Experiment No-10:

Experiment Name: Demonstration of Abstraction & Interfaces.

Task 1: Create an abstract class Shape with abstract method area() and implement it in subclasses Circle and Rectangle.

Objective:

- To understand and demonstrate abstraction in object-oriented programming.
- To create an abstract class Shape with an abstract method area().
- To implement the abstract method in concrete subclasses — Circle and Rectangle — each computing area based on their specific geometry.

Problem analysis:

- Real-world geometric shapes (e.g., circle, rectangle) share a common property — area — but compute it differently.
- Defining a single concrete Shape class with a fixed area() method would be incorrect (no universal formula).
- Need a way to enforce that all shapes must provide their own area() implementation.
- Solution: Use an abstract class to:
    - Declare a contract (abstract double area()) without implementation.
    - Prevent direct instantiation of Shape (e.g., new Shape() is invalid).
    - Enable polymorphic behavior — treat Circle and Rectangle uniformly as Shape.

Algorithm:

1. Define abstract class Shape.
2. Declare abstract method area() → no body.
3. Create class Circle extending Shape.
    - Add radius field.
    - Override area() using π × r^2.

4.Create class Rectangle extending Shape.

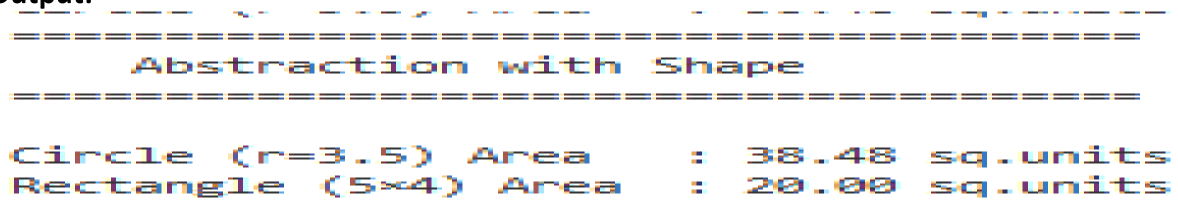
- Add length, width fields.
- Override area() using length × width.

Source code:

```
abstract class Shape {
abstract double area();
}
class Circle extends Shape {
private double radius;
public Circle(double radius) {
this.radius = radius;
}
@Override
public double area() {
return Math.PI * radius * radius;
 }
}
class Rectangle extends Shape {
private double length, width;

public Rectangle(double length, double width) {
this.length = length;
this.width = width;
}
@Override
public double area() {
return length * width;
 }
}
public class ShapeDemo {
public static void main(String[] args) {
System.out.println("=====================================");
System.out.println("    Abstraction with Shape");
System.out.println("=====================================\n");
Shape s1 = new Circle(3.5);
Shape s2 = new Rectangle(5.0, 4.0);
System.out.printf("Circle (r=3.5) Area    : %.2f sq.units%n", s1.area());
System.out.printf("Rectangle (5×4) Area   : %.2f sq.units%n", s2.area());
System.out.println("\n" + "=".repeat(50) + "\n");
 }
}
```

Output:



```
=====================================
    Abstraction with Shape
=====================================

Circle (r=3.5) Area    : 38.48 sq.units
Rectangle (5×4) Area   : 20.00 sq.units
```

Task No-2

Task Name: Create an interface ATMService and implement it in a class DBBL with methods: withdraw(), deposit(), and checkBalance().

Objective:

- To understand the concept and utility of interfaces in Java.
- To design an interface ATMService specifying essential ATM operations: withdraw(), deposit(), and checkBalance().
- To implement this interface in a concrete class DBBL, modeling a real-world banking scenario.

