**🎯 Full Path to Master OOP in C#**

**✅ 1. Foundations of OOP in C#**

**🔹 1.1 Basic Concepts**

* What is OOP?
* OOP vs Procedural Programming
* Four pillars of OOP
  + Encapsulation
  + Inheritance
  + Polymorphism
  + Abstraction

**✅ 2. Classes and Objects**

**🔹 2.1 Creating and Using Classes**

* Class definition
* Creating objects
* Object lifecycle

**🔹 2.2 Fields, Methods, and Properties**

* Instance vs static members
* Readonly, const, static
* Auto-implemented properties
* Property with logic (getter/setter)

**🔹 2.3 Access Modifiers**

* public
* private
* protected
* internal
* protected internal
* private protected

**✅ 3. Constructors in C#**

**🔹 3.1 Types of Constructors**

* Default constructor
* Parameterized constructor
* Constructor overloading
* Static constructor
* Private constructor

**🔹 3.2 Constructor Chaining**

* this keyword
* base keyword

**✅ 4. Encapsulation**

* Hiding fields using private
* Exposing via public properties
* Accessor methods (get/set)
* Benefits of encapsulation
* Practical examples

**✅ 5. Inheritance**

* Base and derived class
* Inheritance hierarchy
* The base keyword
* Constructors in inheritance
* Sealed classes and sealed methods
* When to avoid inheritance (prefer composition)

**✅ 6. Polymorphism**

**🔹 6.1 Compile-Time Polymorphism**

* Method overloading
* Operator overloading

**🔹 6.2 Runtime Polymorphism**

* Method overriding
* Virtual and override keywords
* new keyword (hiding methods)
* Polymorphic behavior with base reference

**✅ 7. Abstraction**

* Abstract classes and abstract methods
* Interfaces
* Interface vs Abstract class
* Multiple interface implementation
* Real-world examples: Payment System, Logger

**✅ 8. Advanced Concepts**

**🔹 8.1 Object Class**

* ToString()
* Equals(), GetHashCode()
* GetType()

**🔹 8.2 Boxing and Unboxing**

* Value types vs reference types
* Object as base class of all types

**🔹 8.3 Static Classes and Static Members**

* When to use static
* Singleton pattern using static

**✅ 9. SOLID Principles in C#**

**🔹 9.1 Single Responsibility Principle**

**🔹 9.2 Open/Closed Principle**

**🔹 9.3 Liskov Substitution Principle**

**🔹 9.4 Interface Segregation Principle**

**🔹 9.5 Dependency Inversion Principle**

**✅ 10. Design Patterns for OOP**

**🔹 10.1 Creational Patterns**

* Singleton
* Factory Method
* Abstract Factory
* Builder

**🔹 10.2 Structural Patterns**

* Adapter
* Decorator
* Facade

**🔹 10.3 Behavioral Patterns**

* Strategy
* Observer
* Command
* Template Method

**✅ 11. Principles and Best Practices**

* Favor composition over inheritance
* DRY (Don't Repeat Yourself)
* YAGNI (You Aren’t Gonna Need It)
* KISS (Keep It Simple, Stupid)
* LOD (Law of Demeter)

**✅ 12. Real-World Application**

**🔹 12.1 Layered Architecture using OOP**

* Presentation Layer
* Business Logic Layer (BLL)
* Data Access Layer (DAL)
* Models Layer
* Interfaces Layer

**✅ 13. Modern C# and OOP Features**

* Records vs Classes
* init-only setters
* Nullable Reference Types
* Pattern Matching in OOP
* Target-typed new expressions
* File-scoped namespace declarations

**✅ 14. Unit Testing OOP Code**

* xUnit / NUnit / MSTest
* Writing testable OOP code
* Mocking with Moq
* Dependency Injection for testing

**✅ 15. Prepare for Interviews**

* Common OOP questions in C#
* Compare: interface vs abstract vs class
* Refactoring to apply SOLID
* Designing systems with OOP
* Practical UML diagrams for C# classes

**Answer**

**🧠 What is OOP (Object-Oriented Programming)?**

**✅ Definition:**

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which are instances of classes. These objects can contain:

* Data (in the form of fields or properties), and
* Behavior (in the form of methods or functions).

OOP focuses on organizing software around real-world entities and their interactions, rather than just functions and logic.

**🎯 Goal of OOP:**

To design flexible, reusable, maintainable, and scalable software by:

* Structuring code into classes
* Encapsulating related data and logic into a single unit (object)
* Promoting code reuse through inheritance
* Supporting dynamic behavior through polymorphism

**🧱 Four Pillars of OOP:**

1. **Encapsulation:** Wrapping data and behavior together; restricting direct access to internal state.
2. **Inheritance**: Reusing code from a base class in a derived class**.**
3. **Polymorphism**: Objects can take many forms (e.g., method overriding).
4. **Abstraction:** Hiding internal complexity and showing only essential features.

**📦 Real-World Example:**

**Imagine you’re modeling a Car in software:**

**csharp**

**CopyEdit**

**public class Car {**

**public string Color { get; set; } // Property (data)**

**public void Drive() { // Method (behavior)**

**Console.WriteLine("Car is driving");**

**}**

**}**

**Here:**

* **Car is a class (a blueprint).**
* **An actual car (e.g., Car myCar = new Car();) is an object.**
* **The car has data (Color) and behavior (Drive()).**

**🧩 Why Use OOP?**

* **Reduces code duplication.**
* **Makes code easier to test and debug.**
* **Encourages modularity and separation of concerns.**
* **Easier to model real-world problems.**
* **Supports extensibility (add features without rewriting existing code).**

**🔍 OOP vs Procedural Programming in C#**

| **Feature** | **OOP (Object-Oriented Programming)** | **Procedural Programming** |
| --- | --- | --- |
| **Definition** | **Based on objects and classes** | **Based on functions and procedures** |
| **Program Structure** | **Divided into objects (data + behavior)** | **Divided into functions that operate on data** |
| **Data Access** | **Uses encapsulation to restrict access** | **Data is typically global/shared** |
| **Code Reuse** | **Achieved via inheritance and polymorphism** | **Reuse through function calls** |
| **Modularity** | **Highly modular due to class-based structure** | **Less modular, especially in large programs** |
| **Flexibility** | **Easy to extend and maintain** | **More difficult to scale or extend** |
| **Example Use Cases** | **Game engines, GUI apps, enterprise systems** | **Scripting, system tools, simple utilities** |
| **Example in C#** | **Classes, objects, interfaces** | **Static methods in a single file** |

