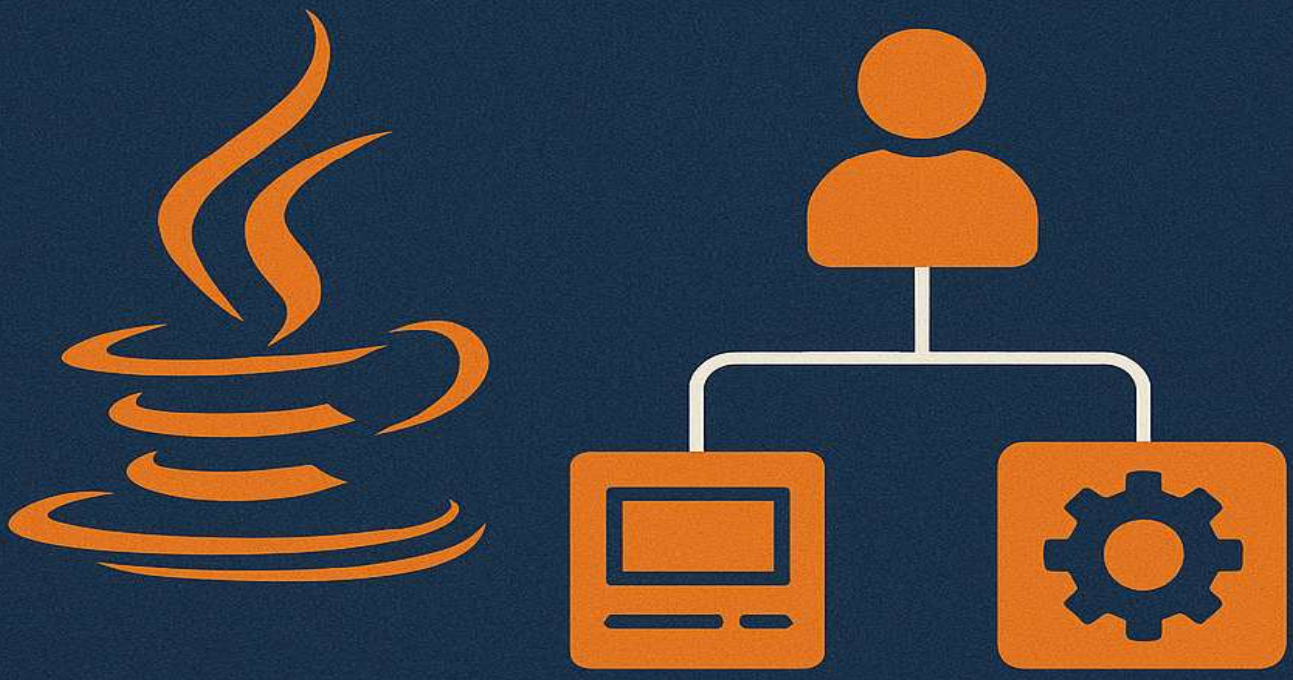
QUESTION BANK

# OBJECT-ORIENTED PROGRAMMING WITH JAVA



PREPARATION FOR THE WRITTEN TEST

# 📝 List of Possible Questions for the Test

1. 🔹 **What is an object in OOP?** Give a real-world example and explain how it translates into code.
2. 🔹 **What's the difference between a class and an object?** Use a real-world analogy.
3. 🔹 **What does it mean to say that an object is an "instance" of a class?**
4. 🔐 **What is encapsulation and why is it important?**
5. ❓ **What's the role of *get* and *set* methods?** Are there disadvantages to using them indiscriminately?
6. 🧱 **What's the role of a class? And of an object?**
7. 🔁 **What's the difference between attributes and methods in a class?**
8. 🛠️ **What is a constructor and what's its main role?**
9. 🔄 **Can a class have more than one constructor?** Give an example.
10. ❔ **What happens if we don't declare any constructor in a class?**
11. 🧬 **What is inheritance?** Give a real-world example.
12. ➕ **What is the main advantage of inheritance?**
13. 🧩 **What does it mean for a subclass to "be a type of" the superclass?** Give an example.
14. 🔧 **What's the purpose of the `super` keyword?** When is it used?
15. 🏚️ **What happens if we don't define a constructor in a subclass?**
16. 🔒 **What are *protected* members and how do they differ from *private* and *public*?**
17. ⚖️ **Explain the difference between inheritance and composition.** Which do you prefer, and why?
18. 🚫 **In what situations can inheritance be harmful?**
19. 🧱 **Why is composition often preferred over inheritance?**
20. 🤝 **What is delegation and how does it differ from inheritance?**
21. 🧠 **What is polymorphism?** Provide a clear and simple example.
22. 🚀 **How does polymorphism contribute to code extensibility?**
23. 🧮 **What's the difference between overloaded and overridden methods?**
24. 🔄 **What is *dynamic binding*?**
25. 📄 **What is an abstract class?** Can it be instantiated?
26. 🖍️ **What is an abstract method and where can it be used?**
27. 🔍 **What's the difference between an abstract class and an interface?**
28. 🧰 **When should we use interfaces instead of inheritance?**

29. 💡 **What does it mean to “program to an interface, not to an implementation”?**
30. ⚙️ **What’s the role of *default* methods in modern Java interfaces?**
31. 🔐 **What’s the role of the access modifiers `private`, `public`, and `protected`?**
32. 🔄 **What are instance variables and static variables for?**
33. 📌 **What is a static method and when should we use it?**
34. 🎈 **What’s the purpose of the `this` keyword?**
35. 🧠 **What does it mean to say that "objects have state"?**
36. 🎭 **What’s the difference between state and behavior in an object?**
