Java Practical

Mayank

Practical 0

Introduction to Java

1 Introduction to Java

JAVA was developed by James Gosling at Sun Microsystems Inc in May 1995 and later acquired by Oracle Corporation. It is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs. Java is a class-based, object-oriented programming language and is designed to have as few implementation dependencies as possible. A general-purpose programming language made for developers to write once run anywhere that is compiled Java code can run on all platforms that support Java. Java applications are compiled to byte code that can run on any Java Virtual Machine. The syntax of Java is similar to C/C++.

Java is widely used for developing applications for desktop, web, and mobile devices. Java is known for its simplicity, robustness, and security features, making it a popular choice for enterprise-level applications.

2 Java Syntax

Java syntax is the set of rules defining how a Java program is written and interpreted.

> Code:

```
1 public class Syntax {
2     public static void main(String[] args) {
3         System.out.println("Hello, World!");
4     }
5 }
```

public class Main

- **public**: An access modifier indicating that the class is accessible from other classes.
- **class**: A keyword used to define a class in Java.
- **Main**: The name of the class. By convention, class names in Java start with an uppercase letter.

public static void main(String[] args)

- **static**: A keyword indicating that the method belongs to the class, not to instances of the class. It can be called without creating an object of the class.

- **void**: The return type of the method, indicating that it does not return any value.
- **main**: The name of the method. This is the entry point of any Java application.
- **String[] args**: An array of String arguments passed to the method. These are command-line arguments.

System.out.println("Hello, World!")

- **System**: A built-in class in the `java.lang` package.
- **out**: A static field in the **System** class, which is an instance of **PrintStream**.
- **println**: A method of **PrintStream** that prints a message to the standard output (usually the console) followed by a newline.
- **"Hello, World!"**: A string literal that is the message to be printed.

3 Variables in Java

Variables are containers for storing data values. In Java, every variable must be declared before it is used. A variable declaration includes the data type followed by the variable name. Java supports different types of variables, including:

- **Local Variables**: Declared inside a method and accessible only within that method.
- **Instance Variables**: Declared inside a class but outside any method. They are accessible from any method in the class.
- **Static Variables**: Declared with the `static` keyword. These are shared among all instances of the class.

Practical 1

Handling Various Data Types

1 Data Types in Java

Java has two categories of data types: **Primitive Data Types** and **Reference/Object Data Types**.

Primitive Data Types

Primitive data types are the most basic data types available in Java.

> Code:

```java
public class Data {
    public static void main(String[] args) {
        int myNum = 5;
        float myFloatNum = 5.99f;
        double myDoubNum = 5.9999d;
        char myLetter = 'D';
        boolean myBool = true;
        String myText = "Hello";

        System.out.println(myNum);
        System.out.println(myFloatNum);
        System.out.println(myDoubNum);
        System.out.println(myLetter);
        System.out.println(myBool);
        System.out.println(myText);
    }
}
```

```
╭──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 765ms
╰─> java Data.java
5
5.99
5.9999
D
true
Hello
╭──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 817ms
```

Figure 1: Primitve Data Types

- **byte**: 8-bit signed integer. Range: -128 to 127.
- **short**: 16-bit signed integer. Range: -32,768 to 32,767.

- **int**: 32-bit signed integer. Range: -2^{31} to $2^{31}-1$.
- **long**: 64-bit signed integer. Range: -2^{63} to $2^{63}-1$.
- **float**: 32-bit floating-point number.
- **double**: 64-bit floating-point number.
- **char**: 16-bit Unicode character.
- **boolean**: Represents two values: true and false.

Non Primitive Data Types

Reference types in Java are Strings and arrays:

> Code:

```java
public class NonPrimitive {
    public static void main(String[] args) {
        // String Data Type
        String stringVar = "Hello, Java!";
        System.out.println("\nString Data Type:");
        System.out.println("String: " + stringVar);

        // Array Data Type
        int[] intArray = {1, 2, 3, 4, 5};
        System.out.println("\nArray Data Type:");
        System.out.print("intArray: ");
        for (int num : intArray) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

```
──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 817ms
└─> java NonPrimitive.java

String Data Type:
String: Hello, Java!

Array Data Type:
intArray: 1 2 3 4 5
──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 800ms
└─>
```

Figure 2: Non-Primitve Data Types

- **Strings**: Sequences of characters.
- **Arrays**: Containers that hold multiple values of the same type.

Practical 2

Type casting

Type casting is when you assign a value of one primitive data type to another type. There are two types of type casting, implicit typecasting and explicit typecasting which are explained below:

1 Implicit Type Casting

Implicit type casting is done automatically when passing a smaller size type to a larger size type.

```
byte → short → char → int → long → float → double
```

> Code:

```java
public class Implicit {
    public static void main(String[] args) {
        int myInt = 9;
        double myDouble = myInt; // Automatic casting: int to double
        System.out.println(myInt); // Outputs 9
        System.out.println(myDouble); // Outputs 9.0
    }
}
```

> Output:

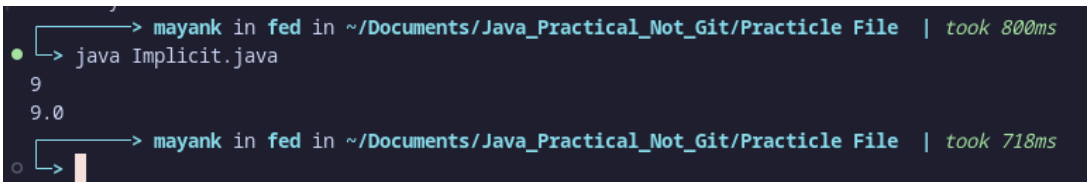

Figure 3: Implicit Type Conversion

2 Explicit Type Casting

Explicit type casting must be done manually by placing the type in parentheses () in front of the value.

