

# **Southern University Bangladesh**

**Department of Computer Science and Engineering**

**Faculty of Science and Engineering**



## **Object Oriented Programming Lab**

### **Lab Report: 06**

**Course code : CSE 0613-108**

**Submitted by:**

**Name : Sharmistha Chowdhury**

**ID: 666-61-60**

**Submitted to :**

**Mohammed Arif Hasan Chowdhury, Assistant Professor**

**Dept. of Computer Science and Engineering**

**Date : 19/05/2024**

	Index	
Sl	Name of Program	Remarks
1	<b>Problem 01:</b> Write a Java program to create an abstract class Animal with an abstract method called sound(). Create subclasses Lion and Tiger that extend the Animal class and implement the sound() method to make a specific sound for each animal.	
2	<b>Problem 02:</b> Write a Java program to create an abstract class Shape with abstract methods calculateArea() and calculatePerimeter(). Create subclasses Circle and Triangle that extend the Shape class and implement the respective methods to calculate the area and perimeter of each shape.	
3	<b>Problem 03:</b> Write a Java program to create an abstract class BankAccount with abstract methods deposit() and withdraw(). Create subclasses: SavingsAccount and CurrentAccount that extend the BankAccount class and implement the respective methods to handle deposits and withdrawals for each account type.	
4	<b>Problem 04:</b> Write a Java program to create an abstract class Animal with abstract methods eat() and sleep(). Create subclasses Lion, Tiger, and Deer that extend the Animal class and implement the eat() and sleep() methods differently based on their specific behavior.	
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

**Problem 01: Write a Java program to create an abstract class `Animal` with an abstract method called `sound()`. Create subclasses `Lion` and `Tiger` that extend the `Animal` class and implement the `sound()` method to make a specific sound for each animal.**

### **Objective:**

The objective of this program is to illustrate the use of abstract methods in Java, where an abstract class `Animal` defines a method `sound()`, and its subclasses `Lion` and `Tiger` provide specific implementations for this method.

### **Algorithm:**

- 1. Define Animal:** An abstract class with an abstract method `sound()`.
- 2. Define Lion and Tiger:** Subclasses of `Animal` that provide their own implementations of `sound()`.
- 3. Create Lion and Tiger instances:** In the main method, create instances of `Lion` and `Tiger`, referenced as `Animal` type.
- 4. Call `sound()` on instances:** Call the `sound()` method on these instances, which calls the appropriate method based on the actual object type.

### **Code:**

```
public class Program1 {
    // Abstract class Animal
    abstract static class Animal {
        public abstract void sound();
    }
    // Subclass Lion
    static class Lion extends Animal {
        @Override
        public void sound() {
            System.out.println("Lion roars!");
        }
    }
    // Subclass Tiger
    static class Tiger extends Animal {
```

```

@Override

public void sound() {

    System.out.println("Tiger growls!");

}

}

public static void main(String[] args) {

    Animal lion = new Lion();

    lion.sound();


    Animal tiger = new Tiger();

    tiger.sound();

}

}

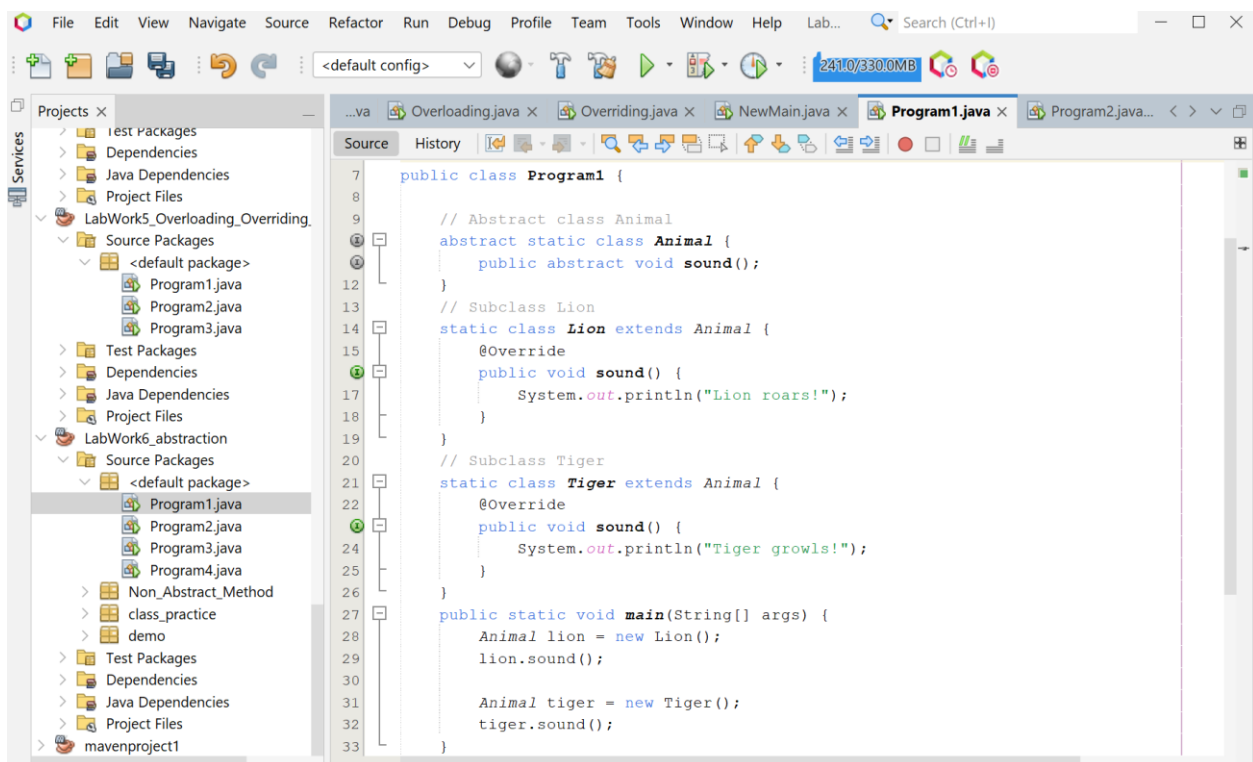
```

### Input/ Expected Output:

Lion roars!

Tiger growls!

### Screenshot of code edition window:



## Screenshot of Output screen/Run screen window:

```

cd C:\Users\Sharmishtha\Documents\NetBeansProjects\LabWork6; "JAVA_HOME=C:\Program Files\Java\jdk-21" cmd /c "%C:\Program
Scanning for projects...

-----< com.mycompany:LabWork6 >-----
Building LabWork6_abstraction 1.0-SNAPSHOT
  from pom.xml
-----[ jar ]-----

--- resources:3.3.1:resources (default-resources) @ LabWork6 ---
skip non existing resourceDirectory C:\Users\Sharmishtha\Documents\NetBeansProjects\LabWork6\src\main\resources

--- compiler:3.11.0:compile (default-compile) @ LabWork6 ---
Changes detected - recompiling the module! :source
Compiling 9 source files with javac [debug target 21] to target\classes

--- exec:3.1.0:exec (default-cli) @ LabWork6 ---
Lion roars!
Tiger growls!

BUILD SUCCESS

Total time:  6.269 s
Finished at: 2024-05-13T21:47:18+06:00
  
```

## Explanation:

The program is a simple Java application that demonstrates the concept of method overloading, a feature in Java that allows a class to have more than one method having the same name, if their argument lists are different. It defines a Product class with two multiply methods, one for calculating the product of two integers and another for three integers. The Program1 class contains the main method where an instance of the Product class is created and both multiply methods are invoked with different sets of numbers.

