What are the four pillars of OOP?

Answer:
Encapsulation, Abstraction, Inheritance, and Polymorphism.

What is a class?

Answer:
A class is a blueprint or template used to create objects. It defines the properties (fields) and behaviors (methods) that the objects created from it will have.

For example, the following `Student` class contains fields like `name`, `id`, and `email`, along with a method `ShowInfo()` to display student details:

```
public class Student
{
    public string name;
    public int id;
    public string email;

    public void ShowInfo()
    {
        Console.WriteLine($"Name: {name}, ID: {id}, Email: {email}");
    }
}
```

You can create and use a `Student` object like this:

```
Student s1 = new Student();
s1.name = "John";
s1.id = 101;
s1.email = "john@example.com";
s1.ShowInfo();
```

Output:

```
Name: John, ID: 101, Email: john@example.com
```

What is an object?

Answer:
An object is an instance of a class. It is created based on the structure defined by the class and holds real values in its fields. Using an object, we can access the class's methods and properties.

For example, using the `Student` class:

```
public class Student
{
    public string name;
    public int id;
    public string email;

    public void ShowInfo()
    {
        Console.WriteLine($"Name: {name}, ID: {id}, Email: {email}");
```

```
    }
}
```

We can create an object like this:

```
Student s1 = new Student();
s1.name = "Alice";
s1.id = 102;
s1.email = "alice@example.com";
s1.ShowInfo();
```

Output:

```
Name: Alice, ID: 102, Email: alice@example.com
```

Explanation:
Here, `s1` is an object of the `Student` class. It holds data for a specific student and can call the `ShowInfo()` method to display that data.

What is a Constructor?

Answer:
A constructor is a special method in a class that is automatically called when an object is created. Its purpose is to initialize the object's data (fields). A constructor has the same name as the class and does not have any return type—not even `void`.

There are two types of constructors:

- **Default Constructor**: Takes no parameters.
- **Parameterized Constructor**: Takes parameters to set values during object creation.

Example with Default Constructor:

```
public class Student
{
    public string name;
    public int id;

    // Default constructor
    public Student()
    {
        name = "Unknown";
        id = 0;
    }

    public void ShowInfo()
    {
        Console.WriteLine($"Name: {name}, ID: {id}");
    }
}
```

Usage:

```
Student s1 = new Student();
s1.ShowInfo();
```

Output:

```
Name: Unknown, ID: 0
```

Example with Parameterized Constructor:

```
public class Student
{
    public string name;
    public int id;

    // Parameterized constructor
    public Student(string name, int id)
    {
        this.name = name;
        this.id = id;
    }

    public void ShowInfo()
    {
        Console.WriteLine($"Name: {name}, ID: {id}");
    }
}
```

Usage:

```
Student s2 = new Student("Karim", 104);
s2.ShowInfo();
```

Output:

```
Name: Karim, ID: 104
```

Explanation:
The constructor allows you to automatically set values when the object is created, making code cleaner and more reliable.

What is an Access Modifier?

Answer:
Access modifiers in OOP define the **visibility and accessibility** of classes, methods, and variables. They help enforce encapsulation by controlling where a member can be accessed from.

Common Access Modifiers:

Modifier	Description
`public`	Accessible from **anywhere** in the program.

`private`	Accessible **only within the same class**.
`protected`	Accessible within the **same class and its derived classes**.

Example:

```
public class Student
{
    public string name;          // Can be accessed from anywhere
    private int id;              // Can only be accessed inside this class
    protected string email;      // Can be accessed in this class and derived classes

    public void SetId(int id)
    {
        this.id = id;
    }

    public void ShowInfo()
    {
        Console.WriteLine($"Name: {name}, ID: {id}, Email: {email}");
    }
}
```

Usage:

```
Student s1 = new Student();
s1.name = "Tanvir";      // OK (public)
```

```
// s1.id = 123;          // ❌ Error (private)
// s1.email = "a@b.com"; // ❌ Error (protected)

s1.SetId(123);           // OK (uses method to set private field)
s1.ShowInfo();
```

Output:

```
Name: Tanvir, ID: 123, Email:
```

Explanation:
Access modifiers help restrict access to data and methods, increasing **security, reusability, and modularity** of your code.

