Tab 1

# **C# Programming – Lecture 5**

## **1. Abstract Classes & Abstract Methods**

An **abstract class in C#** is a class that **cannot be instantiated directly** and is intended to serve as a **base class** for other classes. It represents an **incomplete or generalized concept** that provides a common structure and behavior for its derived classes. An abstract class can contain:

* **Abstract methods** (methods without a body) that **must be implemented** by derived classes
* **Concrete methods** (methods with implementation) to provide **shared functionality**
* **Fields, properties, constructors, and access modifiers**

An **abstract method** defines **only the method signature**—including the method name, return type, and parameters—without providing an implementation. This forces derived classes to **provide their own implementation**, ensuring that specific behavior is defined at the child class level while maintaining a consistent contract.

Abstract classes are a key part of **object-oriented design**, supporting:

* **Abstraction** by hiding implementation details
* **Code reuse** through shared logic
* **Runtime polymorphism** by allowing base class references to invoke derived class methods

They are commonly used when multiple related classes share common behavior but require **different implementations for certain operations**.

### **Syntax**

abstract class Shape

{

public abstract double CalculateArea();

public void Display()

{

Console.WriteLine("This is a shape");

}

}

## **Why Is It Needed?**

### **1. Enforces a Common Contract for Derived Classes**

### Abstract classes ensure that all derived classes follow a predefined structure

### Guarantees that essential methods are implemented

### Prevents incomplete or inconsistent class behavior

### Example: Every Shape must calculate area, but how it does so may differ

### **2. Promotes Code Reusability**

### Common logic can be written once in the abstract class

### Avoids duplication of code across multiple derived classes

### Changes in shared logic need to be made only once

### public void Display()

### {

### Console.WriteLine("This is a shape");

### }

### **3. Supports Dynamic Polymorphism**

### Enables method calls to be resolved at runtime

### Base class references can point to derived objects

### Enhances flexibility and extensibility of applications

### Shape shape = new Circle();

### shape.CalculateArea(); // Calls Circle's implementation

### **4. Prevents Object Creation for Incomplete Base Classes**

### Abstract classes represent incomplete concepts

### Prevents misuse of base classes that lack full implementation

### Enforces proper object creation through derived classes only

### // Shape s = new Shape(); //Not allowed

## **When Is It Applied?**

### **1. When Base Class Should Not Be Instantiated**

### Used when the base class is conceptual

### Serves only as a template or blueprint

### **2. When Common Behavior Exists but Details Vary**

### Shared behavior goes into the abstract class

### Variable behavior is implemented by derived classes

### Example: All employees have a salary, but calculation differs

### **3. When Enforcing Method Implementation in Child Classes**

### Ensures that critical methods must be implemented

### Prevents developers from forgetting essential logic

## **Where Is It Applied?**

### **1. Framework Design**

### ASP.NET Core:

### ControllerBase

### DbContext

### Provides base functionality and enforces extension rules

### **2. Game Engines**

### Abstract classes like:

### Player

### Enemy

### Weapon

### Each derived class defines its own behavior

### **3. Banking, Reporting & Workflow Systems**

### Account, Transaction, Report as abstract classes

### Business rules enforced through abstract methods

## 

## **How Is It Implemented?**

### abstract class Shape

### {

### public abstract double CalculateArea();

### 

### public void Display()

### {

### Console.WriteLine("This is a shape");

### }

### }

### 

### class Circle : Shape

### {

### public override double CalculateArea()

### {

### return 3.14 \* 5 \* 5;

### }

### }

## **Use Case**

### Shape Hierarchy System

### Base class: Shape

### Derived classes: Circle, Rectangle, Triangle

### Each shape calculates area differently while sharing common behavior

## **Code Breakdown**

### 1. abstract Keyword

### Prevents direct instantiation

### Indicates incomplete implementation

### Forces inheritance-based usage

### 2. Abstract Method (CalculateArea)

### Has no body

### Must be overridden in derived classes

### Defines a strict contract

### 

### 3. override Keyword

### Required to implement abstract methods

### Ensures correct method signature

### Enables runtime binding

### 4. Concrete Methods Can Coexist

### Abstract classes can have fully implemented methods

### Shared logic is centralized

### Improves maintainability

### 5. Runtime Method Resolution

### Method call decided at runtime

### Supports extensible designs

### New shapes can be added without modifying existing code

### 6. Partial Implementation Rule

### If a derived class does not implement all abstract methods, it must also be marked abstract

### abstract class Rectangle : Shape

### {

### // No CalculateArea implementation

### }

### 7. Constructor Support

### Abstract classes can have constructors

### Used for initializing shared state

#### 

#### 8. Access Modifiers

* Abstract methods cannot be private
* Common modifiers:  
  + public
  + protected
  + internal

### **Use Case (Real World)**

**Banking System**

abstract class Account

{

public abstract double CalculateInterest();

public void DisplayAccountType()

{

Console.WriteLine("Bank Account");

}

}

class SavingsAccount : Account

{

public override double CalculateInterest()

{

return 0.05;

}

}



* Common behavior → base class
* Business rules → enforced via abstract methods

### **Comparison**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| Methods | Abstract + Concrete | Abstract only (default allowed in C# 8+) |
| Fields | Yes | No |
| Multiple Inheritance | No | Yes |

## **2. Interfaces**

An **interface in C#** defines a **contract or specification** that a class or struct must follow. It contains **only method declarations, properties, events, and indexers**, without providing implementation (prior to C# 8.0). An interface specifies **what a class must do**, but not **how it should do it**.

Interfaces enable **multiple inheritance**, promote **loose coupling**, and allow different, unrelated classes to share common behavior without sharing a class hierarchy. A class that implements an interface **must provide concrete implementations** for all its members.

In modern C# (8.0 and above), interfaces can also contain **default method implementations**, but their primary role remains defining contracts.

**Additionally, interfaces play a vital role in building scalable and maintainable applications**, especially in large enterprise systems. They are widely used in **dependency injection**, **unit testing**, and **framework design**, as they allow developers to swap implementations without changing the dependent code. By programming against interfaces rather than concrete classes, applications become easier to extend, test, and maintain, making interfaces a core principle of robust object-oriented and SOLID-based design.

### **Key Points**

* Defines a **behavioral contract** for classes and structs
* Contains **method, property, event, and indexer declarations**
* Does **not hold instance state or fields**
* Supports **multiple inheritance**
* Promotes **loose coupling and dependency inversion**
* Enables **runtime polymorphism**
* Implementing classes **must define all interface members**
* Commonly used in **APIs, services, dependency injection, and plugin systems**
* Enhanced in **C# 8.0+** with default interface methods

### **Syntax**

interface IPrintable

{

void Print();

}

**Naming Convention:** Interfaces typically start with the letter **I** (e.g., IPrintable, ILogger, IRepository) for clarity.

## 

## **Why Is It Needed?**

## **1. Enables Multiple Inheritance**

C# does **not support multiple inheritance using classes** to avoid complexity and ambiguity. However, many real-world problems require a class to support **multiple independent behaviors**. Interfaces solve this limitation by allowing a class to implement **multiple interfaces**.

class Document : IPrintable, ISerializable

{

public void Print() { }

public void Serialize() { }

}

**Why this matters:**

* A class can behave as both **printable** and **serializable**
* Prevents the *diamond problem* seen in multiple class inheritance
* Encourages **composition over inheritance**

**Real-world example:** A document can be printed, saved, exported, and logged—each represented by a separate interface.

### **2. Promotes Loose Coupling**

Interfaces help reduce **tight dependency** between components by allowing code to depend on **abstractions rather than concrete implementations**. This makes the system easier to change, extend, and maintain.

void PrintData(IPrintable printable)

{

printable.Print();

}

**Benefits:**

* Implementation can change without affecting dependent code
* Supports **Dependency Injection**
* Improves maintainability in large projects

**Enterprise usage:** ASP.NET Core services depend on interfaces, not concrete classes.

### **3. Ensures Consistent Behavior Across Unrelated Classes**

Interfaces allow **unrelated classes** (with no inheritance relationship) to share common behavior by implementing the same interface.

class Invoice : IPrintable { }

class Report : IPrintable { }

class Receipt : IPrintable { }

**Why this is important:**

* Enforces uniform method signatures
* Guarantees predictable behavior
* Enables polymorphism across unrelated objects

**Example:** Any object that implements IPrintable can be printed, regardless of its class hierarchy.

### **4. Improves Testability**

Interfaces make **unit testing and mocking** significantly easier. Instead of using real services like databases or APIs, **mock implementations** of interfaces can be injected during testing.

**Advantages:**

* Enables isolated unit testing
* Reduces dependency on external systems
* Improves test speed and reliability

**Enterprise practice:**

* Widely used with testing frameworks like **Moq, NUnit, xUnit**
* Supports **Test-Driven Development (TDD)**

IPaymentGateway mockPayment = new MockPaymentGateway();

## **When Is It Applied?**

### **1. When Multiple Classes Share Behavior but Not Hierarchy**

* Classes do not fit into a single inheritance tree
* Behavior-based design instead of hierarchy-based design

### **2. When Defining APIs or Services**

* Service contracts in web APIs
* Clear separation between interface and implementation

### **3. When Flexibility and Future Extension Are Required**

* New implementations can be added without modifying existing code

## **Where Is It Applied?**

### **1. Dependency Injection (DI)**

* ASP.NET Core uses interfaces extensively  
  Services are registered and resolved using interfaces

services.AddScoped<IService, ServiceImpl>();

### **2. Plugin Systems**

* External modules implement known interfaces
* Enables plug-and-play architecture

### **3. Repository & Service Layers**

* Common in layered architecture
* Business logic depends on interface, not database implementation

### **4. Framework & Library Design**

* .NET interfaces like:
  + IDisposable
  + IEnumerable
  + IComparable

## **How Is It Implemented?**

class Report : IPrintable

{

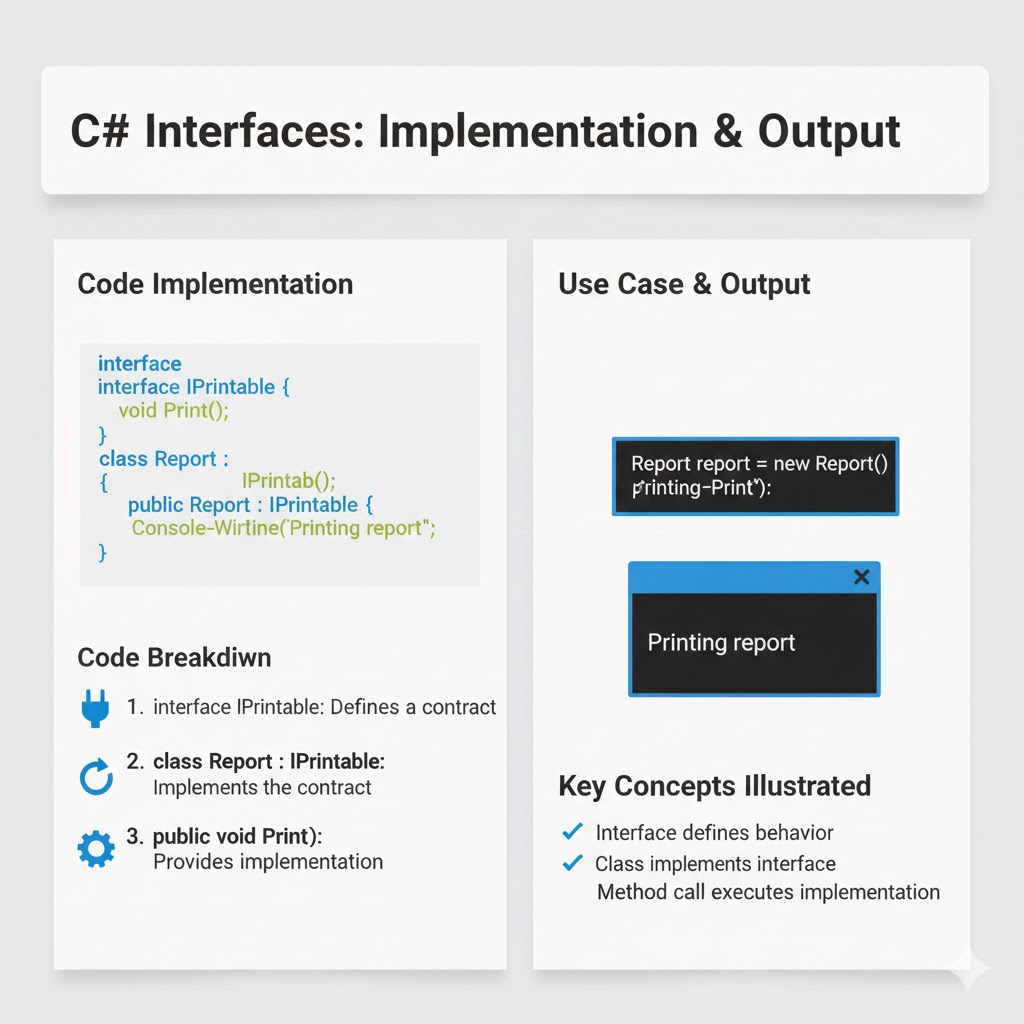
public void Print()

{

Console.WriteLine("Printing report");

}

}



## **Use Case**

### **Document Printing System**

* Interface: IPrintable
* Implementations: Invoice, Report, Receipt
* Any printable object can be processed uniformly

IPrintable doc = new Report();

doc.Print();

## **Code Breakdown**

## **1. Interface Members Are Public by Default**

* No access modifiers allowed
* All members are implicitly public

void Print(); // Always public

## **2. Class Must Implement All Interface Members**

When a class implements an interface, it is **mandatory** for the class to provide concrete implementations for **all interface members**. This ensures that the contract defined by the interface is fully honored.

interface IPrintable

{

void Print();

void Scan();

}

class Report : IPrintable

{

public void Print()

{

Console.WriteLine("Printing report");

}

}

If a class **does not implement all members**, the compiler raises an error. In such cases, the class must be declared **abstract**, allowing derived classes to complete the implementation.

abstract class Document : IPrintable

{

// Print() not implemented

}

**Why this is important:**

* Enforces reliability and consistency
* Prevents incomplete implementations
* Ensures predictable behavior across all implementations

## **3. No Fields or Instance State (Pre-C# 8)**

Before C# 8.0, interfaces could **not contain fields or instance variables**. They were strictly used to define **behavior**, not data.

interface IExample

{

// int count; ❌ Not allowed

void Execute();

}

**Key idea:**

* Interfaces define *what an object can do*, not *what it contains*
* Keeps interfaces lightweight and focused on contracts

**Note:** From C# 8.0 onwards, interfaces can contain static members and default implementations, but they still do not maintain instance state.