37. ⚠️ **What is the `instanceof` operator and what precautions should be taken?**
38. 📬 **What’s the advantage of thinking in terms of messages between objects?**
39. 🧰 **What are utility classes and what are they used for?**
40. 🚪 **Why should we hide implementation details?**
41. 🔗 **What’s the difference between coupling and cohesion?** How do they affect code quality?
42. 🧩 **What kind of problems can arise if we ignore modularity?**
43. ♻️ **Why is code reuse considered good practice?** Give an example.
44. 🏛️ **Explain the Open/Closed Principle in object-oriented design.**
45. 🔧 **What’s the importance of encapsulation for software maintenance and evolution?**
46. 🚫 **Why can using public members indiscriminately be problematic?**
47. ✂️ **What’s the advantage of separating implementation from interface?**
48. 🎯 **Why should we build classes with well-defined responsibilities?**
49. 🧭 **What does responsibility mean in an object?**
50. ❓ **Can an object change its class at runtime?** Justify

The following are suggested answers.

🧠 **1. What is an object in OOP? Give a real-world example and explain how it translates into code.**

An object is an entity with its own identity, that has state (attributes), behavior (methods), and interacts with other objects.

💡 **Real-world example**: imagine a *Car*. It has color, brand, model (state) and it can accelerate, brake, shift gears (behaviors).

🧪 **In Java**:

```
Car myCar = new Car();
myCar.color = "red";
myCar.accelerate();
```

🎯 **Summary**: an object is like a concrete "thing" created from a recipe (the class). It lives in the computer's memory and is what the program interacts with.

---

🧠 **2. What's the difference between a class and an object? Use a real-world analogy.**

Think of a *cookie cutter* (class) and the actual cookies made with it (objects).

• The class defines how something is: its attributes and behaviors.

• The object is a real instance, with actual values.

🛠️ **Java example**:

```
class Person {
    String name;
    void sayHello() {
        System.out.println("Hello!");
    }
}

Person maria = new Person(); // Object
```

👉 Class = blueprint or mold.

👉 Object = the real product we interact with.

---

### 🧠 3. What does it mean to say that an object is an “instance” of a class?

It means the object was created based on the class definition — in other words, it was instantiated.

📦 The class is like a cake recipe.

🎂 The instance (the object) is the actual cake made with that recipe.

🧪 **In Java**:

```
Book b1 = new Book(); // b1 is an instance of class Book
```

Every time you use `new`, you're instantiating — that is, creating a real object based on the class.

