RealTime OOPS Example and Abstraction vs interface

Abstract Class

- Declared using the abstract keyword.
- Can have **both abstract and concrete (implemented) methods**.
- Can have instance variables, constructors, and access modifiers (public, protected, private).
- Supports **single inheritance** (a class can extend only one abstract class).

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

   void sleep() {  // Concrete method (has implementation)
      System.out.println("Sleeping...");
   }
}
```

When to use an abstract class

- An abstract class is a good choice if we are using the inheritance concept since it provides a common base class implementation to derived classes.
- An abstract class is also good if we want to declare non-public members(Abstract classes allow you to use private, protected, and public methods.). In an interface, all methods must be public.
- If we want to add new methods in the future, then an abstract class is a better choice. Because if we add new methods to an interface, then all of the classes that already implemented that interface will have to be changed to implement the new methods.
- If we want to create multiple versions of our component, create an abstract class. Abstract classes provide a simple and easy way to version our components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, we must create a whole new interface.

1. EX:

```
abstract class Car {
```

```
  public void accelerate() {
    System.out.println("Do something to accelerate");
  }
  public void applyBrakes() {
    System.out.println("Do something to apply brakes");
  }
  public abstract void changeGears();
}
```

Now, any Car that wants to be instantiated must implement the changeGears () method.

```
class Alto extends Car {
  public void changeGears() {
    System.out.println("Implement changeGears() method for Alto Car");
  }
}
class Santro extends Car {
  public void changeGears() {
    System.out.println("Implement changeGears() method for Santro Car");
  }
}
```

Interface

- Declared using the interface keyword.
- **Before Java 8**, all methods in an interface were **abstract by default** (implicitly public abstract).
- **From Java 8**, interfaces can have **default methods** (with implementation) and **static methods**.
- **From Java 9**, they can have **private methods** as well.
- Supports **multiple inheritance** (a class can implement multiple interfaces).

```
interface Animal {
   void makeSound(); // Abstract method (implicitly public and abstract)
}
```

When to use an interface

Interfaces are also good when we want to have something similar to multiple inheritances since we can implement multiple interfaces.
Interfaces are a good choice when we think that the API will not change for a while.

```
public interface Actor {
  void perform();
}
public interface Producer {
  void invest();
}
```

Nowadays most of the actors are rich enough to produce their own movie. If we are using interfaces rather than abstract classes, we can implement both Actor and Producer. Also, we can define a new ActorProducer interface that extends both.

```
public interface ActorProducer extends Actor, Producer{
  // some statements
}
```

Key Differences

Feature	Abstract Class	Interface
Methods	Both abstract and concrete	Only abstract (until Java 8); can have default/static methods after Java 8
Variables	Can have instance variables	Only public static final (constants)
Constructors	Can have constructors	No constructors
Inheritance	Single inheritance	Multiple inheritance
Default Access Modifiers	Can be private, protected, or public	Methods are public abstract by default

Example of Both

```java
abstract class Animal {
    abstract void makeSound();  // Abstract method

    void sleep() {  // Concrete method
        System.out.println("Sleeping...");
    }
}

interface Pet {
    void play(); // Abstract method (implicitly public and abstract)
}

class Dog extends Animal implements Pet {
    void makeSound() {
        System.out.println("Bark!");
    }

    public void play() {
        System.out.println("Playing with a ball!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound(); // Output: Bark!
        dog.sleep(); // Output: Sleeping...
        dog.play(); // Output: Playing with a ball!
    }
}
```

Polymorphism in Real-Time Example

Polymorphism in Java allows methods to perform different tasks based on the object that calls them. There are two types:

1. **Compile-time Polymorphism (Method Overloading)**
2. **Run-time Polymorphism (Method Overriding)**

Real-Time Example: Payment Processing System

Consider an **e-commerce application** that supports multiple payment methods like **Credit Card, PayPal, and UPI**.
Each payment method has a different implementation but follows the same structure.

1. Using Method Overriding (Runtime Polymorphism)

```
// Parent Class
class Payment {
    void makePayment() {
        System.out.println("Processing payment...");
    }
}