## **4. Multiple Interface Implementation**

A class in C# can implement **multiple interfaces**, which effectively solves the problem of **multiple inheritance** that is not supported with classes.

interface IPrintable

{

void Print();

}

interface ISerializable

{

void Serialize();

}

class Document : IPrintable, ISerializable

{

public void Print() { }

public void Serialize() { }

}

**Benefits:**

* Combines multiple behaviors in a single class
* Avoids ambiguity issues (diamond problem)
* Encourages modular and flexible design

**Real-world example:**  A file can be printed, saved, logged, and uploaded—each represented by a different interface.

## **5. Explicit Interface Implementation**

Explicit interface implementation is used when:

* Multiple interfaces contain methods with the **same name**
* You want to **hide interface methods** from the class’s public API

interface ILogger

{

void Log();

}

class FileLogger : ILogger

{

void ILogger.Log()

{

Console.WriteLine("Logging to file");

}

}

ILogger logger = new FileLogger();

logger.Log(); // Accessible

// FileLogger fl = new FileLogger();

// fl.Log(); Not accessible directly

**Advantages:**

* Resolves method name conflicts
* Improves encapsulation
* Prevents accidental misuse of interface methods

## **6. Runtime Polymorphism**

Interfaces fully support **runtime polymorphism**, where the method call is resolved **at runtime** based on the actual object type, not the reference type.

IPrintable printable = new Report();

printable.Print();

**Why this matters:**

* Behavior can be changed dynamically
* New implementations can be added without modifying existing code
* Core principle behind **plug-and-play architectures**

**Used extensively in:**

* Dependency Injection
* Plugin systems
* Enterprise frameworks like ASP.NET Core

## **Comparison: Interface vs Abstract Class**

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| Multiple Inheritance | Yes | No |
| Method Implementation | No (Default allowed in C# 8+) | Yes |
| Fields | No | Yes |
| Constructors | No | Yes |
| Access Modifiers | Not allowed | Allowed |
| Use Case | Contract | Base behavior |

# **3. Dynamic Polymorphism (Runtime Polymorphism)**

Dynamic polymorphism in C# is the mechanism that allows a program to **select and execute the appropriate method implementation at runtime**, depending on the **actual type of the object** being referenced, rather than the type of the reference variable. Even when an object is accessed through a **base class or interface reference**, the Common Language Runtime (CLR) determines which overridden method to invoke based on the object’s real type.

This concept is implemented using **inheritance**, where a derived class inherits from a base class, and **method overriding**, where the derived class provides its own implementation of a virtual or abstract method defined in the base class. Dynamic polymorphism enables **runtime flexibility**, supports extensibility, and allows developers to write **generic, loosely coupled code** that can work with multiple object types seamlessly. It is a core feature in enterprise applications, frameworks, and real-time systems where behavior must vary dynamically without changing existing code.

## **Syntax**

Shape s = new Circle();

s.CalculateArea();

Here, the method call is resolved **at runtime**, not at compile time.

## **Why Is It Needed?**

* Provides **runtime flexibility** in program behavior
* Allows writing **generic, reusable, and extensible code**
* Supports the **Open–Closed Principle** (open for extension, closed for modification)
* Helps build **loosely coupled systems**
* Essential for **real-time and enterprise applications**

## **When Is It Applied?**

* When method behavior must be chosen **during program execution**
* When a **base class or interface reference** refers to different derived objects
* When adding new features **without modifying existing code**

## **Where Is It Applied? (Real-World Usage)**

* **Framework-level abstractions**
  + ASP.NET Controllers
  + Middleware pipelines
* **Payment processing systems**
  + CreditCardPayment, UpiPayment, NetBanking
* **Logging systems**
  + FileLogger, DatabaseLogger, CloudLogger
* **Notification systems**
  + Email, SMS, Push Notifications
* **Game engines**
  + Different characters, weapons, or physics behaviors
* **Real-time systems**
  + Sensor data handling
  + Trading and monitoring systems

## **How Is It Implemented? (Code Example)**

### **Base Class**

abstract class Shape

{

public abstract double CalculateArea();

}

### **Derived Class**

class Circle : Shape

{

public override double CalculateArea()

{

return 3.14 \* 5 \* 5;

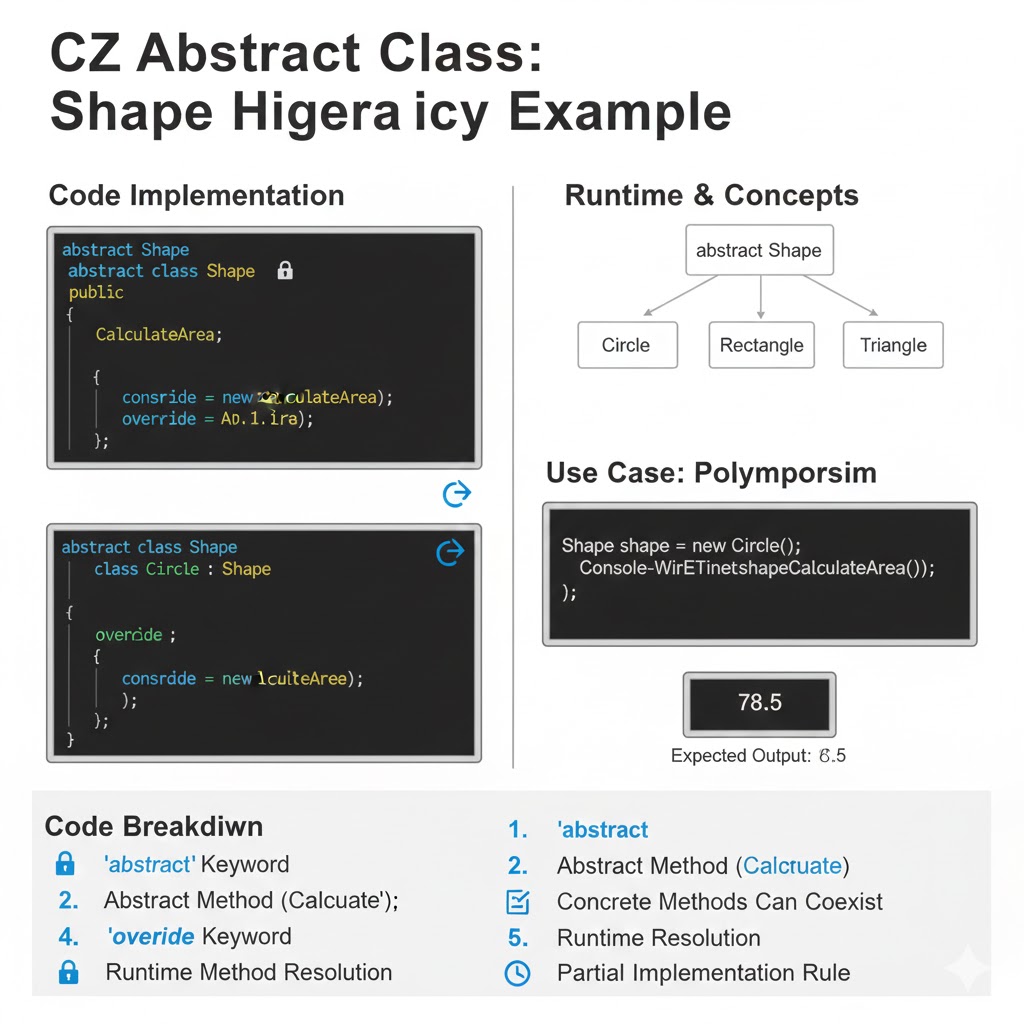
}

}

### **Runtime Execution**

Shape shape = new Circle();

Console.WriteLine(shape.CalculateArea());



### **Use Case (Real-Time Example): Payment Processing System**

Dynamic polymorphism is widely used in real-world payment systems, where the same request must be handled differently based on the payment type, such as Credit Card, UPI, Net Banking, or Wallets—without changing the calling code.

### **Code Example**

abstract class Payment

{

public abstract void ProcessPayment();

}

class CreditCardPayment : Payment

{

public override void ProcessPayment()

{

Console.WriteLine("Processing credit card payment");

}

}

class UpiPayment : Payment

{

public override void ProcessPayment()

{

Console.WriteLine("Processing UPI payment");

}

}

Payment payment = new UpiPayment();

payment.ProcessPayment(); // Decided at runtime

### **Explanation of Runtime Behavior**

* The variable payment is of type Payment (base class reference)
* At runtime, it holds an object of UpiPayment (derived class)
* The CLR invokes the overridden method of UpiPayment, not the base class
* The method call is resolved dynamically at runtime

### **Why This Is a Perfect Real-Time Example**

Same code path handles all payment typesDifferent behavior based on actual object typeRuntime decision making without if-else or switchNew payment methods can be added without modifying existing code

### 

### **Code Breakdown**

1. **Base Class Reference :**

Payment payment;

* Allows handling all payment types using a single reference
* Promotes loose coupling and abstraction

1. **Abstract Method**public abstract void ProcessPayment();

* Forces all derived classes to implement their own payment logic
* Ensures a common contract across all payment types

1. **Method Overriding**public override void ProcessPayment()

* Replaces the base class method with specific implementation
* Enables runtime method selection

1. **Runtime Binding by CLR**payment.ProcessPayment();

* The CLR checks the actual object type at runtime
* Executes UpiPayment.ProcessPayment()
* This is known as late binding

### **Key Advantages Demonstrated**

* Eliminates complex conditional logic
* Improves scalability and maintainability
* Supports Open–Closed Principle
* Essential for enterprise and real-time systems

### **Compile-Time vs Runtime Polymorphism**

| **Feature** | **Compile-Time Polymorphism** | **Runtime Polymorphism** |
| --- | --- | --- |
| Also Called | Static Polymorphism | Dynamic Polymorphism |
| Achieved By | Method Overloading | Method Overriding |
| Binding Time | Compile time | Runtime |
| Flexibility | Low | High |
| Performance | Faster | Slightly slower |

## **4. Multiple Inheritance (C#)**

## **Multiple inheritance** refers to the ability of a class to inherit features from **more than one parent**. In C#, **multiple inheritance using classes is not supported** to avoid ambiguity and complexity (such as the diamond problem). However, C# **supports multiple inheritance through interfaces**, allowing a class to implement **multiple behaviors** safely.

### **Syntax**

#### **Not Allowed (Class-Based Multiple Inheritance)**

## 

## // This is NOT allowed in C#

## class C : A, B { }

#### **Allowed (Interface-Based Multiple Inheritance)**

## 

## interface IA

## {

## void MethodA();

## }

## 

## interface IB

## {

## void MethodB();

## }

## 

## class Test : IA, IB

## {

## public void MethodA()

## {

## Console.WriteLine("Method A implemented");

## }

## 

## public void MethodB()

## {

## Console.WriteLine("Method B implemented");

## }

## }

### **Why Is Multiple Inheritance Needed?**

### **Enables Multiple Independent Behaviors** In real-world applications, a single class often needs to perform more than one role. For example, a service may need to log information, send notifications, and process data at the same time. Multiple inheritance through interfaces allows a class to adopt all these behaviors without being restricted to a single base class.

### **Avoids Ambiguity of Multiple Base Classes** Traditional multiple inheritance with classes can lead to problems such as method name conflicts and the *diamond problem*. C# avoids these issues by disallowing multiple class inheritance and instead using interfaces, which contain no state and therefore eliminate ambiguity.

### **Encourages Modular and Reusable Design** Interfaces promote separation of concerns. Each interface defines a small, focused behavior that can be reused across multiple classes. This results in cleaner, more maintainable, and highly reusable code components.

### **Solves Limitations of Single Inheritance** Since C# allows a class to inherit from only one base class, interfaces provide a way to overcome this limitation by enabling a class to implement multiple contracts and participate in different roles simultaneously.

### **When Is It Applied? (Expanded)**

### **When a Class Must Follow Multiple Contracts** If a class needs to guarantee the implementation of multiple behaviors (e.g., logging, validation, serialization), interfaces allow it to comply with all required contracts.

### **When Behaviors Are Unrelated but Required Together** In many systems, behaviors are logically independent. For example, printing and saving data are unrelated operations but may both be required in the same class. Interfaces allow combining such unrelated functionalities.

### **When Designing Extensible Systems** Multiple inheritance through interfaces is ideal for systems that are expected to grow. New behaviors can be introduced as new interfaces without affecting existing class hierarchies.

### **Where Is It Applied? (Expanded)**

### **Enterprise Applications** Used extensively in layered architectures where services implement multiple interfaces such as business logic, logging, and security.