---

### 🧠 4. What is encapsulation and why is it important?

🔒 Encapsulation is the principle of hiding an object’s internal details, revealing only what’s necessary for external use.

🎯 **Why?**

• Protects data from accidental or malicious changes.
• Ensures the object is used correctly.
• Allows internal changes without affecting users of the object.

🧪 **Example**:

```
class BankAccount {
    private double balance;

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) balance += amount;
    }
}
```

Here, `balance` is protected. It can only be read or modified through the `getBalance()` and `deposit()` methods.

---

🧠 **5. What's the role of get and set methods? Are there disadvantages to using them indiscriminately?**

🕹️ Get/set methods are used to access (get) or modify (set) private attributes in a controlled way.

🛑 But be careful: if you use get and set for *every* attribute without adding extra logic, you're just pretending to encapsulate — you're exposing everything anyway!

🧪 **Example with useful logic**:

```
public void setAge(int age) {
    if (age >= 0) this.age = age;
}
```

🎯 **Summary**: Use get/set when needed, but add rules to ensure the object stays in a valid state.

---

🧠 **6. What's the role of a class? And of an object?**

📦 **Class** = defines what the object can do and what data it can have. It's the structure.

🧍 **Object** = the actual thing, with concrete data and active behavior, created from the class.

💬 **Analogy**:

• The class "Person" says all people have a name and can speak.
• The object "Ana" has the name "Ana" and can say "Hello!".

---

🧠 **7. What's the difference between attributes and methods in a class?**

🧬 **Attributes** (or instance variables): store the object's state (e.g., name, age).

⚙️ **Methods**: define behaviors (actions the object can perform).

🧪 **Example**:

```
class Cat {
    String name;          // attribute
    void meow() {         // method
        System.out.println("Meow!");
    }
}
```

🎯 Attributes are data; methods are actions.

---

### 🧠 8. What is a constructor and what's its main role?

🧱 A constructor is a special method called when an object is created (`new`), whose role is to initialize the object's attributes.

🧪 **Example**:

```
class Book {
    String title;

    Book(String initialTitle) {
        title = initialTitle;
    }
}
```

When you write `new Book("Don Quixote")`, the constructor immediately sets the title value.

---

### 🧠 9. Can a class have more than one constructor? Give an example.

Yes! This is called **constructor overloading**. You can have several as long as they have different parameters.

🧪 **Example**:

```
class Person {
    String name;
    int age;

    Person(String name) {
        this.name = name;
    }

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

🧠 This is useful to provide different options when creating objects.

---

### 🧠 10. What happens if we don't declare any constructor in a class?

If no constructor is declared, Java automatically provides a default constructor (with no parameters).

🧪 **Example**:

```
class Animal {
    // no constructor defined
}
```

You can do:

```
Animal a = new Animal(); // valid!
```

❗ But if you define *any* constructor, the default one is no longer generated. You must create it yourself if needed.

---

### 🧠 11. What is inheritance? Give a real-world example.

👑 Inheritance is when a class (the subclass) reuses code and behaviors from another (the superclass), while also adding or modifying features.

💡 **Real-world example**:

• *Vehicle* is a generic class.
• *Car*, *Motorcycle*, and *Truck* can inherit from *Vehicle*.

🧪 **In Java**:

```
class Vehicle {
    void move() { System.out.println("Moving..."); }
}

class Car extends Vehicle {
    void honk() { System.out.println("Honk!"); }
}
```

*Car* inherits the `move()` method from *Vehicle* but can also have its own methods.

---

🧠 **12. What is the main advantage of inheritance?**

🚀 **Code reuse**. It avoids duplication, simplifies maintenance, and helps build complex systems more cleanly.

🎯 With inheritance, we can create a hierarchical structure where common code is in one place (superclass), and subclasses inherit or adapt it as needed.

---

🧠 **13. What does it mean for a subclass to "be a type of" the superclass? Give an example.**

It's the basis of the "is-a" principle. If *Car* inherits from *Vehicle*, then a car *is a* vehicle.

🧪 **In Java**:

