Experiment No-8

Experiment Name: Demonstration of Polymorphism Using Method Overriding

Task 1: Create the Animal class and override the sound() method; Animal sound(): prints “Some generic sound”, Dog sound(): prints “Dog barks”, Cat sound(): prints “Cat meows”.

Objective:

To demonstrate runtime (dynamic) polymorphism in Java through method overriding, by:

1. Implementing an inheritance hierarchy where subclasses (Dog, Cat) override a method (sound()) from a common superclass (Animal), producing distinct outputs.

Problem analysis:

Animal & sound() Overriding-

- All animals produce sound, but the sound is species-specific.
- A generic sound() method in superclass Animal is insufficient for real behavior.
- Need runtime decision: same method call (sound()) → different output based on actual object (Dog, Cat).
- Requires inheritance and method overriding (not overloading).
- Upcasting (Animal ref = new Dog()) must still invoke subclass behavior.
- Superclass method must be overridable (non-private, non-final, non-static).

Algorithm:

Step 1: Create a superclass Animal with a method sound() that prints a generic message.

Step 2: Create subclasses Dog and Cat that extend Animal.

Step 3: In each subclass, override the sound() method using @Override with species-specific output.

Step 4: In main(), declare reference variables of type Animal.

Step 5: Assign objects of Dog and Cat to Animal references (upcasting).

Step 6: Call sound() on each reference — verify that subclass versions execute at runtime.

Source code:

```
class Animal {
    void sound() {
        System.out.println("Animal:Some generic sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog:Bark, Bark");
    }
}
class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat:meow, meow");
    }
}
```

```java
public class AnimalTask1 {
    public static void main(String[] args) {
        System.out.println("==Animal Sound==");
        Animal animal = new Animal();
        Cat cat = new Cat();
        Dog dog = new Dog();
        animal.sound();
        cat.sound();
        dog.sound();
    }
}
```

Output:

```
==Animal Sound==
Animal:Some generic sound
Cat:meow, meow
Dog:Bark, Bark

D:\java lab report VS code>
```

Task No-2

Task name: Create the Shape class and override the draw() method; Shape draw(): prints “Drawing shape”, Circle draw(): prints “Drawing Circle”, Rectangle draw(): prints “Drawing Rectangle”.

Objective:

To understand and demonstrate runtime (dynamic) polymorphism in Java using method overriding, by:

1. Creating a superclass Shape with a generic draw() method, and subclasses Circle and Rectangle that override it to reflect shape-specific drawing behavior.

Problem analysis:

Shape & draw() Overriding-

- All shapes can be drawn, but drawing instruction varies by shape type.
- Generic draw() in Shape serves as placeholder/default behavior.
- Concrete shapes (Circle, Rectangle) must provide specialized implementations.
- Polymorphism allows uniform interface (Shape.draw()) with variant execution.
- Enables extensibility (e.g., adding Triangle later without modifying client code).
- Signature must match exactly to ensure overriding (not accidental overloading).

Algorithm:

Step 1: Create a superclass Shape with a method draw() that prints a generic message.

Step 2: Create subclasses Circle and Rectangle that extend Shape.

Step 3: Override the draw() method in each subclass with shape-specific output.

Step 4: In main(), use Shape-type references to hold Circle and Rectangle objects.