## Discussion

This program effectively illustrates the power of abstract methods and polymorphism in object-oriented programming. By defining a common method in an abstract class and allowing subclasses to provide their own implementations, we can create more flexible and reusable code. This is a fundamental concept in Java and many other object-oriented languages. It allows us to write code that is open to extension (by adding new subclasses) but closed for modification (since we don't need to change the abstract class or existing subclasses). This principle is known as the Open-Closed Principle in software design and is a key aspect of writing maintainable and scalable code.

**Problem 02: Write a Java program to create an abstract class Shape with abstract methods calculateArea() and calculatePerimeter(). Create subclasses Circle and Triangle that extend the Shape class and implement the respective methods to calculate the area and perimeter of each shape.**

### **Objective:**

The objective of this Java program is to demonstrate the concept of abstraction and polymorphism in object-oriented programming by calculating and printing the area and perimeter of a circle and a triangle.

### **Algorithm:**

- 1. Define the abstract class 'Shape':** This class contains two abstract methods, 'calculateArea()' and 'calculatePerimeter()', which will be implemented in the subclasses.
- 2. Define the subclass 'Circle' that extends 'Shape':** This class has a private variable 'radius' and a constructor to initialize it. It also provides the implementation for the abstract methods:
  - 'calculateArea()': Returns the area of the circle using the formula ' $\pi * \text{radius} * \text{radius}$ '.
  - 'calculatePerimeter()': Returns the perimeter of the circle using the formula ' $2 * \pi * \text{radius}$ '.
- 3. Define the subclass 'Triangle' that extends 'Shape':** This class has private variables 'side1', 'side2', and 'side3' and a constructor to initialize them. It also provides the implementation for the abstract methods:
  - 'calculateArea()': Returns the area of the triangle using Heron's formula ' $\sqrt{s * (s - \text{side1}) * (s - \text{side2}) * (s - \text{side3})}$ ', where 's' is the semi-perimeter of the triangle ' $((\text{side1} + \text{side2} + \text{side3}) / 2)$ '.
  - 'calculatePerimeter()': Returns the perimeter of the triangle using the formula ' $\text{side1} + \text{side2} + \text{side3}$ '.
- 4. In the 'main' method:**
  - Create a 'Circle' object with radius 4.0 and calculate and print its area and perimeter.
  - Create a 'Triangle' object with sides 3.0, 4.0, and 5.0 and calculate and print its area and perimeter.

**Code:**

```
public class Program2 {

    // Abstract class Shape
    abstract static class Shape {
        abstract double calculateArea();
        abstract double calculatePerimeter();
    }

    // Subclass Circle
    static class Circle extends Shape {
        private double radius;

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

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

        @Override
        double calculatePerimeter() {
            return 2 * Math.PI * radius;
        }
    }

    // Subclass Triangle
    static class Triangle extends Shape {
        private double side1;
        private double side2;
```

```

private double side3;

public Triangle(double side1, double side2, double side3) {
    this.side1 = side1;
    this.side2 = side2;
    this.side3 = side3;
}

@Override
double calculateArea() {
    double s = (side1 + side2 + side3) / 2; // Semi-perimeter
    return Math.sqrt(s * (s - side1) * (s - side2) * (s - side3));
}

@Override
double calculatePerimeter() {
    return side1 + side2 + side3;
}
}

public static void main(String[] args) {
    double r = 4.0;
    Circle circle = new Circle(r);

    double ts1 = 3.0, ts2 = 4.0, ts3 = 5.0;
    Triangle triangle = new Triangle(ts1, ts2, ts3);

    System.out.println("Radius of the Circle: " + r);
    System.out.println("Area of the Circle: " + circle.calculateArea());
    System.out.println("Perimeter of the Circle: " + circle.calculatePerimeter());

    System.out.println("\nSides of the Triangle are: " + ts1 + ", " + ts2 + ", " + ts3);
    System.out.println("Area of the Triangle: " + triangle.calculateArea());
}

```



```

        System.out.println("Perimeter of the Triangle: " + triangle.calculatePerimeter());
    }
}

```

### Input/ Expected Output:

Radius of the Circle: 4.0

Area of the Circle: 50.26548245743669

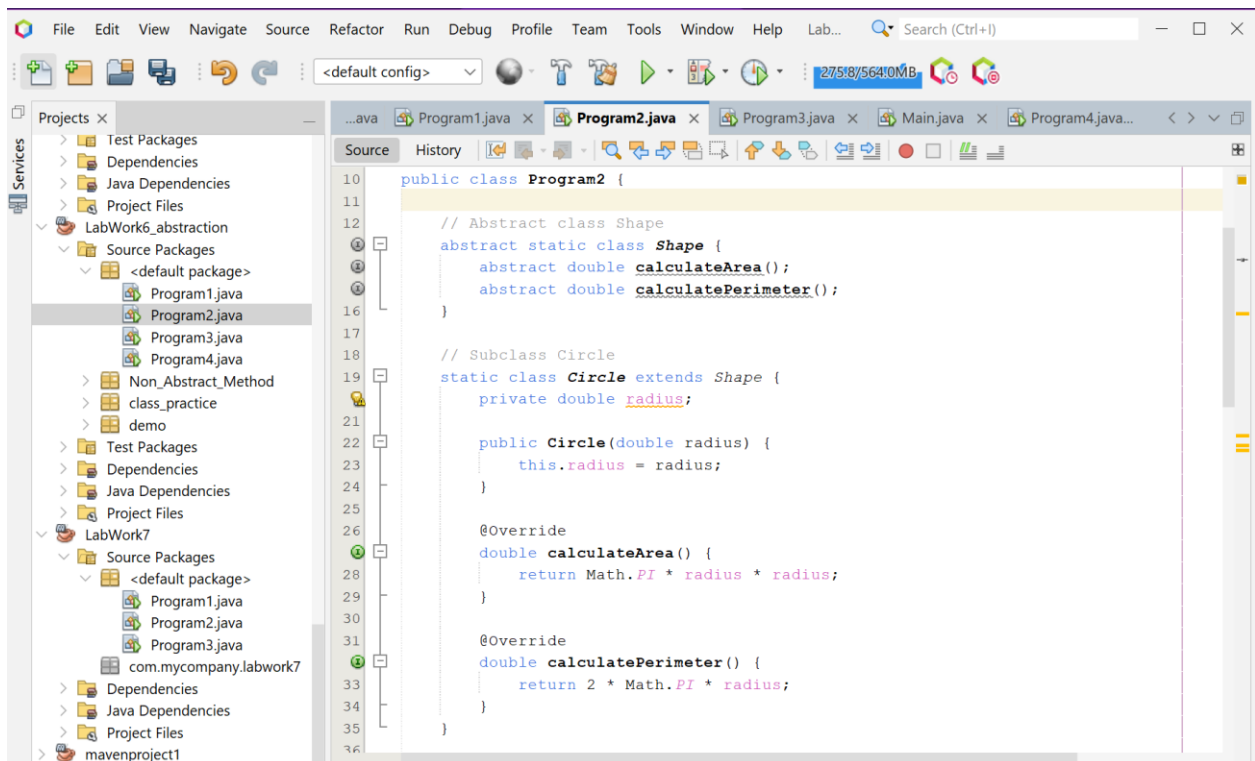
Perimeter of the Circle: 25.132741228718345

