Experiment No-7: Demonstration of Polymorphism Using Method Overloading
Task No-1
Task Name: Create a Calculator class with Method Overloading for Addition- Add two integers, Add two fractions, Add three integers.

Objective:

To understand and demonstrate compile-time polymorphism (method overloading) in Java by:

- Creating a Calculator class with overloaded add() methods for integers and fractions.

Problem analysis:

Calculator with Overloaded add() Methods:

- Requirement: A single intuitive method name (add) should support multiple arithmetic use cases without forcing the user to remember different method names (e.g., add2Ints, add3Ints, addDoubles).
- Challenge: Java does not support default arguments (like C++/Python), so multiple signatures are needed to handle:
- Sum of two integers (most common case),
- Sum of three integers (frequent in real calculations),
- Sum of two fractional numbers (e.g., decimals, measurements).
- Design Decision: Overload add() with:
    - add(int, int)
    - add(int, int, int)
    - add(double, double)

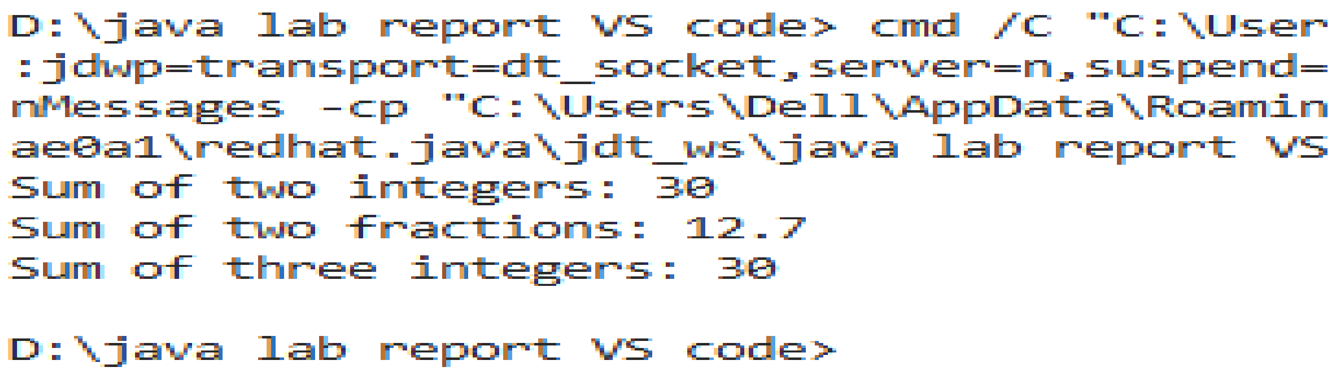
Algorithm:

1. Define class Calculator.
2. Define method add(int a, int b) → return a + b.
3. Define method add(int a, int b, int c) → return a + b + c.
4. Define method add(double a, double b) → return a + b.
5. In main(), create object and call all versions.

Source code:

```java
public class Calculator {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
    int add(int a, int b, int c) {
        return a + b + c;
    }
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum of two integers: " + calc.add(10, 20));
        System.out.println("Sum of two doubles: " + calc.add(5.5, 7.2));
        System.out.println("Sum of three integers: " + calc.add(5, 10, 15));
    }
}
```

