Object-Oriented Programming (OOP) Concepts in Java

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which contain data and methods that operate on the data. Java, as an object-oriented language, strongly emphasizes OOP principles. The four main concepts of OOP are **Encapsulation**, **Inheritance**, **Abstraction**, and **Polymorphism**.

---

1. Encapsulation

Encapsulation is the concept of wrapping data (variables) and code (methods) together into a single unit, usually a class. It also refers to the idea of restricting access to certain components of an object, making the object’s state hidden from the outside world and only accessible through a set of public methods.

Example: Bank Account

```
class BankAccount {
    // Private variables are encapsulated within the class
    private double balance;
    private String accountNumber;

    // Constructor to initialize the account
    public BankAccount(String accountNumber, double initialBa
lance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }

    // Public method to deposit money
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    // Public method to withdraw money
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }

    // Public method to check the balance
    public double getBalance() {
        return balance;
    }

    // Public method to get the account number
    public String getAccountNumber() {
        return accountNumber;
    }
}

public class EncapsulationExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("123456789", 50
0.00);

        account.deposit(200);
        account.withdraw(100);

        System.out.println("Account Number: " + account.getAc
countNumber());
        System.out.println("Balance: " + account.getBalance
());
    }
}
```

Explanation:

- The `BankAccount` class encapsulates the data (`balance`, `accountNumber`) and provides public methods (`deposit`, `withdraw`, `getBalance`, `getAccountNumber`) to interact with this data. The private variables cannot be accessed directly from outside the class, ensuring that the internal state is protected.

---

2. Inheritance

Inheritance is a mechanism wherein a new class (subclass or derived class) inherits properties and behavior (methods) from an existing class (superclass or base class). This promotes code reuse and establishes a natural hierarchy between classes.

Example: Animal, Dog, and Cat

```
// Base class
class Animal {
    String name;

    // Constructor
    public Animal(String name) {
        this.name = name;
    }

    // Method to make sound
    public void makeSound() {
        System.out.println(name + " makes a sound.");
    }
}

// Derived class
class Dog extends Animal {
    public Dog(String name) {
        super(name); // Calling the superclass constructor
    }
    // Overriding makeSound method
    @Override
    public void makeSound() {
        System.out.println(name + " barks.");
    }
}

// Derived class
class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    // Overriding makeSound method
    @Override
    public void makeSound() {
        System.out.println(name + " meows.");
    }
}

public class InheritanceExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        Cat cat = new Cat("Whiskers");

        dog.makeSound(); // Outputs: Buddy barks.
        cat.makeSound(); // Outputs: Whiskers meows.
    }
}
```

Explanation:

- The `Animal` class is the base class with a method `makeSound()`. The `Dog` and `Cat` classes inherit from `Animal` and override the `makeSound()` method to provide specific implementations.

3. Abstraction

Abstraction is the concept of hiding the complex implementation details and exposing only the essential features of an object. It helps to reduce complexity and allows the programmer to focus on interactions at a higher level.

Example: Abstract Shape Class

```
// Abstract class
abstract class Shape {
    // Abstract method
    abstract double calculateArea();

    // Concrete method
    public void display() {
        System.out.println("This is a shape.");
    }
}

// Concrete subclass
class Circle extends Shape {
    double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// Concrete subclass
class Rectangle extends Shape {
    double width, height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    double calculateArea() {
        return width * height;
    }
}

public class AbstractionExample {
    public static void main(String[] args) {
        Shape circle = new Circle(5.0);
        Shape rectangle = new Rectangle(4.0, 6.0);

        System.out.println("Circle Area: " + circle.calculate
Area());
        System.out.println("Rectangle Area: " + rectangle.cal
culateArea());
    }
}
```

Explanation:

- The `Shape` class is an abstract class with an abstract method `calculateArea()`. The `Circle` and `Rectangle` classes extend `Shape` and provide implementations for `calculateArea()`. The details of how the area is calculated are hidden, providing a simplified interface.

---

4. Polymorphism

Polymorphism is the ability of a single action to operate in different forms. In Java, it allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

Example: Animal Sound

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

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks.");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows.");
    }
}

public class PolymorphismExample {
    public static void main(String[] args) {
        Animal myAnimal = new Animal(); // Animal reference a
nd object
        Animal myDog = new Dog(); // Animal reference but Dog
object
        Animal myCat = new Cat(); // Animal reference but Cat
object

        myAnimal.makeSound(); // Outputs: Animal makes a soun
d.
        myDog.makeSound(); // Outputs: Dog barks.
        myCat.makeSound(); // Outputs: Cat meows.
    }
}
```

Explanation:

- Here, `makeSound()` is called on an `Animal` reference, but the actual method that gets executed depends on the object type. This is runtime polymorphism (method overriding) where the method to be invoked is determined at runtime.

Summary

- **Encapsulation**: Protects the internal state of an object and allows controlled access through methods.
- **Inheritance**: Enables new classes to inherit properties and behaviors from existing classes, promoting code reuse.
- **Abstraction**: Hides the implementation details and exposes only the necessary aspects, reducing complexity.
- **Polymorphism**: Allows methods to behave differently based on the object that invokes them, providing flexibility in code.

These four OOP concepts work together to create modular, reusable, and maintainable code in Java. Understanding and applying these principles is essential for effective object-oriented programming.