Sides of the Triangle are: 3.0, 4.0, 5.0

Area of the Triangle: 6.0

Perimeter of the Triangle: 12.0

### Screenshot of code edition window:



The screenshot shows an IDE with the following components:

- Top Bar:** File, Edit, View, Navigate, Source, Refactor, Run, Debug, Profile, Team, Tools, Window, Help, Lab... Search (Ctrl+I)
- Left Panel (Project Explorer):**
  - Test Packages
  - Dependencies
  - Java Dependencies
  - Project Files
  - LabWork6\_abstraction
    - Source Packages
      - <default package>
        - Program1.java
        - Program2.java** (selected)
        - Program3.java
        - Program4.java
      - Non\_Abstract\_Method
      - class\_practice
      - demo
    - Test Packages
    - Dependencies
    - Java Dependencies
    - Project Files
    - LabWork7
      - Source Packages
        - <default package>
          - Program1.java
          - Program2.java
          - Program3.java
        - com.mycompany.labwork7

- Right Panel (Source Editor):**

```

37 // Subclass Triangle
38 static class Triangle extends Shape {
39     private double side1;
40     private double side2;
41     private double side3;
42
43     public Triangle(double side1, double side2, double side3) {
44         this.side1 = side1;
45         this.side2 = side2;
46         this.side3 = side3;
47     }
48
49     @Override
50     double calculateArea() {
51         double s = (side1 + side2 + side3) / 2; // Semi-perimeter
52         return Math.sqrt(s * (s - side1) * (s - side2) * (s - side3));
53     }
54
55     @Override
56     double calculatePerimeter() {
57         return side1 + side2 + side3;
58     }
59 }
60
61 public static void main(String[] args) {
62     double r = 4.0;
63     Circle circle = new Circle(r);
64
65     double ts1 = 3.0, ts2 = 4.0, ts3 = 5.0;
66     Triangle triangle = new Triangle(ts1, ts2, ts3);
67
68     System.out.println("Radius of the Circle: " + r);
69     System.out.println("Area of the Circle: " + circle.calculateArea());
70     System.out.println("Perimeter of the Circle: " + circle.calculatePerimeter());
71
72     System.out.println("\nSides of the Triangle are: " + ts1 + ", " + ts2 + ", " + ts3);
73     System.out.println("Area of the Triangle: " + triangle.calculateArea());
74     System.out.println("Perimeter of the Triangle: " + triangle.calculatePerimeter());
75 }
76
77
78

```

## Screenshot of Output screen/Run screen window:

```

-----[ jar ]-----
--- resources:3.3.1:resources (default-resources) @ LabWork6 ---
skip non existing resourceDirectory C:\Users\Sharmishtha\Documents\NetBeansProjects\LabWork6\src\main\resources

--- compiler:3.11.0:compile (default-compile) @ LabWork6 ---
Changes detected - recompiling the module! :source
Compiling 9 source files with javac [debug target 21] to target\classes

--- exec:3.1.0:exec (default-cli) @ LabWork6 ---
Radius of the Circle: 4.0
Area of the Circle: 50.26548245743669
Perimeter of the Circle: 25.132741228718345

Sides of the Triangle are: 3.0, 4.0, 5.0
Area of the Triangle: 6.0
Perimeter of the Triangle: 12.0

-----
BUILD SUCCESS
-----
Total time: 5.660 s
Finished at: 2024-05-17T14:48:52+06:00
-----

```

## Explanation :

This Java program demonstrates the use of abstraction and polymorphism in object-oriented programming through the calculation of area and perimeter for two shapes: a circle and a triangle. The Shape class is an abstract class with two abstract methods: `calculateArea()` and `calculatePerimeter()`. These methods are implemented in the subclasses Circle and Triangle. The Circle class calculates the area and perimeter of a circle using the formulas  $\pi * \text{radius} * \text{radius}$  and  $2 * \pi * \text{radius}$  respectively. The Triangle class calculates the area using Heron's formula  $\sqrt{s * (s - \text{side1}) * (s - \text{side2}) * (s - \text{side3})}$ , where  $s$  is the semi-perimeter of the triangle  $((\text{side1} + \text{side2} + \text{side3}) / 2)$ , and the perimeter using the formula  $\text{side1} + \text{side2} + \text{side3}$ . In the main method, a Circle object and a Triangle object are created, and their areas and perimeters are calculated and printed.

## Discussion:

The program effectively illustrates the power of object-oriented programming principles like abstraction and polymorphism. Abstraction is used to define a general Shape class that outlines the methods any shape should have, without specifying how these methods are implemented. This is where polymorphism comes into play. The Circle and Triangle classes, which extend the Shape class, provide their own specific implementations of the `calculateArea()` and `calculatePerimeter()` methods. This allows for code that is more organized and easier to understand, as each shape class is responsible for its own calculations. Furthermore, it allows for flexibility, as you can easily add more shape classes in the future, each with their own specific implementations, without having to modify the existing code. This makes the code more maintainable and adaptable to changes, which are key aspects of good software design.

**Problem 03: Write a Java program to create an abstract class `BankAccount` with abstract methods `deposit()` and `withdraw()`. Create subclasses: `SavingsAccount` and `CurrentAccount` that extend the `BankAccount` class and implement the respective methods to handle deposits and withdrawals for each account type.**

### **Objective:**

The objective of this Java program is to simulate a simple banking system with two types of accounts (Savings and Current), allowing deposits and withdrawals, demonstrating the use of abstract classes, inheritance, and polymorphism in Java.