**🧠 Example Comparison**

**✅ Procedural Style in C#:**

**int speed = 0;**

**void Accelerate() {**

**speed += 10;**

**Console.WriteLine("Speed: " + speed);**

**}**

**Accelerate();**

**✅ Object-Oriented Style in C#:**

**public class Car {**

**private int speed;**

**public void Accelerate() {**

**speed += 10;**

**Console.WriteLine("Speed: " + speed);**

**}**

**}**

**Car myCar = new Car();**

**myCar.Accelerate();**

**📌 Key Differences (Explained)**

**1. Data & Behavior Grouping**

* **OOP: Data and behavior live together inside the class.**
* **Procedural: Data is separate; functions manipulate global/shared data.**

**2. Scalability**

* **OOP: Scales better in large systems (supports design patterns, testing).**
* **Procedural: Becomes harder to manage as the codebase grows.**

**3. Maintenance**

* **OOP: Easier to maintain and change code without breaking everything.**
* **Procedural: Tight coupling between logic and data makes changes risky.**

**🔧 When to Use What?**

| **Scenario** | **Use** |
| --- | --- |
| **Small scripts, quick tasks** | **Procedural** |
| **Large systems, apps with many entities** | **OOP** |
| **Systems needing maintainability & reusability** | **OOP** |

**🧱 Four Pillars of OOP in C#**

**✅ 1. Encapsulation**

**Definition: Binding data (fields) and behavior (methods) into a single unit (class), and restricting access to the inner workings of that object.**

**🔐 Purpose:**

* **Hide complexity from the outside world**
* **Protect internal object state**
* **Control how data is accessed/modified**

**🧩 Example:**

**public class BankAccount {**

**private decimal balance; // Private field**

**public void Deposit(decimal amount) {**

**if (amount > 0) {**

**balance += amount;**

**}**

**}**

**public decimal GetBalance() {**

**return balance;**

**}**

**}**

**🔸 Outside code can’t directly access balance, only through public methods.**

**✅ 2. Inheritance**

**Definition: Enables a class to inherit members (fields, properties, methods) from another class.**

**🔐 Purpose:**

* **Reuse common code (code reuse)**
* **Support hierarchical relationships (is-a relationship)**

**🧩 Example:**

**public class Animal {**

**public void Eat() {**

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

**}**

**}**

**public class Dog : Animal {**

**public void Bark() {**

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

**}**

**}**

**// Usage**

**Dog dog = new Dog();**

**dog.Eat(); // Inherited from Animal**

**dog.Bark();**

**🔸 Dog inherits the behavior of Animal.**

**✅ 3. Polymorphism**

**Definition: The ability for different classes to be treated as instances of the same parent class, and for the same method name to behave differently depending on the object.**

**📌 Types:**

* **Compile-time (Static) Polymorphism → Method Overloading**
* **Run-time (Dynamic) Polymorphism → Method Overriding**

**🧩 Example:**

**public class Animal {**

**public virtual void MakeSound() {**

**Console.WriteLine("Animal makes a sound");**

**}**

**}**

**public class Dog : Animal {**

**public override void MakeSound() {**

**Console.WriteLine("Dog barks");**

**}**

**}**

**// Usage**

**Animal myAnimal = new Dog(); // Upcasting**

**myAnimal.MakeSound(); // Outputs: "Dog barks"**

**🔸 The same method MakeSound() works differently depending on the object.**

**✅ 4. Abstraction**

**Definition: Hiding internal implementation details and showing only essential features to the outside world.**

**🔐 Purpose:**

* **Focus on what an object does, not how it does it**
* **Simplifies usage of complex systems**

**🧩 Achieved By:**

* **Abstract classes**
* **Interfaces**

**🧩 Example (Interface):**

**public interface IPayment {**

**void ProcessPayment(decimal amount);**

**}**

**public class CreditCardPayment : IPayment {**

**public void ProcessPayment(decimal amount) {**

**Console.WriteLine("Paid " + amount + " with credit card.");**

**}**

**}**

**🔸 The user just calls ProcessPayment() without knowing internal implementation.**

**🔁 Summary Table**

| **Pillar** | **Key Concept** | **Benefit** |
| --- | --- | --- |
| **Encapsulation** | **Hide data using private fields and public methods** | **Security, control, modularity** |
| **Inheritance** | **Reuse code from base classes** | **Code reuse, hierarchy modeling** |
| **Polymorphism** | **Same method behaves differently** | **Flexibility, extensibility** |
| **Abstraction** | **Show only essential features, hide details** | **Simplicity, separation of concerns** |

**Absolutely, Atanu! Let’s go deep into "Classes and Objects" in C#, including interview-level understanding, best practices, and real-world examples. This is core to mastering OOP and cracking top-level interviews.**

**🔹 2.1 Creating and Using Classes in C#**

**🧱 What is a Class?**

**A class is a blueprint or template for creating objects. It defines the properties (data) and methods (behavior) an object will have.**

**✅ Syntax of a Class:**

**public class Car**

**{**

**// Fields (data)**

**private string color;**

**private int speed;**

**// Properties**

**public string Color {**

**get { return color; }**

**set { color = value; }**

**}**

**// Method (behavior)**

**public void Drive() {**

**Console.WriteLine("Car is driving...");**

**}**

**}**

**🧱 What is an Object?**

**An object is an instance of a class. It is a real-world entity created from the class blueprint.**

**✅ Creating an Object**

**Car myCar = new Car(); // Object creation**

**myCar.Color = "Red"; // Set property**

**myCar.Drive(); // Call method**

**📦 Key Terms**

| **Term** | **Description** |
| --- | --- |
| **Class** | **Blueprint or template** |
| **Object** | **Instance of a class** |
| **Instance** | **Synonym for object** |
| **Field** | **Private data inside the class** |
| **Property** | **Public access point for fields (get/set)** |
| **Method** | **Defines behavior of object** |

**🧠 Object Lifecycle in C#**

**Let’s understand the lifecycle of an object step-by-step:**

**🔹 1. Declaration**

**Car myCar;**

**This is just a reference. No memory is allocated yet.**

**🔹 2. Instantiation**

**myCar = new Car();**

**This allocates memory and creates the actual object.**

**🔹 3. Initialization**

**myCar.Color = "Black";**

**Set values to object properties.**

**🔹 4. Usage**

**myCar.Drive();**

**You can now use the object’s behavior.**

**🔹 5. Destruction (Garbage Collection)**

* **C# uses automatic garbage collection.**
* **You don’t need to manually delete objects.**
* **If an object is no longer referenced, it is eligible for GC.**

**🎯 Interview-Proof Understanding: Key Concepts**

**✅ Object Reference vs Object Instance**

**Car car1 = new Car();**

**Car car2 = car1;**

**car1.Color = "Red";**

**Console.WriteLine(car2.Color); // Red (same reference!)**

**🔸 car1 and car2 point to the same object in memory.**

**✅ Memory Allocation:**

* **Objects are created in the Heap.**
* **Reference variables are stored in the Stack.**

