Practise Questions with Answers

OOPS

Q1. You are designing a **BankAccount** class with fields of**accountNumber**, **balance**, and **accountHolderName**.

- Should balance be public or not? Justify

Answer: No, balance should not be public.
It should be private.

Justification:

- If balance is public, anyone can change it directly.
- account.balance = -5000; // invalid but possible
- This breaks data security and integrity, which is critical in banking systems.
- Making it private ensures that the balance can be changed only through controlled methods.

- How would you control deposits and withdrawals using methods?

Answer: we can control access by providing public methods that validate rules before modifying the balance.
Example :

```
class BankAccount {
    private String accountNumber;
    private double balance;
    private String accountHolderName;
```

```
    // Constructor
    public BankAccount(String accountNumber, String
accountHolderName, double balance) {
        this.accountNumber = accountNumber;
        this.accountHolderName = accountHolderName;
        this.balance = balance;
    }

    // Deposit method
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        } else {
            System.out.println("Invalid deposit amount");
        }
    }

    // Withdraw method
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Insufficient balance or invalid
amount");
        }
    }

    // Getter
    public double getBalance() {
        return balance;
    }
}
```

Rules enforced here:

- Deposit amount must be positive

- Withdrawal amount must be ≤ balance
- Balance cannot become negative

- How does encapsulation help enforce banking rules?

Answer: Encapsulation helps enforce banking rules by restricting direct access to account data and allowing changes only through controlled methods that apply business validations or constraints.

Explanation (Banking Context)Data protection:
Account details like balance are kept private, so no one can directly change them.

- Rule enforcement through methods:
  Operations such as deposit() and withdraw() act as gatekeepers.
  They check rules like:
  - Withdrawal amount ≤ available balance
  - Deposit amount > 0
  - Minimum balance must be maintained
-
- Prevents invalid states:
  Encapsulation ensures the account never reaches an illegal state (e.g., negative balance).
- Centralized business logic:
  All banking rules exist in one place (inside the class), making the system:
  - Easier to maintain
  - Safer to modify

- Less error-prone

Sample Example :

```
private double balance;


public void withdraw(double amount) {
   if (amount > 0 && balance - amount >= 1000) {
      balance -= amount;
   }
}
```

Here, encapsulation ensures:

- Balance cannot be accessed directly
- Minimum balance rule is always followed

Q2. A withdrawal fails due to **insufficient balance**.

- Would you use a checked or unchecked exception?

Answer : I use unchecked exception(Runtime exception) because

- Insufficient balance is a business rule violation, not a system failure.
- It occurs at runtime based on user input or account state.
- Forcing callers to handle it using throws (checked exception) makes the code noisy and less flexible.
- Example :

```
public class InsufficientBalanceException extends RuntimeException {
   public InsufficientBalanceException(String message) {
      super(message);
   }
```

}

- Why is exception handling critical?

Answer: Exception handling is critical because it:

- Prevents application crashes
- Maintains system stability
- Provides meaningful error messages
- Ensures transaction safety (no partial updates)
- Improves user experience

Without exception handling:

- The program may terminate abruptly
- Account balance may become inconsistent
- Users won’t know why the transaction failed.

- How would a custom exception improve code clarity?

Answer : Custom exception make the code:

- More readable
- More domain-specific
- Easier to debug and maintain

Instead of this:

- throw new RuntimeException("Error");

Use this:

- throw new InsufficientBalanceException("Withdrawal failed: insufficient balance");

Benefits:

- Clearly indicates what went wrong
- Separates business errors from system errors
- Allows specific handling:
- catch (InsufficientBalanceException e) {
- System.out.println(e.getMessage());
- }

Q3. An e-commerce system supports **PhysicalProduct** and **DigitalProduct**.

- What common behaviors would you abstract

Answer : Common behaviors that apply to both product types should be abstracted, such as:

- getPrice()
- applyDiscount()
- calculateFinalPrice()
- getProductDetails()
- validateProduct()

Common properties:

- productId
- name

- price

•Would you use an abstract class or interface?
Answer : I use an abstract class
Because:

- Both products share state (fields like id, name, price)
- Both need partial implementation (shared logic like discount calculation)
- Only one product type is expected per class (no multiple inheritance needed).