### **Algorithm:**

1. **Define the abstract class `BankAccount`:** This class has a protected variable `balance` and a constructor to initialize it. It also contains two abstract methods, `deposit(double amount)` and `withdraw(double amount)`, which will be implemented in the subclasses.
2. **Define the subclass `SavingsAccount` that extends `BankAccount`:** This class has a constructor to initialize the `balance` using the superclass constructor. It also provides the implementation for the abstract methods:
  - `deposit(double amount)`: Increases the `balance` by the `amount` and prints a success message along with the current balance.
  - `withdraw(double amount)`: Checks if the `balance` is greater than or equal to the `amount`. If it is, decreases the `balance` by the `amount` and prints a success message along with the current balance. If not, prints a failure message about insufficient funds.
3. **Define the subclass `CurrentAccount` that extends `BankAccount`:** This class has a constructor to initialize the `balance` using the superclass constructor. It also provides the implementation for the abstract methods, which are identical to those in the `SavingsAccount` class.
4. **In the `main` method:**
  - Create a `Scanner` object to read user input.
  - Create a `SavingsAccount` object with an initial balance of 1000. Print the initial balance, prompt the user to enter a deposit amount, perform the deposit, and print the new balance. Then, prompt the user to enter a withdrawal amount, perform the withdrawal, and print the new balance.
  - Create a `CurrentAccount` object with an initial balance of 2000. Print the initial balance, prompt the user to enter a deposit amount, perform the deposit, and print the new balance. Then, prompt the user to enter a withdrawal amount, perform the withdrawal, and print the new balance.
  - Close the `Scanner` object.

**Code:**

```

import java.util.Scanner;

public class Program3 {

    // Abstract class BankAccount
    abstract static class BankAccount {
        protected double balance;

        public BankAccount(double balance) {
            this.balance = balance;
        }

        public abstract void deposit(double amount);
        public abstract void withdraw(double amount);
    }

    // Subclass SavingsAccount
    static class SavingsAccount extends BankAccount {
        public SavingsAccount(double balance) {
            super(balance);
        }

        @Override
        public void deposit(double amount) {
            balance += amount;

            System.out.println("Deposit of " + amount + " successful. Current balance: " + balance);
        }

        @Override
        public void withdraw(double amount) {
            if (balance >= amount) {
                balance -= amount;
            }
        }
    }
}

```

```

        System.out.println("Withdrawal of " + amount + " successful. Current balance: " + balance);
    } else {
        System.out.println("Insufficient funds. Withdrawal failed.");
    }
}

// Subclass CurrentAccount
static class CurrentAccount extends BankAccount {
    public CurrentAccount(double balance) {
        super(balance);
    }
    @Override
    public void deposit(double amount) {
        balance += amount;
        System.out.println("Deposit of " + amount + " successful. Current balance: " + balance);
    }
    @Override
    public void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrawal of " + amount + " successful. Current balance: " + balance);
        } else {
            System.out.println("Insufficient funds. Withdrawal failed.");
        }
    }
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    BankAccount savings = new SavingsAccount(1000);
    System.out.println("Savings A/c: Initial Balance: " + savings.balance);
    System.out.println("Enter deposit amount for savings account:");

```

```
double depositAmount = scanner.nextDouble();
savings.deposit(depositAmount);
```

```
System.out.println("Enter withdrawal amount for savings account:");
double withdrawalAmount = scanner.nextDouble();
savings.withdraw(withdrawalAmount);
```

```
BankAccount current = new CurrentAccount(2000);
System.out.println("\nCurrent A/c: Initial Balance: " + current.balance);
System.out.println("Enter deposit amount for current account:");
depositAmount = scanner.nextDouble();
current.deposit(depositAmount);
```

```
System.out.println("Enter withdrawal amount for current account:");
withdrawalAmount = scanner.nextDouble();
current.withdraw(withdrawalAmount);
```

```
scanner.close();
```

```
}
```

```
}
```

### **Input/ Expected Output:**

Savings A/c: Initial Balance: 1000.0

Enter deposit amount for savings account: 2000000

Deposit of 2000000.0 successful. Current balance: 2001000.0

Enter withdrawal amount for savings account: 60000

Withdrawal of 60000.0 successful. Current balance: 1941000.0

Current A/c: Initial Balance: 2000.0

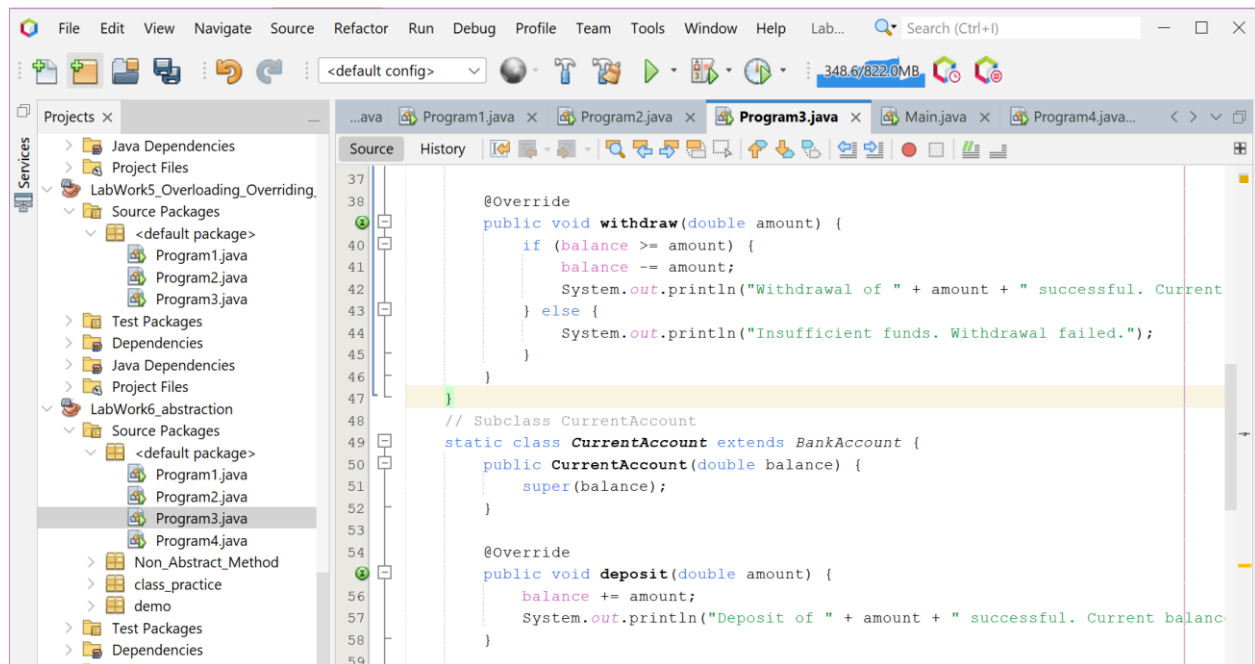
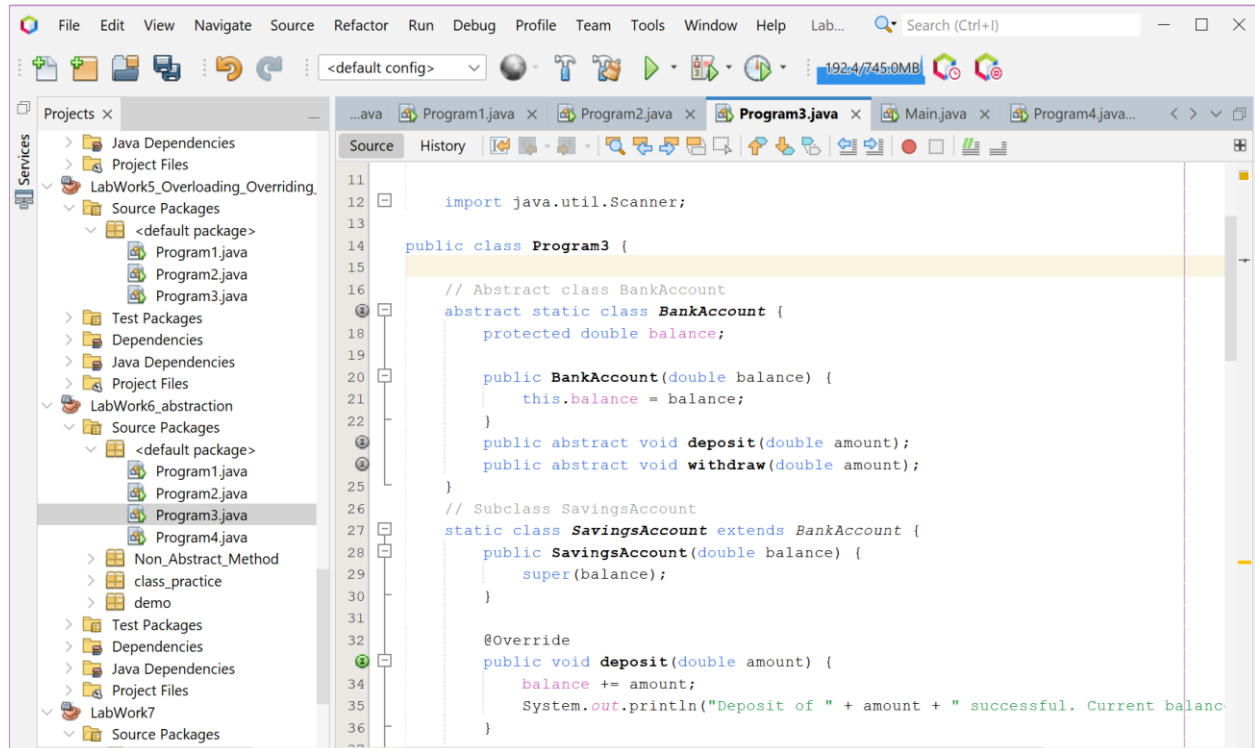
Enter deposit amount for current account: 500000