```
double → float → long → int → char → short → byte
```

```java
public class Explicit {
    public static void main(String[] args) {
        double myDouble = 9.78d;
        int myInt = (int) myDouble; // Manual casting: double to int

        System.out.println(myDouble);    // Outputs 9.78
        System.out.println(myInt);       // Outputs 9
    }
}
```

> **Output:**

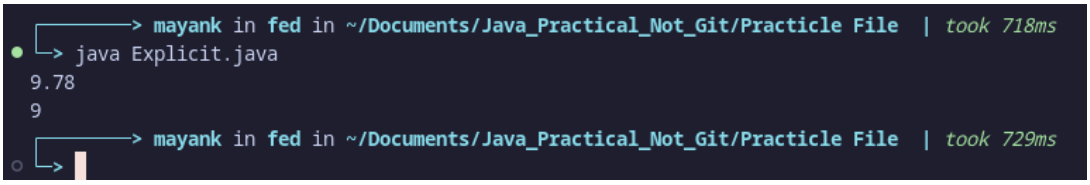

Figure 4: Explicit Type Conversion

Practical 3

Array 1D and 2D

1 1-Dimensional Array

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

> Code:

```java
public class Array {
    public static void main(String[] args) {
        String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
        System.out.println(cars[0]);
        System.out.println(cars[1]);
        System.out.println(cars[2]);
        System.out.println(cars[3]);
        // Changing an element of an array
        cars[0] = "audi";
        System.out.println(cars[0]);
        // Length of an array
        System.out.println(cars.length);
        // Loop through an array
        for (String arr  : cars) {
            System.out.println(arr);
        }
    }
}
```

> Output:

```
┌──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 753ms
└─> java Array.java
Volvo
BMW
Ford
Mazda
audi
4
audi
BMW
Ford
Mazda
┌──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 779ms
```

Figure 5: output of 1-D array

2 Multi Dimensional Array

A multidimensional array is an array of arrays. Multidimensional arrays are useful when you want to store data as a tabular form, like a table with rows and columns.

> Code:

```java
    // Program for Multi-Dimensional Array

public class Array2D {
    public static void main(String[] args) {
        int[][] my2DArr = {{10, 20, 30, 40}, {50, 60, 70}};
        // Accessing Elemensts of array
        System.out.println(my2DArr[0][0]); // 10
        System.out.println(my2DArr[1][2]); // 70
        // change element of array
        my2DArr[0][0] = 100;
        System.out.println(my2DArr[0][0]); //100

        // Loop through a multi dimensional array
        System.out.println("Looping through an array");
        for (int[] row : my2DArr) {
            for(int i : row) {
                System.out.println(i);
            }
        }
    }
}
```

> Output:

```
──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 754ms
└─> java Array2D.java
10
70
100
Looping through an array
100
20
30
40
50
60
70
──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 866ms
└─>
```

Figure 6: output of Multi-D array

Practical 4

Various Control Strucutures

1 For loop

For loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

> Code:

```java
import java.util.Scanner;

public class ForLoop {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int n = sc.nextInt();
        for (int i = 0; i < n; i++) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
}
```

> Output:

```
╭──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 7ms
╰─> java ForLoop.java
Enter a number: 10
0 1 2 3 4 5 6 7 8 9
╭──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 2s
╰─>
```

Figure 7: output of for loop

2 While loop

A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

> Code:

```java
import java.util.Scanner;

public class WhileLoop {

    public static void main(String[] args) {
// Sum of first n numbers
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int n = sc.nextInt();
        int sum = 0;
        while (n > 0) {
            sum += n--;
        }
        System.out.println(sum);
    }
}
```

> **Output:**

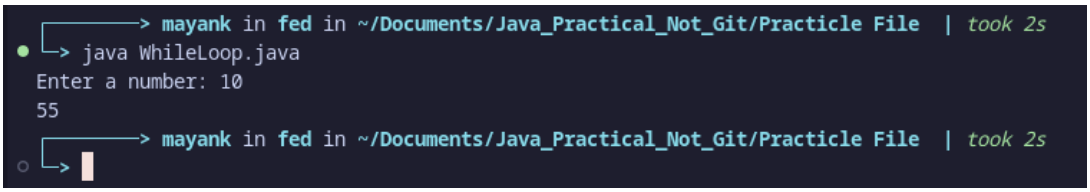

Figure 8: output of while loop

3 Do-While loop

Do-While loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of Exit Control Loop.

> **Code:**

```java
import java.util.Scanner;

public class DoWhile {
    public static void main(String[] args) {
        // Sum of first n numbers
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int n = sc.nextInt();
        int sum = 0;
        do {
            sum += n--;
        } while (n > 0);
        System.out.println(sum);
    }
}
```

> **Output:**

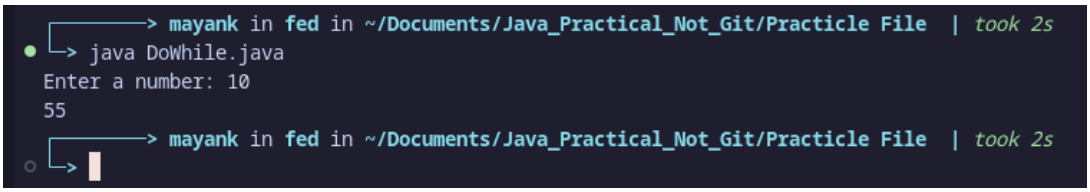

Figure 9: output of do-while loop

Practical 5
Various Decision Strucutures

1 The IF statement

Use the if statement to specify a block of Java code to be executed if a condition is true.

> Code:

```java
public class If {
    public static void main(String[] args) {
        if (20 > 18) {
            System.out.println("20 is greater than 18");
        }
    }
}
```

> Output:

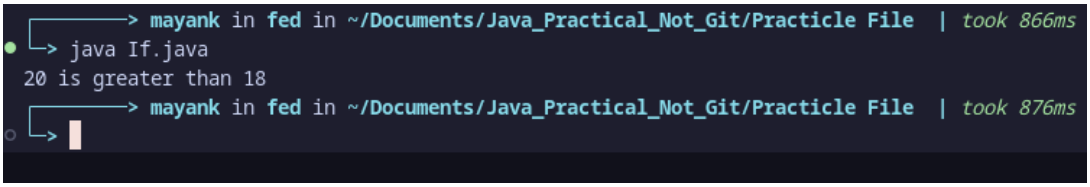

Figure 10: output of if statement

2 The IF-Else statement

Executes one block of code if its condition evaluates to true, and another block of code if it evaluates to false.

> Code:

```java
public class IfElse {
    public static void main(String[] args) {
        int time = 20;
        if (time < 18) {
            System.out.println("Good day.");
        } else {
            System.out.println("Good evening.");
        }
    }
}
```

> **Output:**

```
┌──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 881ms
└─> java IfElse.java
Good evening.
┌──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 741ms
```

Figure 11: output of if-else statement

3 The IF-Else ladder

Executes one block of code if its condition evaluates to true, and then checks other coditions given in else if statements if it is false, or executes the last else block if nothing is true

> **Code:**

```
1  import java.util.Scanner;
2
3  public class IfElseLad {
4      public static void main(String[] args) {
5          Scanner sc = new Scanner(System.in);
6          System.out.print("Enter age -> ");
7          int age = sc.nextInt();
8          if (age < 12) {
9              System.out.println("Child");
10         }else if (age < 18) {
11             System.out.println("Teenager");
12         }else {
13             System.out.println("Adult");
14         }
15     }
16 }
```

> **Output:**

```
┌──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 2s
└─> java IfElseLad.java
Enter age -> 20
Adult
┌──────> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File  | took 3s
```

Figure 12: output of if-else-ladder statement

4 Nested If-Else

We can put If-Else statements inside otehr If-Else statemetns in order to build more complex logic

> **Code:**

```java
import java.util.Scanner;

public class NestedIf {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a -> ");
        int a = sc.nextInt();
        System.out.print("Enter b -> ");
        int b = sc.nextInt();
        System.out.print("Enter c -> ");
        int c = sc.nextInt();
        if (a > b) {
            if (a > c) {
                System.out.println("A is greatest");
            }else {
                System.out.println("C is greatest");
            }
        } else {
            if (b > c) {
                System.out.println("B is greatest");
            }else {
                System.out.println("C is greatest");
            }
        }
    }
}
```

> **Output:**

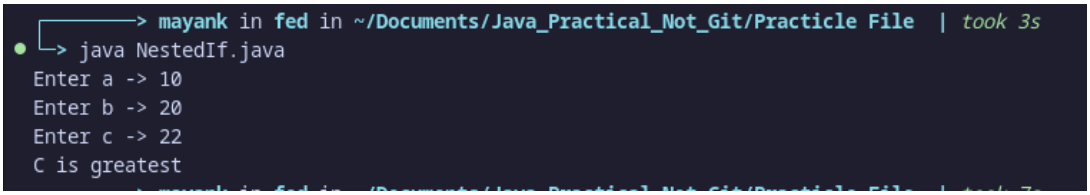

Figure 13: output of nested-if statement

5 Switch statement

The switch statement in Java is a multi-way branch statement. In simple words, the Java switch statement executes one statement from multiple conditions.

> **Code:**

```java
import java.util.Scanner;

```

```java
public class Switch {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int day = sc.nextInt();
        switch (day) {
            case 1:
                System.out.println("Sunday");
                break;
            case 2:
                System.out.println("Monday");
                break;
            case 3:
                System.out.println("Tuesday");
                break;
            case 4:
                System.out.println("Wednesday");
                break;
            case 5:
                System.out.println("Thursday");
                break;
            case 6:
                System.out.println("Friday");
                break;
            case 7:
                System.out.println("Saturday");
                break;
            default:
                System.out.println("Invalid day");
        }
    }
}
```

> **Output:**

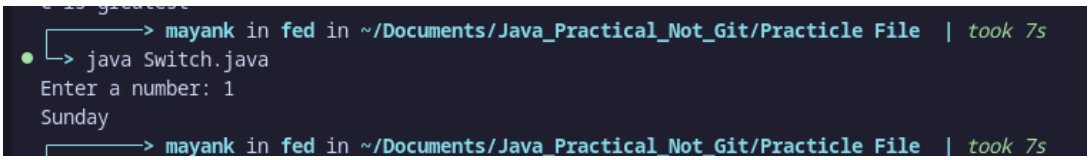

Figure 14: output of switch statement

Practical 6

Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

1 Example of Recursion

Code:

```java
public class Recursion {
    public static void main(String[] args) {
        int result = sum(10);
        System.out.println(result);
    }
    public static int sum(int k) {
        if (k > 0) {
        return k + sum(k - 1);
        } else {
        return 0;
        }
    }
}
```

> Output:

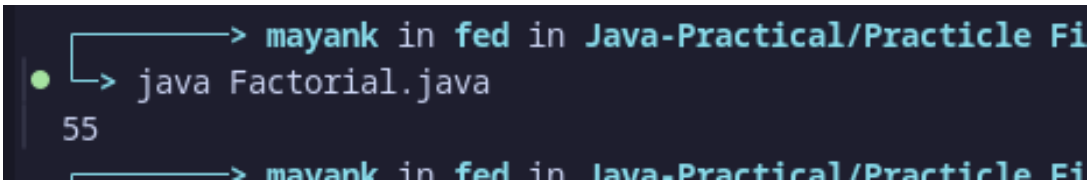

Figure 15: Recursion Example

2 Factorial of a number using Recursion

Code:

```java
public class Factorial {
    static int factorial(int n) {
        if (n == 0 || n == 1)
            return 1;
        return n * factorial(n - 1);
    }
```