**✅ Anonymous Objects:**

**new Car().Drive(); // No reference stored**

**✅ Multiple Objects from One Class:**

**Car car1 = new Car();**

**Car car2 = new Car();**

**Each object has its own copy of fields/properties.**

**🛠 Real-Life Example: Employee Class**

**public class Employee**

**{**

**public int Id { get; set; }**

**public string Name { get; set; }**

**public void DisplayInfo() {**

**Console.WriteLine($"ID: {Id}, Name: {Name}");**

**}**

**}**

**// Usage**

**Employee emp1 = new Employee();**

**emp1.Id = 101;**

**emp1.Name = "Atanu";**

**emp1.DisplayInfo();**

**❓ Interview Questions You Must Be Ready For**

**Q1: What is the difference between a class and an object?**

**A class is a definition (blueprint), while an object is a real-world instance of that class.**

**Q2: How is memory allocated for objects in C#?**

**Objects are allocated on the heap, and reference variables on the stack.**

**Q3: What is the default value of a class field?**

**If not initialized, reference types are null, and value types are their default (e.g., 0 for int).**

**Q4: Can you instantiate a class without a reference?**

**Yes, using anonymous objects: new ClassName().MethodName();**

**Q5: How does garbage collection work in relation to objects?**

**Once no reference points to an object, the GC (Garbage Collector) will clean it up automatically.**

**🚀 Pro Tips**

* **Always use properties instead of public fields (for future validation/logging).**
* **Use constructors for safe object initialization (we’ll dive into that next).**
* **Avoid null reference exceptions by using null checks or nullable reference types.**

**🔹 2.2 Fields, Methods, and Properties (C# OOP)**

**✅ 1. Fields**

**Fields are variables declared inside a class, used to store data.**

**🔸 Syntax:**

**public class Person {**

**public string name; // Field**

**}**

**🔹 By default:**

* **Fields are instance members (per object).**
* **Can be public or private (use private + properties in best practices).**

**✅ 2. Methods**

**Methods define the behavior of a class — they perform actions.**

**🔸 Syntax:**

**public void SayHello() {**

**Console.WriteLine("Hello!");**

**}**

**🔹 Can take parameters, return values, and be static or instance-based.**

**✅ 3. Properties**

**Properties provide controlled access to fields.**

**🔸 Use properties instead of exposing fields directly.**

**private string name; // Backing field**

**public string Name {**

**get { return name; }**

**set { name = value; }**

**}**

**🔄 Instance vs Static Members**

| **Feature** | **Instance** | **Static** |
| --- | --- | --- |
| **Scope** | **Per object** | **Shared across all instances** |
| **Access** | **Via object reference** | **Via class name** |
| **Memory** | **Stored per object** | **Stored once in memory (in AppDomain)** |
| **Use Case** | **Object-specific data/behavior** | **Shared logic/data like counters, helpers** |

**🧩 Example:**

**public class Counter {**

**public int instanceCount = 0;**

**public static int staticCount = 0;**

**public void Increment() {**

**instanceCount++;**

**staticCount++;**

**}**

**}**

**🔐 readonly, const, and static — Key Differences**

| **Keyword** | **Value Settable** | **When Set?** | **Modifier For** | **Mutability** |
| --- | --- | --- | --- | --- |
| **const** | **No (compile-time)** | **At declaration** | **Fields** | **Immutable** |
| **readonly** | **Yes (once)** | **In constructor only** | **Fields** | **Immutable** |
| **static** | **Shared** | **Class-level (1 copy)** | **Fields/Props/Methods** | **Mutable or not** |

**🧩 Examples:**

**public class Sample {**

**public const double PI = 3.14; // Must assign at declaration**

**public readonly int CreatedAt; // Set once in constructor**

**public static int SharedCount = 0; // Shared among all objects**

**public Sample() {**

**CreatedAt = DateTime.Now.Second;**

**}**

**}**

**⚙️ Auto-Implemented Properties**

**Shorthand for defining properties when no logic is needed in get or set.**

**🔸 Syntax:**

**public string Name { get; set; } // Auto-property**

**🔹 Behind the scenes, compiler creates a private anonymous backing field.**

**🧠 Property with Logic (Getters and Setters)**

**When you need to:**

* **Validate input**
* **Transform data**
* **Add logging**

**🔸 Example:**

**private int age;**

**public int Age {**

**get {**

**return age;**

**}**

**set {**

**if (value >= 0) age = value;**

**else Console.WriteLine("Age cannot be negative");**

**}**

**}**

**🛠 Full Example — All Concepts Together**

**public class Product**

**{**

**private string name;**

**public static int TotalProducts = 0;**

**public readonly DateTime CreatedOn;**

**public const string Company = "TechCorp";**

**public string Name {**

**get { return name; }**

**set {**

**if (!string.IsNullOrEmpty(value)) {**

**name = value;**

**}**

**}**

**}**

**// Auto-property**

**public decimal Price { get; set; }**

**public Product() {**

**CreatedOn = DateTime.Now;**

**TotalProducts++;**

**}**

**public void Display() {**

**Console.WriteLine($"Name: {Name}, Price: {Price}, Created: {CreatedOn}");**

**}**

**public static void ShowCompany() {**

**Console.WriteLine($"Company: {Company}");**

**}**

**}**

**🧠 Must-Know Interview Questions:**

**Q1: Difference between field and property?**

* **Field: raw storage.**
* **Property: provides controlled access, can have logic.**

**Q2: Can you override a property?**

* **Yes, using virtual, override, or abstract in inheritance.**

**Q3: When to use readonly vs const?**

* **const: for known-at-compile-time values (e.g., PI)**
* **readonly: for values known only at runtime but never changed after that.**

**Q4: Can static members access instance data?**

* **❌ No. Static methods can’t access instance members directly.**

**Q5: What happens if you don’t provide a setter?**

**public string Name { get; } // Read-only property**

* **It's a read-only property. Can only be set in constructor (or init-only in C# 9+).**

**🔒 2.3 Access Modifiers in C#**

**✅ What Are Access Modifiers?**

**Access modifiers define where a class member (field, method, property, class) can be accessed from. They control visibility and encapsulation.**