Deposit of 500000.0 successful. Current balance: 502000.0

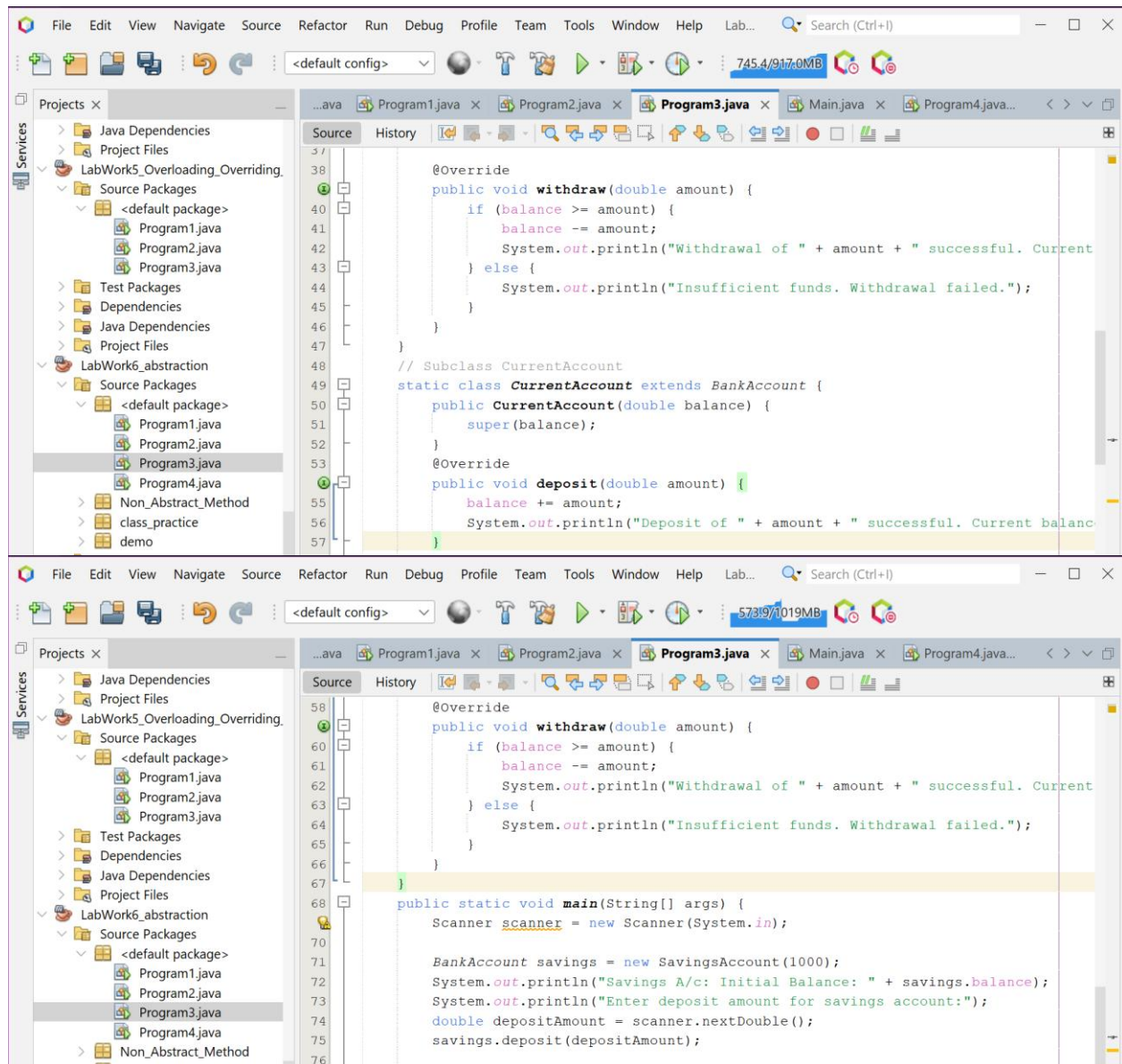
Enter withdrawal amount for current account: 7000