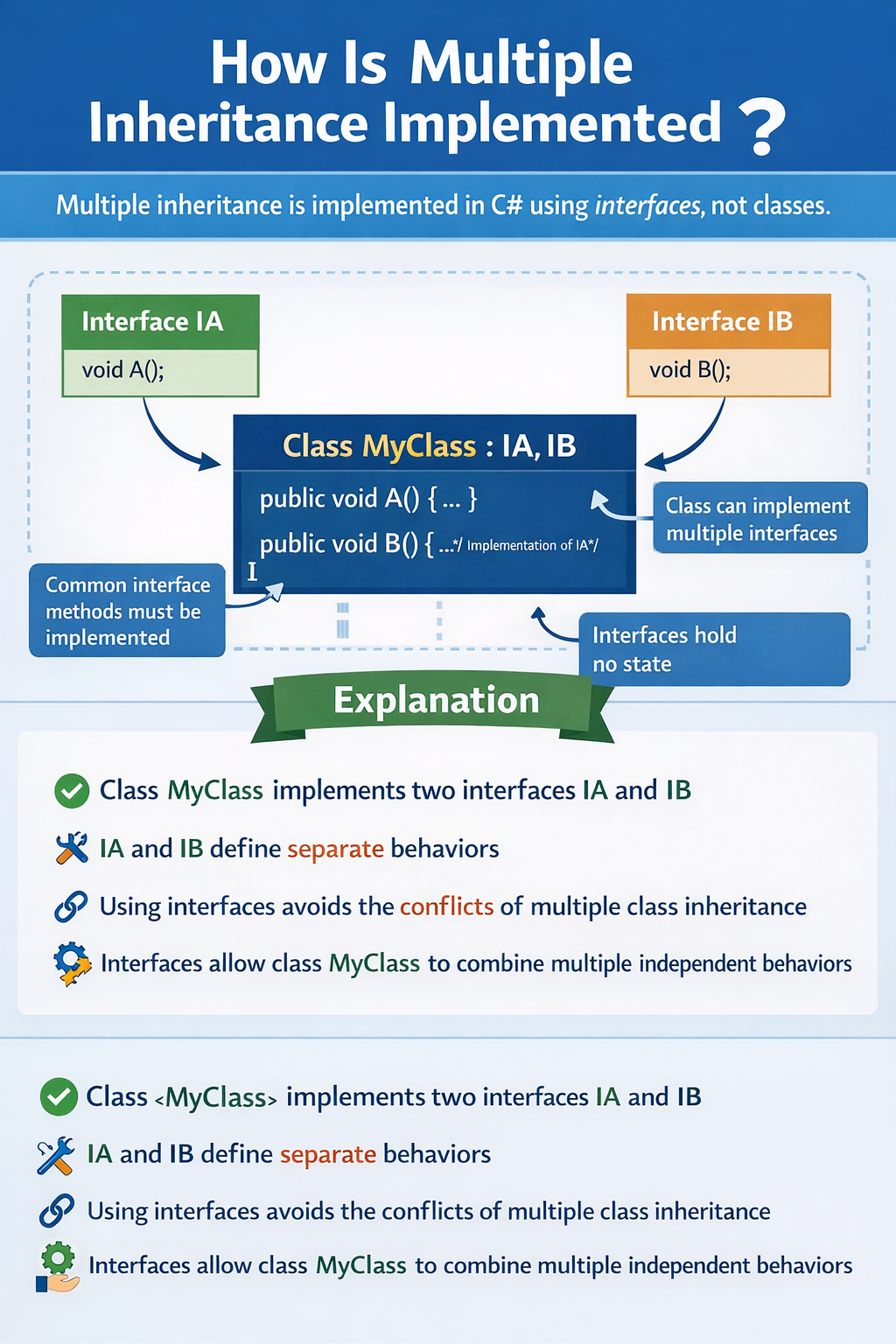
### **Plugin-Based Architectures** Plugins implement common interfaces to integrate seamlessly with the main application while providing different functionalities.

### **Payment and Notification Systems** A single service may handle payments, send alerts, and log transactions by implementing multiple interfaces.

### **Microservices and API Contracts** Interfaces define service contracts and APIs, allowing different services or implementations to adhere to the same specifications while evolving independently.

### **How Is It Implemented?**

### Multiple inheritance is implemented using **interfaces**, not classes.



### **Advantages**

## Safe alternative to multiple class inheritance

## Promotes loose coupling

## Supports runtime polymorphism

## Enhances scalability and testability

## Improves code maintainability

## Encourages modular design

## Facilitates dependency injection

## Ideal for enterprise and framework development

### **Real-Time Use Case: Enterprise Order Management System**

In large-scale **enterprise applications**, a single service class often needs to handle **multiple responsibilities** such as logging system activities, sending notifications, validating data, and interacting with external services. C# achieves this cleanly using **multiple inheritance through interfaces**.

### **Code Example**

interface ILogger

{

void Log(string message);

}

interface INotifier

{

void Notify(string message);

}

class OrderService : ILogger, INotifier

{

public void Log(string message)

{

Console.WriteLine("Log: " + message);

}

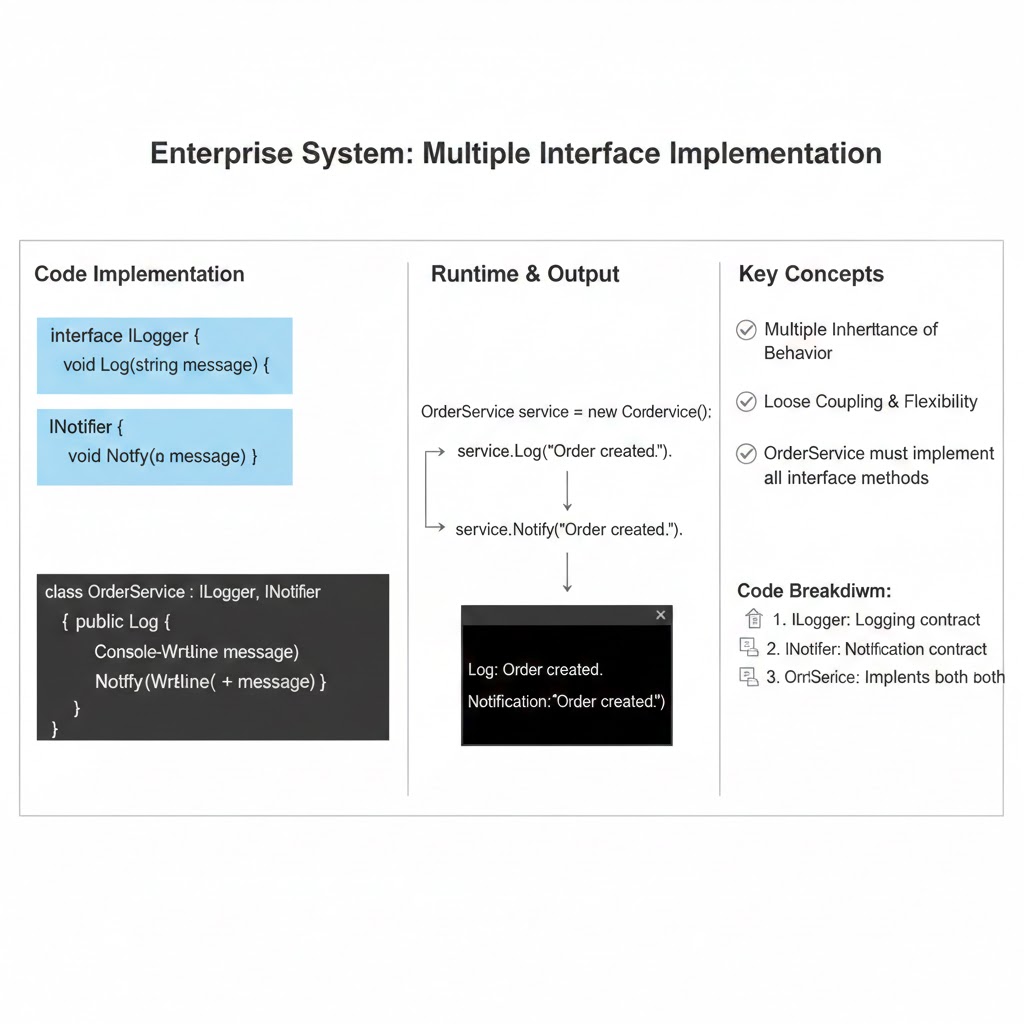
public void Notify(string message)

{

Console.WriteLine("Notification: " + message);

}

}



### **Explanation**

1. **Multiple Responsibilities in One Service**OrderService needs to log operations (such as order creation or failure) and notify users or administrators. Instead of inheriting from multiple classes (which is not allowed), it **implements multiple interfaces**, each representing a specific behavior.
2. **Clean Separation of Concerns**
   * ILogger handles logging-related behavior
   * INotifier handles notification-related behavior  
      This separation keeps the system modular and easy to maintain.
3. **No Shared State – No Ambiguity** Interfaces do not store data or state. As a result, there is no confusion over which implementation or variable to use, eliminating ambiguity and runtime conflicts.
4. **Easy Extensibility** If new behavior is required (e.g., auditing or caching), a new interface can be introduced and implemented without modifying existing interfaces or breaking current code.

### **Code Breakdown**

#### **1. Multiple Interface Implementation**

class OrderService : ILogger, INotifier

* The class implements **more than one interface**
* This enables **multiple inheritance safely**
* Provides flexibility to combine unrelated behaviors

#### **2. Mandatory Method Implementation**

public void Log(string message)

public void Notify(string message)

* All interface methods **must be implemented**
* Ensures the class adheres strictly to the defined contracts
* If not implemented, the class must be declared abstract

#### **3. No State Sharing**

interface ILogger { }

interface INotifier { }

* Interfaces do **not contain instance variables**
* They define behavior only, not data
* Prevents conflicts that occur in multiple class inheritance

### **Key Advantages Demonstrated**

* Supports multiple behaviors in one class
* Improves maintainability and testability
* Enables loose coupling and dependency injection
* Ideal for enterprise-scale systems

### 

### **Comparison: Class vs Interface Inheritance**

| **Feature** | **Class Inheritance** | **Interface Inheritance** |
| --- | --- | --- |
| Multiple Inheritance | Not Supported | Supported |
| State | Allowed | Not Allowed |
| Method Implementation | Yes | Optional (C# 8+) |
| Flexibility | Limited | High |

### **5. Explicit Interface Implementation**

**Explicit interface implementation** in C# allows a class to implement interface members **without making them publicly accessible through the class instance**. The implemented methods can only be accessed **via an interface reference**, which helps resolve conflicts when multiple interfaces have methods with the same name.

### **Syntax**

interface ILogger

{

void Log();

}

interface INotifier

{

void Log();

}

class FileLogger : ILogger, INotifier

{

void ILogger.Log()

{

Console.WriteLine("Logging to file via ILogger");

}

void INotifier.Log()

{

Console.WriteLine("Logging to notification via INotifier");

}

}

### **When & Where Applied**

* When **multiple interfaces contain identical method names**
* In **framework-level APIs** where methods must be separated per interface
* When **class should not expose all interface members publicly**

### **Use Case: Implementing IDisposable Alongside Custom Interfaces**

class ResourceHandler : IDisposable, INotifier

{

void IDisposable.Dispose()

{

Console.WriteLine("Resource disposed");

}

void INotifier.Log()

{

Console.WriteLine("Notification sent");

}

}

// Accessing via interface references

IDisposable resource = new ResourceHandler();

resource.Dispose(); // Works

INotifier notifier = new ResourceHandler();

notifier.Log(); // Works

ResourceHandler obj = new ResourceHandler();

obj.Dispose(); // ERROR: not accessible directly

obj.Log(); // ERROR: not accessible directly

### 

### **How Is It Implemented**

1. Prefix the interface name with the method:

void InterfaceName.MethodName() { ... }

1. Method is **only callable via interface reference**:

ILogger logger = new FileLogger();

logger.Log(); // Works

FileLogger fileLogger = new FileLogger();

fileLogger.Log(); // ERROR: method not accessible

### **Code Breakdown**

### **1. Interface Reference Required**

* Explicitly implemented methods are **not part of the class’s public API**
* They can be accessed **only through interface-type variables**
* Enforces strict usage of interfaces as contracts

ILogger logger = new FileLogger(); // Correct access

### **2.Separate Implementations**

* A class can implement **multiple interfaces with identical method names**
* Each interface can have its **own distinct implementation**
* Eliminates naming conflicts completely

void ILogger.Log() { }

void INotifier.Log() { }

### **3.Controlled Access**

* Interface methods are hidden from direct class usage
* Improves **encapsulation and abstraction**
* Ensures users interact with the object only through the intended interface

### 

### **Advantages**

* Resolves **method name conflicts**
* Enhances **encapsulation and access control**
* Supports **multiple interface implementation** safely
* Prevents accidental invocation of interface methods

## **1. Interface Definition (Common for Both)**

interface ILogger

{

void Log();

}

## **2. Implicit Interface Implementation**

### **Code**

class FileLogger : ILogger

{

// Implicit implementation

public void Log()

{

Console.WriteLine("Logging to file");

}

}

### **Usage**

FileLogger f = new FileLogger();

f.Log(); // Works

ILogger i1 = f;

i1.Log(); // Works

### **Key Observations**

* Method must be public
* Can be accessed via:  
  + Class object
  + Interface reference
* Most commonly used

## **3. Explicit Interface Implementation**

### **Code**

class DbLogger : ILogger

{

// Explicit implementation

void ILogger.Log()

{

Console.WriteLine("Logging to database");

}

}

### **Usage**

DbLogger d = new DbLogger();

// d.Log(); // Compile-time error

ILogger i2 = d;

i2.Log(); // Works

### **Key Observations**

* No access modifier allowed
* Cannot be accessed using class object
* Accessible only through interface reference
* Used to hide implementation or resolve conflicts

## **4. Same Class Using Both (Interview Favorite)**

class HybridLogger : ILogger

{

// Implicit

public void Log()

{

Console.WriteLine("General logging");

}

// Explicit

void ILogger.Log()

{

Console.WriteLine("Interface-specific logging");

}

}

### **Usage**

HybridLogger h = new HybridLogger();

h.Log(); // General logging

ILogger i3 = h;

i3.Log(); // Interface-specific logging

## **5. Comparison Summary (Code-Level)**

| **Feature** | **Implicit** | **Explicit** |
| --- | --- | --- |
| Access modifier | public required | Not allowed |
| Callable via class object | Yes | No |
| Callable via interface reference | Yes | Yes |
| Method visibility | Public | Hidden |
| Used for conflicts | No | Yes |

## **6. One-Line Interview Answer**

Implicit interface methods are public and accessible via both class and interface references, whereas explicit interface methods are accessible only through the interface reference.