// Child Classes - Overriding the method differently
class CreditCardPayment extends Payment {
    @Override
    void makePayment() {
        System.out.println("Processing payment using Credit Card.");
    }
}

class PayPalPayment extends Payment {
    @Override
    void makePayment() {
        System.out.println("Processing payment using PayPal.");
    }
}

class UPIPayment extends Payment {
    @Override
    void makePayment() {
        System.out.println("Processing payment using UPI.");
    }
}

// Main Class
public class Main {
    public static void main(String[] args) {
        Payment payment; // Reference of parent class

        payment = new CreditCardPayment();
```

```
        payment.makePayment(); // Output: Processing payment using Credit Card.

        payment = new PayPalPayment();
        payment.makePayment(); // Output: Processing payment using PayPal.

        payment = new UPIPayment();
        payment.makePayment(); // Output: Processing payment using UPI.
    }
}
```

Explanation

✅ The Payment class provides a generic makePayment() method.
✅ Each payment method **overrides** the method to provide its specific implementation.
✅ The payment reference dynamically binds to different objects, demonstrating **runtime polymorphism**.

Using Method Overloading (Compile-time Polymorphism)

A real-time example is a **calculator application** that supports different types of operations using **method overloading**.

```
class Calculator {
    // Adding two numbers
    int add(int a, int b) {
        return a + b;
    }

    // Adding three numbers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Adding double values
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        System.out.println(calc.add(10, 20));       // Output: 30
        System.out.println(calc.add(10, 20, 30));   // Output: 60
        System.out.println(calc.add(5.5, 2.5));     // Output: 8.0
    }
}
```

Explanation

✅ The add() method is **overloaded** with different parameter types.
✅ The compiler determines which method to call based on the arguments.
✅ This is an example of **compile-time polymorphism**.

Real-Time Example of Inheritance

Inheritance allows a **child class** to reuse the **attributes and methods** of a **parent class**, reducing code duplication and promoting reusability.

Real-Time Example: Employee Management System

Imagine a company where we have different types of employees: **Permanent Employees** and **Contract Employees**.
All employees share common attributes like name and salary, but **contract employees** have additional attributes like contractDuration.

Implementing Inheritance

```
// Parent Class
class Employee {
    String name;
    double salary;

    Employee(String name, double salary) {
        this.name = name;
```

```java
        this.salary = salary;
    }

    void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Salary: " + salary);
    }
}

// Child Class 1 (Permanent Employee)
class PermanentEmployee extends Employee {
    int benefits;

    PermanentEmployee(String name, double salary, int benefits) {
        super(name, salary);  // Reusing Parent Constructor
        this.benefits = benefits;
    }

    void displayDetails() {
        super.displayDetails(); // Calling parent method
        System.out.println("Benefits: " + benefits);
    }
}

// Child Class 2 (Contract Employee)
class ContractEmployee extends Employee {
    int contractDuration; // in months

    ContractEmployee(String name, double salary, int contractDuration) {
        super(name, salary);
        this.contractDuration = contractDuration;
    }

    void displayDetails() {
        super.displayDetails();
        System.out.println("Contract Duration: " + contractDuration + " months");
    }
}

// Main Class
```

```java
public class Main {
    public static void main(String[] args) {
        PermanentEmployee emp1 = new PermanentEmployee("John Doe", 50000, 5000);
        ContractEmployee emp2 = new ContractEmployee("Jane Smith", 40000, 12);

        System.out.println("Permanent Employee Details:");
        emp1.displayDetails();

        System.out.println("\nContract Employee Details:");
        emp2.displayDetails();
    }
}
```

Output