What is Encapsulation?

Answer:
Encapsulation is one of the core principles of object-oriented programming. It is the process of **hiding the internal details** of an object and **only exposing necessary parts** through methods or properties. This is typically done by making fields `private` and providing `public` getters and setters to access and update them.

Encapsulation helps in:

- Protecting data from unauthorized access.
- Making the code easier to maintain and modify.

Example:

```
public class Student
{
    private string name;
    private int id;
    private string email;

    public void SetStudentInfo(string name, int id, string email)
    {
        this.name = name;
        this.id = id;
        this.email = email;
    }

    public void ShowInfo()
    {
        Console.WriteLine($"Name: {name}, ID: {id}, Email: {email}");
    }
}
```

Usage:

```
Student s1 = new Student();
s1.SetStudentInfo("Rahim", 103, "rahim@example.com");
s1.ShowInfo();
```

Output:

```
Name: Rahim, ID: 103, Email: rahim@example.com
```

Explanation:
Here, the internal data of the `Student` class is hidden from direct access. Instead, it can only be modified and retrieved using the provided methods, which is the essence of encapsulation.

What are Setter and Getter?

Answer:
Setters and **getters** are user-defined methods that allow controlled access to private fields in a class.

- A **setter** sets or updates the value of a private field.
- A **getter** returns the value of a private field.
  This helps in achieving **encapsulation** by hiding internal data from direct access.

Example: Using User-Defined Setter and Getter

```
public class Student
{
    private string name;
    private int id;

    // Setter for name
```

```
    public void SetName(string name)
    {
        this.name = name;
    }

    // Getter for name
    public string GetName()
    {
        return name;
    }

    // Setter for id
    public void SetId(int id)
    {
        this.id = id;
    }

    // Getter for id
    public int GetId()
    {
        return id;
    }

    public void ShowInfo()
    {
        Console.WriteLine($"Name: {name}, ID: {id}");
    }
}
```

Usage:

```
Student s1 = new Student();
s1.SetName("Amin");
s1.SetId(106);

Console.WriteLine("Name: " + s1.GetName());
Console.WriteLine("ID: " + s1.GetId());
```

Output:

```
Name: Amin
ID: 106
```

Explanation:
Instead of giving direct access to `name` and `id`, we control it using `SetName()`, `SetId()`, `GetName()`, and `GetId()` methods. This is useful for validation, formatting, or security before reading/writing data.

What is Inheritance?

Answer:
Inheritance is a feature of object-oriented programming where a **child class** inherits properties and methods from a **parent class**. It allows code reuse and helps organize related classes in a hierarchy.

Example:

Suppose we have a base `Course` class and two specialized courses, `JobInterviewCourse` and `CompetitiveProgrammingCourse`, which extend the `Course` class.

```
// Base class
public class Course
{
    public string title;
    public int duration;      // in hours
    public double courseFee;

    public void EnrollCourse()
    {
        Console.WriteLine($"Enrolled in {title} course.");
    }

    public void ShowInfo()
    {
        Console.WriteLine($"Course: {title}, Duration: {duration} hours, Fee: {courseFee} USD");
    }
}

// Derived class 1
public class JobInterviewCourse : Course
{
```

```
    public void PrepareInterview()
    {
        Console.WriteLine("Preparing for job interviews...");
    }
}

// Derived class 2
public class CompetitiveProgrammingCourse : Course
{
    public void PracticeProblems()
    {
        Console.WriteLine("Practicing competitive programming problems...");
    }
}
```

Usage:

```
JobInterviewCourse jobCourse = new JobInterviewCourse();
jobCourse.title = "Job Interview Preparation";
jobCourse.duration = 40;
jobCourse.courseFee = 300;
jobCourse.EnrollCourse();
jobCourse.ShowInfo();
jobCourse.PrepareInterview();

CompetitiveProgrammingCourse cpCourse = new CompetitiveProgrammingCourse();
cpCourse.title = "Competitive Programming";
```

```
cpCourse.duration = 60;
cpCourse.courseFee = 500;
cpCourse.EnrollCourse();
cpCourse.ShowInfo();
cpCourse.PracticeProblems();
```