## **7. Common Interview Traps**

* Explicit methods are implicitly private
* Explicit implementation does not override
* Same method name can exist in both implicit and explicit forms
* Explicit methods are useful when multiple interfaces have same method names

## **Why is Explicit Interface Implementation Needed in C#?**

Explicit interface implementation is needed to control how and when interface methods are exposed and resolved, especially in complex, real-world systems.

Below are the exact technical reasons—this is how you should explain it in interviews and exams.

## **1. To Resolve Method Name Conflicts (MOST IMPORTANT)**

### **Problem**

Two interfaces can have same method names with same signatures.

interface IFile

{

void Open();

}

interface IDatabase

{

void Open();

}

A class implementing both cannot decide automatically which Open() belongs to which interface.

### **Solution – Explicit Implementation**

class Resource : IFile, IDatabase

{

void IFile.Open()

{

Console.WriteLine("Opening file");

}

void IDatabase.Open()

{

Console.WriteLine("Opening database");

}

}

### **Why Implicit Fails Here**

* One public void Open() cannot satisfy both different behaviors
* Explicit implementation clearly separates responsibilities

## **2. To Hide Interface Methods from Class Users**

### **Scenario**

Some interface methods are not meant to be called directly by class consumers.

### **Example**

interface IInternalAudit

{

void Audit();

}

class BankAccount : IInternalAudit

{

void IInternalAudit.Audit()

{

Console.WriteLine("Internal audit process");

}

}

### **Why This Matters**

* BankAccount users cannot see or misuse Audit()
* Only systems working via IInternalAudit can access it

This improves encapsulation and API safety.

## **3. To Provide Different Behaviors for Same Method Name**

### **Scenario**

A class must behave differently based on role.

interface IUser

{

void Access();

}

interface IAdmin

{

void Access();

}

### **Explicit Implementation**

class Account : IUser, IAdmin

{

void IUser.Access()

{

Console.WriteLine("User-level access");

}

void IAdmin.Access()

{

Console.WriteLine("Admin-level access");

}

}

Here, behavior changes based on interface reference, not object.

## **4. To Enforce Interface-Based Programming**

Explicit implementation forces consumers to use interfaces, not concrete classes.

### **Why This is Good Design**

* Loose coupling
* Better testability
* Clear separation of responsibilities
* Supports Dependency Injection

## **5. To Avoid Polluting Public Class API**

Without explicit implementation:

* All interface methods become public
* Class API becomes bloated

With explicit implementation:

* Interface methods stay hidden
* Class remains clean and focused

## **6. Summary Table – Why Explicit Is Needed**

| **Reason** | **Why Explicit Helps** |
| --- | --- |
| Method conflicts | Separates same-named methods |
| API control | Hides methods from class |
| Multiple behaviors | Different logic per interface |
| Clean design | Prevents class API pollution |
| Enterprise design | Supports SOLID principles |

## **7. One-Line Interview Answer**

Explicit interface implementation is needed to resolve method name conflicts, hide interface methods from class consumers, and enforce strict interface-based access in complex designs.

## **8. When NOT to Use Explicit Implementation**

* When interface has no method conflicts
* When methods should be part of public class API
* For simple or beginner-level designs

## **6. Namespaces**

A namespace in C# is a **logical container** that groups related programming elements such as **classes, interfaces, structures, enumerations, and delegates** under a common name. It provides a **hierarchical organization** of code, making large applications easier to understand, manage, and maintain. By using namespaces, developers can structure applications in a way that reflects real-world modules, layers, or business domains.

Namespaces also play a crucial role in **preventing naming conflicts**, especially when multiple libraries or assemblies define types with the same class name. By qualifying a type with its namespace, the C# compiler can clearly distinguish between identically named classes. This makes namespaces essential in **large-scale, enterprise-level applications**, reusable libraries, frameworks, and collaborative projects where code from multiple sources is combined.

### **Syntax**

namespace Company.Project.Module

{

class Employee

{

public void Display()

{

Console.WriteLine("Employee Details");

}

}

}

### **Why Is a Namespace Needed? (Expanded)**

* **Organizes Large Codebases into Logical Groups** Namespaces help structure large applications by grouping related classes and components together, making the codebase easier to navigate and understand.
* **Avoids Class Name Collisions** In real-world projects, different modules or libraries may define classes with the same name. Namespaces uniquely identify each class and prevent ambiguity during compilation.
* **Improves Code Readability and Maintainability** A well-designed namespace hierarchy clearly communicates the purpose of classes, making the code easier to read, debug, and maintain over time.
* **Helps Manage Large Teams and Enterprise Projects** Teams can work on different namespaces independently without interfering with each other’s code, enabling parallel development and better collaboration.
* **Supports Modular and Layered Architecture** Namespaces align naturally with architectural layers such as Presentation, Business Logic, and Data Access, promoting clean and scalable application design.

### **When Is It Applied? (Expanded)**

* **When a Project Contains Multiple Classes and Modules**As soon as an application grows beyond a few classes, namespaces become essential for proper organization.
* **When Building Libraries, Frameworks, or APIs**Public libraries use namespaces to expose well-structured and versioned APIs to consumers.
* **When Integrating Third-Party Assemblies**Namespaces help distinguish custom code from external libraries and avoid naming conflicts.
* **When Following Clean Architecture Principles**Namespaces help separate concerns and enforce boundaries between different layers of the application.

### **Where Is It Applied? (Expanded)**

* **Large-Scale Enterprise Applications**Used to structure complex systems with hundreds or thousands of classes.
* **.NET Framework and .NET Core Libraries**Core libraries such as System, System.Collections, and System.Linq rely heavily on namespaces.
* **SDKs and Reusable Class Libraries**Namespaces ensure clarity and reusability when distributing shared code.
* **Web Applications (ASP.NET MVC, Web API)**Commonly used to separate Controllers, Models, Services, and Data layers.

### **Example: Two Classes with the Same Name in Different Namespaces**

namespace HR

{

class Employee

{

public void Info()

{

Console.WriteLine("HR Employee");

}

}

}

namespace Finance

{

class Employee

{

public void Info()

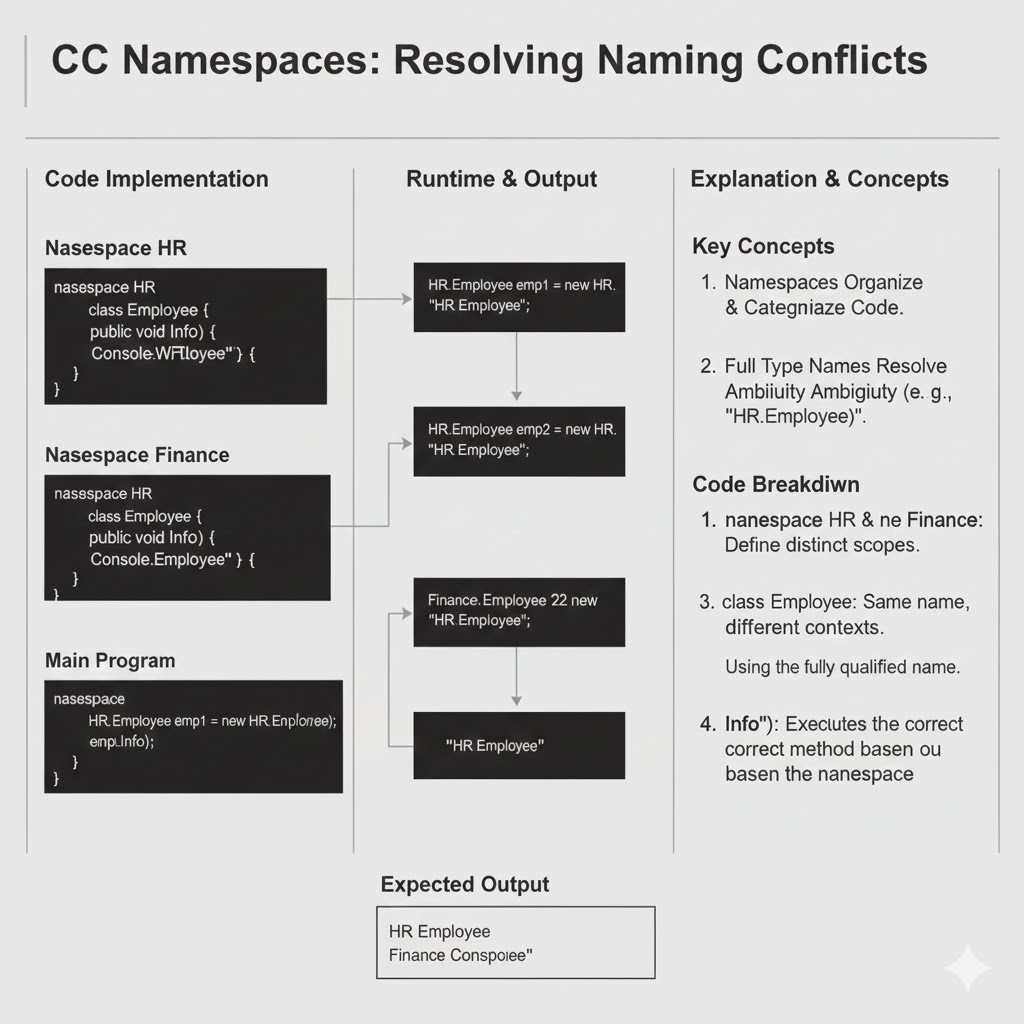
{

Console.WriteLine("Finance Employee");

}

}

}



In this example, both the HR and Finance namespaces contain a class named Employee. Even though the class names are identical, there is **no conflict** because each class belongs to a different namespace.

### **Using the Namespace**

HR.Employee emp1 = new HR.Employee();

Finance.Employee emp2 = new Finance.Employee();

* The **fully qualified name** (Namespace.ClassName) tells the compiler exactly which class to use
* Both classes can coexist in the same project without ambiguity

### **Use Case**

* Common in enterprise applications where different modules (HR, Finance, Sales) may define similar entities
* Useful when integrating multiple libraries that contain identically named classes
* Supports clean separation of business domains

### **Code Breakdown**

### **1. namespace Keyword**

* The namespace keyword is used to define a **logical container** for grouping related types such as **classes, interfaces, structures, enums, and delegates**.
* It does not affect how code runs at runtime but plays a critical role in **code organization and structure**.
* Helps separate different functional areas of an application into clear boundaries.

### **2. Dot Notation**

Dot notation allows creating **hierarchical namespaces**, such as:  
 namespace Company.Project.HR

* Represents parent–child relationships between modules.
* Improves clarity by reflecting real-world application structure (company → project → module).
* Makes large applications easier to navigate and understand.

### **3. Prevents Ambiguity**

* When multiple classes have the same name, namespaces uniquely identify them using **fully qualified names**.
* This prevents naming conflicts at compile time and ensures the compiler knows exactly which class to use.
* Essential when working with **third-party libraries** or multiple project modules.

### **4. Improves Scalability**

* New modules, features, or components can be added under new namespaces without impacting existing code.
* Encourages modular development, allowing applications to grow smoothly over time.
* Ideal for enterprise systems that evolve continuously.

### **5. Enhances Maintainability**

* Well-organized namespaces make code **easier to locate, read, debug, and modify**.
* Reduces complexity in large projects with many files and contributors.
* Improves collaboration among developers by enforcing a consistent structure.

### **Comparison with Similar Concepts**

| **Feature** | **Namespace** | **Class** |
| --- | --- | --- |
| Purpose | Logical grouping | Blueprint of objects |
| Prevents Name Conflicts | Yes | No |
| Contains | Classes, interfaces, enums | Fields & methods |
| Scope | Logical | Physical |

## **7. Nested Namespaces**

A **Nested Namespace** is a namespace defined **inside another namespace** to form a **hierarchical and logical structure** for organizing related program elements such as **classes, interfaces, structs, enums, and delegates**.

In simple terms, **nested namespaces function like folders within folders in a file system**. Just as folders help organize files on a computer, nested namespaces help organize code in a structured, meaningful way. This hierarchical grouping allows developers to separate functionality based on **application domains, modules, layers, or business units**, making large codebases easier to understand and maintain.

Nested namespaces play a critical role in **large-scale and enterprise-level applications** by:

* Clearly indicating ownership and responsibility of code
* Reducing naming conflicts between types with the same name
* Supporting modular and layered architecture
* Improving code readability and navigation
* Enabling better collaboration among development teams
* Making applications easier to extend and scale over time

By using nested namespaces, developers can design software that is **well-structured, self-documenting, and aligned with real-world business organization**, ultimately resulting in cleaner, more maintainable, and professional C# applications.

Example idea:

Company

└── Project

└── HR

└── Employee

## **Syntax**

### **Compact (Dot Notation) Syntax**

namespace A.B.C

{

class MyClass

{

}

}

**Expanded Syntax**

namespace A

{

namespace B

{

namespace C

{

class MyClass

{

}

}

}

}

**Both syntaxes are functionally identical**.  
The **dot notation** is preferred in modern C# for **better readability and simplicity**, as it allows developers to define nested namespaces in a **clear, concise, and intuitive manner**. It reduces unnecessary nesting and indentation, making the code easier to read, write, and maintain.