```
7       public static void main(String[] args) {
8           int ans = factorial(5);
9           System.out.println("Factorial of 5 is :" + ans);
10      }
11  }
```

> **Output:**

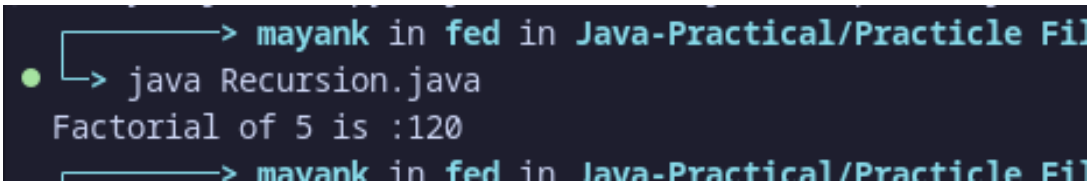

Figure 16: Factorial using example

Practical 7

Method Overloading by passing objects as arguments

In object-oriented programming, method overloading is a feature that allows you to define multiple methods with the same name but different parameters. In the context of passing objects as arguments, method overloading can be used to handle different types or classes of objects.

1 Example of Method Overloading by passing objects as arguments

Code:

```
class Circle {
    double radius;
    Circle(double radius) {
        this.radius = radius;
    }
}

class Rectangle {
    double length, width;
    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
}

class AreaCalculator {
    double calculateArea(Circle circle) {
        return Math.PI * circle.radius * circle.radius;
    }
    double calculateArea(Rectangle rectangle) {
        return rectangle.length * rectangle.width;
    }
}

public class MOPOAA {
    public static void main(String[] args) {
        Circle circle = new Circle(5);
        Rectangle rectangle = new Rectangle(4, 6);
        AreaCalculator calculator = new AreaCalculator();
        System.out.println
        ("Area of Circle: "+calculator.calculateArea(circle));
        System.out.println
```

```
33          ("Area of Rectangle:"+calculator.calculateArea(rectangle));
34      }
35  }
```

> **Output:**

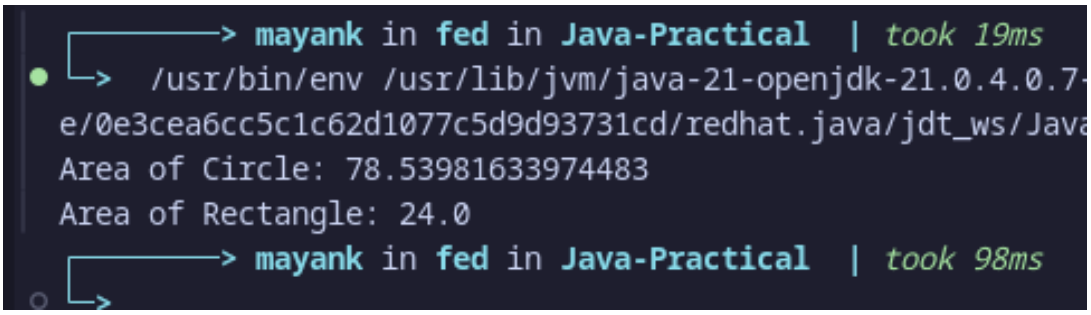

Figure 17: Method Overloading by passing objects as arguments

Practical 8

Constructor Overloading by passing objects as arguments

Constructor overloading in object-oriented programming allows a class to have multiple constructors with different parameter lists. This enables the creation of objects in different ways. When you overload constructors by passing objects as arguments, you can create new objects based on existing objects.

1 Example of Constructor Overloading by passing objects as arguments

Code:

```java
class Book {
    String title;
    String author;
    int pages;
    // Constructor 1: No arguments
    Book() {
        this.title = "Unknown";
        this.author = "Unknown";
        this.pages = 0;
    }
    // Constructor 2: Passing title, author, and pages
    Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }
    /*
    Constructor 3: Passing an existing Book object
    (copy constructor)
    */
    Book(Book existingBook) {
        this.title = existingBook.title;
        this.author = existingBook.author;
        this.pages = existingBook.pages;
    }
    void displayDetails() {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Pages: " + pages);
    }
}

```

```java
public class COBPOAA {
    public static void main(String[] args) {
        // Using Constructor 1
        Book book1 = new Book();
        System.out.println("Book 1 details:");
        book1.displayDetails();

        // Using Constructor 2
        Book book2 = new Book("1984", "George Orwell", 328);
        System.out.println("\nBook 2 details:");
        book2.displayDetails();