**C# supports 6 access modifiers:**

**🔹 1. public**

**Accessible from anywhere — no restriction.**

**✅ Usage:**

**public class Car {**

**public string Brand;**

**public void Drive() {**

**Console.WriteLine("Driving...");**

**}**

**}**

**🟢 Used for:**

* **Exposing APIs**
* **Making class libraries reusable**

**🔹 2. private**

**Accessible only within the same class.**

**✅ Usage:**

**public class Car {**

**private int speed;**

**public void Accelerate() {**

**speed += 10;**

**Console.WriteLine($"Speed: {speed}");**

**}**

**}**

**🔒 Best for:**

* **Hiding internal logic**
* **Enforcing encapsulation**

**🔹 3. protected**

**Accessible within the same class and derived classes (even in other assemblies).**

**✅ Usage:**

**public class Animal {**

**protected void Sleep() {**

**Console.WriteLine("Sleeping...");**

**}**

**}**

**public class Dog : Animal {**

**public void Rest() {**

**Sleep(); // ✅ Allowed**

**}**

**}**

**🟡 Used for:**

* **Letting child classes reuse internal logic**

**🔹 4. internal**

**Accessible only within the same assembly/project (not accessible outside DLL/EXE).**

**✅ Usage:**

**internal class Engine {**

**internal void Start() {**

**Console.WriteLine("Engine started.");**

**}**

**}**

**⚙️ Good for:**

* **Hiding implementation in libraries**
* **Creating internal APIs**

**🔹 5. protected internal**

**Accessible within the same assembly OR from derived classes in other assemblies.**

**✅ Usage:**

**public class Base {**

**protected internal void Show() {**

**Console.WriteLine("Accessible from derived OR same assembly.");**

**}**

**}**

**🟠 Combines protected + internal.**

**🔹 6. private protected (C# 7.2+)**

**Accessible within the same class or derived classes, but only within the same assembly.**

**✅ Usage:**

**public class Base {**

**private protected void Info() {**

**Console.WriteLine("Only for this class and derived in same assembly");**

**}**

**}**

**🔴 Most restrictive for inheritance scenarios.**

**📊 Summary Table**

| **Modifier** | **Same Class** | **Derived Class (Same Assembly)** | **Other Class (Same Assembly)** | **Derived (Other Assembly)** | **Other Class (Other Assembly)** |
| --- | --- | --- | --- | --- | --- |
| **public** | **✅** | **✅** | **✅** | **✅** | **✅** |
| **private** | **✅** | **❌** | **❌** | **❌** | **❌** |
| **protected** | **✅** | **✅** | **❌** | **✅** | **❌** |
| **internal** | **✅** | **✅** | **✅** | **❌** | **❌** |
| **protected internal** | **✅** | **✅** | **✅** | **✅** | **❌** |
| **private protected** | **✅** | **✅** | **❌** | **❌** | **❌** |

**🎯 Real-World Use Cases**

* **public → Service classes, API models**
* **private → Internal variables, helper methods**
* **protected → Shared logic across base/child classes**
* **internal → Utility classes inside a library**
* **protected internal → Base class logic + same assembly flexibility**
* **private protected → Strictest form of inheritance control**

**❓ Common Interview Questions**

**Q1: What's the difference between protected internal and private protected?**

* **protected internal: accessible to derived classes OR same assembly.**
* **private protected: accessible to derived classes AND in same assembly only.**

**Q2: Can a method be both private and protected?**

* **Yes, via private protected.**

**Q3: Can you apply access modifiers to classes?**

* **✅ Top-level classes: public, internal**
* **❌ private, protected, etc. are not allowed for top-level classes.**

**🔨 3.1 Types of Constructors in C#**

**✅ What is a Constructor?**

**A constructor is a special method in a class that gets called automatically when an object is created.  
Its job is to initialize the object.**

**🔸 Syntax:**

**public class MyClass {**

**public MyClass() {**

**// This is a constructor**

**}**

**}**

* **Has no return type (not even void)**
* **Same name as the class**
* **Can be overloaded (multiple constructors with different signatures)**

**🔹 1. Default Constructor**

**A constructor with no parameters. Used to initialize default values.**

**✅ Example:**

**public class Car {**

**public string Brand;**

**public Car() {**

**Brand = "Unknown";**

**}**

**}**

**🔹 Usage:**

**Car c1 = new Car();**

**Console.WriteLine(c1.Brand); // Output: Unknown**

**🔹 2. Parameterized Constructor**

**A constructor that takes arguments. Used to initialize objects with custom values.**

**✅ Example:**

**public class Car {**

**public string Brand;**

**public Car(string brand) {**

**Brand = brand;**

**}**

**}**

**🔹 Usage:**

**Car c2 = new Car("Toyota");**

**Console.WriteLine(c2.Brand); // Output: Toyota**

**🔹 3. Constructor Overloading**

**Defining multiple constructors with different parameters (method overloading concept).**

**✅ Example:**

**public class Employee {**

**public string Name;**

**public int Age;**

**// Default**

**public Employee() {**

**Name = "Unknown";**

**Age = 0;**

**}**

**// Overloaded**

**public Employee(string name) {**

**Name = name;**

**Age = 25; // default age**

**}**

**public Employee(string name, int age) {**

**Name = name;**

**Age = age;**

**}**

**}**

**🔹 4. Static Constructor**

**A constructor that:**

* **Is called automatically once**
* **Initializes static members**
* **Cannot have parameters**
* **Cannot be called explicitly**

**✅ Syntax:**

**public class Config {**

**public static string AppName;**

**static Config() {**

**Console.WriteLine("Static constructor called.");**

**AppName = "MyApp";**

**}**

**}**

**🔹 Usage:**

**Console.WriteLine(Config.AppName);**

**// Output:**

**// Static constructor called.**

**// MyApp**

**🧠 Rules:**

* **Runs once per type (before first usage)**
* **Cannot use access modifiers (always private)**
* **No this or base call**
* **Used for logging, config reading, etc.**

**🔹 5. Private Constructor**

**A constructor declared private:**

* **Prevents instantiation from outside**
* **Common in Singleton or Static Classes**

**✅ Example: Singleton Pattern**

**public class Logger {**

**private static Logger instance;**

**private Logger() {**

**Console.WriteLine("Logger initialized.");**

**}**

**public static Logger GetInstance() {**

**if (instance == null) {**

**instance = new Logger();**

**}**

**return instance;**

**}**

**}**

**🔒 Prevents creating:**

**// Logger log = new Logger(); ❌ Not allowed**

**Logger log = Logger.GetInstance(); ✅**

**🧠 Object Initialization Order (Important for Interviews)**

1. **Static constructor (once per type)**
2. **Instance field initializers**
3. **Instance constructor**

**📊 Summary Table**

| **Constructor Type** | **Parameters** | **Can Overload?** | **Use Case** |
| --- | --- | --- | --- |
| **Default** | **No** | **Yes** | **Default values, basic instantiation** |
| **Parameterized** | **Yes** | **Yes** | **Custom data at object creation** |
| **Overloaded** | **Optional** | **Yes** | **Flexibility in initialization** |
| **Static** | **No** | **No** | **Static field setup (once per class)** |
| **Private** | **Optional** | **Yes** | **Restrict instantiation, Singleton** |