## **Why Is It Needed?**

Nested namespaces are essential in **real-world, large-scale applications** for the following reasons:

### **1. Logical Organization**

* Groups related functionality together
* Makes the project structure easy to understand
* Reflects business or system architecture clearly

### **2. Prevents Naming Conflicts**

Two classes can have the same name if they belong to different namespaces:

Company.Project.HR.Employee

Company.Project.Finance.Employee

### **3. Improves Readability**

* Developers instantly know where a class belongs
* Code becomes self-documenting

### **4. Improves Scalability**

* New modules can be added without disturbing existing code
* Ideal for long-term projects

### **5. Team Collaboration**

* Different teams can work on different namespaces
* Reduces merge conflicts in large teams

### **6. Enterprise-Level Maintainability**

* Code becomes easier to debug, refactor, and extend
* Encourages clean architecture principles

## **When Is It Applied?**

Nested namespaces are applied when:

* The application is **large or growing**
* Multiple modules or domains exist
* Following **layered architecture**
* Creating **reusable libraries or frameworks**
* Working with **enterprise software systems**

### **Common Scenarios:**

* Multi-department business applications
* Microservices-based systems
* API-based architectures
* SDK and framework development

## **Where Is It Applied?**

Nested namespaces are widely used in:

### **1. Banking Software**

Banking.Accounts

Banking.Loans

Banking.Payments

### **2. ERP Systems**

ERP.Inventory

ERP.HR

ERP.Finance

### **3. Web APIs**

MyApp.Controllers

MyApp.Services

MyApp.Repositories

### **4. Desktop Applications**

DesktopApp.UI

DesktopApp.BusinessLogic

DesktopApp.DataAccess

### **5. Game Engines**

Game.Physics

Game.Graphics

Game.AI

## **How Is It Implemented**

using System;

namespace Company.Project.HR

{

class Employee

{

public void Display()

{

Console.WriteLine("Employee from HR Department");

}

}

}

class Program

{

static void Main()

{

Company.Project.HR.Employee emp = new Company.Project.HR.Employee();

emp.Display();

}

}

## **Detailed Code Breakdown**

### **1. namespace Company.Project.HR**

namespace Company.Project.HR

{

...

}

This line defines a **nested namespace** using **dot notation**.

* **Company** → Represents the **organization or company name**
* **Project** → Represents a **specific application or product**
* **HR** → Represents a **module or department** (Human Resources)

✔ This hierarchical naming clearly communicates **ownership, responsibility, and purpose** of the code.  
✔ It mirrors real-world business structure, making the application intuitive to understand.

### 

### **2. class Employee**

class Employee

{

public void Display()

{

Console.WriteLine("Employee from HR Department");

}

}

* The Employee class contains **HR-specific logic**
* It is **scoped only to the HR namespace**
* Encapsulation ensures that the class does not interfere with other Employee classes in different modules

This avoids conflicts such as:

Company.Project.Finance.Employee

Company.Project.IT.Employee

Each Employee class can have its own behavior without collision.

### **3. Fully Qualified Name Usage**

Company.Project.HR.Employee emp = new Company.Project.HR.Employee();

This is called a **Fully Qualified Name**.

✔ Ensures **zero ambiguity**✔ Compiler knows exactly which Employee class is being referenced  
✔ Especially useful when multiple namespaces contain classes with the same name

Without a fully qualified name, the compiler might not know which class to use.

### **4. Method Execution**

emp.Display();

* Calls the Display() method of the HR Employee class
* Outputs:

Employee from HR Department

This confirms correct namespace resolution and execution.

## **5. Scalability Advantage**

Nested namespaces allow you to expand the application **without breaking existing code**.

Example:

namespace Company.Project.Finance

{

class Employee

{

public void Display()

{

Console.WriteLine("Employee from Finance Department");

}

}

}

✔ HR and Finance modules remain independent  
✔ No naming conflict occurs  
✔ Existing HR code continues to work without modification  
✔ New departments can be added seamlessly

## **Why This Implementation Is Powerful**

* Encourages **modular design**
* Supports **enterprise scalability**
* Improves **code maintainability**
* Enhances **team collaboration**
* Aligns with **industry coding standards**

## **Real-World Analogy**

Just like a company has multiple departments under one organization, nested namespaces allow an application to have multiple **functional areas under a single project**, each cleanly separated and easy to manage.

## **Use Cases**

### **1. Department Separation**

Company.Project.HR

Company.Project.Finance

Company.Project.Sales

Each department remains independent and manageable.

### **2. Layered Architecture**

MyApp.Controllers

MyApp.Services

MyApp.Repositories

MyApp.Models

Encourages clean separation of concerns.

### **3. Library / Module Organization**

Utilities.Logging

Utilities.Security

Utilities.Validation

Makes reusable components easy to maintain.

### **4. Framework Development**

Framework.Core

Framework.UI

Framework.Network

Ensures extensibility and modularity.

## **Advantages of Nested Namespaces**

✔ Cleaner project structure  
✔ Reduced naming conflicts  
✔ Easy maintenance  
✔ Better collaboration  
✔ Professional coding standards  
✔ Industry best practice

**8.Using Alias, Static Imports**

# **8.1. Using Alias in C#**

## A **Using Alias** in C# allows a developer to create a **short, meaningful alternate name (alias)** for a **namespace or a specific type**. This alias can then be used throughout the code instead of repeatedly writing long or complex namespace names.

## Using alias is especially useful when:

## Namespace names are lengthy

## Multiple libraries contain classes with the same name

## Code readability and maintainability are priorities

## In simple terms, a **using alias acts as a shortcut** that makes code **cleaner, clearer, and easier to manage**, without changing the underlying functionality.

## **Syntax**

## using AliasName = NamespaceName;

### **Example**

## using Col = System.Collections.Generic;

## Here:

## Col is the alias

## System.Collections.Generic is the original namespace

## **Why Is Using Alias Needed?**

## Using alias is needed for several important reasons in professional C# development:

### **1. Improves Readability**

## Long namespace names can make code difficult to read:

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

## With alias:

## Col.List<int> list = new Col.List<int>();

## Code becomes shorter and more readable.

### **2. Reduces Repetition and Errors**

## Repeated typing of long namespaces:

## Increases chances of spelling mistakes

## Makes code verbose and cluttered

## Using alias minimizes repetition and improves coding efficiency.

### **3. Resolves Naming Conflicts**

## Different libraries may contain classes with the **same name**, such as:

## ProjectA.File

## ProjectB.File

## Without alias, the compiler cannot determine which class to use.

### **4. Essential for Large and Enterprise Projects**

## In enterprise applications:

## Many third-party libraries are used

## Codebases are large

## Multiple teams work together

## Using alias ensures **clarity, consistency, and maintainability**.

## **When Is It Applied?**

## Using alias is applied when:

## Namespace names are long or deeply nested

## Multiple namespaces contain identically named classes

## Code clarity and maintainability are important

## Working on large, modular, or enterprise-level applications

## Integrating multiple external libraries or frameworks

## **Where Is It Applied?**

## Using alias is commonly applied in:

## Enterprise applications

## Web APIs

## Desktop applications

## Library and framework development

## Large team-based projects

## Microservices and modular systems

## **How Is It Implemented**

## using System;

## using Col = System.Collections.Generic;

## 

## class Program

## {

## static void Main()

## {

## Col.List<int> numbers = new Col.List<int>();

## 

## numbers.Add(10);

## numbers.Add(20);

## numbers.Add(30);

## 

## foreach (int n in numbers)

## {

## Console.WriteLine(n);

## }

## }

## }

## **Detailed Code Breakdown**

### **1. Alias Declaration**

## using Col = System.Collections.Generic;

## Creates a short alias Col

## Replaces the long namespace System.Collections.Generic

### **2. Using the Alias**

## Col.List<int> numbers = new Col.List<int>();

## Col.List<int> replaces System.Collections.Generic.List<int>

## No change in behavior or performance

## Only improves readability and clarity

## **Resolving Ambiguity Using Alias**

### **Problem**

## Two namespaces contain a class with the same name:

## ProjectA.File

## ProjectB.File

## Using both without alias causes ambiguity.

## **Solution Using Alias**

## using FileA = ProjectA.File;

## using FileB = ProjectB.File;

## 

## class Program

## {

## static void Main()

## {

## FileA.Read();

## FileB.Read();

## }

## }

## **Explanation**

## FileA and FileB clearly distinguish the classes

## Compiler knows exactly which File class to use

## Prevents compile-time and runtime confusion

## **Advantages of Using Alias**

## ✔ Cleaner and shorter code ✔ Resolves naming conflicts ✔ Improves readability ✔ Reduces repetition ✔ Enhances maintainability ✔ Ideal for enterprise-level applications

## **Use Cases of Using Alias**

## Shortening long or complex namespaces

## Resolving class name conflicts

## Improving code clarity

## Managing large and complex projects

## Integrating multiple libraries smoothly

**8.2. Static Imports**

A **Static Import** in C# allows direct access to the **static members (methods, properties, constants)** of a class **without using the class name**. Once a static class is imported, its static members can be used as if they are defined in the current scope. This feature was introduced to reduce verbosity and make code cleaner and more readable, especially when static members are used frequently.

**Syntax**

using static ClassName;

**Example**

using static System.Math;

**Why Is Static Import Needed?**Static imports are needed because repeatedly writing class names for static members makes code lengthy and cluttered. In applications involving heavy calculations or utility methods, static imports reduce redundancy, improve readability, and make the logic easier to follow. They also help developers focus on the business logic rather than repetitive syntax.

**When Is It Applied?** Static imports are applied when:

* Static methods or constants are used frequently
* Code involves mathematical or scientific calculations
* Utility or helper classes are heavily used
* Cleaner and concise syntax is required

**Where Is It Applied?** Static imports are commonly used in:

* Mathematical and scientific applications
* Financial and accounting systems
* Game development logic
* Simulation software
* Utility-heavy enterprise applications

**How Is It Implemented**

using System;

using static System.Math;

class Program

{

static void Main()

{

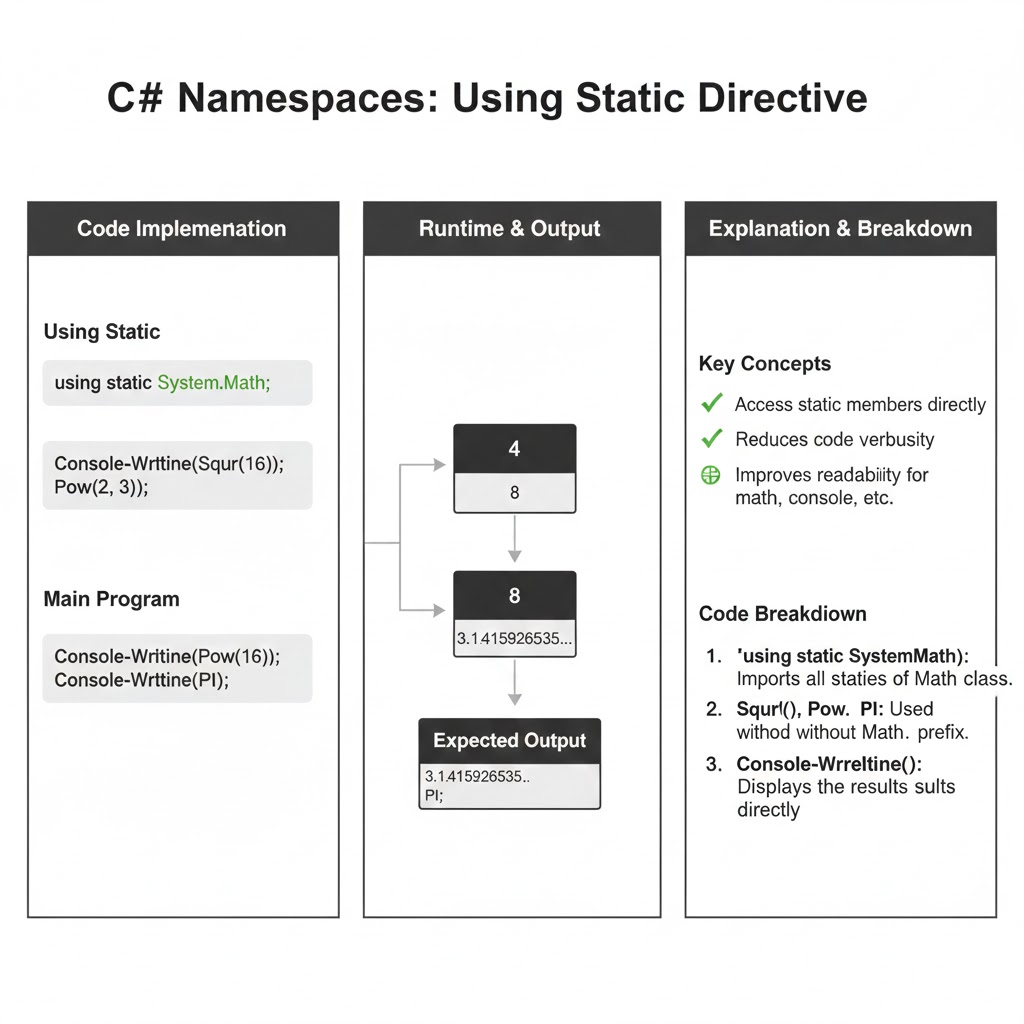
Console.WriteLine(Sqrt(16));

Console.WriteLine(Pow(2, 3));

Console.WriteLine(PI);

}

}

**Code Breakdown**

* using static System.Math imports all static members of the Math class
* Sqrt(16) is used instead of Math.Sqrt(16)
* Pow(2,3) is used instead of Math.Pow(2,3)
* PI is accessed directly without Math.PI
* Code becomes shorter and more readable without affecting functionality

**Without Static Import (Comparison)**

Console.WriteLine(Math.Sqrt(16));

Console.WriteLine(Math.Pow(2, 3));

Console.WriteLine(Math.PI);

This works correctly but is more repetitive and less clean when used multiple times.

**Use Cases of Static Imports**

* Mathematical calculations
* Accessing constants
* Utility and helper classes
* Cleaner business logic implementation

**Advantages**

* Reduces repetitive code
* Improves readability
* Makes logic concise and expressive

**Resolving Ambiguity Using Alias in Import Statements**

Ambiguity occurs when **two or more namespaces contain types with the same name** and are imported into the same file. In such cases, the compiler cannot determine which type to use. **Using alias in import (using) statements** resolves this ambiguity by assigning **distinct, meaningful names** to each conflicting type or namespace.