```
Permanent Employee Details:
Name: John Doe
Salary: 50000.0
Benefits: 5000

Contract Employee Details:
Name: Jane Smith
Salary: 40000.0
Contract Duration: 12 months
```

Explanation

✅ **Employee (Parent Class)** contains common attributes (name, salary) and methods (displayDetails).
✅ **PermanentEmployee (Child Class)** adds benefits.
✅ **ContractEmployee (Child Class)** adds contractDuration.
✅ **super() is used** to call the parent class constructor and methods.

Real-Time Example of Encapsulation

Encapsulation is the concept of **hiding data** and allowing controlled access through **getter and setter methods**. This ensures **data security and integrity** by restricting direct modifications.

Real-Time Example: Banking System (Encapsulating Account Details)

In a banking system, we should not allow direct access to sensitive data like accountNumber or balance. Instead, we use **private variables** and provide **getter and setter methods** for controlled access.

Encapsulation in Action

```
// Bank Account Class with Encapsulation
class BankAccount {
    private String accountHolder;
    private String accountNumber;
    private double balance;

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

    // Getter method to access accountHolder
    public String getAccountHolder() {
        return accountHolder;
    }

    // Setter method to modify accountHolder (if needed)
    public void setAccountHolder(String accountHolder) {
        this.accountHolder = accountHolder;
    }

    // Getter method to access balance
    public double getBalance() {
        return balance;
    }

    // Setter method to deposit money (Controlled Access)
    public void deposit(double amount) {
        if (amount > 0) {
```

```java
            balance += amount;
            System.out.println("Deposited: " + amount);
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }

    // Setter method to withdraw money with validation
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
        } else {
            System.out.println("Invalid withdrawal amount or insufficient balance.");
        }
    }

    // Display account details
    public void displayAccountDetails() {
        System.out.println("Account Holder: " + accountHolder);
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Current Balance: " + balance);
    }
}

// Main Class
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("John Doe", "1234567890", 5000);

        // Display initial account details
        account.displayAccountDetails();

        // Deposit money
        account.deposit(2000);
        account.displayAccountDetails();

        // Withdraw money
        account.withdraw(3000);
        account.displayAccountDetails();
```

```
        // Trying to withdraw more than balance
        account.withdraw(5000);  // Invalid transaction
    }
}
```

Output

```
Account Holder: John Doe
Account Number: 1234567890
Current Balance: 5000.0

Deposited: 2000.0
Account Holder: John Doe
Account Number: 1234567890
Current Balance: 7000.0

Withdrawn: 3000.0
Account Holder: John Doe
Account Number: 1234567890
Current Balance: 4000.0

Invalid withdrawal amount or insufficient balance.
```

Key Features of Encapsulation in the Example

✅ **Data Hiding:** The variables (accountNumber, balance) are private, preventing direct access.
✅ **Controlled Access:** The **getter methods (getBalance())** allow reading, and **setter methods (deposit(), withdraw())** allow modification with validation.
✅ **Security:** Unauthorized modifications are restricted (e.g., preventing negative deposits or over-withdrawals).

A real-world example of **abstraction** in Object-Oriented Programming (OOP) can be seen in the **use of a TV remote control**.

Real-World Example: TV Remote Control

- **Abstraction in TV Remote:** A TV remote has many buttons to control various aspects of the TV, like volume, channels, and power. However, as a user, you don't need to understand the internal workings of how the TV communicates with the remote or how the signal is processed. You only need to interact with simple buttons (e.g., power button, volume up/down, channel change) to control the TV.

Breaking it down:

1. **The TV Remote**:
    - The **interface** you interact with consists of simple, user-friendly buttons (e.g., power, volume, mute).
    - **Abstraction**: You don't need to know how the remote's signals are being transmitted to the TV or how the TV processes these signals; the details are abstracted away. You just press the button, and the result occurs.
2. **The TV**:
    - The **TV** has complex operations happening inside (e.g., signal processing, channel switching, display control). These processes are **hidden** from the user.
    - You don't need to know the technical workings behind how the TV processes the signal from the remote to adjust the volume or change channels. You only need to interact with the **simple interface** (buttons on the remote).

How Abstraction Works Here:

- **Complexity is hidden**: The complicated processes (signal transmission, processing, etc.) are hidden from you. You don't need to know how the electronics work inside the TV to use it.
- **Simplified interface**: The remote provides a simplified interface to control a complex device without needing to understand its inner mechanisms.