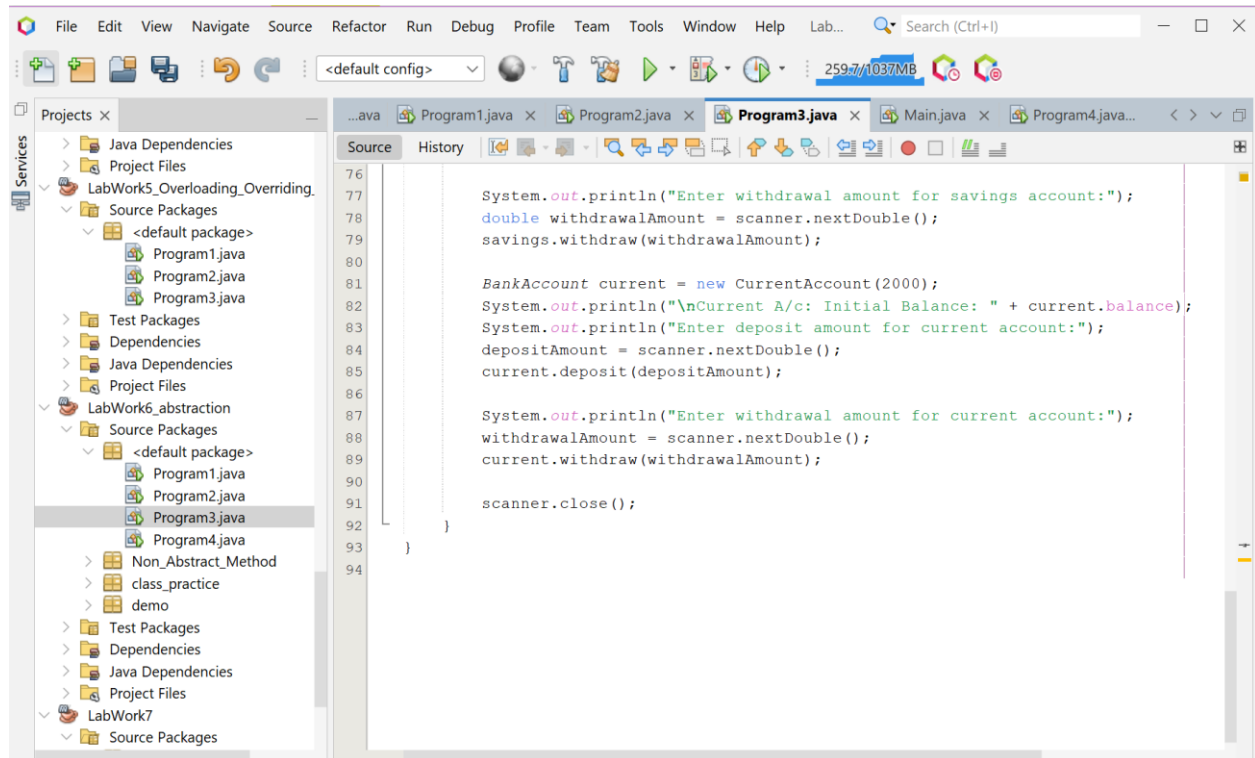
Withdrawal of 7000.0 successful. Current balance: 495000.00

## Screenshot of code edition window:

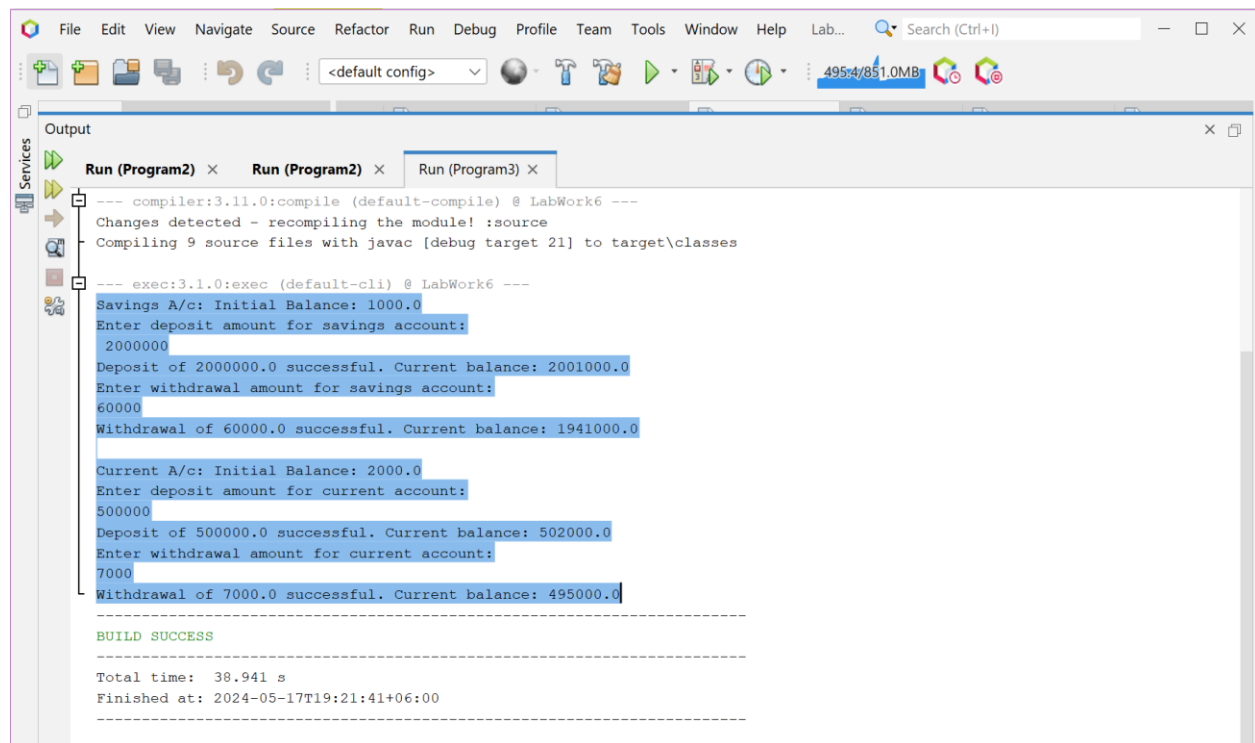








### Screenshot of Output screen/Run screen window:



**Explanation :**

This Java program simulates a simple banking system with two types of accounts: Savings and Current. Both account types extend an abstract class BankAccount that outlines common behaviors such as depositing and withdrawing money. The SavingsAccount and CurrentAccount classes implement these behaviors, checking for sufficient funds before withdrawal. In the main method, a SavingsAccount and a CurrentAccount are created, and the user is prompted to enter deposit and withdrawal amounts for each account, which are then processed and the resulting balances printed.

**Discussion:**

The program effectively demonstrates the principles of object-oriented programming, specifically abstraction and polymorphism. The BankAccount class serves as a template for creating more specific account classes, and each specific account class provides its own implementation of the deposit and withdraw methods. This design allows for code that is organized and easy to understand, as each account class is responsible for its own specific behaviors. Furthermore, it allows for flexibility, as more account types can be easily added in the future without having to modify the existing code, making the code more maintainable and adaptable to changes.