**Problem Scenario (Ambiguous Imports)**Assume two namespaces both define a class named File.

ProjectA.IO.File

ProjectB.IO.File

If both namespaces are imported normally, the compiler raises an ambiguity error.

using ProjectA.IO;

using ProjectB.IO;

class Program

{

static void Main()

{

File.Read(); // Ambiguous reference

}

}

**Solution: Using Alias in Import Statements**

using System;

using AFile = ProjectA.IO.File;

using BFile = ProjectB.IO.File;

namespace ProjectA.IO

{

class File

{

public static void Read()

{

Console.WriteLine("Reading from ProjectA File");

}

}

}

namespace ProjectB.IO

{

class File

{

public static void Read()

{

Console.WriteLine("Reading from ProjectB File");

}

}

}

class Program

{

static void Main()

{

AFile.Read();

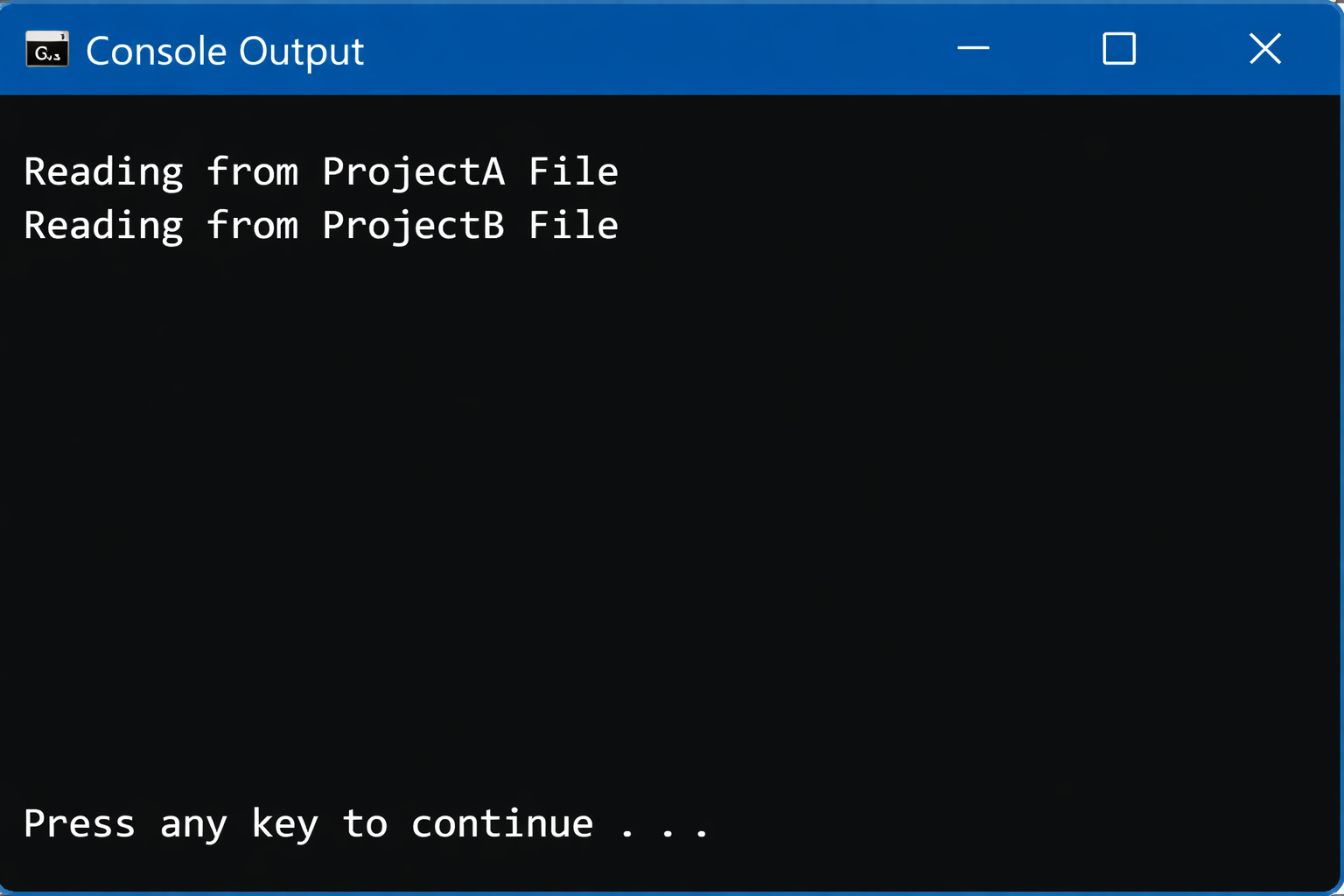
BFile.Read();

}

}

**Explanation**

* AFile and BFile are aliases defined directly in the **import (using) statements**
* Each alias uniquely identifies a specific File class
* The compiler clearly knows which class is being referenced
* Eliminates both compile-time and runtime ambiguity



# **9. Partial Classes & Static Classes in C#**

C# provides **Partial Classes** and **Static Classes** to solve different design and architectural problems.

* **Partial classes** help manage **large or auto-generated code** by splitting a class across multiple files.
* **Static classes** provide **utility or helper functionality** without allowing object creation.

Both are widely used in **enterprise, framework, and real-world applications**.

# **9.1. Partial Class**

**A Partial Class allows a single class to be split across multiple files, while the compiler treats all parts as one single class at compile time**  
This means developers can organize large or complex classes into logical sections without changing how the class behaves at runtime.**All partial class definitions must use the partial keyword, have the same class name, and exist within the same namespace.** Partial classes improve code readability, maintainability, and teamwork by allowing different developers or tools (such as code generators) to work on separate parts of the same class without causing conflicts.

All parts must:

* Use the partial keyword
* Have the same class name
* Belong to the same namespace

## **Syntax**

partial class ClassName

{

}

## **Why Is It Needed?**

Partial classes are needed because:

* Large classes become difficult to manage in one file
* Auto-generated code should not be modified manually
* Separation of concerns improves maintainability
* Multiple developers can work on the same class safely

## **When Is It Applied?**

Partial classes are applied when:

* Classes are very large
* Code is auto-generated (Designer, ORM, API clients)
* Team-based development is used
* Clean separation of logic is required

## **Where Is It Applied?**

Partial classes are commonly used in:

* Windows Forms (Designer files)
* ASP.NET Web Forms and MVC
* Entity Framework models
* Large enterprise applications
* Code generation tools

## **How Is It Implemented**

### **Employee\_Part1.cs**

namespace Company

{

partial class Employee

{

public void Work()

{

Console.WriteLine("Employee is working");

}

}

}

### **Employee\_Part2.cs**

namespace Company

{

partial class Employee

{

public void GetSalary()

{

Console.WriteLine("Employee salary processed");

}

}

}

### **Program.cs**

using System;

using Company;

class Program

{

static void Main()

{

Employee emp = new Employee();

emp.Work();

emp.GetSalary();

}

}

## 

## **Code Breakdown**

## **Use of the partial Keyword** The partial keyword explicitly tells the C# compiler that the Employee class definition is incomplete in this file and that other parts of the same class may exist elsewhere in the project.

## **Same Class Name and Namespace** Both Employee\_Part1.cs and Employee\_Part2.cs define a class named Employee inside the same Company namespace. This is mandatory—partial classes must share:

## The same class name

## The same namespace

## Compatible access modifiers

## Logical Separation of Responsibilities Each file focuses on a specific responsibility:

## Employee\_Part1.cs handles work-related behavior

## Employee\_Part2.cs handles salary-related behavior

## This improves readability and keeps files smaller and easier to manage.

## **Compile-Time Merging** At compile time, the C# compiler automatically combines all partial class definitions into a single Employee class. No extra runtime cost or special handling is required.

## **Normal Object Creation and Usage** In Program.cs, the Employee object is created just like any other class. Even though methods are defined across different files, the object can access all of them seamlessly.

## **Encapsulation Is Preserved** From the outside, Employee behaves as one complete class. The fact that it is split across files is completely hidden from the consumer of the class.

## **Use Cases of Partial Classes**

* UI designer-generated code
* Separating business logic from UI logic
* Large domain models
* Team collaboration
* Cleaner file organization

# **9.2. Static Class**

A **Static Class** is a special type of class in C# designed to **group related utility or helper methods** that do not depend on object state. It represents behavior rather than data and is commonly used where creating objects is unnecessary or even undesirable.

A static class **cannot be instantiated**, meaning you cannot create an object using the new keyword. This restriction enforces the idea that the class is meant to be accessed directly through its class name.

A static class **contains only static members**, such as static methods, properties, and fields. All members belong to the class itself rather than to any object, ensuring a single shared implementation throughout the application.

Static classes are typically used to **provide common functionality** that is reused across the application, such as mathematical calculations, validation logic, logging helpers, configuration access, or string utilities.

**Key characteristics :**

• No object creation  
• Implicitly sealed  
• Global accessibility

• No instance state  
• Thread-safe design (when stateless)

## **Syntax**

static class ClassName

{

static members

}

## **Why Is It Needed?**

Static classes are needed because:

* Some functionality does not require object state
* Prevents unnecessary object creation
* Improves performance
* Clearly represents utility behavior

## **When Is It Applied?**

Static classes are applied when:

* Only helper methods are needed
* Shared logic is required across application
* No instance data is involved
* Constants or configuration helpers are used

## **Where Is It Applied?**

Static classes are used in:

* Utility/helper libraries
* Logging helpers
* Validation helpers
* Mathematical calculations
* Configuration managers

## **How Is It Implemented**

using System;

static class Utility

{

public static void Helper()

{

Console.WriteLine("Utility helper method executed");

}

}

class Program

{

static void Main()

{

Utility.Helper();

}

}

## 

## **Code Breakdown**

#### **1. static class Utility**

* The static keyword makes the class:
  + **Non-instantiable** (you cannot write new Utility())
  + **Implicitly sealed** (cannot be inherited)
* The compiler enforces these rules at compile time, preventing misuse.
* This ensures the class is used only as a **function provider**, not as a data holder.

#### **2. public static void Helper()**

* All members inside a static class **must also be static**
* This method:
  + Belongs to the class itself, not to any object
  + Has a single shared implementation throughout the application
* There is **no instance state**, which improves memory efficiency and predictability.

#### **3. Console.WriteLine(...)**

* Demonstrates that static methods can:
  + Perform actions
  + Access other static members
* Common real-world examples include:
  + Math.Sqrt()
  + Convert.ToInt32()
  + File.ReadAllText()

#### **4. Utility.Helper();**

* The method is accessed directly using the **class name**
* No object creation is required
* This makes the code: Cleaner, Faster (no constructor call)

### **Why Object Creation Is Not Required**

* Static classes represent **behavior**, not **entities**
* Since there is no instance:
  + No memory is allocated per object
  + No constructor is executed
  + No object lifecycle to manage

### **What the Compiler Enforces Automatically**

If you try any of the following, the compiler will throw an error:

Utility u = new Utility(); // Not allowed

class MyUtility : Utility {} // Not allowed

This ensures:

* Consistent usage
* Better architectural discipline

## **Use Cases of Static Classes**

* Helper methods
* Common validations
* Mathematical utilities
* Constants storage
* Logging utilities

## **Partial Class vs Normal Class vs Static Class**

| **Feature** | **Partial Class** | **Normal Class** | **Static Class** |
| --- | --- | --- | --- |
| Can Split Across Files | Yes | No | No |
| Instantiation | Yes | Yes | No |
| Contains Instance Members | Yes | Yes | No |
| Contains Static Members | Yes | Yes | Yes |
| Used for Code Generation | Yes | No | No |
| Used as Utility | No | Sometimes | Yes |

## **Key Differences Summary**

* **Partial Class** → Code organization and scalability
* **Static Class** → Utility and helper functionality
* **Normal Class** → Object-oriented behavior

## **10. Enumerations (Enums)**

An **Enumeration (enum)** **is a special value type in C# that defines a set of named constant values.** It is used to represent a fixed collection of related values in a **type-safe**, readable, and meaningful way instead of using raw numbers or strings.

In simple terms, enums replace **magic numbers** with **clear, self-descriptive names**, improving code clarity and safety.

### **Syntax**

enum EnumName

{

Value1,

Value2,

Value3

}

With explicit values:

enum Status

{

Pending = 1,

Approved = 2,

Rejected = 3

}

### **Why Is It Needed**

Enums are needed because:

• Avoid use of hard-coded numbers and strings  
• Improve readability and self-documentation  
• Provide compile-time type safety  
• Reduce logical and comparison errors  
• Make code easier to maintain and modify  
• Represent fixed sets of values clearly

Without enums:

if(status == 1) { }

With enums:

if(status == Status.Approved) { }

The second approach is **clear, safe, and maintainable**.

