OOPS

Class & Object

A **class** is a blueprint that defines the structure and behavior of objects, while an **object** is an instance of a class. Classes contain fields (attributes) and methods (behaviors) that objects can use. You create objects using the `new` keyword, and each object has its own copy of the instance variables.

```
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

public class TestClassObject {
    public static void main(String[] args) {
        Animal dog = new Animal();
        dog.eat();
    }
}
```

Output:

```
Animal is eating
```

Abstraction

Abstraction hides implementation details and shows only essential features to the user. Abstract classes cannot be instantiated and may contain abstract methods (without implementation) that must be overridden by subclasses. This helps in achieving a clear separation between "what" an object does and "how" it does it, improving code maintainability.

```
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing Circle");
    }
}

public class TestAbstraction {
    public static void main(String[] args) {
        Shape s = new Circle();
        s.draw();
    }
}
```

Output:

```
Drawing Circle
```

Encapsulation

Encapsulation is the bundling of data (fields) and methods that operate on the data into a single unit (class), while restricting direct access to some components. By making fields private and providing public getter and setter methods, you control how data is accessed and modified. This protects data integrity and makes code more maintainable and flexible.

```
class Person {
    private String name;

    public void setName(String n) {
        name = n;
    }
    public String getName() {
        return name;
    }
}

public class TestEncapsulation {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Alice");
        System.out.println(p.getName());
    }
}
```

Output:

```
Alice
```

Inheritance

Inheritance allows a new class (subclass/child) to acquire properties and methods from an existing class (superclass/parent). This promotes code reusability and establishes a natural hierarchy between classes. The child class can add its own methods and fields while inheriting functionality from the parent, following an "is-a" relationship.

Types of Inheritance:

- **Single Inheritance:** A class extends one parent class
- **Multilevel Inheritance:** A class extends another class which extends another class (A → B → C)
- **Hierarchical Inheritance:** Multiple classes extend the same parent class
- **Multiple Inheritance:** Not supported directly in Java (use interfaces instead)
- **Hybrid Inheritance:** Combination of multiple types (achieved through interfaces)

```
// Single Inheritance
class Animal {
    void eat() {
        System.out.println("Eating");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking");
    }
}

// Multilevel Inheritance
class Puppy extends Dog {
    void play() {
        System.out.println("Playing");
    }
}

public class TestInheritance {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
        d.bark();

        Puppy p = new Puppy();
        p.eat();   // from Animal
        p.bark();  // from Dog
        p.play();  // from Puppy
    }
}
```

Output:

```
Eating
Barking
Eating
Barking
Playing
```

Polymorphism (Method Overloading)

Method Overloading is a form of compile-time polymorphism where multiple methods in the same class have the same name but different parameters (different number, type, or order of parameters). The compiler determines which method to call based on the method signature. This improves code readability by using the same method name for related operations.

```
class Calculator {
    // Same method name, different parameters
    int add(int a, int b) {
        return a + b;
    }

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

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

public class TestOverloading {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 10));          // calls first method
        System.out.println(calc.add(5, 10, 15));      // calls second method
        System.out.println(calc.add(5.5, 10.5));      // calls third method
    }
}
```

Output:

```
15
30
16.0
```

Polymorphism (Method Overriding)

Polymorphism means "many forms" and allows objects of different classes to be treated as objects of a common superclass. Method overriding is a form of runtime polymorphism where a subclass provides a specific implementation of a method already defined in its parent class. The actual method called is determined at runtime based on the object type, not the reference type.

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.sound();
    }
}
```

Output:

```
Dog barks
```