is thrown at runtime. All exceptions are derived from `System.Exception` class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

## **C# Checked and Unchecked**

C# provides `checked` and `unchecked` keyword to handle integral type exceptions. `checked` and `unchecked` keywords specify checked context and unchecked context respectively. In checked context, arithmetic overflow raises an exception whereas, in an unchecked context, arithmetic overflow is ignored and result is truncated.

## 2. What is the Difference Between **throw** and **throw ex**?

Both are used to rethrow an exception, but **throw** maintains the original stack trace, whereas **throw ex** resets it.

## 6. What Are Exception Filters (C# 6.0 Feature)?

Exception filters allow you to specify conditions in **catch** blocks.

## C# Serialization

In C#, serialization is the process of converting object into byte stream so that it can be saved to memory, file or database. The reverse process of serialization is called deserialization.

## C# Delegates

In C#, delegate is a reference to the method. It works like function pointer in C and C++. But it is objected-oriented, secured and type-safe than function pointer. For static method, delegate encapsulates method only. But for instance method, it encapsulates method and instance both. The best use of delegate is to use as event.

**Internally a delegate declaration defines a class which is the derived class of System.Delegate.**

**Delegates is the typed safed function pointer it holds the reference of the current method and call the method. Its like pointer in c and c++;**

**Delegates help to call the method**

- Create the delegate using the delegate key word :- `public delegate void addDelegarte(int a, int b)`
  - Parameter and return type of the method should be same as the calling method
  - Signature should be match means return type and parameters type and number of parameter
  - Define the delegate unde the namespace or above the class not in the class
  - Create the instance of the delegate pass the method into the delegate as the parameter
- ✓ Delegates **increase flexibility**, **reduce code duplication**, and **support event-driven programming**.
- ✓ They are **type-safe**, making them **safer than traditional function pointers** in C++.
- ✓ **Best Practice:** Use **Func<>** and **Action<>** for built-in delegate functionality.

## Multithreading in C#

Multithreading is a technique in C# where multiple threads execute concurrently to improve application performance and responsiveness. Each thread runs independently, enabling tasks to run in parallel.



# Why Use Multithreading?

- ✓ **Improves Performance** – Multiple tasks execute simultaneously, utilizing CPU cores efficiently.
- ✓ **Enhances Responsiveness** – UI applications remain responsive while performing background tasks.
- ✓ **Parallel Processing** – Best suited for CPU-intensive tasks like computations, data processing, and rendering.

✓ Use **Task** for most cases unless low-level thread control is needed.

## 2. What is Thread Safety? How do you achieve it?

✓ **Thread Safety** means multiple threads can **safely** access shared resources **without conflicts**.

✓ **Achieved using:**

- **lock** keyword
- **Mutex** and **Semaphore**
- **Monitor** class
- **Interlocked** class

### Lock

- A **general term** for synchronization mechanisms used to restrict access to a shared resource.
- Can be **exclusive** (only one thread can access) or **shared** (multiple threads can read, but only one can write).
- Typically used in **multithreading** environments.

### Mutex (Mutual Exclusion)

- A **binary lock** (only one thread can hold it at a time).
- Used to protect **critical sections** in a program.
- The thread **holding the mutex must release it**; no other thread can release it.
- Prevents **race conditions**.

### Semaphore

- Can be **binary** (similar to mutex) or **counting** (allows multiple threads to access a resource up to a limit).
- Used for **resource management** (e.g., controlling access to a pool of connections).
- Any thread can **release a semaphore**, unlike a mutex, which must be released by the owner.
- Useful for **limiting concurrency** in applications.

## 6. How do you run multiple threads in parallel?

✓ Use `Task.Run()`, `Parallel.ForEach()`, or `Thread` class.

## 7. What is the purpose of Thread.Join() and Thread.IsAlive()?

- ✓ **Thread.Join()** – Waits for a thread to complete before proceeding.
- ✓ **Thread.IsAlive** – Checks if a thread is still running.

## 8. What is a Thread Pool in C#?

- ✓ **Thread Pool** is a collection of pre-created threads used to optimize performance.
- ✓ Uses `ThreadPool.QueueUserWorkItem()` to schedule tasks.

## 10. How to Handle Exceptions in Multithreading?

- ✓ Use **try-catch** inside the thread.
- ✓ In **Task**, use **ContinueWith()** to handle exceptions.