Problem analysis:

- Multiple banks (e.g., DBBL, BRAC, UCB) must offer the same core services: *withdraw*, *deposit*, *check balance*.
- However, internal logic (e.g., fee rules, validation, logging) may differ per bank.
- Using inheritance (e.g., abstract class Bank) would restrict a bank from extending another class (Java doesn't support multiple inheritance of classes).
- Solution: Use an interface to:
    - Define a *standardized service contract* independent of implementation.
    - Allow a class to implement multiple interfaces (future extensibility).
    - Achieve loose coupling and support plug-and-play design (e.g., swap DBBL with another bank easily).
- Interface ensures compile-time safety — any implementing class *must* define all declared methods.

Algorithm:

1. Define interface ATMService.
    - Declare methods: withdraw(amount), deposit(amount), checkBalance().
2. Create class DBBL implementing ATMService.
    - Maintain private balance.
    - Implement:
        - deposit(): Add amount if > 0.
        - withdraw(): Subtract only if sufficient balance.
        - checkBalance(): Return current balance.
3. In main(), instantiate objects and demonstrate runtime polymorphism.

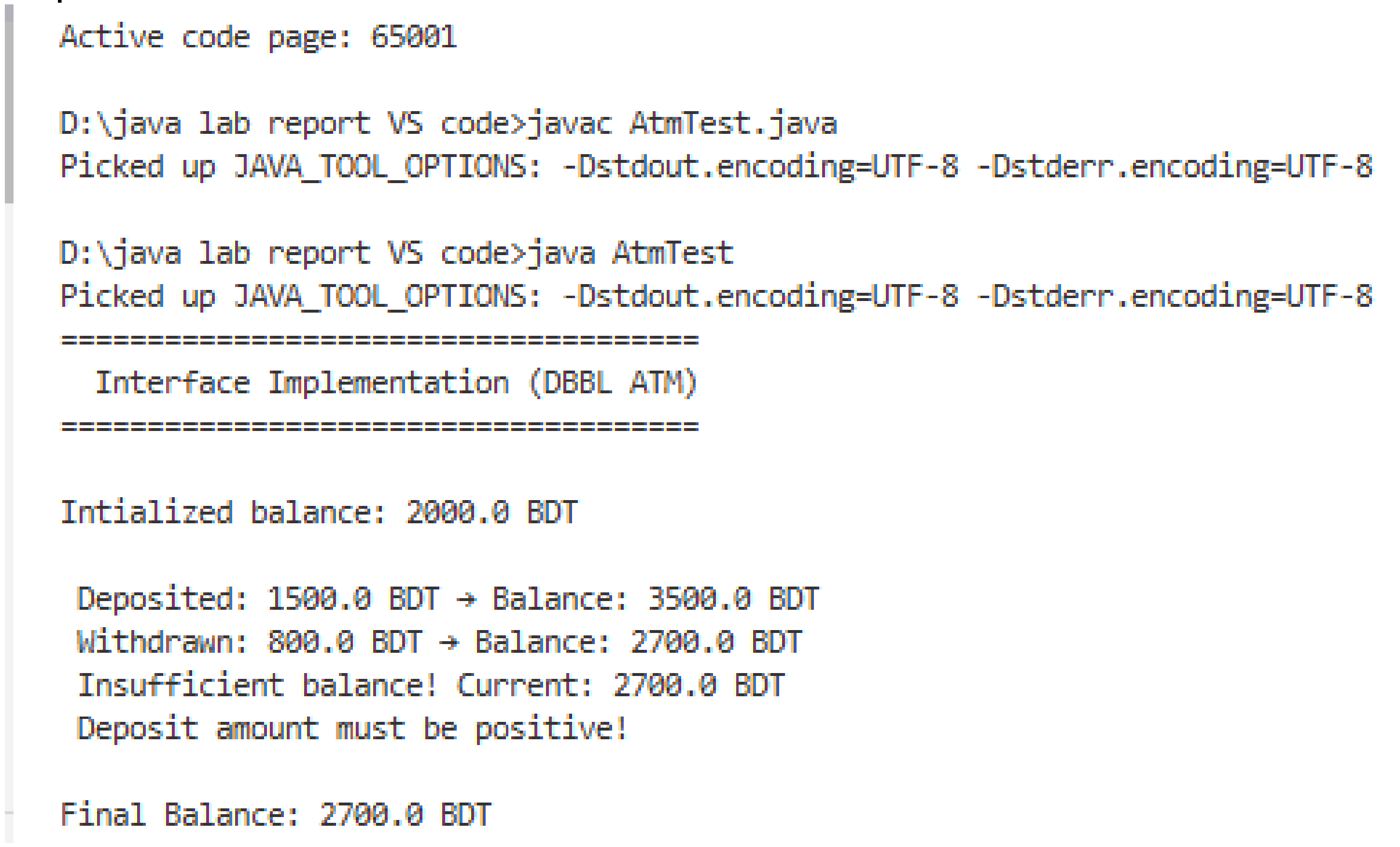
Source code:

```java
interface ATMService {
    void withdraw(double amount);
    void deposit(double amount);
    double checkBalance();
}
class DBBL implements ATMService {
    private double balance;

    public DBBL(double initialBalance) {
        this.balance = initialBalance;
    }
    @Override
    public void withdraw(double amount) {
        if (amount <= 0) {
            System.out.println(" Invalid amount!");
        } else if (amount > balance) {
            System.out.println(" Insufficient balance! Current: " + balance + " BDT");
        } else {
            balance -= amount;
     System.out.println(" Withdrawn: " + amount + " BDT → Balance: " + balance + " BDT");
        }
    }
    @Override
    public void deposit(double amount) {
        if (amount <= 0) {
            System.out.println(" Deposit amount must be positive!");
        } else {
            balance += amount;
     System.out.println(" Deposited: " + amount + " BDT → Balance: " + balance + " BDT");
        }
    }
    @Override
    public double checkBalance() {
        return balance;
    }
}
public class AtmTest {
    public static void main(String[] args) {
        System.out.println("  Interface Implementation (DBBL ATM)");
        DBBL dbblAccount = new DBBL(2000.0);
      System.out.println("Intialized balance: " + dbblAccount.checkBalance() + " BDT\n");
```

```java
        dbblAccount.deposit(1500.0);
        dbblAccount.withdraw(800.0);
        dbblAccount.withdraw(5000.0);
        dbblAccount.deposit(-200);
        System.out.println("\nFinal Balance: " + dbblAccount.checkBalance() + " BDT");
    }
}
```