### **When Is It Applied**

Enums are applied when:

• A variable can have only a limited set of values  
• Representing states, types, or categories  
• Replacing flags, modes, or statuses  
• Improving decision-making logic  
• Designing APIs and domain models

### **Where Is It Applied**

Enums are commonly used in:

• Banking and finance systems  
• Web and REST APIs  
• Game development  
• Operating systems and drivers  
• Enterprise applications  
• UI state management  
• E-commerce platforms

### **How Is It Implemented**

using System;

enum OrderStatus

{

Pending,

Shipped,

Delivered,

Cancelled

}

class Program

{

static void Main()

{

OrderStatus status = OrderStatus.Shipped;

Console.WriteLine("Order Status: " + status);

if (status == OrderStatus.Shipped)

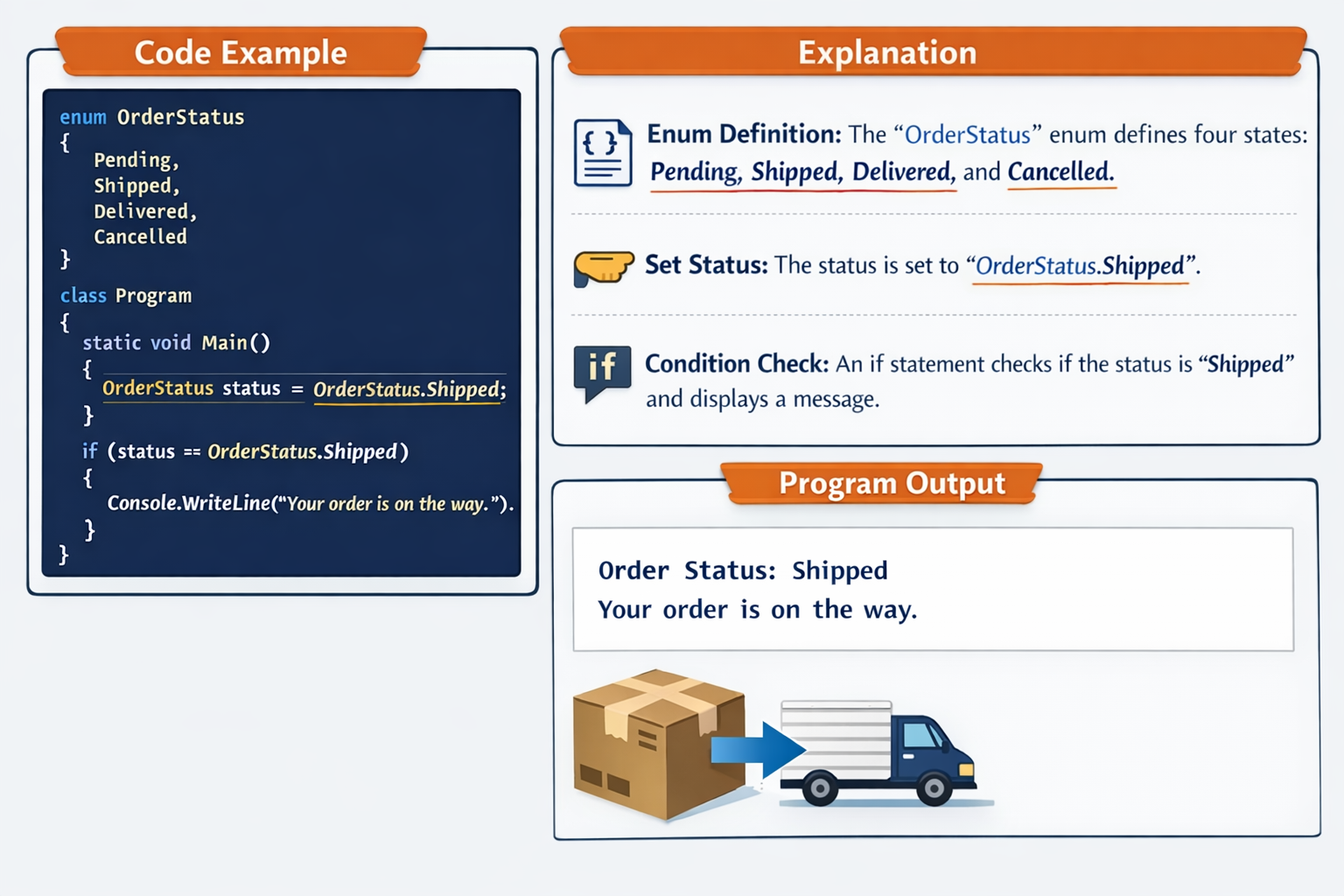
{

Console.WriteLine("Your order is on the way.");

}

}

}



### **Use Cases**

• Order processing states  
 • User roles (Admin, User, Guest)  
 • Payment types (Cash, Card, UPI)  
 • Game states (Start, Pause, End)  
 • Traffic light signals  
 • Days of the week

### **Code Breakdown**

#### **1. enum OrderStatus**

Defines a fixed set of allowed values  
Compiler assigns values starting from 0 by default

Pending = 0

Shipped = 1

Delivered = 2

Cancelled = 3

#### **2. OrderStatus status = OrderStatus.Shipped;**

Creates a variable of enum type  
Only valid enum values are allowed

#### **3. Enum in Condition**

if (status == OrderStatus.Shipped)

• Clear comparison  
 • No risk of invalid values  
 • Compile-time checking

### **Enum with Explicit Values**

enum Priority

{

Low = 1,

Medium = 2,

High = 3

}

Useful when mapping to:  
 • Database values  
 • API response codes

### **Enum with Switch Statement**

switch (status)

{

case OrderStatus.Pending:

Console.WriteLine("Order pending");

break;

case OrderStatus.Shipped:

Console.WriteLine("Order shipped");

break;

case OrderStatus.Delivered:

Console.WriteLine("Order delivered");

break;

}

### **Comparison with Other Similar Constructs**

| **Feature** | **Enum** | **Constant** | **String** | **Integer** |
| --- | --- | --- | --- | --- |
| Type Safety | Yes | Partial | No | No |
| Readability | High | Medium | Medium | Low |
| Compile-Time Checking | Yes | Yes | No | No |
| Fixed Values | Yes | Yes | No | No |
| Maintainability | High | Medium | Low | Low |

### **Enum vs Constant**

• Enums group related constants  
 • Constants do not enforce grouping  
 • Enums prevent invalid assignments

### **Enum vs String**

• Strings allow invalid values  
 • Enums restrict to predefined options  
 • Enums are faster and safer

### **Enum vs Integer**

• Integers are unclear and error-prone  
 • Enums are expressive and meaningful

# **C# 1-Hour Assignment:**

## **Scenario (Use Case)**

You are developing a **Library Management System** that handles **Books, eBooks, and Magazines**, where you need to manage borrowing, returning, and calculating fines. The system also includes **user notifications**, **library analytics**, and **role-based access control**.

This scenario allows you to use all the above concepts effectively.

## **Assignment Tasks**

### **Task 1: Abstract Classes & Methods**

**Requirement:**Create an abstract class LibraryItem representing any item in the library.

abstract class LibraryItem

{

public string Title { get; set; }

public string Author { get; set; }

public int ItemID { get; set; }

public abstract void DisplayItemDetails();

public abstract double CalculateLateFee(int daysLate);

}

**Task:**

* Implement two derived classes: Book and Magazine.
* Override the abstract methods with proper logic:
  + Book: $1 per day late fee.
  + Magazine: $0.5 per day late fee.

**Expected Output:**Display details of a Book and a Magazine along with late fee for 3 days.

### **Task 2: Interfaces & Multiple Inheritance**

**Requirement:**Create interfaces for additional functionalities.

interface IReservable

{

void ReserveItem();

}

interface INotifiable

{

void SendNotification(string message);

}

**Task:**

* Implement Book as IReservable and INotifiable.
* Show how multiple interfaces can be implemented in a single class.

### **Task 3: Dynamic Polymorphism**

**Requirement:**Use **polymorphism** to handle all library items dynamically.

List<LibraryItem> libraryItems = new List<LibraryItem>();

libraryItems.Add(new Book { Title = "C# Fundamentals", Author = "John Doe", ItemID = 101 });

libraryItems.Add(new Magazine { Title = "Tech Today", Author = "Jane Doe", ItemID = 201 });

foreach (LibraryItem item in libraryItems)

{

item.DisplayItemDetails();

}

**Task:**

* Ensure correct method overrides are invoked at runtime.

### **Task 4: Explicit Interface Implementation**

**Requirement:** Implement **explicit interface methods** for Book.

class Book : LibraryItem, IReservable, INotifiable

{

public override void DisplayItemDetails() { /\* implementation \*/ }

public override double CalculateLateFee(int daysLate) { /\* implementation \*/ }

void IReservable.ReserveItem() { Console.WriteLine("Book reserved successfully."); }

void INotifiable.SendNotification(string message) { Console.WriteLine($"Notification: {message}"); }

}

**Task:**

* Demonstrate calling explicit methods using interface reference.

### **Task 5: Namespaces & Nested Namespaces**

**Requirement:**Organize your project:

namespace LibrarySystem

{

namespace Items

{

class Book { /\*...\*/ }

class Magazine { /\*...\*/ }

}

namespace Users

{

class Member { /\*...\*/ }

}

}

**Task:**

* Use using alias for nested namespaces:

using LibraryItems = LibrarySystem.Items;

### **Task 6: Partial & Static Classes**

**Requirement:**Create a partial class LibraryAnalytics:

partial class LibraryAnalytics

{

public static int TotalItemsBorrowed { get; set; }

}

partial class LibraryAnalytics

{

public static void ShowAnalytics() => Console.WriteLine($"Total Items Borrowed: {TotalItemsBorrowed}");

}

**Task:**

* Demonstrate using static properties/methods to keep track of borrowing.

### **Task 7: Enumerations**

**Requirement:**Create enums for **UserRole** and **ItemStatus**:

enum UserRole { Admin, Librarian, Member }

enum ItemStatus { Available, Borrowed, Reserved, Lost }

**Task:**

* Assign roles and statuses in your objects and display them.

## **Bonus Challenge**

* Implement a **notification system** that sends messages differently based on user role using interfaces.
* Extend your LibraryItem class to handle **eBooks** with a digital download method.

## **Complete Code:**

using System;

using System.Collections.Generic;

// ENUMERATIONS

enum UserRole { Admin, Librarian, Member }

enum ItemStatus { Available, Borrowed, Reserved, Lost }

// NAMESPACES

namespace LibrarySystem

{

namespace Items

{

// ABSTRACT CLASS

abstract class LibraryItem

{

public string Title { get; set; }

public string Author { get; set; }

public int ItemID { get; set; }

public ItemStatus Status { get; set; }

public abstract void DisplayItemDetails();

public abstract double CalculateLateFee(int daysLate);

}

// INTERFACES

interface IReservable

{

void ReserveItem();

}

interface INotifiable

{

void SendNotification(string message);

}

// BOOK CLASS IMPLEMENTING MULTIPLE INTERFACES

class Book : LibraryItem, IReservable, INotifiable

{

public override void DisplayItemDetails()

{

Console.WriteLine($"Book: {Title}, Author: {Author}, ID: {ItemID}, Status: {Status}");

}

public override double CalculateLateFee(int daysLate)

{

return daysLate \* 1.0; // $1 per day

}

// EXPLICIT INTERFACE IMPLEMENTATION

void IReservable.ReserveItem()

{

Status = ItemStatus.Reserved;

Console.WriteLine($"Book '{Title}' reserved successfully.");

}

void INotifiable.SendNotification(string message)

{

Console.WriteLine($"Notification for Book '{Title}': {message}");

}

}

// MAGAZINE CLASS

class Magazine : LibraryItem

{

public override void DisplayItemDetails()

{

Console.WriteLine($"Magazine: {Title}, Author: {Author}, ID: {ItemID}, Status: {Status}");

}

public override double CalculateLateFee(int daysLate)

{

return daysLate \* 0.5; // $0.5 per day

}

}

}

namespace Analytics

{

// PARTIAL & STATIC CLASS

partial class LibraryAnalytics

{

public static int TotalItemsBorrowed { get; set; }

}

partial class LibraryAnalytics

{

public static void ShowAnalytics()

{

Console.WriteLine($"Total Items Borrowed: {TotalItemsBorrowed}");

}

}

}

namespace Users

{

class Member

{

public string Name { get; set; }

public UserRole Role { get; set; }

public void DisplayRole()

{

Console.WriteLine($"{Name} has role: {Role}");

}

}

}

}

// USING ALIAS

using LibraryItems = LibrarySystem.Items;

using LibraryAnalytics = LibrarySystem.Analytics.LibraryAnalytics;

using LibrarySystem.Users;

class Program

{

static void Main()

{

// CREATE ITEMS

LibraryItems.Book book1 = new LibraryItems.Book

{

Title = "C# Fundamentals",

Author = "John Doe",

ItemID = 101,

Status = ItemStatus.Available

};

LibraryItems.Magazine mag1 = new LibraryItems.Magazine

{

Title = "Tech Today",

Author = "Jane Doe",

ItemID = 201,

Status = ItemStatus.Available

};

// LIST OF ITEMS (POLYMORPHISM)

List<LibraryItems.LibraryItem> libraryItems = new List<LibraryItems.LibraryItem> { book1, mag1 };

Console.WriteLine("=== LIBRARY ITEMS ===");

foreach (var item in libraryItems)

{

item.DisplayItemDetails();

Console.WriteLine($"Late Fee for 3 days: ${item.CalculateLateFee(3)}\n");

}

// EXPLICIT INTERFACE CALLS

IReservable reservableBook = book1;

INotifiable notifiableBook = book1;

reservableBook.ReserveItem();

notifiableBook.SendNotification("Your reserved book is ready for pickup.\n");

// CREATE USERS

Member member1 = new Member { Name = "Alice", Role = UserRole.Member };

Member librarian1 = new Member { Name = "Bob", Role = UserRole.Librarian };

member1.DisplayRole();

librarian1.DisplayRole();

Console.WriteLine();