**❓ Must-Know Interview Questions**

**Q1: Can we have a static and instance constructor in the same class?**

**✅ Yes. Static runs once per type, instance runs every time object is created.**

**Q2: Can a constructor be virtual or abstract?**

**❌ No. Constructors cannot be virtual, override, or abstract.**

**Q3: What happens if you don’t write a constructor?**

**✅ C# provides a default parameterless constructor automatically if no constructor is written.**

**Q4: Can constructors return values?**

**❌ No. Constructors do not have return types, not even void.**

**Q5: Can a constructor call another constructor?**

**✅ Yes, using constructor chaining with this() and base() — we can explore this next if you want.**

**🔄 3.2 Constructor Chaining in C#**

**✅ What is Constructor Chaining?**

**Constructor chaining means one constructor calls another constructor in the same class or base class using this or base.**

**This improves code reuse and maintainability, and reduces duplication.**

**🔹 this Keyword — Chaining Constructors in the Same Class**

**Use this() to call another constructor in the same class.**

**✅ Syntax:**

**public ClassName() : this(params) {**

**// body**

**}**

**🔸 Example:**

**public class Person {**

**public string Name;**

**public int Age;**

**// Default constructor**

**public Person() : this("Unknown", 0) {**

**Console.WriteLine("Default constructor called.");**

**}**

**// Parameterized constructor**

**public Person(string name, int age) {**

**Name = name;**

**Age = age;**

**Console.WriteLine("Parameterized constructor called.");**

**}**

**}**

**🔹 Output:**

**Person p = new Person();**

**// Output:**

**// Parameterized constructor called.**

**// Default constructor called.**

**🔹 base Keyword — Calling Base Class Constructor**

**Use base() to call the parent class constructor from the child class.**

**✅ Syntax:**

**public DerivedClass() : base(params) {**

**// child constructor body**

**}**

**🔸 Example:**

**public class Animal {**

**public Animal(string species) {**

**Console.WriteLine($"Animal created: {species}");**

**}**

**}**

**public class Dog : Animal {**

**public Dog() : base("Dog") {**

**Console.WriteLine("Dog constructor called.");**

**}**

**}**

**🔹 Output:**

**Dog d = new Dog();**

**// Output:**

**// Animal created: Dog**

**// Dog constructor called.**

**📌 Rules of Constructor Chaining**

| **Rule** | **Applies To** |
| --- | --- |
| **Must be the first statement in constructor** | **this / base** |
| **Can only chain to one constructor** | **this or base (not both)** |
| **Cannot call a method before this() or base()** | **❌** |
| **Static constructors cannot use this or base** | **❌** |
| **Can be used to avoid repeating logic in overloaded constructors** | **✅** |

**🧠 Full Example (Chaining in Both Classes)**

**public class Vehicle {**

**public Vehicle(string type) {**

**Console.WriteLine($"Vehicle Type: {type}");**

**}**

**}**

**public class Car : Vehicle {**

**public string Model;**

**public Car() : this("DefaultModel") {**

**Console.WriteLine("Default Car constructor");**

**}**

**public Car(string model) : base("Car") {**

**Model = model;**

**Console.WriteLine($"Car Model: {model}");**

**}**

**}**

**🔹 Output:**

**Car c = new Car();**

**// Output:**

**// Vehicle Type: Car**

**// Car Model: DefaultModel**

**// Default Car constructor**

**❓ Interview Questions on Constructor Chaining**

**Q1: Can you call this() and base() in the same constructor?**

**❌ No. Only one can be used, and it must be the first statement.**

**Q2: Why use constructor chaining?**

**✅ To:**

* **Reduce code duplication**
* **Maintain initialization logic in one place**

**Q3: Can constructor chaining throw exceptions?**

**✅ Yes. If an exception occurs in the chained constructor, object creation will fail.**

**Q4: Can static constructors use this() or base()?**

**❌ No. Static constructors are not part of instance chaining logic.**

**✅ Summary**

| **Keyword** | **Used For** | **Calls** | **Use Case** |
| --- | --- | --- | --- |
| **this** | **Within same class** | **Another constructor** | **Reuse logic in overloaded constructors** |
| **base** | **In derived/child class constructor** | **Base class constructor** | **Inheritance, passing values to base** |

**🔐 4. Encapsulation in C#**

**✅ What is Encapsulation?**

**Encapsulation means hiding the internal details (data) of a class and only exposing a controlled interface to the outside world.  
It helps protect the data from unauthorized access and misuse.**

**🔹 Hiding Fields Using private**

**Fields (variables) inside a class should be declared private to restrict direct access.**

**public class BankAccount {**

**private double balance; // hidden from outside**

**}**

**🔹 Exposing via Public Properties**

**Use public properties with get and set accessors to control how fields are accessed or modified.**

**public class BankAccount {**

**private double balance;**

**// Property with getter and setter**

**public double Balance {**

**get { return balance; }**

**set {**

**if (value >= 0) // validation**

**balance = value;**

**}**

**}**

**}**

**🔹 Accessor Methods (Get/Set)**

**Before C# introduced properties, we used explicit get/set methods:**

**public class BankAccount {**

**private double balance;**

**public double GetBalance() {**

**return balance;**

**}**

**public void SetBalance(double value) {**

**if (value >= 0)**

**balance = value;**

**}**

**}**

**Properties are now preferred as a cleaner, more intuitive way.**

**🔹 Auto-Implemented Properties (Shortcut)**

**When no special logic is needed, C# lets you write properties simply:**

**public class Person {**

**public string Name { get; set; }**

**}**

**This auto-generates a private backing field behind the scenes.**

**🔹 Benefits of Encapsulation**

| **Benefit** | **Explanation** |
| --- | --- |
| **Control access** | **Control how data is read or written (e.g., validation)** |
| **Maintainability** | **Change internal implementation without affecting users** |
| **Data hiding** | **Protect sensitive data from unwanted modification** |
| **Increased flexibility** | **Properties can be extended with logic (lazy loading, events)** |
| **Improved security** | **Prevent invalid or harmful data assignments** |

**🔹 Practical Example: Encapsulated Class**

**public class Employee {**

**private int age;**

**public int Age {**

**get { return age; }**

**set {**

**if (value >= 18 && value <= 65)**

**age = value;**

**else**

**throw new ArgumentException("Age must be between 18 and 65.");**

**}**

**}**

**}**

**Usage:**

**Employee emp = new Employee();**

**emp.Age = 25; // OK**

**Console.WriteLine(emp.Age); // 25**

**emp.Age = 10; // Throws exception: Age must be between 18 and 65.**

**🔹 Encapsulation in Action (Full Example)**

**public class Product {**

**private double price;**

**public double Price {**

**get { return price; }**

**set {**

**if (value < 0)**

**throw new ArgumentException("Price cannot be negative");**

**price = value;**

**}**

**}**

**public void ApplyDiscount(double percent) {**

**if (percent < 0 || percent > 100)**

**throw new ArgumentException("Invalid discount percentage");**

**price -= price \* (percent / 100);**

**}**

**}**

**❓ Common Interview Questions on Encapsulation**

**Q1: Why do we use private fields instead of public?**

* **To protect internal state and prevent arbitrary external modification.**

**Q2: What’s the difference between fields and properties?**

* **Fields store data; properties control access to that data.**

**Q3: Can properties be read-only or write-only?**

* **Yes.**
  + **Read-only: public int Age { get; } (no setter)**
  + **Write-only: public int Password { set; } (no getter)**