Output:



```
Active code page: 65001

D:\java lab report VS code>javac AtmTest.java
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8

D:\java lab report VS code>java AtmTest
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
========================================
  Interface Implementation (DBBL ATM)
========================================

Intialized balance: 2000.0 BDT

 Deposited: 1500.0 BDT → Balance: 3500.0 BDT
 Withdrawn: 800.0 BDT → Balance: 2700.0 BDT
 Insufficient balance! Current: 2700.0 BDT
 Deposit amount must be positive!

Final Balance: 2700.0 BDT
```

Discussion:

In this experiment, two fundamental Object-Oriented Programming concepts—abstraction and interfaces—were demonstrated. In Task 1, abstraction was implemented using an abstract class Shape, which contains the abstract method area(). This method provides a general structure for calculating area but leaves the actual implementation to its subclasses. The subclasses Circle and Rectangle override the area() method to provide shape-specific formulas. This shows how abstraction helps hide unnecessary details and allows subclasses to implement only what is required.

In Task 2, an interface named ATMService was created to define essential ATM operations such as deposit(), withdraw(), and checkBalance(). The class DBBL implements this interface and provides concrete definitions for each method. This demonstrates how interfaces help in achieving multiple implementations and ensure that specific functionalities must be implemented by any class using the interface. Together, both tasks clearly show how abstraction and interfaces improve code modularity, flexibility, and reusability, which are key principles of Object-Oriented Programming.