**Problem 04: Write a Java program to create an abstract class `Animal` with abstract methods `eat()` and `sleep()`. Create subclasses `Lion`, `Tiger`, and `Deer` that extend the `Animal` class and implement the `eat()` and `sleep()` methods differently based on their specific behavior.**

**Objective:**

The objective of this Java program is to demonstrate the principles of abstraction and polymorphism in object-oriented programming by modeling the behaviors of different types of animals (`Lion`, `Tiger`, and `Deer`) through their eating and sleeping habits.

**Algorithm:**

1. **Define the abstract class `Animal`:** This class contains two abstract methods, `eat()` and `sleep()`, which will be implemented in the subclasses.
2. **Define the subclass `Lion` that extends `Animal`:** This class provides the implementation for the abstract methods:
  - `eat()`: Prints "Lion eats meat".
  - `sleep()`: Prints "Lion sleeps at night".
3. **Define the subclass `Tiger` that extends `Animal`:** This class provides the implementation for the abstract methods:
  - `eat()`: Prints "Tiger eats meat".
  - `sleep()`: Prints "Tiger sleeps at day".
4. **Define the subclass `Deer` that extends `Animal`:** This class provides the implementation for the abstract methods:
  - `eat()`: Prints "Deer eats grass".
  - `sleep()`: Prints "Deer sleeps at night".
5. **In the main method:**
  - Create a `Lion` object and call its `eat()` and `sleep()` methods.
  - Create a `Tiger` object and call its `eat()` and `sleep()` methods.
  - Create a `Deer` object and call its `eat()` and `sleep()` methods.

**Code:**

```
public class Program4 {

    // Abstract class Animal
    abstract static class Animal {
        public abstract void eat();
        public abstract void sleep();
    }

    // Subclass Lion
    static class Lion extends Animal {
        @Override
        public void eat() {
            System.out.println("Lion eats meat");
        }

        @Override
        public void sleep() {
            System.out.println("Lion sleeps at night");
        }
    }

    // Subclass Tiger
    static class Tiger extends Animal {
        @Override
        public void eat() {
            System.out.println("Tiger eats meat");
        }

        @Override
        public void sleep() {
            System.out.println("Tiger sleeps at day");
        }
    }

    // Subclass Deer
    static class Deer extends Animal {
        @Override
        public void eat() {
            System.out.println("Deer eats grass");
        }

        @Override
```

```
        public void sleep() {  
            System.out.println("Deer sleeps at night");  
        }  
    }  
  
    public static void main(String[] args) {  
        Animal lion = new Lion();  
        lion.eat();  
        lion.sleep();  
  
        Animal tiger = new Tiger();  
        tiger.eat();  
        tiger.sleep();  
  
        Animal deer = new Deer();  
        deer.eat();  
        deer.sleep();  
    }  
}
```

**Input/ Expected Output:**

Lion eats meat

Lion sleeps at night

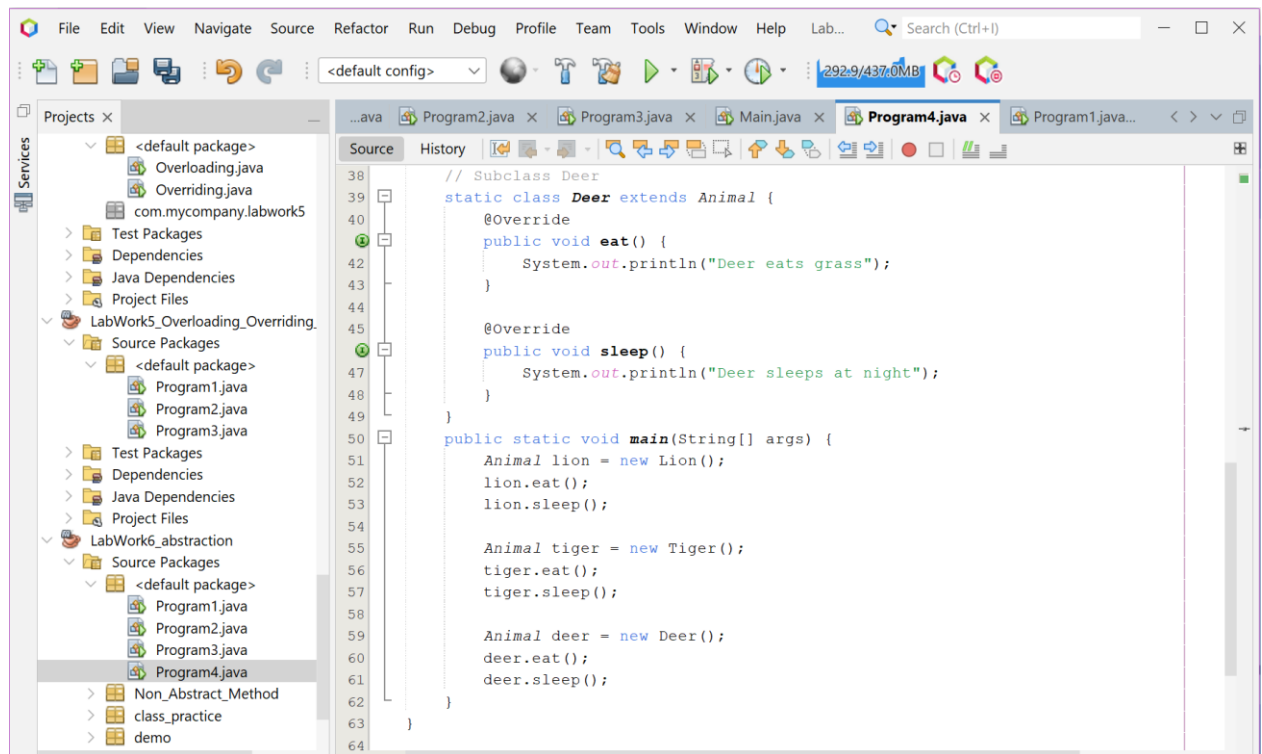
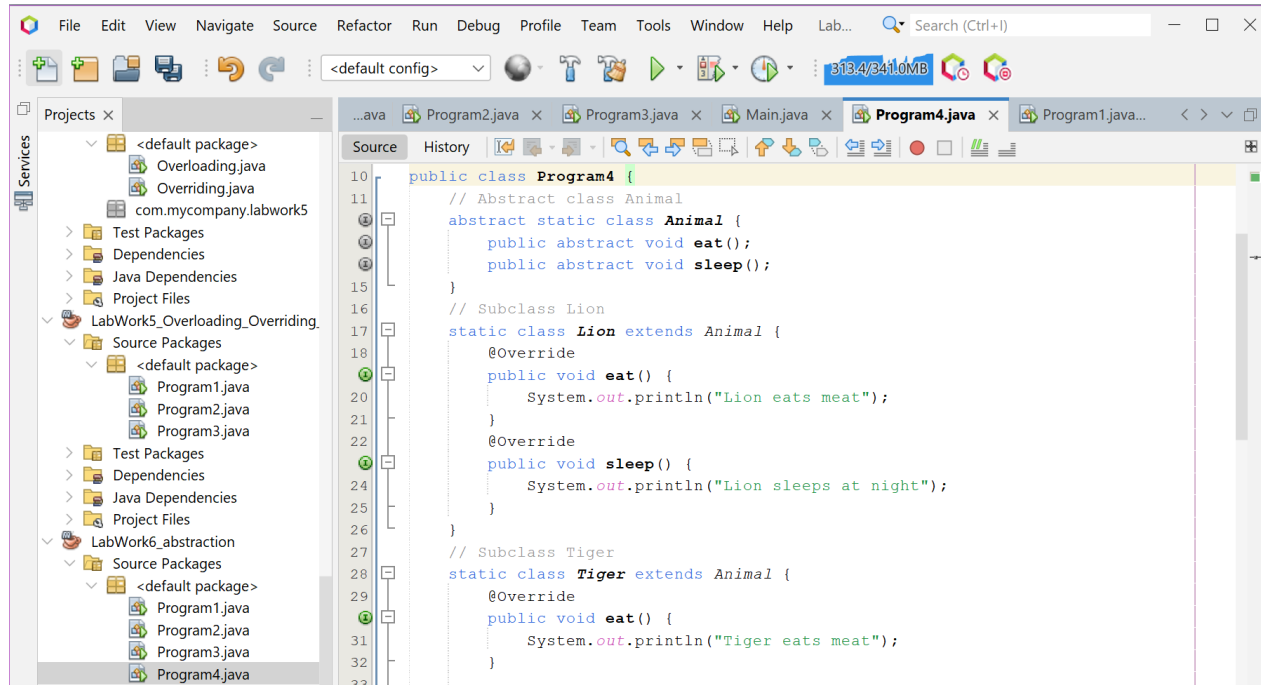
Tiger eats meat