**Q4: What happens if you expose fields as public directly?**

* **No control over data; anyone can modify data improperly leading to bugs.**

**🧬 5. Inheritance in C#**

**✅ What is Inheritance?**

**Inheritance is the mechanism by which a new class (derived class) inherits properties, methods, and behaviors from an existing class (base class).  
It promotes code reuse and logical hierarchy.**

**🔹 Base and Derived Class**

* **Base Class (Parent/Superclass): The class being inherited from.**
* **Derived Class (Child/Subclass): The class that inherits from the base.**

**Example:**

**// Base class**

**public class Animal {**

**public void Eat() {**

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

**}**

**}**

**// Derived class**

**public class Dog : Animal {**

**public void Bark() {**

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

**}**

**}**

**Usage:**

**Dog dog = new Dog();**

**dog.Eat(); // inherited method**

**dog.Bark(); // own method**

**🔹 Inheritance Hierarchy**

* **C# supports single inheritance: A class can only inherit from one base class.**
* **You can build multi-level inheritance:**

**public class Animal { }**

**public class Mammal : Animal { }**

**public class Dog : Mammal { }**

**🔹 The base Keyword**

**Used to:**

* **Access base class members (methods, properties)**
* **Call base class constructor**

**Example: Calling base class constructor**

**public class Animal {**

**public string Name;**

**public Animal(string name) {**

**Name = name;**

**}**

**}**

**public class Dog : Animal {**

**public Dog(string name) : base(name) {**

**Console.WriteLine("Dog created: " + Name);**

**}**

**}**

**🔹 Constructors in Inheritance**

* **Derived class constructors can call base class constructors using base() (must be the first statement).**
* **If base class has no parameterless constructor, derived must explicitly call base constructor.**

**public class Vehicle {**

**public Vehicle(string type) {**

**Console.WriteLine("Vehicle type: " + type);**

**}**

**}**

**public class Car : Vehicle {**

**public Car() : base("Car") {**

**Console.WriteLine("Car created");**

**}**

**}**

**🔹 Sealed Classes and Sealed Methods**

**1. Sealed Class**

**Cannot be inherited further.**

**public sealed class MathHelper {**

**public static int Add(int a, int b) => a + b;**

**}**

* **Useful to prevent inheritance and protect class behavior.**

**2. Sealed Method**

**Prevents a virtual method from being overridden in derived classes.**

**public class BaseClass {**

**public virtual void Display() {**

**Console.WriteLine("Base Display");**

**}**

**}**

**public class DerivedClass : BaseClass {**

**public sealed override void Display() {**

**Console.WriteLine("Derived Display");**

**}**

**}**

**public class MoreDerived : DerivedClass {**

**// Cannot override Display() here — sealed in DerivedClass**

**}**

**🔹 When to Avoid Inheritance — Prefer Composition**

| **Inheritance** | **Composition** |
| --- | --- |
| **Is-a relationship (Dog *is-a* Animal)** | **Has-a relationship (Car *has-a* Engine)** |
| **Tightly couples child and parent** | **More flexible, loosely coupled** |
| **Hard to change hierarchy later** | **Easier to modify and extend** |

**Problems with Overusing Inheritance:**

* **Fragile base class problem (changes in base affect many classes)**
* **Deep inheritance trees → hard to understand**
* **Violates encapsulation if used wrongly**

**Composition Example:**

**public class Engine {**

**public void Start() {**

**Console.WriteLine("Engine started");**

**}**

**}**

**public class Car {**

**private Engine engine = new Engine();**

**public void StartCar() {**

**engine.Start(); // delegation**

**Console.WriteLine("Car started");**

**}**

**}**

**❓ Interview Questions on Inheritance**

* **What is the difference between inheritance and composition?**
* **How do you call a base class constructor from a derived class?**
* **Can a sealed class be inherited?**
* **Explain constructor chaining with inheritance.**
* **What are the advantages and disadvantages of inheritance?**
* **When should you avoid inheritance?**

**🎭 6. Polymorphism in C#**

**🔹 6.1 Compile-Time Polymorphism (Static Polymorphism)**

**What is it?**

* **Decisions about which method to call are made at compile time.**
* **Achieved mainly by method overloading and operator overloading.**

**🛠️ Method Overloading**

**Multiple methods in the same class share the same name but differ in parameters (type, number, or order).**

**public class Calculator {**

**public int Add(int a, int b) {**

**return a + b;**

**}**

**public double Add(double a, double b) {**

**return a + b;**

**}**

**public int Add(int a, int b, int c) {**

**return a + b + c;**

**}**

**}**

**Usage:**

**Calculator calc = new Calculator();**

**Console.WriteLine(calc.Add(2, 3)); // 5**

**Console.WriteLine(calc.Add(2.5, 3.5)); // 6.0**

**Console.WriteLine(calc.Add(1, 2, 3)); // 6**

**🛠️ Operator Overloading**

**Define custom behavior for operators (+, -, etc.) for your own types.**

**public class Point {**

**public int X, Y;**

**public Point(int x, int y) {**

**X = x; Y = y;**

**}**

**// Overload + operator**

**public static Point operator +(Point p1, Point p2) {**

**return new Point(p1.X + p2.X, p1.Y + p2.Y);**

**}**

**public override string ToString() => $"({X}, {Y})";**

**}**

**Usage:**

**Point p1 = new Point(1, 2);**

**Point p2 = new Point(3, 4);**

**Point p3 = p1 + p2; // Calls overloaded +**

**Console.WriteLine(p3); // (4, 6)**

**🔹 6.2 Runtime Polymorphism (Dynamic Polymorphism)**

**What is it?**

* **Decisions about which method to call are made at runtime.**
* **Achieved by method overriding using virtual and override keywords.**

**🛠️ Method Overriding**

* **A base class defines a method as virtual.**
* **A derived class overrides it with override.**

**public class Animal {**

**public virtual void Speak() {**

**Console.WriteLine("Animal makes a sound");**

**}**

**}**

**public class Dog : Animal {**

**public override void Speak() {**

**Console.WriteLine("Dog barks");**

**}**

**}**

**public class Cat : Animal {**

**public override void Speak() {**

**Console.WriteLine("Cat meows");**

**}**

**}**

**Polymorphic Behavior with Base Reference**

**Animal animal1 = new Dog();**

**Animal animal2 = new Cat();**

**animal1.Speak(); // Dog barks**

**animal2.Speak(); // Cat meows**

**🛠️ new Keyword (Method Hiding)**

* **Instead of overriding, a derived class can hide the base class method by declaring a method with the same name and new keyword.**
* **This does not participate in runtime polymorphism.**