## Extension Method

Extension method allow us to add the new method in the existing class without creating the instance of the class or inheriting the class

### What is an Extension Method?

An **extension method** is a **static method** that allows you to **add new functionality** to an **existing class** **without modifying** its source code. It is defined inside a **static class** and uses **this** keyword before the first parameter to specify the type it extends.

### Why Use Extension Methods?

- ✓ **Enhances existing classes** without modifying their original code.
- ✓ **Improves code readability** by allowing method chaining.
- ✓ **Encapsulates reusable logic** in a structured way.

### How to Create an Extension Method?

- 1 Create a static class.
- 2 Define a static method inside the class.
- 3 Use **this** before the first parameter to indicate the extended type.
- 4 Call the method as if it were a built-in method.

- ✓ The class **must be static**.
- ✓ The method **must be static**.
- ✓ The **first parameter must have this** keyword followed by the type being extended.
- ✓ Extension methods **cannot override existing methods**.
- ✓ They **only work in the namespace where they are defined** (unless imported).

- ✓ **Extending Built-in .NET Types** – E.g., `string`, `int`, `DateTime`.
- ✓ **Enhancing LINQ Queries** – Custom query operations.
- ✓ **Extending Third-Party Libraries** – Add features to classes you **don't own**.
- ✓ **Code Reusability** – Keep helper methods separate without modifying core classes.

### 3. What is the difference between constant and readonly in C#?

A `const` keyword in C# is used to declare a constant field throughout the program. That means once a variable has been declared `const`, its value cannot be changed throughout the program.

In C#, a constant is a number, string, null reference, or boolean values.

### 4. What is Reflection in C#?

Reflection in C# extracts metadata from the datatypes during runtime.

To add reflection in the .NET framework, simply use `System.Reflection` namespace in your program to retrieve the type which can be anything from:

Assembly

Module

Enum

MethodInfo

ConstructorInfo

MemberInfo

ParameterInfo

Type

FieldInfo

EventInfo

PropertyInfo

### Difference Between `ref` and `out` in C#

In C#, both `ref` and `out` are **pass-by-reference** parameters, meaning they allow methods to modify the value of a variable outside their scope. However, they have key differences in usage and behavior.

### ◆ When to Use `ref` vs. `out`

✓ Use `ref` when:

- You **already have a value** and want to modify it inside the method.
- The variable will be **used before modification**.

✓ Use `out` when:

- The variable is **only meant to return a value** from the method.
- You need to **return multiple values** from a method (alternative to `Tuple` or `out parameters`).

## Difference Between `string` and `StringBuilder` in C#

### ◆ Key Differences

#### 1 Mutability

- `string` is **immutable** (cannot be changed after creation).
- `StringBuilder` is **mutable** (modifications happen in the same object).

#### 2 Performance

- `string` is **slow** for frequent modifications because each change creates a new object.
- `StringBuilder` is **faster** for multiple string manipulations as it modifies the same object.

#### 3 Memory Usage

- `string` creates a **new instance** every time it is modified, leading to high memory usage.
- `StringBuilder` **modifies existing memory**, reducing unnecessary allocations.

#### 4 String Concatenation

- Using `string` for concatenation (`+=`) creates multiple objects, impacting performance.
- `StringBuilder.Append()` is **optimized** for concatenation and avoids multiple allocations.

#### 5 Thread Safety

- `string` is **thread-safe** since it is immutable.
- `StringBuilder` is **not thread-safe** unless explicitly synchronized.

#### 6 Use Case

- Use **string** when the text **does not change frequently**.
- Use **StringBuilder** when **frequent modifications** (append, replace, insert) are required.

## What are Properties in C#?

Properties in C# are public members of a class where they provide the ability to access private members of a class. The basic principle of encapsulation lets you hide some sensitive properties from the users by making the variables private. The private members are not accessible otherwise in a class. Therefore, by using properties in C# you can easily access the private members and set their values. Used in encapsulation to use the private fields

### ◆ What is Boxing and Unboxing?

Boxing and Unboxing are processes that allow value types (**int**, **double**, **bool**, etc.) to be converted into reference types (**object**) and vice versa.

#### ◆ 1. Boxing

✓ **Converting a value type (e.g., **int**, **double**) into an object (reference type).**

✓ Happens **implicitly** when assigning a value type to an **object** or **dynamic**.

✓ Allocates memory on the **heap** (increases memory usage).

#### 2. Unboxing

✓ **Extracting a value type from an object (converting back to its original type).**

✓ Requires **explicit type casting**.

✓ The value is copied from the **heap to the stack**.