Output:



```
D:\java lab report VS code> cmd /C "C:\User
:jdwp=transport=dt_socket,server=n,suspend=
nMessages -cp "C:\Users\Dell\AppData\Roamin
ae0a1\redhat.java\jdt_ws\java lab report VS
Sum of two integers: 30
Sum of two fractions: 12.7
Sum of three integers: 30

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

Discussion:

In this experiment, method overloading was successfully demonstrated through two practical tasks. In Task 1, the Calculator class implemented three versions of the add() method: one for two integers, one for three integers, and one for two fractional (double) values. This allowed the same method name to handle different arithmetic scenarios without ambiguity, showcasing how compile-time polymorphism improves code readability and usability. The Java compiler resolved each method call correctly based solely on the number and type of arguments passed — for instance, add(4, 5) invoked the two-integer version, while add(2.5, 3.7) triggered the double-precision version. This eliminates the need for multiple method names like addTwoInts() or addFractions(), leading to a cleaner and more intuitive interface.

Task No-2

Task Name: Create a Shape class that will have method overloading to find the area - one method for Rectangle (takes length, width), another method for Circle (takes radius).

Objective:

To understand and demonstrate compile-time polymorphism (method overloading) in Java by:

- Creating a Shape class with overloaded area() methods for rectangle and circle.

Problem analysis:

- Shape Area Calculator with Overloaded area() Methods
- Requirement: Compute area of different 2D shapes using a consistent interface.
- Challenge: Rectangle and circle require different numbers of inputs:
    - Rectangle needs length and width (2 parameters),
    - Circle needs only radius (1 parameter).
- Design Decision: Overload area() in the Shape class as:
    - area(double length, double width)
    - area(double radius)

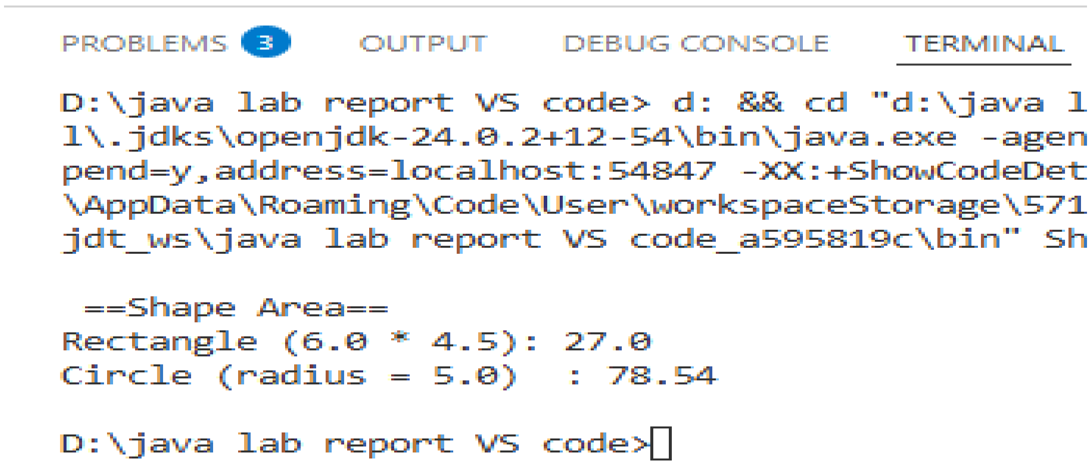
Algorithm:

1. Define class Shape.
2. Define method area(double length, double width) → return length * width.
3. Define method area(double radius) → return 3.1416 * radius * radius.
4. In main(), create object and test both area calculations.

Source code:

```java
class Shape {
    public double area(double length, double width) {
        return length * width;
    }
    public double area(double radius) {
        return 3.1416 * radius * radius;
    }
    public static void main(String[] args) {
        Shape s = new Shape();
        System.out.println("\n ==Shape Area==");
        System.out.println("Rectangle (6.0 × 4.5): " + s.area(6.0, 4.5));
        System.out.println("Circle (radius = 5.0)  : " + s.area(5.0));
    }
}
```

Output:



```
D:\java lab report VS code> d: && cd "d:\java l
l\.jdks\openjdk-24.0.2+12-54\bin\java.exe -agen
pend=y,address=localhost:54847 -XX:+ShowCodeDet
\AppData\Roaming\Code\User\workspaceStorage\571
jdt_ws\java lab report VS code_a595819c\bin" Sh

 ==Shape Area==
Rectangle (6.0 * 4.5): 27.0
Circle (radius = 5.0)  : 78.54

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

Discussion:

In Task 2, the Shape class used method overloading to compute the area of different geometric figures — specifically, a rectangle and a circle — using the same method name area(). The rectangle version accepted two parameters (length and width), whereas the circle version required only one (radius). This design mirrors real-world APIs where operations share conceptual similarity (e.g., “calculate area”) but differ in required inputs. The successful execution confirmed that the compiler distinguished between area(5.0, 3.0) and area(4.0) without errors, reinforcing the reliability of signature-based method resolution. However, care must be taken to avoid ambiguous overloads—for example, defining both area(int, int) and area(double, double) could lead to confusion when literals (e.g., area(2, 3)) are passed—though such cases were avoided here by using consistent double parameters.