**public class BaseClass {**

**public void Display() {**

**Console.WriteLine("Base Display");**

**}**

**}**

**public class DerivedClass : BaseClass {**

**public new void Display() {**

**Console.WriteLine("Derived Display");**

**}**

**}**

**BaseClass obj1 = new DerivedClass();**

**obj1.Display(); // Output: Base Display (calls base method)**

**DerivedClass obj2 = new DerivedClass();**

**obj2.Display(); // Output: Derived Display**

**Summary Table**

| **Type** | **How It Works** | **Keywords** | **When Determined** |
| --- | --- | --- | --- |
| **Compile-Time Polymorphism** | **Method or operator signature determines method** | **method overloading, operator overloading** | **Compile time** |
| **Runtime Polymorphism** | **Base class reference calls overridden derived method** | **virtual, override** | **Runtime** |
| **Method Hiding** | **Derived method hides base method** | **new** | **Compile time** |

**Common Interview Questions**

* **What's the difference between method overloading and overriding?**
* **What is the difference between override and new keyword?**
* **Can private methods be overridden? (No)**
* **What happens if you forget to mark base method as virtual but try to override?**
* **Explain polymorphism with examples.**

**🎭 7. Abstraction in C#**

**✅ What is Abstraction?**

**Abstraction means hiding the complex implementation details and showing only the essential features of an object/class.  
It focuses on *what* an object does, not *how* it does it.**

**🔹 Abstract Classes and Abstract Methods**

* **Abstract class: A class that cannot be instantiated and is meant to be a base class.**
* **Can contain abstract methods (methods without implementation) and normal methods with implementation.**
* **Abstract methods must be implemented by derived classes.**

**Syntax and Example:**

**public abstract class Vehicle {**

**public abstract void Start(); // abstract method**

**public void Stop() { // concrete method**

**Console.WriteLine("Vehicle stopped");**

**}**

**}**

**public class Car : Vehicle {**

**public override void Start() {**

**Console.WriteLine("Car started");**

**}**

**}**

**Usage:**

**Vehicle myCar = new Car();**

**myCar.Start(); // Car started**

**myCar.Stop(); // Vehicle stopped**

**🔹 Interfaces**

* **Interfaces only declare methods/properties/events (no implementation, except default interface methods in recent C#).**
* **A class implements interfaces and provides method implementations.**
* **A class can implement multiple interfaces (unlike inheritance).**

**Example:**

**public interface IPayment {**

**void Pay(double amount);**

**}**

**public class CreditCardPayment : IPayment {**

**public void Pay(double amount) {**

**Console.WriteLine($"Paid {amount} using Credit Card");**

**}**

**}**

**public class PaypalPayment : IPayment {**

**public void Pay(double amount) {**

**Console.WriteLine($"Paid {amount} using Paypal");**

**}**

**}**

**🔹 Interface vs Abstract Class**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Can have implemented methods** | **Yes** | **Mostly no (until C# 8.0 default methods)** |
| **Can contain fields** | **Yes** | **No** |
| **Multiple inheritance** | **No (only single inheritance)** | **Yes (a class can implement multiple interfaces)** |
| **Can be instantiated** | **No** | **No** |
| **Purpose** | **Share base behavior + enforce method implementation** | **Define a contract (only method signatures)** |
| **Constructors** | **Yes** | **No** |

**🔹 Multiple Interface Implementation**

**public interface ILogger {**

**void Log(string message);**

**}**

**public interface INotifier {**

**void Notify(string message);**

**}**

**public class NotificationService : ILogger, INotifier {**

**public void Log(string message) {**

**Console.WriteLine($"Log: {message}");**

**}**

**public void Notify(string message) {**

**Console.WriteLine($"Notify: {message}");**

**}**

**}**

**🔹 Real-World Examples**

**1. Payment System**

**public interface IPaymentGateway {**

**void ProcessPayment(double amount);**

**}**

**public class StripePayment : IPaymentGateway {**

**public void ProcessPayment(double amount) {**

**Console.WriteLine($"Processing Stripe payment of {amount}");**

**}**

**}**

**public class PaypalPayment : IPaymentGateway {**

**public void ProcessPayment(double amount) {**

**Console.WriteLine($"Processing Paypal payment of {amount}");**

**}**

**}**

**Usage:**

**IPaymentGateway payment = new StripePayment();**

**payment.ProcessPayment(1000);**

**2. Logger System**

**public abstract class Logger {**

**public abstract void Log(string message);**

**public void LogWithTimestamp(string message) {**

**Console.WriteLine($"{DateTime.Now}: {message}");**

**}**

**}**

**public class FileLogger : Logger {**

**public override void Log(string message) {**

**Console.WriteLine($"Writing to file: {message}");**

**}**

**}**

**❓ Common Interview Questions on Abstraction**

* **What is the difference between abstract classes and interfaces?**
* **When would you choose an abstract class over an interface?**
* **Can a class inherit multiple abstract classes? (No)**
* **Can a class implement multiple interfaces? (Yes)**
* **Explain real-world use cases of abstraction.**

**⚙️ 8. Advanced Concepts in C#**

**🔹 8.1 Object Class**

* **In C#, every type implicitly inherits from the base class object.**
* **It provides several important methods available to all objects.**

**Key Methods:**

**1. ToString()**

* **Returns a string representation of the object.**
* **You can override it to customize output.**

**public class Person {**

**public string Name { get; set; }**

**public int Age { get; set; }**

**public override string ToString() {**

**return $"Person(Name: {Name}, Age: {Age})";**

**}**

**}**

**Person p = new Person { Name = "Atanu", Age = 30 };**

**Console.WriteLine(p.ToString());**

**// Output: Person(Name: Atanu, Age: 30)**

**2. Equals(object obj)**

* **Used to determine if two objects are equal.**
* **By default, compares references for reference types.**
* **Override to define custom equality logic.**

**public class Point {**

**public int X, Y;**

**public override bool Equals(object obj) {**

**if (obj == null || !(obj is Point)) return false;**

**Point other = (Point)obj;**

**return this.X == other.X && this.Y == other.Y;**

**}**

**public override int GetHashCode() {**

**return X.GetHashCode() ^ Y.GetHashCode();**

**}**

**}**

**Point p1 = new Point { X = 1, Y = 2 };**

**Point p2 = new Point { X = 1, Y = 2 };**

**Console.WriteLine(p1.Equals(p2)); // True**

**3. GetHashCode()**

* **Returns a hash code for the object, used in hash-based collections (like Dictionary).**
* **When overriding Equals, always override GetHashCode.**