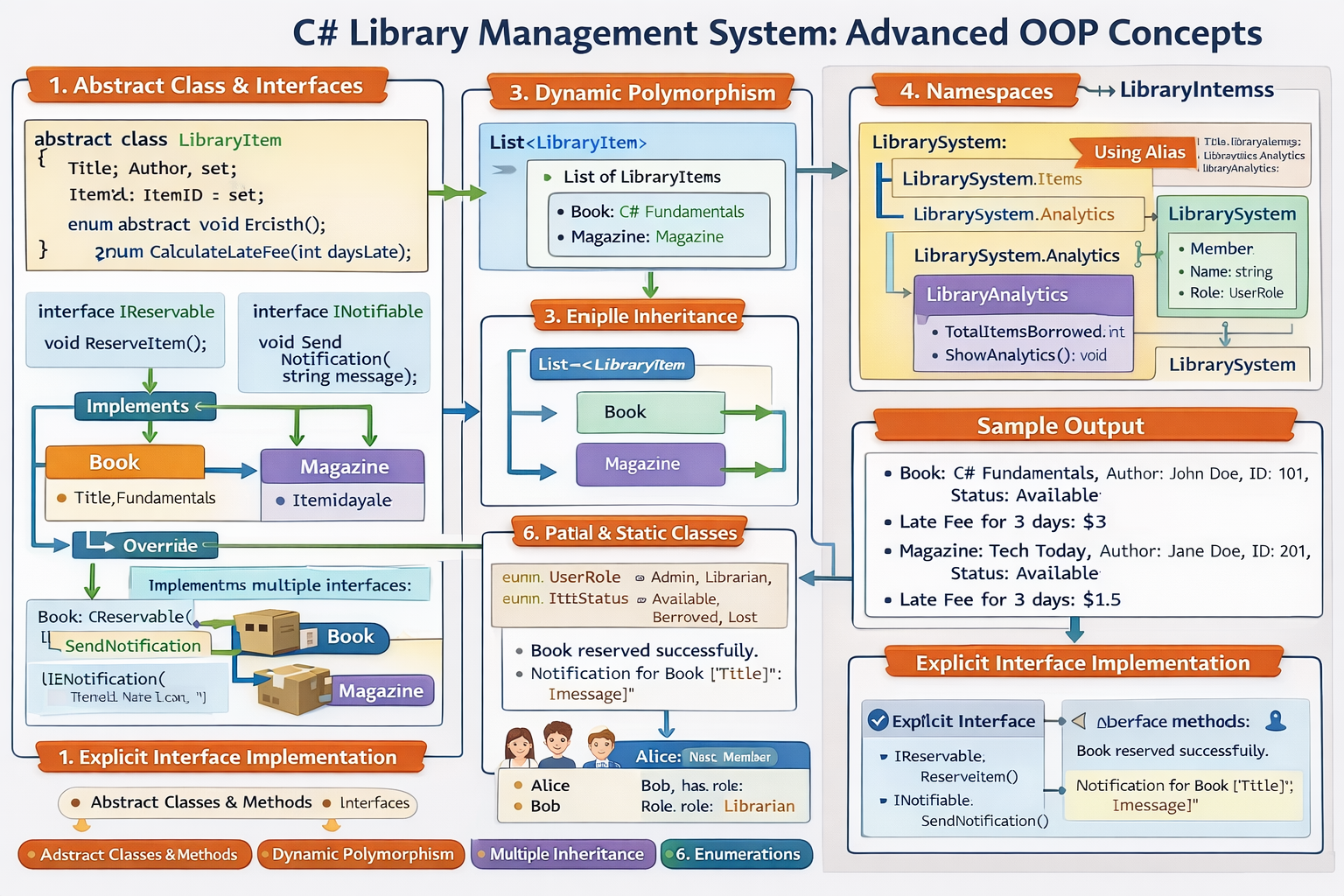
// UPDATE ANALYTICS

LibraryAnalytics.TotalItemsBorrowed = 5;

LibraryAnalytics.ShowAnalytics();

}

}



Interface vs Abstract

# **Interface vs Abstract Class in C# (Exact Technical Difference)**

## **1. Definition (C# Language Specific)**

### **Interface (C#)**

In C#, an interface is a reference type that defines a contract consisting of method, property, event, or indexer declarations without implementation.

A class that implements an interface must implement all its members.

### **Abstract Class (C#)**

In C#, an abstract class is a class that cannot be instantiated and can contain both abstract (unimplemented) and concrete (implemented) members.

It is designed to be inherited.

## **2. Method Implementation (Key Technical Difference)**

| **Feature** | **Interface (C#)** | **Abstract Class (C#)** |
| --- | --- | --- |
| Abstract methods | Implicitly abstract | Explicitly marked abstract |
| Method body | Not allowed (before C# 8) | Allowed |
| Concrete methods | Not allowed (traditional) | Allowed |

Important exam note:

* Interface methods are public by default
* Abstract class methods can be public, protected, internal

## **3. Fields and State Handling**

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| Instance fields | Not allowed | Allowed |
| State (data) | Cannot store state | Can store state |
| Constants | Allowed | Allowed |

Technical reason:

* Interfaces do not support instance variables
* Abstract classes support encapsulation

## **4. Inheritance Rules (Very Important in C#)**

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| Multiple inheritance | Supported | Not supported |
| Keyword used | implements (conceptually) | : (inherits) |
| Base types allowed | Multiple interfaces | Only one base class |

C# allows:

* One base class
* Multiple interfaces

## **5. Constructors**

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| Constructor allowed | No | Yes |
| Object initialization | Not possible | Possible |

Technical explanation:

* Interfaces do not create objects
* Abstract classes participate in object construction through derived classes

## **6. Access Modifiers**

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| Default access | public | No default |
| Modifier flexibility | Fixed | Flexible |

Exam tip:

Interface members must be public, abstract class members do not have to be public.

## **7. Usage and Design Intent (C# Design Principle)**

| **Aspect** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| Abstraction level | Full abstraction | Partial abstraction |
| Coupling | Loose | Tight |
| Reusability | Behavioral contract | Code + behavior reuse |
| Dependency Injection | Preferred | Secondary |

## **8. When to Use (C# Decision Rule)**

### **Use Interface in C# when:**

* Multiple inheritance is required
* You are defining a capability
* Classes are unrelated
* Dependency Injection is used
* API flexibility is important

### **Use Abstract Class in C# when:**

* Classes share common logic
* State must be maintained
* Partial implementation is needed
* You want to provide default behavior

## **9. One-Line Interview Answer (C#)**

In C#, an interface defines a contract without implementation and supports multiple inheritance, whereas an abstract class provides partial implementation, state, and supports single inheritance.

## **10. Common Interview Trap (C#)**

Question:

Why not replace abstract classes with interfaces everywhere?

Correct Answer:

Interfaces cannot hold state or constructors and cannot share implementation logic, which abstract classes can.

## **11. Final Memory Rule (C#)**

* Interface → What a class must do
* Abstract Class → What a class is + how it partly behaves
* Method → How work is actually done

Library Management System

# **Library Management System**

## **Scenario-Based Assignment (Strict Task Instructions)**

## **TASK 1: Abstract Class & Abstract Methods**

### **What the student MUST create**

1. **ONE abstract base class** named:
   * LibraryItem
2. The abstract class MUST contain:
   * **Exactly 3 properties**
     + Title
     + Author
     + ItemID
   * **Exactly 2 abstract methods**
     + One method to display item details
     + One method to calculate late fee
3. **TWO concrete child classes**:
   * Book
   * Magazine
4. Each child class MUST:
   * Inherit from LibraryItem
   * Override **both abstract methods**
5. Late fee rules (MANDATORY):
   * Book → 1 unit per day
   * Magazine → 0.5 units per day

### **What the student MUST show as proof**

* Create **1 Book object** and **1 Magazine object**
* Display:
  + Title
  + Author
  + ItemID
  + Late fee for **exactly 3 days**

## **TASK 2: Interfaces & Multiple Inheritance**

### **What the student MUST create**

1. **ONE interface** named:
   * IReservable
   * Must contain **one method** for reserving an item
2. **ONE interface** named:
   * INotifiable
   * Must contain **one method** that accepts a message
3. The Book class MUST:
   * Implement **both interfaces**
   * Implement **all interface methods**

### **What the student MUST show as proof**

* Call:
  + The reservation method
  + The notification method
* Output MUST clearly indicate:
  + Reservation success
  + Notification message sent

## **TASK 3: Dynamic Polymorphism**

### **What the student MUST create**

1. **ONE collection** that stores:
   * Objects of type LibraryItem
2. Add:
   * **1 Book**
   * **1 Magazine**
3. Use:
   * A loop that accesses items **only using the parent type**

### **What the student MUST show as proof**

* The correct DisplayItemDetails() method must execute:
  + Book version for Book
  + Magazine version for Magazine
* Student MUST write **1–2 lines explanation** stating:
  + “Method selection happens at runtime”

## **TASK 4: Explicit Interface Implementation**

### **What the student MUST create**

1. Modify Book class so that:
   * Interface methods are **explicitly implemented**
2. The class MUST NOT allow:
   * Direct calling of reservation or notification methods using Book object

### **What the student MUST show as proof**

* Call interface methods using:
  + IReservable reference
  + INotifiable reference
* Student MUST write **1 clear sentence** explaining:
  + Why direct access is restricted

## **TASK 5: Namespaces & Nested Namespaces**

### **What the student MUST create**

1. **ONE main namespace**:
   * LibrarySystem
2. Inside it, create **TWO nested namespaces**:
   * Items
   * Users
3. Place:
   * Book and Magazine inside Items
   * One user class (Member) inside Users
4. Create **ONE namespace alias**:
   * Alias LibrarySystem.Items

### **What the student MUST show as proof**

* Use alias to create Book and Magazine objects
* Write **2 points** explaining:
  + Why nested namespaces are useful

## **TASK 6: Partial & Static Classes**

### **What the student MUST create**

1. **ONE partial class**:
   * LibraryAnalytics
2. The class MUST be split into:
   * **Two separate parts**
3. The class MUST contain:
   * **ONE static property** to track total borrowed items
   * **ONE static method** to display analytics

### **What the student MUST show as proof**

* Increase borrowed count
* Display total borrowed items
* Write **1 line** explaining:
  + Why static members are used

## **TASK 7: Enumerations (Enums)**

### **What the student MUST create**

1. **ONE enum** named:
   * UserRole
     + Admin
     + Librarian
     + Member
2. **ONE enum** named:
   * ItemStatus
     + Available
     + Borrowed
     + Reserved
     + Lost
3. Assign:
   * A role to a user
   * A status to a library item

### **What the student MUST show as proof**

* Display:
  + User role
  + Item status
* Write **1 sentence** explaining:
  + Why enums are better than strings

## **BONUS TASK (HIGH WEIGHTAGE)**

### **What the student MUST create**

1. Modify notification logic so that:
   * Admin → system alert
   * Member → borrowing update
2. Add **ONE new item type**:
   * eBook
3. eBook MUST:
   * Inherit from LibraryItem
   * Have **one additional digital-specific behavior**

### **What the student MUST show as proof**

* Demonstrate:
  + Notification difference by role
  + eBook behavior execution

Below is the **EXPECTED OUTCOME / EXPECTED OUTPUT** for **EACH TASK**, written in a **clear, observable, examiner-friendly format**.

# **EXPECTED OUTCOME**

## **Library Management System Assignment**

## **TASK 1: Abstract Class & Abstract Methods**

### **Expected Outcome (What MUST appear)**

**Displayed Output:**

Item Type: Book

Title: C# Fundamentals

Author: John Doe

Item ID: 101

Late Fee for 3 days: 3.0

Item Type: Magazine

Title: Tech Today

Author: Jane Doe

Item ID: 201

Late Fee for 3 days: 1.5

**Concept Confirmation:**

* Book late fee = 3 days × 1 = 3.0
* Magazine late fee = 3 days × 0.5 = 1.5

**What this proves:**

* Abstract methods are correctly overridden
* Different logic is applied based on item type

## **TASK 2: Interfaces & Multiple Inheritance**

### **Expected Outcome**

**Displayed Output:**

Book reserved successfully.

Notification sent: Your reserved book is ready for pickup.

**What this proves:**

* One class implements **multiple interfaces**
* Optional behaviors are added without inheritance misuse

## **TASK 3: Dynamic Polymorphism**

### **Expected Outcome**

**Displayed Output (from loop):**

Item Type: Book

Title: C# Fundamentals

Author: John Doe

Item ID: 101

Item Type: Magazine

Title: Tech Today

Author: Jane Doe

Item ID: 201

**Mandatory Student Explanation (1–2 lines):**

The method executed depends on the object type at runtime, not the reference type.

**What this proves:**

* Runtime polymorphism is working
* Correct method binding occurs dynamically

## **TASK 4: Explicit Interface Implementation**

### **Expected Outcome**

**Displayed Output:**

Book reserved successfully.

Notification: Please return the book on time.

**How it must be called (conceptually):**

* Through IReservable reference
* Through INotifiable reference

**Mandatory Explanation (1 sentence):**

Explicit implementation prevents direct access and exposes functionality only through interfaces.

**What this proves:**

* Controlled access
* Proper use of explicit interface implementation

## **TASK 5: Namespaces & Nested Namespaces**

### **Expected Outcome**

**Successful Object Creation Using Alias**

Book and Magazine objects created using namespace alias.

**Mandatory Explanation (2 points):**

1. Nested namespaces organize large projects logically.

2. Aliases reduce long namespace references and improve readability.

**What this proves:**

* Correct namespace hierarchy
* Professional project organization

## **TASK 6: Partial & Static Classes**

### **Expected Outcome**

**Displayed Output:**

Total Items Borrowed: 5

**Mandatory Explanation (1 line):**

Static members store system-wide data shared across all objects.

**What this proves:**

* Static data persists globally
* Partial classes are merged into one logical unit

## **TASK 7: Enumerations (Enums)**

### **Expected Outcome**

**Displayed Output:**

User Role: Member

Item Status: Borrowed

**Mandatory Explanation (1 sentence):**

Enums prevent invalid values and improve code readability.

**What this proves:**

* Controlled value sets
* Safer and cleaner logic

## **BONUS TASK: Advanced Design**

### **Expected Outcome**

**Notification Based on Role:**

Admin Alert: System maintenance scheduled.

Member Notification: Your borrowed item is due tomorrow.

**eBook Behavior Output:**

eBook downloaded successfully.

**What this proves:**

* Role-based behavior variation
* Digital item extension without breaking design
* Scalable and extensible architecture

## **FINAL EVALUATION CHECKLIST (For Faculty)**

| **Concept** | **Verified By** |
| --- | --- |
| Abstraction | Different late fee outputs |
| Interfaces | Reservation + Notification |
| Polymorphism | Correct runtime method calls |
| Explicit Interface | Interface-only access |
| Namespaces | Alias usage |
| Static | Shared analytics |
| Enums | Controlled roles/status |
| Design Quality | Bonus task |