Q1. Designing a BankAccount Class

Should `balance` be public? Why?

No, `balance` should never be public.

Reason:
If `balance` is public, anyone can directly change it:

```
account.balance = -5000;
```

This breaks banking rules.

Instead, `balance` should be private so that it can only be changed through controlled methods.

How do you control deposits and withdrawals?

By providing public methods that validate inputs before updating balance.

Example:

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

    public BankAccount(String accNo, String name, double initialBalance) {
        accountNumber = accNo;
        accountHolderName = name;
        balance = initialBalance;
    }

    public void deposit(double amount) {
        if (amount <= 0) {
            System.out.println("Invalid deposit amount");
            return;
        }
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount > balance) {
            System.out.println("Insufficient balance");
            return;
        }
        balance -= amount;
    }
    public double getBalance() {
        return balance;
    }
}
```

How does encapsulation help enforce banking rules?

Encapsulation:

- Hides internal data (balance)
- Forces users to follow rules via methods
- Prevents illegal state changes

Result:
Account balance can never be negative or tampered with directly.

Q2. Withdrawal Failure Due to Insufficient Balance

Checked or Unchecked Exception?

Unchecked Exception is preferred.

Why?

- Insufficient balance is a runtime business rule violation
- It doesn't need mandatory handling at compile time

Why is exception handling important?

Exception handling:

- Prevents application crashes
- Gives meaningful error messages
- Keeps code stable and predictable

How does a custom exception improve clarity?

Instead of throwing a generic exception, a custom exception clearly explains the problem.

Example:

```
class InsufficientBalanceException extends RuntimeException {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}
public void withdraw(double amount) {
    if (amount > balance) {
        throw new InsufficientBalanceException("Not enough balance to withdraw");
    }
    balance -= amount;
}
```

Benefit:
Anyone reading the code immediately understands what went wrong.

Q3. PhysicalProduct vs DigitalProduct

What common behaviors can be abstracted?

Common behaviors:

- getPrice()
- getProductDetails()
- applyDiscount()

Abstract Class or Interface?

Abstract class is a better choice.

Why?

- Both product types share behavior
- Both may have common fields (price, name)

Example:

```
abstract class Product {
    protected double price;

    abstract double getPrice();

    void applyDiscount(double percentage) {
        price = price - (price * percentage / 100);
    }
}
class PhysicalProduct extends Product {
    double getPrice() {
        return price + 50; // shipping charge
    }
}
class DigitalProduct extends Product {
    double getPrice() {
        return price; // no shipping
    }
}
```

Q4. Polymorphism in Java

What is polymorphism?

Polymorphism means one method call behaving differently depending on the object type at runtime.

Compile-time vs Runtime Polymorphism

Type	Achieved Using	Binding Time
Compile-time	Method Overloading	Compile time
Runtime	Method Overriding	Runtime

Which uses method overriding?

Runtime polymorphism

Why is runtime polymorphism important?

- Enables loose coupling
- Improves flexibility
- Makes code extensible
- Supports real-world behavior changes

Hands-on Example:

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

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

public class Main {
    public static void main(String[] args) {
        Employee emp = new FullTimeEmployee();
        System.out.println(emp.calculateSalary());
    }
}
```

Output:

```
50000
```

Method called depends on object type, not reference type.

Rules for Method Overriding (Interview Friendly)

1. Same method signature

```
// Overloading, NOT overriding
double calculateSalary(int bonus) { }
```

---

2. Access level cannot be reduced

Invalid:

```
protected double calculateSalary() { }
```

Valid:

```
public double calculateSalary() { }
```

---

3. Return type must be same or covariant

---

4. `@Override` annotation is recommended

- Avoids mistakes
- Improves readability