**4. GetType()**

* **Returns the runtime type of the current instance.**

**Person p = new Person();**

**Console.WriteLine(p.GetType()); // Outputs: Namespace.Person**

**🔹 8.2 Boxing and Unboxing**

**What are they?**

* **Boxing: Converting a value type (e.g., int, struct) to a reference type (object).**
* **Unboxing: Extracting the value type from the boxed object.**

**Why?**

**Because value types and reference types are stored differently in memory.**

**Example:**

**int x = 123; // value type**

**object obj = x; // Boxing: int -> object (heap allocation)**

**int y = (int)obj; // Unboxing: object -> int (value type)**

**Important:**

* **Boxing is expensive (performance overhead) because it involves heap allocation.**
* **Avoid unnecessary boxing/unboxing in performance-critical code.**

**🔹 8.3 Static Classes and Static Members**

**Static Members**

* **Belong to the class itself, not instances.**
* **Shared by all instances.**
* **Accessed using the class name.**

**public class MathUtils {**

**public static double Pi = 3.14159;**

**public static int Square(int x) {**

**return x \* x;**

**}**

**}**

**Console.WriteLine(MathUtils.Pi);**

**Console.WriteLine(MathUtils.Square(5));**

**Static Classes**

* **Cannot be instantiated or inherited.**
* **Can contain only static members.**
* **Used for utility/helper classes.**

**public static class Logger {**

**public static void Log(string message) {**

**Console.WriteLine($"Log: {message}");**

**}**

**}**

**Logger.Log("This is a static log message");**

**When to use Static?**

* **When the class only contains stateless utility functions.**
* **When you want a single instance shared globally (Singleton pattern).**

**Singleton Pattern Using Static**

**Ensures a class has only one instance and provides a global access point.**

**public sealed class Singleton {**

**private static readonly Singleton instance = new Singleton();**

**// Private constructor prevents instantiation outside**

**private Singleton() {}**

**public static Singleton Instance {**

**get { return instance; }**

**}**

**public void ShowMessage() {**

**Console.WriteLine("Singleton instance called");**

**}**

**}**

**Usage:**

**Singleton s1 = Singleton.Instance;**

**Singleton s2 = Singleton.Instance;**

**s1.ShowMessage(); // Singleton instance called**

**Console.WriteLine(object.ReferenceEquals(s1, s2)); // True (both references same instance)**

**Summary**

| **Concept** | **Description** | **Example Keyword** |
| --- | --- | --- |
| **Object class** | **Base class with methods like ToString, Equals** | **override ToString()** |
| **Boxing/Unboxing** | **Convert value types to object and back** | **Implicit conversion** |
| **Static members** | **Shared class-level variables/methods** | **static keyword** |
| **Static class** | **Class with only static members, no instance** | **static class** |
| **Singleton pattern** | **Single global instance using static members** | **Private ctor + static property** |

**🚀 Modern C# & OOP Features**

**1. Records vs Classes**

**What are Records?**

* **Introduced in C# 9.0.**
* **Designed to create immutable data objects with value-based equality.**
* **Simplifies defining data carriers (like DTOs).**
* **Records provide built-in value equality, whereas classes use reference equality by default.**

**Example:**

**public record PersonRecord(string FirstName, string LastName);**

**public class PersonClass {**

**public string FirstName { get; set; }**

**public string LastName { get; set; }**

**}**

**Equality Check:**

**var r1 = new PersonRecord("Atanu", "Das");**

**var r2 = new PersonRecord("Atanu", "Das");**

**Console.WriteLine(r1 == r2); // True (value equality)**

**var c1 = new PersonClass { FirstName = "Atanu", LastName = "Das" };**

**var c2 = new PersonClass { FirstName = "Atanu", LastName = "Das" };**

**Console.WriteLine(c1 == c2); // False (reference equality)**

**Immutability with Records**

**Records have init-only properties by default:**

**var person = new PersonRecord("Atanu", "Das");**

**// person.FirstName = "NewName"; // Error! init-only setter**

**2. init-only setters**

* **Introduced in C# 9.0.**
* **Allows properties to be set only during object initialization, making the object immutable after creation.**

**Example:**

**public class Employee {**

**public string Name { get; init; }**

**public int Age { get; init; }**

**}**

**var emp = new Employee { Name = "Atanu", Age = 30 };**

**// emp.Name = "New Name"; // Compilation error: cannot assign after initialization**

**3. Nullable Reference Types**

* **Introduced in C# 8.0.**
* **Helps avoid null reference exceptions by explicitly marking reference types as nullable or non-nullable.**
* **Requires nullable context enabled (#nullable enable or project setting).**

**Example:**

**#nullable enable**

**string? nullableString = null; // Allowed**

**string nonNullableString = "Hi";**

**// Console.WriteLine(nonNullableString.Length); // Safe**

**// Console.WriteLine(nullableString.Length); // Warning: possible null reference**

**4. Pattern Matching in OOP**

* **Enhanced in C# 8.0 and later.**
* **Allows concise, readable conditional logic for checking types, properties, and values.**

**Examples:**

**object obj = "Hello";**

**if (obj is string s) {**

**Console.WriteLine($"String of length {s.Length}");**

**}**

**int? number = 5;**

**switch (number) {**

**case int n when n > 0:**

**Console.WriteLine("Positive number");**

**break;**

**case null:**

**Console.WriteLine("Null value");**

**break;**

**}**

**5. Target-typed new expressions**

* **Introduced in C# 9.0.**
* **Simplifies object creation by omitting the type on the right side when the compiler can infer it.**

**Example:**

**List<string> names = new(); // Instead of new List<string>()**

**var emp = new Employee { Name = "Atanu" }; // Type inferred**

**6. File-scoped namespace declarations**

* **Introduced in C# 10.0.**
* **Simplifies namespace declaration by removing indentation.**

**Traditional way:**

**namespace MyApp**

**{**

**class Program**

**{**

**static void Main() { }**

**}**

**}**

**File-scoped namespace:**

**namespace MyApp;**

**class Program**

**{**

**static void Main() { }**

**}**

**Summary Table**

| **Feature** | **Purpose** | **C# Version** | **Example Keyword/Concept** |
| --- | --- | --- | --- |
| **Records** | **Immutable data classes with value equality** | **9.0** | **record** |
| **init-only setters** | **Immutable properties after initialization** | **9.0** | **init** |
| **Nullable Reference Types** | **Avoid null errors with nullable annotations** | **8.0** | **string?, #nullable enable** |
| **Pattern Matching** | **Concise type/value checking** | **7.0+** | **is, switch, when** |
| **Target-typed new expressions** | **Simplify object instantiation** | **9.0** | **new()** |
| **File-scoped namespaces** | **Cleaner namespace declaration** | **10.0** | **namespace MyApp;** |