        // Using Constructor 3 (Copy Constructor)
        Book book3 = new Book(book2);
        System.out.println
        ("\nBook 3 details (copied from Book 2):");
        book3.displayDetails();
    }
}
```

> **Output:**

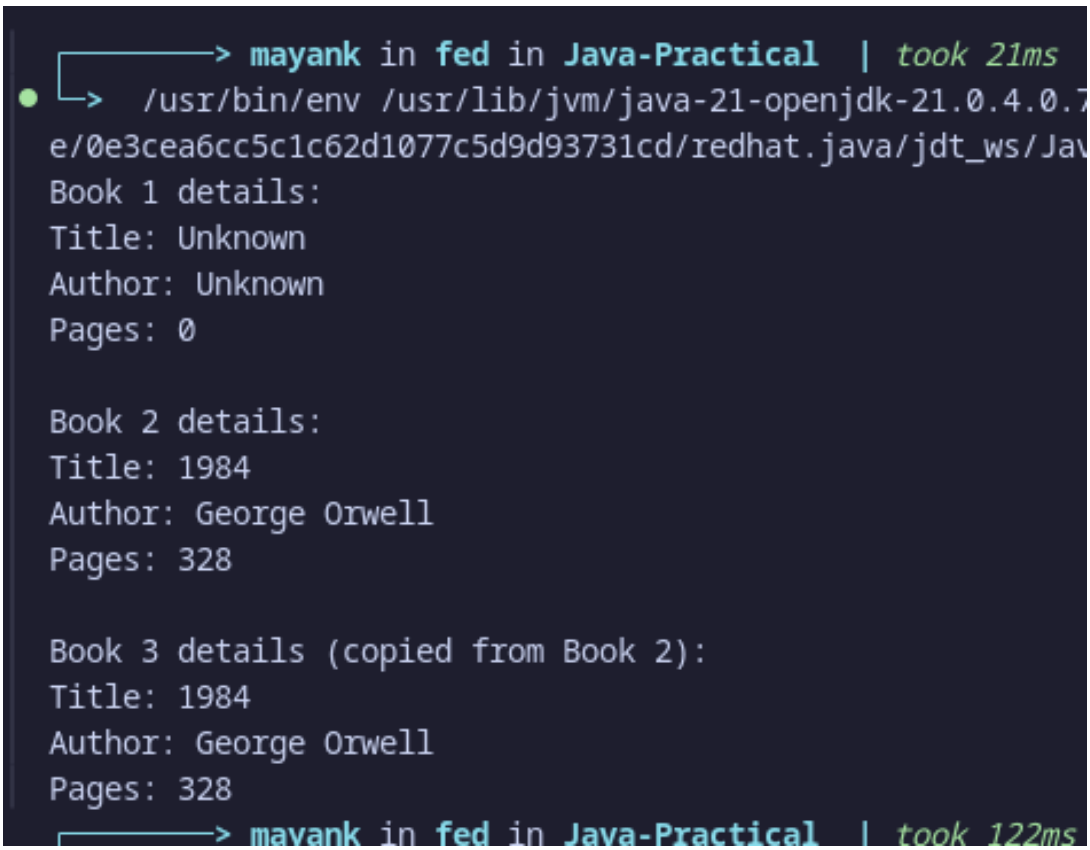

Figure 18: Constructor Overloading by passing objects as arguments

Practical 9

Access Control and the Usage of Static, Final, and Finalize

In Java, access control mechanisms and the usage of keywords such as `static`, `final`, and the method `finalize()` are essential components of object-oriented programming. They help in controlling the visibility of class members, managing shared resources, and enhancing performance.

1 Access Control

Java provides four types of access control modifiers:

- **Public:** Members are accessible from any other class.
- **Protected:** Members are accessible within the same package and by subclasses.
- **Default:** (No modifier) Members are accessible within the same package.
- **Private:** Members are accessible only within the class itself.

Example of Access Control Modifiers

Code:

```
class Example {
    public int publicVar = 10;
    protected int protectedVar = 20;
    int defaultVar = 30;
    private int privateVar = 40;
}
```

2 Static Keyword

The `static` keyword in Java is used for memory management. It can be applied to variables, methods, blocks, and nested classes. A `static` member belongs to the class rather than an instance of the class.

Usage of Static Variables and Methods

Code:

```java
class StaticExample {
    static int count = 0;

    // Static method
    static void increment() {
        count++;
    }
}
```

In this example, the static variable `count` is shared across all instances of the `StaticExample` class, and the `increment()` method can be called without creating an object.

3 Final Keyword

The `final` keyword can be used to define constants, prevent method overriding, and prevent inheritance. When applied to:

- **Variables:** Makes the variable constant, meaning it cannot be reassigned.
- **Methods:** Prevents the method from being overridden in subclasses.
- **Classes:** Prevents the class from being subclassed.

Example of Final Variable, Method, and Class

Code:

```java
final class FinalClass {
    final int MAX_VALUE = 100;

    final void display() {
        System.out.println("This is a final method.");
    }
}

// Uncommenting the below class will result in a compile-time error
// class SubClass extends FinalClass {}
```

4 Finalize Method

The `finalize()` method is called by the garbage collector before an object is destroyed. It is used to perform cleanup actions before an object is garbage collected.

Usage of Finalize Method

Code:

```java
class FinalizeExample {
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Finalize method called.");
    }

    public static void main(String[] args) {
        FinalizeExample obj = new FinalizeExample();
        obj = null;  // Make the object eligible for garbage collection
        System.gc(); // Request garbage collection
    }
}
```

5 Example

```java
class AccessDemo {
    public int publicVar = 10;
    private int privateVar = 20;

    static int staticVar = 30;

    final int finalVar = 40;

    public void show() {
        System.out.println("Public Variable: " + publicVar);
        System.out.println("Private Variable: " + privateVar);
        System.out.println("Static Variable: " + staticVar);
        System.out.println("Final Variable: " + finalVar);
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("Finalize method called.");
    }

    public static void main(String[] args) {
        AccessDemo demo = new AccessDemo();
        demo.show();

        // Static variable can be accessed directly without creating an object
        System.out.println("Accessing static variable without object: " + Access

        // Making object eligible for garbage collection
        demo = null;
        System.gc();  // Request garbage collection
    }
}
%
```

> **Output:**

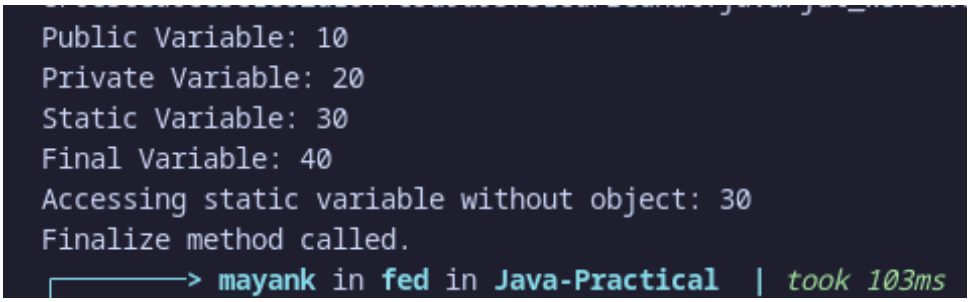

Figure 19: Example Program Output

Practical 10

Command Line Arguments

Command line arguments are parameters passed to the main method when you run a program from the command line.

1 Syntax

Code:

```
public static void main(String[] args) {}
```

Here, args is an array of String objects that holds the command line arguments passed to the program.

2 Example of Command Line Argumesnts

Code:

```
public class CommandLineExample {
    public static void main(String[] args) {
        // Check if any arguments were passed
        if (args.length > 0) {
            System.out.println("Command line arguments:");

            // Iterate over the arguments and print each one
            for (int i = 0; i < args.length; i++) {
                System.out.println
                ("Argument " + i + ": " + args[i]);
            }
        } else {
            System.out.println
            ("No command line arguments were passed.");
        }
    }
}
```

3 How to Run the Program with Command Line Arguments

Run the program with following command:

```
java CommandLineExample.java John Jinny Kia
```

> **Output:**

```
┌──────> mayank in fed in Java-Practical/Practicle File  | took 9ms
└─> java CommandLineExample.java John Jinny Kia
Command line arguments:
Argument 0: John
Argument 1: Jinny
Argument 2: Kia
```

Figure 20: Command Line Arguments

Practical 11

Various types of inheritance by applying various access controls to its data members and methods

Inheritance in Java is a mechanism where a new class (child/subclass) inherits the properties (fields) and behaviours (methods) of an existing class (parent/superclass). It promotes code reuse and allows for an organized hierarchy.

Types of Inheritance in Java: Java supports different types of inheritance, but multiple inheritance (where a class inherits from more than one class) is not supported directly to avoid ambiguity. Below are the types Java supports:

1 Single Inheritance:

A class inherits from a single superclass.

Code:

```java
class Employee {
    float salary = 40000;
}

class Programmer extends Employee {
    int bonus = 10000;
    public static void main(String args[]) {
        Programmer p = new Programmer();
        System.out.println("Programmer salary is:" + p.salary);
        System.out.println("Bonus of Programmer is:" + p.bonus);
    }
}
```

> Output:

```
> mayank in fed in Java-Practical |
> /usr/bin/env /usr/lib/jvm/java-21-openjdk-21.0.5.0.1
e3cea6cc5c1c62d1077c5d9d93731cd/redhat.java/jdt_ws/Java-P
Programmer salary is:40000.0
Bonus of Programmer is:10000
> mayank in fed in Java-Practical | took 107ms
```

Figure 21: ouput of Single Inheritance

2 Multilevel Inheritance:

A class inherits from a class that is also a subclass of another class, forming a chain of inheritance.

Code:

```java
class animal {
    public static void eat() {
        System.out.println("Animal eats food");
    }
}

class dog extends animal {
    public static void bark() {
        System.out.println("Dog barks");
    }
}

class puppy extends dog {

    public static void play() {
        System.out.println("Puppy plays");
    }

    public static void main(String[] args) {
        puppy obj = new puppy();
        obj.eat();
        obj.bark();
        obj.play();
    }
}
```

> Output:

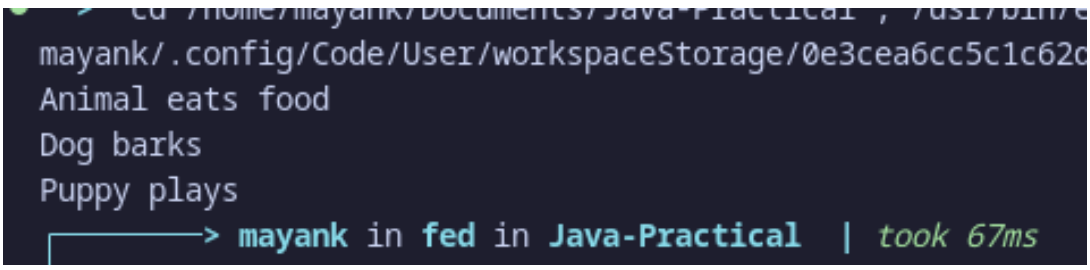

Figure 22: ouput of Multilevel Inheritance

3 Hierarchical Inheritance:

Multiple classes inherit from a single superclass.

Code:

```java
class Animal {
    public void eat() {
        System.out.println("Animal eats");
    }
}

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

class Cat extends Animal {
    public void meow() {
        System.out.println("Cat meows");
    }
}

public class HInherit {
    public static void main(String[] args) {
        // Create Dog object and callmethods
        Dog dog = new Dog();
        dog.eat(); // Inherited method from Animal
        dog.bark(); // Specific method of Dog
        // Create Cat object and call methods
        Cat cat = new Cat();
        cat.eat(); // Inherited method from Animal
        cat.meow(); // Specific method of Cat
    }
}
```

> Output:

```
mayank/.config/Code/User/workspaceStorage/0e3cea6cc5c1c62c
Animal eats
Dog barks
Animal eats
Cat meows
╭──────> mayank in fed in Java-Practical  | took 68ms
```

Figure 23: ouput of Hierarchical Inheritance

Practical 12

Method overriding

Method overriding in Java is a key concept of Object-Oriented Programming (OOP), where a subclass (child class) provides a specific implementation of a method that is already defined in its superclass (parent class). The method in the subclass should have the same name, return type, and parameters as in the superclass. The idea is to allow a subclass to modify or enhance the behavior of the method that it inherits from its parent class

1 Example Using Code:

```
class Car {
    public String bestModel() {
        return "";
    }
}

class BMW extends Car {
    @Override
    public String bestModel() {
        return "M4 Competition";
    }
}

class Audi extends Car {
    @Override
    public String bestModel() {
        return "RS7";
    }
}

class Porsche extends Car {
    @Override
    public String bestModel() {
        return "911 gt3rs";
    }
}

public class MetOvr {
    public static void main(String[] args) {
        BMW bmw = new BMW();
        Audi audi = new Audi();
        Porsche porsche = new Porsche();