Output:

```
Enrolled in Job Interview Preparation course.
Course: Job Interview Preparation, Duration: 40 hours, Fee: 300 USD
Preparing for job interviews...

Enrolled in Competitive Programming course.
Course: Competitive Programming, Duration: 60 hours, Fee: 500 USD
Practicing competitive programming problems...
```

Explanation:

- `JobInterviewCourse` and `CompetitiveProgrammingCourse` inherit fields and methods from `Course` (`title`, `duration`, `courseFee`, `EnrollCourse()`, `ShowInfo()`).
- They also add their own specific behaviors like `PrepareInterview()` and `PracticeProblems()`.

9. What is Polymorphism?

Answer:
Polymorphism means **“many forms.”** It allows a method or object to behave differently depending on how it is used. There are two common types of polymorphism:

- **Method Overloading (Compile-time Polymorphism):** Same method name with different parameters in the same class.
- **Method Overriding (Runtime Polymorphism):** A derived class changes the behavior of a method inherited from the base class.

Example of Method Overloading:

```
public class Course
{
    public void ShowInfo()
    {
        Console.WriteLine("This is a course.");
    }

    // Overloaded method with one parameter
    public void ShowInfo(string title)
    {
        Console.WriteLine($"Course Title: {title}");
    }

    // Overloaded method with two parameters
    public void ShowInfo(string title, int duration)
```

```
    {
        Console.WriteLine($"Course Title: {title}, Duration: {duration} hours");
    }
}
```

Example of Method Overriding:

```
public class Course
{
    public virtual void ShowDetails()
    {
        Console.WriteLine("General Course details");
    }
}

public class JobInterviewCourse : Course
{
    public override void ShowDetails()
    {
        Console.WriteLine("Job Interview Course - Learn how to prepare for interviews.");
    }
}

public class CompetitiveProgrammingCourse : Course
{
    public override void ShowDetails()
```

```
    {
        Console.WriteLine("Competitive Programming Course - Practice algorithms and
problem-solving.");
    }
}
```

Usage:

```
// Method Overloading
Course c = new Course();
c.ShowInfo();
c.ShowInfo("Math 101");
c.ShowInfo("Math 101", 40);

// Method Overriding (polymorphism)
Course course1 = new JobInterviewCourse();
Course course2 = new CompetitiveProgrammingCourse();

course1.ShowDetails();  // Calls JobInterviewCourse version
course2.ShowDetails();  // Calls CompetitiveProgrammingCourse version
```

Output:

```
This is a course.
Course Title: Math 101
Course Title: Math 101, Duration: 40 hours
```

```
Job Interview Course - Learn how to prepare for interviews.
Competitive Programming Course - Practice algorithms and problem-solving.
```

Simple Interview Explanation:

“Polymorphism means one thing in many forms. Method overloading is when a method has the same name but different inputs. Method overriding is when a child class changes how a method from the parent class works.”

Why is Method Overloading Called Compile-time Polymorphism?

Answer:
Method Overloading is called **compile-time polymorphism** because the decision about which method to call is made by the **compiler** during the program compilation, not at runtime.

When you call an overloaded method, the compiler looks at the **method name** and the **number or types of arguments** to decide exactly which method version to use. This process is called **static binding** or **early binding** because it happens before the program runs.