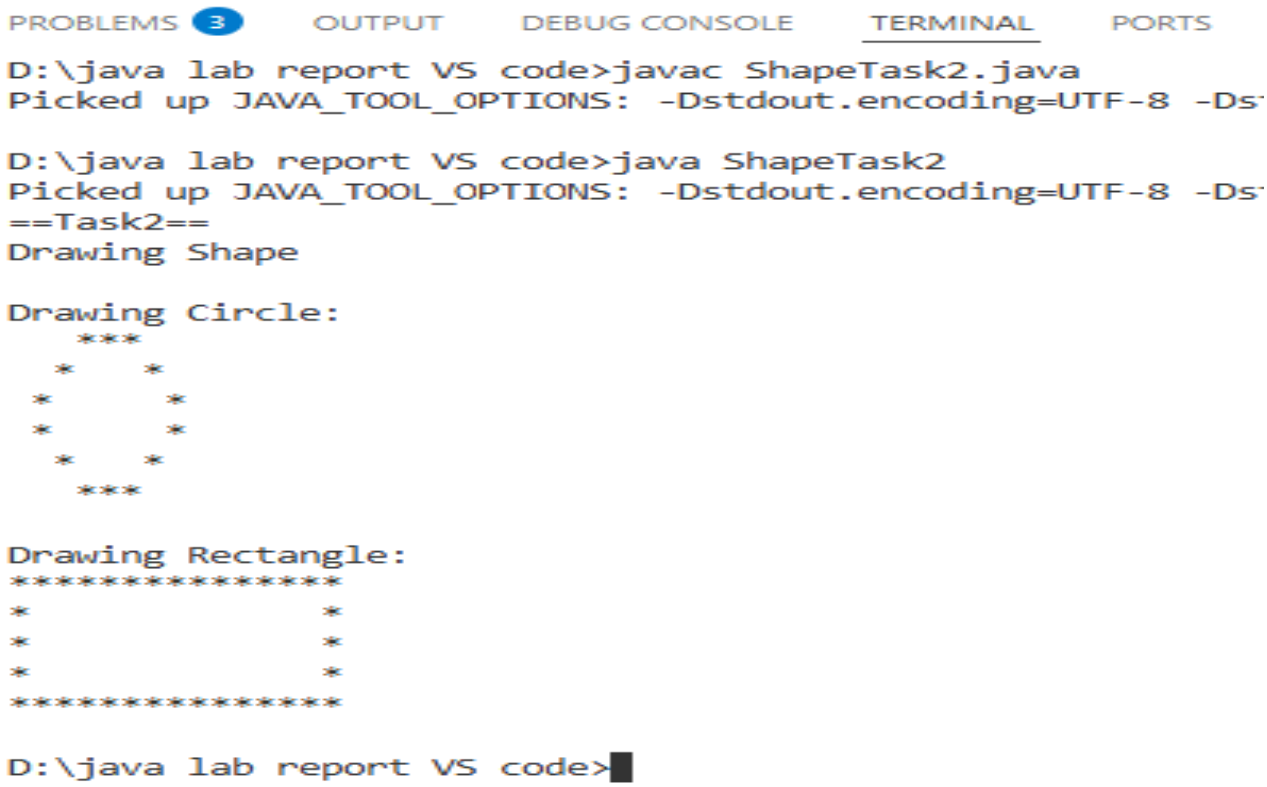
Step 5: Invoke draw() on each reference — confirm dynamic dispatch (subclass method runs).

Source code:

```
class Shape {
    void draw() {
        System.out.println("Drawing Shape");
    }
}
class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing Circle:");
        System.out.println("   ***   ");
        System.out.println("  *   *  ");
        System.out.println(" *     * ");
        System.out.println(" *     * ");
        System.out.println("  *   *  ");
        System.out.println("   ***   ");
    }
}
class Rectangle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing Rectangle:");
        System.out.println("***************");
        System.out.println("*             *");
        System.out.println("*             *");
        System.out.println("*             *");
        System.out.println("***************");

    }
}
public class ShapeTask2 {
    public static void main(String[] args) {
        System.out.println("==Task2==");
        Shape s;
        s = new Shape();
        s.draw();
        System.out.println();
        s = new Circle();
        s.draw();
        System.out.println();
        s = new Rectangle();
        s.draw();
    }
}
```

Output:



Discussion:

In this experiment, runtime polymorphism was demonstrated through method overriding. Both Animal and Shape classes contained base methods (sound() and draw()), which were overridden by their respective subclasses. When parent class references were used to refer to child class objects, the overridden methods in the child classes were executed instead of the parent's version. This shows that Java determines the method to call at runtime, not at compile time. The experiment successfully proved that method overriding enables dynamic behavior, increases flexibility, and allows the same method name to behave differently depending on the object type. This is a core feature of object-oriented programming.