Tiger sleeps at day

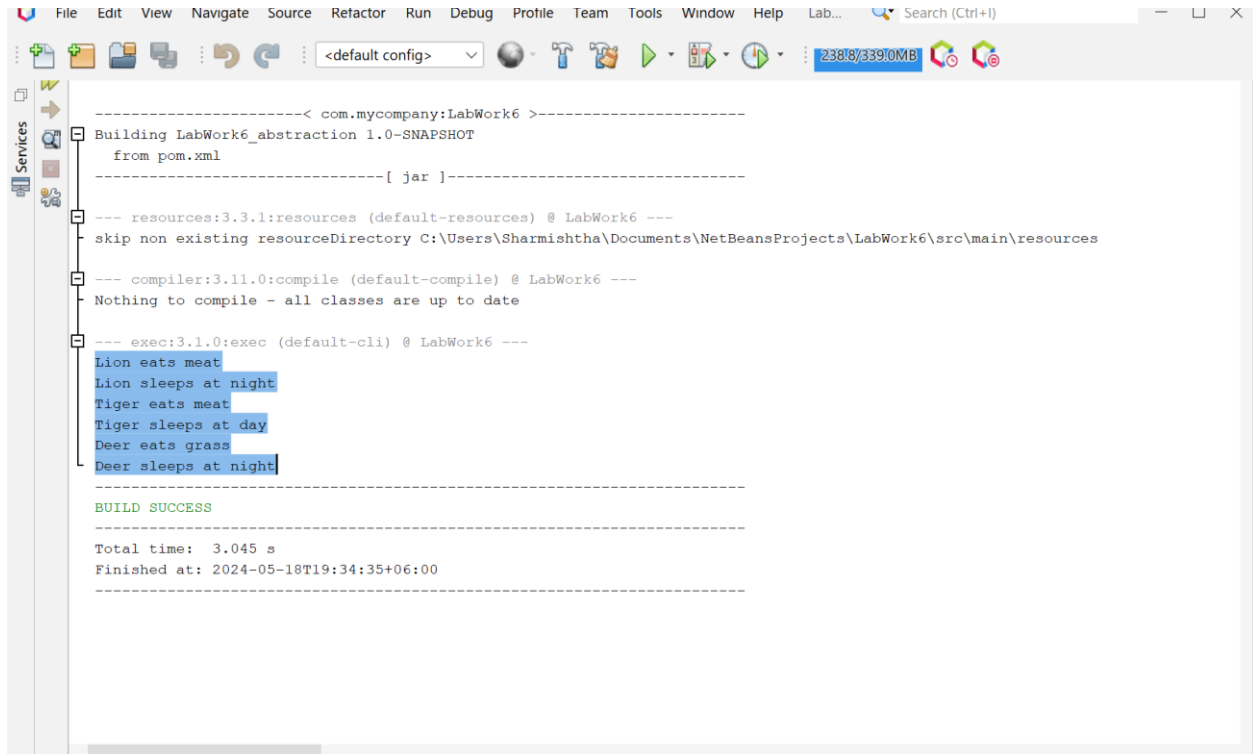
Deer eats grass

Deer sleeps at night

## Screenshot of code edition window:



## Screenshot of Output screen/Run screen window:



```

-----< com.mycompany:LabWork6 >-----
Building LabWork6_abstraction 1.0-SNAPSHOT
from pom.xml
-----[ jar ]-----

--- resources:3.3.1:resources (default-resources) @ LabWork6 ---
skip non existing resourceDirectory C:\Users\Sharmishtha\Documents\NetBeansProjects\LabWork6\src\main\resources

--- compiler:3.11.0:compile (default-compile) @ LabWork6 ---
Nothing to compile - all classes are up to date

--- exec:3.1.0:exec (default-cli) @ LabWork6 ---
Lion eats meat
Lion sleeps at night
Tiger eats meat
Tiger sleeps at day
Deer eats grass
Deer sleeps at night

BUILD SUCCESS

Total time: 3.045 s
Finished at: 2024-05-18T19:34:35+06:00
  
```

## Explanation :

This Java program models the behaviors of different types of animals (Lion, Tiger, and Deer) using the principles of abstraction and polymorphism in object-oriented programming. An abstract Animal class is defined with two abstract methods: eat() and sleep(). These methods are implemented in the subclasses Lion, Tiger, and Deer, each representing a specific type of animal. In the main method, an instance of each animal is created and their eat() and sleep() methods are called, resulting in each animal displaying its unique behavior.

## Discussion:

The program effectively demonstrates the principles of object-oriented programming, specifically abstraction and polymorphism. The Animal class serves as a template for creating more specific animal classes, and each specific animal class provides its own implementation of the eat() and sleep() methods. This design allows for code that is organized and easy to understand, as each animal class is responsible for its own specific behaviors. Furthermore, it allows for flexibility, as more animal types can be easily added in the future without having to modify the existing code, making the code more maintainable and adaptable to changes. This program is a great example of good software design using object-oriented programming principles..