Example:

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
```

```
    }

    public double Add(double a, double b)
    {
        return a + b;
    }
}
```

When you call:

```
Calculator calc = new Calculator();
calc.Add(5, 10);       // Calls Add(int, int)
calc.Add(5.5, 3.2);    // Calls Add(double, double)
```

The compiler knows **at compile time** which `Add` method to use based on the arguments you provide.

Summary:

- Method Overloading means multiple methods with the same name but different parameters.
- The compiler decides which method to call **before the program runs**.
- Because of this, method overloading is called **compile-time polymorphism** or **static polymorphism**.

What is Static Binding?

Answer:
Static binding (also called **early binding**) is the process where the compiler determines **which method or function to call** at **compile time**, before the program runs.

This happens when the method call is resolved based on the **reference type** and the **method signature** is known in advance. Static binding is used with:

- Method overloading
- Non-virtual methods

Example:

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public int Multiply(int a, int b)
    {
        return a * b;
    }
}
```

```
Calculator calc = new Calculator();
int sum = calc.Add(5, 3);          // Compiler knows exactly which Add to call
int product = calc.Multiply(4, 2); // Compiler knows exactly which Multiply to call
```

The compiler **fixes** the method calls (`Add` and `Multiply`) during compilation itself — this is static binding.

Summary:

- Static binding means method calls are resolved during compilation.
- It makes program execution faster because no method lookup is needed at runtime.
- It is the opposite of **dynamic binding** (runtime binding), where method calls are resolved during program execution.

Why is Method Overriding Called Runtime Polymorphism?

Answer:
Method overriding is called **runtime polymorphism** because the decision about **which method to call** is made at **runtime (when the program is running)**, not during compilation.

When a base class reference points to a derived class object, the overridden method in the derived class is called **dynamically** based on the actual object type during execution. This is known as **dynamic binding** or **late binding**.

Example:

```
public class Course
{
    public virtual void ShowDetails()
    {
        Console.WriteLine("General Course details");
    }
}

public class JobInterviewCourse : Course
{
    public override void ShowDetails()
    {
        Console.WriteLine("Job Interview Course details");
    }
}


Course course = new JobInterviewCourse();
course.ShowDetails();  // Calls JobInterviewCourse's ShowDetails() at runtime
```

Even though `course` is declared as a `Course` type, the actual method called depends on the **runtime type** (`JobInterviewCourse`), not the compile-time type. That's why it is called **runtime polymorphism**.

Summary:

- Method overriding allows a child class to provide its own version of a method.
- The method call is resolved **during program execution**, based on the object's actual type.
- This behavior is called **runtime polymorphism** or **dynamic binding**.

13. What is Dynamic Binding?

Answer:
Dynamic binding (also called **late binding**) is the process where the decision about **which method to call** is made **at runtime** instead of compile time.

This happens in situations like **method overriding**, where a base class reference refers to a derived class object, and the program decides which overridden method to execute based on the **actual object type** during execution.

Example:

```
public class Course
{
    public virtual void ShowDetails()
    {
        Console.WriteLine("General course details");
    }
}
```

```
public class CompetitiveProgrammingCourse : Course
{
    public override void ShowDetails()
    {
        Console.WriteLine("Competitive Programming course details");
    }
}


Course course = new CompetitiveProgrammingCourse();
course.ShowDetails();  // Calls CompetitiveProgrammingCourse's ShowDetails() at runtime
```

Even though `course` is declared as type `Course`, the overridden method in `CompetitiveProgrammingCourse` is called because the decision happens at runtime.

Summary:

- Dynamic binding means the program decides **which method to call while running**.
- It enables **runtime polymorphism** and allows flexible and extendable programs.
- It contrasts with **static binding**, where the method call is decided at compile time.

Difference Between Method Overloading and Method Overriding (Interview Style):

Answer: Method Overloading is when you have **multiple methods with the same name but different parameters** in the same class. It allows you to perform similar actions but with different inputs. The compiler decides which method to call based on the arguments you provide. This is called **compile-time polymorphism**.

Example:

```
void Add(int a, int b) { }
void Add(double a, double b) { }
```

Method Overriding is when a **child class provides its own version of a method** that is already defined in the parent class. This lets the child class change or extend the behavior of that method. The method to call is decided at runtime based on the actual object type. This is called **runtime polymorphism**.

