Day-18

Access Modifiers, OOPS Concept – Data Abstraction

Access Modifiers in Java

Access modifiers define the scope of accessibility for classes, attributes, methods, and constructors. They determine how different parts of your code can interact with each other.

- **public**: Accessible from everywhere.
- **protected**: Accessible within the same package and by subclasses in different packages.
- **default** (no modifier): Accessible only within the same package.
- **private**: Accessible only within the class where it is defined.

Here's a tabular representation of the access modifiers in Java:

Modifier	Within the Class	Within the Package	In Other Packages (Subclass)	Everywhere (Other classes)
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No
private	Yes	No	No	No

Abstraction in Java

- **Abstraction** is a process of **hiding the implementation** details and **showing only functionality** to the user.
- Another way, it **shows only essential things to the user** and **hides the internal details**.
- For example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- It allows you to focus on **what an object does, not how it works**.

Achieving Abstraction in Java

There are two ways to achieve abstraction:

1. **Abstract class** (0% to 100% abstraction)
2. **Interface** (100% abstraction) before Java8.

Abstract Class

- An abstract class is a class that is declared with the **abstract** keyword.
- It can contain both **abstract** (without a body) and **non-abstract** methods (with a body).
- Abstract classes **cannot be instantiated** and **must be extended by other cla**sses that provide implementations for the abstract methods.

Key Points to Remember:

- Declared using the **abstract** keyword.
- Can have both **abstract** and **non-abstract** methods.
- **Cannot** be instantiated directly.
- Can include **constructors** and **static methods**.
- Can have **final methods** to prevent subclasses from modifying them.

Abstract Method

An abstract method is a method declared without an implementation (no method body). It must be implemented by subclasses.

Syntax:

```
abstract void method();  // abstract method with no body
```

Lab Assignment: Abstraction in Java using Abstract Classes

Objective:

The objective of this lab assignment is to understand and implement abstraction in Java by creating an abstract class that contains both abstract and non-abstract methods. You will implement a real-time use case of a banking system where different types of accounts (Savings and Current) will have common and specific behaviors.

Task Overview:

You will create:

1. An abstract class (BankAccount) that defines the common structure for bank accounts.
2. Subclasses (SavingsAccount and CurrentAccount) that provide specific implementations for different types of bank accounts.
3. A main class (BankDemo) to demonstrate the functionality.

Key Requirements:

1. The **abstract class** should include:
    - An **abstract method** for calculating interest.
    - A **constructor** to initialize the account number and initial balance.
    - **Non-abstract** methods to handle deposits and withdrawals.
    - A **final method** to display the account balance.
    - A **static method** to provide bank details.
2. The **subclasses** should:
    - Implement the abstract method to calculate interest based on account type.
    - Provide constructors that call the superclass constructor.
3. The **main class** should:
    - Create objects for both account types.
    - Demonstrate deposit, withdrawal, interest calculation, and balance display functionalities.

Step-by-Step Instructions:

Step 1: Create the Abstract Class BankAccount

- Define the abstract class with attributes accountNumber and balance.
- Define an abstract method calculateInterest().
- Add a constructor to initialize accountNumber and balance.

- Create methods deposit() and withdraw() to handle deposits and withdrawals.
- Implement a final method displayBalance() to show the balance.
- Create a static method bankDetails() to display bank information.

Step 2: Create the Subclass SavingsAccount

- Define the subclass SavingsAccount that extends BankAccount.
- Implement the constructor that calls the superclass constructor.
- Implement the abstract method calculateInterest() to add interest to the balance.

Step 3: Create the Subclass CurrentAccount

- Define the subclass CurrentAccount that extends BankAccount.
- Implement the constructor that calls the superclass constructor.
- Implement the abstract method calculateInterest() (no interest is added for current accounts).

Step 4: Create the BankDemo Class

- Create objects of SavingsAccount and CurrentAccount.
- Perform a series of operations (deposit, withdraw, calculate interest, display balance) for each account type.

Tasks to Complete:

1. Implement the abstract class and its subclasses based on the structure provided.
2. Write your own logic for interest calculation in the SavingsAccount.
3. Test the deposit(), withdraw(), and displayBalance() methods.
4. Verify that the calculateInterest() method works differently for SavingsAccount and CurrentAccount.