```
Vehicle v = new Car(); // allowed!
```

✅ This enables **polymorphism**, where we treat different objects in a uniform way as long as they share the same base type.

---

🧠 **14. What's the purpose of the `super` keyword? When is it used?**

🧠 `super` is used to:

51. Access superclass members that have been overridden.
52. Call the superclass constructor.

🧪 **Example**:

```
class Animal {
    Animal(String name) {
        System.out.println("Animal: " + name);
    }
}

class Cat extends Animal {
    Cat() {
        super("Felix"); // calls the superclass constructor
    }
}
```

You can also use `super.method()` to call superclass methods that have been overridden.

---

🧠 **15. What happens if we don't define a constructor in a subclass?**
If the subclass doesn't have a constructor, Java tries to automatically call the superclass's default constructor (`super()`).
❗ **Common mistake**: if the superclass doesn't have a no-argument constructor, and we don't explicitly call `super(...)`, the code won't compile.

Com todo o gosto, Luís! Aqui vai a continuação da tradução fiel, mantendo o estilo original, emojis e clareza pedagógica:

🧠 **16. What are protected members and how do they differ from private and public?**
🔐 Access modifiers control who can see or modify a class's members:

| Modifier | Visible to... |
|---|---|
| `private` | Only within the class itself |
| `protected` | Within the class and its subclasses |
| `public` | From any class |

🎯 `protected` is useful when we want to give limited access to subclasses, without fully exposing members like `public` does.

---

🧠 **17. Explain the difference between inheritance and composition. Which do you prefer, and why?**
🧬 **Inheritance**: creates an "is-a" relationship.
🧩 **Composition**: used when a class contains another as part of its functionality ("has-a").
💡 **Example**:
• *Car* inherits from *Vehicle* (inheritance).
• *Car* has an *Engine* (composition).
🎯 In many cases, **composition is preferred** because:
• It avoids rigid dependencies.
• It offers more flexibility and reusability.

---

### 🧠 18. In what situations can inheritance be harmful?

🚫 When used carelessly, it can:

• Create rigid and hard-to-maintain structures.
• Expose unnecessary or undesired behaviors.
• Make the system harder to test and evolve.

**Bad practice**: using inheritance just to "reuse code" when **composition** or **delegation** would be more appropriate.

---

### 🧠 19. Why is composition often preferred over inheritance?

Because composition favors **encapsulation** and **flexibility**. We can swap components without changing the entire structure.

💡 **Analogy**: if a car has an engine, we can replace the engine without redesigning the whole car.

**In Java**:

```
class Engine { void start() { System.out.println("Engine started"); } }

class Car {
    private Engine engine = new Engine();
    void start() { engine.start(); }
}
```

✅ This pattern makes the system more **modular** and **evolvable**.

🧠 **20. What is delegation and how does it differ from inheritance?**

🧭 **Delegation** is when an object relies on another class instance to perform a task.

⚙️ It differs from inheritance because:

• There's no *is-a* relationship — it's *has-a*.

• We use composition to "pass" responsibilities.

💡 **Example**:

```
class Printer {
    void print(String text) {
        System.out.println(text);
    }
}

class Document {
    private Printer printer = new Printer();
    void print() {
        printer.print("Document content");
    }
}
```

📌 The `Document` delegates the printing task to `Printer`.

---

🧠 **21. What is polymorphism? Give a clear and simple example.**

🎭 Polymorphism means “many forms”. In OOP, it refers to the ability to treat objects of different classes as if they were the same type — as long as they share a common interface.

💡 **Example**: different animals can make sounds, but each in their own way:

```
class Animal {
    void makeSound() {
        System.out.println("Generic sound");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Meow");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof");
    }
}
```

Even using `Animal a = new Cat();`, the correct method (`Meow`) is called. That’s polymorphism in action!

---

🧠 **22. How does polymorphism contribute to code extensibility?**

🚀 It allows code to:

• Work with generic types (e.g., `Animal`).

• Be easily extended with new classes (e.g., `Parrot`) without changing existing code.

🎯 It makes the system more **open to extension and closed to modification** (Open/Closed Principle).

---

🧠 **23. What's the difference between overloaded and overridden methods?**

🧪 **Overloading**: methods with the same name but different parameters, in the **same class**.

🧪 **Overriding**: a subclass redefines a method it inherited from a superclass.

📌 **Overloading example**:

```
void show(String s) { ... }
void show(int n) { ... }
```

📌 **Overriding example**:

```
class Parent {
    void greet() { System.out.println("Hello from parent!"); }
}

class Child extends Parent {
    @Override
    void greet() { System.out.println("Hello from child!"); }
}
```

---

🧠 **24. What is dynamic binding?**

🧠 It's the ability to decide **at runtime** which method to call, depending on the actual object type.

💡 **Example with polymorphism**:

```
Animal a = new Cat();
a.makeSound(); // Executes Cat's makeSound(), not Animal's
```

🎯 The decision is based not on the variable type, but on the actual object in memory. This allows for more flexible and adaptive behavior.

---

### 🧠 25. What is an abstract class? Can it be instantiated?

📦 An **abstract class** is an incomplete model that cannot be instantiated directly. It serves as a base for other classes.

🧪 **Example**:

```
abstract class Shape {
    abstract void draw();
}
```

Only concrete classes that extend `Shape` and implement `draw()` can be instantiated.

❌ This is not allowed:

```
Shape s = new Shape(); // Error!
```

---

### 🧠 26. What is an abstract method and where can it be used?

🔧 An abstract method is a method without a body, which **must** be implemented by subclasses.
It can only exist in an **abstract class**.

🧪 **Example**:

```
abstract class Animal {
    abstract void makeSound(); // no implementation
}
```

Subclasses must implement `makeSound()`.

---

### 🧠 27. What's the difference between an abstract class and an interface?

| Feature | Abstract Class | Interface |
|---|---|---|
| Can have attributes | Yes (with state) | Yes (since Java 8, with limits) |
| Methods with body | Yes | Yes (since Java 8, with `default`) |
| Multiple inheritance | No | Yes (can implement many) |
| Main purpose | Reuse behavior | Define contracts (what must be done) |

💡 Use an **abstract class** when you want to provide some common implementation.
Use an **interface** when you only want to define what must be done, without specifying how.

---

🧠 **28. When should we use interfaces instead of inheritance?**

📌 When we want multiple classes to share a set of methods — even if they're not related by inheritance.

💡 **Example**: both a `Document` and an `Image` can be *printable* (i.e., `Printable`), even if they don't share a common superclass.

```
interface Printable {
    void print();
}
```

🎯 Interfaces help create **decoupled, flexible, and testable** code.

---

🧠 **29. What does it mean to "program to an interface, not to an implementation"?**

💭 It means using **generic types (interfaces)** instead of concrete ones. This makes it easier to switch implementations.

📌 Instead of this:

```
ArrayList<String> list = new ArrayList<>();
```

Do this:

```
List<String> list = new ArrayList<>();
```

That way, if you want to switch to `LinkedList`, you only change the constructor — the rest of the code stays the same!

---

### 🧠 30. What's the role of default methods in modern Java interfaces?

🧠 Since Java 8, **default methods** allow interfaces to include implementations, without breaking older classes that use them.

💡 They're useful for evolving APIs without forcing all classes to implement new methods.

🧪 **Example**:

```
interface Healthy {
    default void breathe() {
        System.out.println("Breathing is essential!");
    }
}
```

A class that implements `Healthy` inherits the `breathe()` method by default.

---

### 🧠 31. What's the role of the access modifiers private, public, and protected?

🔐 Access modifiers control **who can access what** in a class:

| Modifier | Visible to… |
|---|---|
| `private` | Only within the class itself |
| `public` | Anywhere in the program |
| `protected` | Same class, subclasses, and same package |

💡 **Example**:

```
class Person {
    private String name; // only accessible inside the class
    public void introduce() {
        System.out.println("Hi, I'm " + name);
    }
}
```

🎯 Using `private` protects the data, and `public` creates safe interfaces for interaction.

---

🧠 **32. What are instance variables and static variables for?**

🧬 **Instance variables** are unique per object — each instance has its own copy.

⚙️ **Static variables** (using `static`) are shared by all objects of the class.

💡 **Example**:

```
class Student {
    String name;             // instance variable
    static int totalStudents; // class variable (static)
}
```

Each student has a different name, but all share the same `totalStudents`.

---

🧠 **33. What is a static method and when should we use it?**

🔧 A **static method** belongs to the class, not to any specific object. It can be called without creating an instance.

💡 **Classic example**:

```
Math.sqrt(25); // static method from the Math class
```

✅ Use it when:

• The operation doesn't depend on an object's state.
• You want to create utility methods (e.g., converters, calculators, etc.).

Com todo o gosto! Aqui está a tradução para inglês desta última parte, mantendo o tom didático, emojis, clareza e estilo consistente:

---

### 🧠 34. What's the purpose of the `this` keyword?

📌 `this` refers to the current object — the one executing the method.

It's used when:

• We want to distinguish attributes from parameters with the same name.

• We call another constructor from the same class.

💡 **Example**:

```
class Person {
    String name;

    Person(String name) {
        this.name = name; // 'this.name' refers to the attribute
    }
}
```

---

### 🧠 35. What does it mean to say that "objects have state"?

📦 An object's **state** is the set of values stored in its attributes.

💡 **Example**:

```
Car c = new Car();
c.brand = "Renault";
c.speed = 60;
```

In this case, the car's state is: brand = "Renault", speed = 60.

🎯 The state changes over time, as methods act on attributes (e.g., `accelerate()` changes speed).

---

🧠 **36. What's the difference between state and behavior in an object?**

📌 **State**: the data (attributes).

📌 **Behavior**: the actions (methods).

💡 **Example from a Lamp class**:

```
class Lamp {
    boolean isOn; // state

    void turnOn() { isOn = true; }   // behavior
    void turnOff() { isOn = false; }
}
```

🎯 Behavior changes the state — it's this dynamic that brings the object to life!

---

🧠 **37. What is the `instanceof` operator and what precautions should be taken?**

🔍 `instanceof` checks if an object belongs to a class (or subclass).

💡 **Example**:

```
if (a instanceof Cat) {
    System.out.println("It's a cat!");
}
```

⚠️ **Be careful**:

• Overusing `instanceof` can violate the principle of polymorphism.

• Instead, we often prefer polymorphic methods (e.g., `animal.makeSound()` without knowing if it's a Cat or Dog).

---

🧠 **Summary of this section**:

| Concept | Essence |
|---|---|
| private/public/protected | Control access |
| static | Shared by all objects |
| this | Refers to the current object |
| State vs Behavior | Data vs Actions |
| instanceof | Checks object type |

---

🧠 **38. What's the advantage of thinking in terms of messages between objects?**

💌 In OOP, instead of seeing objects as "data boxes", we see them as autonomous entities that exchange **messages**.

🎯 This promotes:

• Low coupling.
• High cohesion.
• Flexibility and reuse.

💡 **Real-life example**:

A customer orders a meal from a restaurant, without knowing how the chef prepares it. Only the result matters.

📌 In code, this translates to methods representing those "messages":

```
customer.order("Pizza Margherita");
```

---

🧠 **39. What are utility classes and what are they used for?**

🧰 **Utility classes** are collections of `static` methods that provide general-purpose services and don't require instances.

💡 **Classic example**: `Math` in Java.

```
double root = Math.sqrt(25);
```

📌 You can create your own:

```
class Converter {
    public static double celsiusToFahrenheit(double c) {
        return c * 1.8 + 32;
    }
}
```

🎯 They're useful, but should remain simple and stateless. For behaviors involving state, use objects.

---

### 🧠 40. Why should we hide implementation details?

🔐 Hiding technical details prevents users of a class from:

• Becoming dependent on how it's built.
• Accessing sensitive data directly.
• Breaking the code through careless changes.

💡 Just like we use a microwave without knowing its circuit, we should use a class without needing to know how its methods work.

🎯 This makes software more **robust**, **flexible**, and **easier to maintain**.

---

### 🧠 41. What's the difference between coupling and cohesion? How do they affect code quality?

| Concept | Simple Definition | Good or Bad? |
|---|---|---|
| Coupling | Degree of dependency between classes | The lower, the better |
| Cohesion | Degree of focus of a class | The higher, the better |

💡 A class with **high cohesion** does one thing, but does it well.

💡 Classes with **low coupling** work independently and don't rely too much on others.

🎯 Result: a **modular**, **testable**, and **extensible** system.

---

### 🧠 42. What kind of problems can arise if we ignore modularity?

🚨 Ignoring modularity leads to:

• Code that's hard to understand.
• Changes breaking unrelated parts of the system.
• Near-impossible reuse.

📌 It's like a giant cake with all ingredients mixed — if something's wrong, you can't tell what to fix.

🎯 With modularity, each "ingredient" is in its own labeled jar — easier to use, test, and swap.

---

### 🧠 43. Why is code reuse considered good practice? Give an example.

♻️ Reusing code:

• Avoids duplication.
• Reduces errors.
• Increases productivity.
• Promotes consistency.

💡 **Example**:

```
class EmailValidator {
    public static boolean validate(String email) {
        return email.contains("@");
    }
}
```

🎯 This method can be used in forms, APIs, mobile apps... without reinventing the wheel!

---

### 🧠 44. Explain the Open/Closed principle in object-oriented design.

🔐 This principle (the "O" in SOLID) states that **code should be open for extension, but closed for modification**.

💡 In other words: you can add new behavior, but you shouldn't need to change existing code.

📌 **Example**: Instead of using `if` with many types, use polymorphism:

```
interface Payment {
    void process();
}

class Card implements Payment {
    public void process() { ... }
}

class MBWay implements Payment {
    public void process() { ... }
}

// Later
Payment p = new MBWay();
p.process(); // We don't care how — we just know it processes!
```

---

### 🧠 45. What's the importance of encapsulation for software maintenance and evolution?

🔧 Encapsulation protects a class's data and internal logic, allowing:
• Safe changes.
• Minimal impact on other system parts.
• Fewer bugs.
💡 If everyone only uses the "public interface", you can swap the engine inside without anyone noticing — just like replacing a car engine while keeping the same steering wheel and pedals.

---

### 🧠 46. Why can using public members indiscriminately be problematic?

🚫 Making everything `public` turns the object into an "open battlefield", where any code can:
• Modify data without control.
• Break internal rules of the class.
• Make the system fragile and insecure.
🎯 The idea of "privacy" in programming is essential to protect the inner workings of the object.

---

### 🧠 47. What's the advantage of separating implementation from interface?

📄 Separating **"what it does"** from **"how it does it"** allows you to:
• Replace the implementation without affecting users.
• Hide complexity.
• Test and maintain more easily.
💡 It's like a power outlet: you know where to plug in (the interface), but don't need to know how the circuit inside works (the implementation).

---

### 🧠 48. Why should we build classes with well-defined responsibilities?

📦 A class with a **single responsibility**:

• Is easier to test, understand, and evolve.
• Avoids unintended side effects.
• Follows the **Single Responsibility Principle** (the "S" in SOLID).

💡 If you have a `ClientClass` that validates and also prints invoices… that's two different things! Time to split it up.

---

### 🧠 49. What does responsibility mean in an object?

🧠 An object's **responsibility** is what it should know and what it should do.

🎯 Good design assigns responsibilities clearly and fairly.

💡 A `Clock` knows the time and can update it.
It shouldn't process salaries or print labels — that's beyond its role.

---

### 🧠 50. Can an object change its class at runtime? Justify.

❌ In Java (and most OO languages), an object **cannot** change its class after being created.

📌 The class defines the object's structure and behavior. Changing class would be like turning a banana into a wrench halfway through the program.
However, you **can**:

• Replace the object with another that implements the same interface.
• Use **polymorphism** to change the perceived behavior.

💡 **Example**:

```
Drawable shape = new Circle(); // it's a circle
shape = new Rectangle();       // now it's a rectangle
```

🎯 The reference type stays the same, but the pointed object changes — as long as it respects the contract (e.g., interface).