Example:

```
public class Parent {
    public virtual void Show() { Console.WriteLine("Parent"); }
}

public class Child : Parent {
    public override void Show() { Console.WriteLine("Child"); }
}
```

In short:

- Overloading = same method name, different parameters, same class, decided at compile time.

- Overriding = same method signature, different class (child overrides parent), decided at runtime.

What is a Virtual Method?

A **virtual method** is a method in a base class that is marked with the `virtual` keyword, which means it is **allowed to be overridden** by derived (child) classes.

It enables **runtime polymorphism**, so when a derived class provides its own version of that method using `override`, the correct method is called depending on the actual object type at runtime.

Example:

```
public class Course
{
    public virtual void ShowDetails()
    {
        Console.WriteLine("General course details");
    }
}

public class JobInterviewCourse : Course
{
    public override void ShowDetails()
    {
        Console.WriteLine("Job Interview course details");
    }
```

```
}
```

If you call:

```
Course c = new JobInterviewCourse();
c.ShowDetails();  // Calls JobInterviewCourse's ShowDetails because it's virtual and overridden
```

In short:

A **virtual method** is a base class method that can be **overridden** in a child class to provide specific behavior, enabling **dynamic method dispatch** at runtime.

What is Abstraction?

Abstraction means **hiding the complex internal details** and **showing only the necessary features** of an object. It helps you focus on *what* an object does instead of *how* it does it.

In C#, abstraction is achieved using:

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

✅ Real-Life Example:

When you drive a car, you just use the **steering wheel, brake, and accelerator** — you don't need to know how the engine works internally. That's abstraction.

Code Example: Using Abstract Class

```
public abstract class Course
{
    public string title;

    // Abstract method - no body here
    public abstract void ShowDetails();
}

public class CompetitiveProgrammingCourse : Course
{
    // Must override the abstract method
    public override void ShowDetails()
    {
        Console.WriteLine("This is a Competitive Programming course.");
    }
}
```

Usage:

```
Course c = new CompetitiveProgrammingCourse();
c.title = "CP Bootcamp";
```

```
c.ShowDetails();
```

✅ In short (Interview style):

"Abstraction means showing only the important information and hiding unnecessary details. We use abstract classes or interfaces to define *what* should be done, and the derived class defines *how* it should be done."

What is an Abstract Class?

An **abstract class** is a class that **cannot be instantiated directly**. It may contain both:

- **Abstract methods** (methods without a body that **must be overridden** in derived classes)
- **Non-abstract methods** (methods with a body, shared among all child classes)

We use abstract classes when we want to define a **common structure** for all subclasses, but **leave the specific implementation** to the subclasses.

✅ Real-Life Example: Payment System

Let's say we are building a payment system. All payment types have an amount and transaction ID, and all must **process the payment** and **verify the details** — but the **process is different** for each type (Credit Card or Bkash).

Code Example:

```
// Abstract base class
public abstract class CoursePayment
{
    public double amount;
    public string transactionId;

    // Abstract methods – must be implemented in child classes
    public abstract void ProcessPayment();
    public abstract void VerifyDetails();

    // Common method
    public void ShowInfo()
    {
        Console.WriteLine($"Amount: {amount}, Transaction ID: {transactionId}");
    }
}
```

Derived Class 1: Credit Card Payment

```
public class CreditCardPayment : CoursePayment
{
    public string creditCardNumber;

    public override void ProcessPayment()
    {
        Console.WriteLine("Processing payment through Credit Card...");
```

```
    }

    public override void VerifyDetails()
    {
        Console.WriteLine($"Verifying credit card number: {creditCardNumber}");
    }
}
```

Derived Class 2: Bkash Payment

```
public class BkashPayment : CoursePayment
{
    public string bkashNumber;

    public override void ProcessPayment()
    {
        Console.WriteLine("Processing payment through Bkash...");
    }

    public override void VerifyDetails()
    {
        Console.WriteLine($"Verifying Bkash number: {bkashNumber}");
    }
}
```