Example Design

Abstract Class

```
public abstract class Product {
    protected String productId;
    protected String name;
    protected double price;

    public Product(String productId, String name, double price) {
        this.productId = productId;
        this.name = name;
        this.price = price;
    }

    public double getPrice() {
        return price;
```

```
    }

    public double calculateFinalPrice() {
        return price - applyDiscount();
    }

    protected abstract double applyDiscount();
    public abstract void deliver();
}
```

Physical Product

```
public class PhysicalProduct extends Product {

    public PhysicalProduct(String productId, String name, double price) {
        super(productId, name, price);
    }

    @Override
    protected double applyDiscount() {
        return price * 0.1;
    }

    @Override
    public void deliver() {
        System.out.println("Shipping product to address");
```

```
    }
}
```

Digital Product

```
public class DigitalProduct extends Product {

    public DigitalProduct(String productId, String name, double price) {
        super(productId, name, price);
    }

    @Override
    protected double applyDiscount() {
        return price * 0.2;
    }

    @Override
    public void deliver() {
        System.out.println("Providing download link");
    }
}
```

Q4. What is **polymorphism** in Java?

Answer: Polymorphism means "one interface, many forms."

In Java, it allows the same method call to behave differently depending on the object type at runtime.

Example :

```
class Employee {
    double calculateSalary() {
        return 30000;
    }
}

class FullTimeEmployee extends Employee {
    @Override
    double calculateSalary() {
        return 50000;
    }
}

class ContractEmployee extends Employee {
    @Override
    double calculateSalary() {
        return 20000;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Employee emp1 = new FullTimeEmployee();
        Employee emp2 = new ContractEmployee();

        System.out.println(emp1.calculateSalary()); // 50000
        System.out.println(emp2.calculateSalary()); // 20000
    }
}
```

Output : 50000

20000

Explanation

- Reference type is Employee
- Object type changes at runtime
- JVM decides which overridden method to execute
- This is runtime polymorphism.

- Explain **compile-time** vs **runtime polymorphism**.

Answer :

Compile-time Polymorphism (Static Binding)

- Achieved using method overloading

- Method resolution happens at compile time
- Same method name, different parameter list

Example : class Calculator {

```
    int add(int a, int b) {
        return a + b;
    }

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

- Decided at compile time
- Faster, but less flexible

Runtime Polymorphism (Dynamic Binding)

- Achieved using method overriding
- Method resolution happens at runtime
- Uses inheritance + method overriding

Example:class Product {

```
    void deliver() {
        System.out.println("Delivering product");
    }
```

```
}


class DigitalProduct extends Product {
    @Override
    void deliver() {
        System.out.println("Providing download link");
    }
}
```

```
Product p = new DigitalProduct();
p.deliver(); // Calls DigitalProduct's method
```

- Decided at runtime
- More flexible and extensible

- Which one is achieved using method overriding?

Answer : Runtime polymorphism is achieved using method overriding.
Method overriding allows a child class to provide its own implementation of a parent class method.

At runtime, Java decides which method to execute based on the actual object, not the reference type.

```
Employee emp = new FullTimeEmployee();
emp.calculateSalary(); // Calls FullTimeEmployee's method at runtime
```

Method overriding achieves runtime polymorphism in Java because method execution is decided at runtime based on the object type.

- Why is runtime polymorphism important in real applications.
- **Answer : Runtime polymorphism is important because it:**
  - Supports loose coupling
  - Enables dynamic behavior selection
  - Improves code extensibility
  - Follows Open–Closed Principle
  - Makes systems easier to maintain

Real-world example:

- Payment system (CreditCard, UPI, NetBanking)
- Product delivery (PhysicalProduct, DigitalProduct)
- Notification service (Email, SMS, Push)

```
Payment payment = new UPIPayment();

payment.pay();
```

Same code → different behavior at runtime.