✅ Usage:

```
CreditCardPayment ccPayment = new CreditCardPayment();
ccPayment.amount = 5000;
ccPayment.transactionId = "TXN123";
ccPayment.creditCardNumber = "1234-5678-9999-0000";
ccPayment.ShowInfo();
ccPayment.ProcessPayment();
ccPayment.VerifyDetails();

Console.WriteLine();

BkashPayment bkash = new BkashPayment();
bkash.amount = 3500;
bkash.transactionId = "BK123";
bkash.bkashNumber = "017xxxxxxx";
bkash.ShowInfo();
bkash.ProcessPayment();
bkash.VerifyDetails();
```

Output:

```
Amount: 5000, Transaction ID: TXN123
Processing payment through Credit Card...
Verifying credit card number: 1234-5678-9999-0000

Amount: 3500, Transaction ID: BK123
Processing payment through Bkash...
```

```
Verifying Bkash number: 017xxxxxxxx
```

✅ In Short (Interview Style):

"An abstract class defines a common structure but lets child classes define the specific behavior. In this example, `CoursePayment` defines that all payment types must have `ProcessPayment()` and `VerifyDetails()`, but how they do it depends on the class — Credit Card and Bkash have different ways."

What is an Interface?

An **interface** is like a **contract** that defines **what a class must do**, but **not how** it does it.

- It only contains method **signatures** (no implementation).
- A class that implements the interface **must provide the implementation** for all its methods.
- Interfaces support **multiple inheritance** (a class can implement multiple interfaces).

Interfaces are used when you want to define **common behavior** across unrelated classes.

✅ Real-Life Example (Payment System)

All payment methods (Credit Card, Bkash, etc.) must have `ProcessPayment()` and `VerifyDetails()`, but they work differently.

Instead of using a base class, we define an **interface**:

Code Example:

```
// Define interface
public interface IPayment
{
    void ProcessPayment();
    void VerifyDetails();
}
```

Class 1: CreditCardPayment implements IPayment

csharp
CopyEdit

```
public class CreditCardPayment : IPayment
{
    public string creditCardNumber;

    public void ProcessPayment()
    {
        Console.WriteLine("Processing payment using Credit Card...");
    }

    public void VerifyDetails()
    {
        Console.WriteLine($"Verifying Credit Card: {creditCardNumber}");
```

```
    }
}
```

Class 2: BkashPayment implements IPayment

```
public class BkashPayment : IPayment
{
    public string bkashNumber;

    public void ProcessPayment()
    {
        Console.WriteLine("Processing payment using Bkash...");
    }

    public void VerifyDetails()
    {
        Console.WriteLine($"Verifying Bkash Number: {bkashNumber}");
    }
}
```

Usage:

```
IPayment payment1 = new CreditCardPayment { creditCardNumber = "1234-5678-9012-3456" };
payment1.ProcessPayment();
payment1.VerifyDetails();
```

```
Console.WriteLine();

IPayment payment2 = new BkashPayment { bkashNumber = "017xxxxxxxx" };
payment2.ProcessPayment();
payment2.VerifyDetails();
```

✅ Output:

```
Processing payment using Credit Card...
Verifying Credit Card: 1234-5678-9012-3456

Processing payment using Bkash...
Verifying Bkash Number: 017xxxxxxxx
```

✅ In Short (Interview Style):

"An interface defines *what* methods a class should have, but not *how* they work. It's like a rulebook. The classes that implement the interface must provide their own implementation. It helps create flexible and decoupled code."

What is the Difference between an Abstract class and an Interface?

An abstract class is like a partially complete class. It can have both implemented methods and abstract methods. That means it can have some code already written that child classes can use. I use an abstract class when I want to share common behavior among related classes and also force them to implement specific methods.

On the other hand, an interface is like a full contract — it only has method names, no implementation. When a class implements an interface, it must write code for all the methods. I use interfaces when I just want to define a set of rules that different classes, even if they're not related, have to follow.

One main difference is that I can only inherit from one abstract class, but I can implement multiple interfaces.

So, if I need shared code with some flexibility, I go for abstract class. If I just need structure or multiple abilities like logging, saving, etc., I use interfaces.