        System.out.println(bmw.bestModel());
```

```
35              System.out.println(audi.bestModel());
36              System.out.println(porsche.bestModel());
37          }
38      }
```

> **Output:**

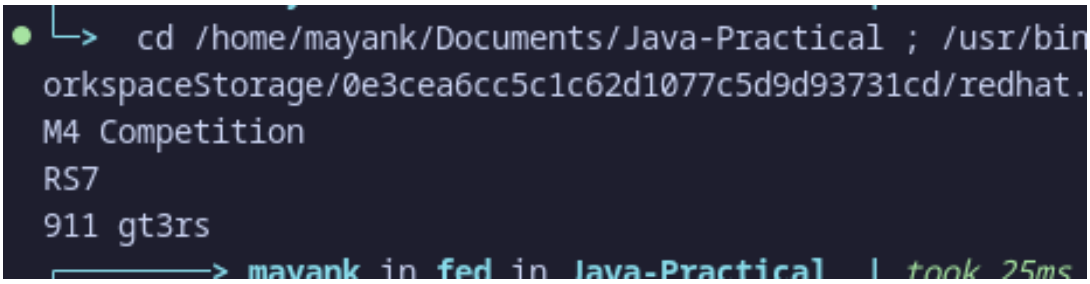

Figure 24: ouput of Method overriding

> **Example of Method overriding by Diagram:**

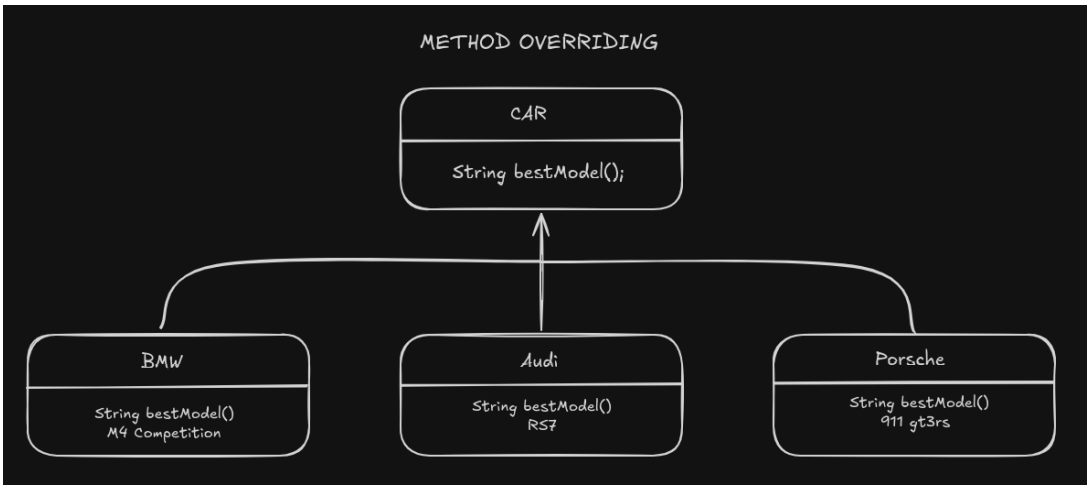

Figure 25: Example of Method overriding

Practical 13

Abstract class

An abstract class in Java is a class that cannot be instantiated on its own. It is used to represent an abstract concept that other classes can inherit from, providing a common structure. An abstract class can have both abstract methods (methods without a body, meant to be implemented by subclasses) and concrete methods (methods with a body)

1 Syntax:

```
    abstract class ClassName {
    // Abstract method (no body)
    public abstract void methodName();

    // Concrete method (with body)
    public void anotherMethod() {
        System.out.println("This is a concrete method.");
    }
}
```

Code:

```
abstract class Payment {
    public abstract void makePayment(double amount);

    public void paymentDetails(double amount) {
        System.out.println("Payment of $" + amount + " is being processed.");
    }
}

class CreditCardPayment extends Payment {
    @Override
    public void makePayment(double amount) {
        System.out.println("Processing Credit Card Payment of $" + amount);
    }
}

class PayPalPayment extends Payment {
    @Override
    public void makePayment(double amount) {
        System.out.println("Processing PayPal Payment of $" + amount);
    }
}

public class Abs {
```

```
24      public static void main(String[] args) {
25          Payment creditCard = new CreditCardPayment();
26          Payment payPal = new PayPalPayment();
27
28          creditCard.paymentDetails(100.50);
29          creditCard.makePayment(100.50);
30
31          System.out.println();
32
33          payPal.paymentDetails(50.75);
34          payPal.makePayment(50.75);
35      }
36  }
```

> **Output:**

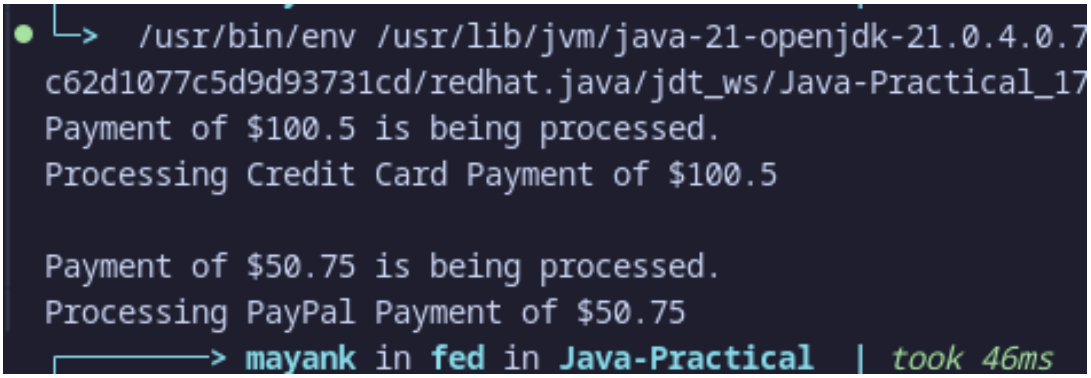

Figure 26: ouput of abstract class

Practical 14

Nested class

A nested class in Java is a class defined within another class. Nested classes can be used to logically group classes that are only used in one place, improving encapsulation and making the code more readable. Nested classes have access to the members (both static and non- static) of the outer class, depending on whether they are static or non-static themselves.

1 Static Nested Class :

A static nested class in Java is a nested class that is declared static. Since it is a static member of the outer class, it can be accessed without creating an instance of the outer class. However, unlike non-static inner classes, a static nested class cannot access non-static members (fields or methods) of the outer class directly. It can only access the static members (both fields and methods) of the outer class.

Code:

```
class outerClass {
    static int outer_x = 34; // Static member
    int outer_y = 102;
    // Non-static member
    private static int outerPrivate = 44; // Static member
    static class innerClass {
        // Static nested class
        void display() {
            outerClass outer = new outerClass();
            System.out.println("Value of x: " + outer_x);
            System.out.println("Value of y: " + outer.outer_y);
            System.out.println("Value of private variable: " + outerPrivate);
        }
    }
}

public class Nest { // Separate class to run the main method
    public static void main(String[] args) {
        System.out.println("Static Nested Class");
        outerClass.innerClass obj = new outerClass.innerClass();
        obj.display(); // Call the display method}
    }
}
```

> **Output:**

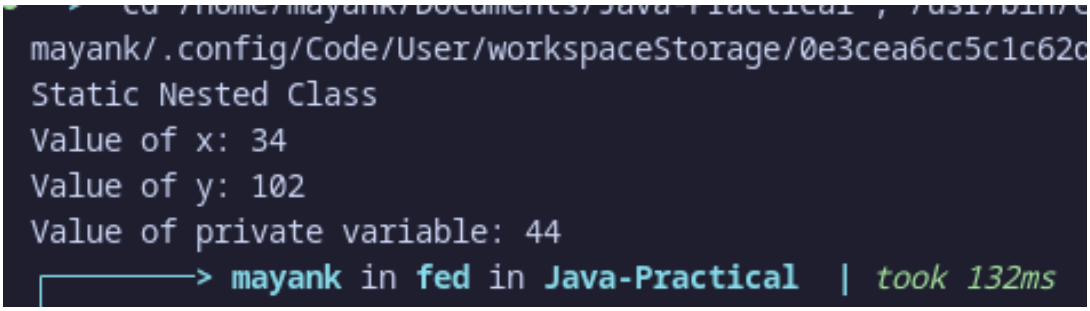

Figure 27: ouput of nested class

Practical 15

Constructor chaining

In Java, constructor chaining is a sequence of invoking constructors upon initializing an object. It is used when we want to invoke a number of constructors, one after another by using only an instance.

Code:

```java
class Animal {
    String name;
    String speak;

    Animal(){
        this("Cat", "Meow");
        System.out.println("This is Default Constructor");
    }

    Animal(String name, String speak) {
        this.name = name;
        this.speak = speak;
        System.out.println("This is Parameterised Constructor");
        System.out.println(name + " | " + speak);
    }
}

public class Chain {
    public static void main(String[] args) {
        Animal obj1 = new Animal();
    }
}
```

> Output:

```
┌──────> mayank in fed in Java-Practical/Practicle File
└─> java Chain.java
This is Parameterised Constructor
Cat | Meow
┌──────> mayank in fed in Java-Practical/Practicle File
```

Figure 28: ouput of constructor chaining

Practical 16

Importing Classes from User-defined Package and Creating Packages Using Access Protection

1 Packages

In Java, a package is a namespace that organizes a set of related classes and interfaces. Packages serve as containers for classes, helping to avoid name conflicts and making code easier to locate and manage. Java's standard library is organized into packages (like `java.util` or `java.io`), allowing developers to reuse classes easily without having to rewrite common functionalities. By using packages, we can logically separate different components of a program and control their visibility to other classes.

Packages also play a critical role in implementing access control. Java provides four levels of access control: **public**, **protected**, **default** (package-private), and **private**. These access levels determine how classes and members (fields and methods) can be accessed across different packages.

- **Public**: Accessible from any other class, irrespective of the package.
- **Protected**: Accessible within the same package and by subclasses in other packages.
- **Default (Package-private)**: Accessible only within the same package; no modifier is needed.
- **Private**: Accessible only within the class in which it is defined.

2 Creating and Importing a User-defined Package

The following code demonstrates how to define a package, create a class within the package with various access levels, and then import and use that class in a main program.

Code for the Package (`mypackage/MyClass.java`):

```
1  package mypackage;
2
3  public class MyClass {
4      public int publicVar = 10;
5      protected int protectedVar = 20;
6      int defaultVar = 30; // default access
7      private int privateVar = 40;
8
9      public void display() {
10         System.out.println("Public variable: " + publicVar);
11         System.out.println("Protected variable: " + protectedVar);
```

```
        System.out.println("Default variable: " + defaultVar);
        System.out.println("Private variable: " + privateVar);
    }
}
```

Code for Importing the Package (`ImportPackage.java`):

```
import mypackage.MyClass;

public class ImportPackage {
    public static void main(String[] args) {
        MyClass obj = new MyClass();

        // Accessing variables with different access modifiers
        System.out.println("Accessing public variable: " + obj.publicVar);
        // System.out.println("Accessing protected variable: " + obj.protectedVa
        // System.out.println("Accessing default variable: " + obj.defaultVar);
        // System.out.println("Accessing private variable: " + obj.privateVar);

        obj.display(); // Method displays all variables within the same class
    }
}
```

Output:

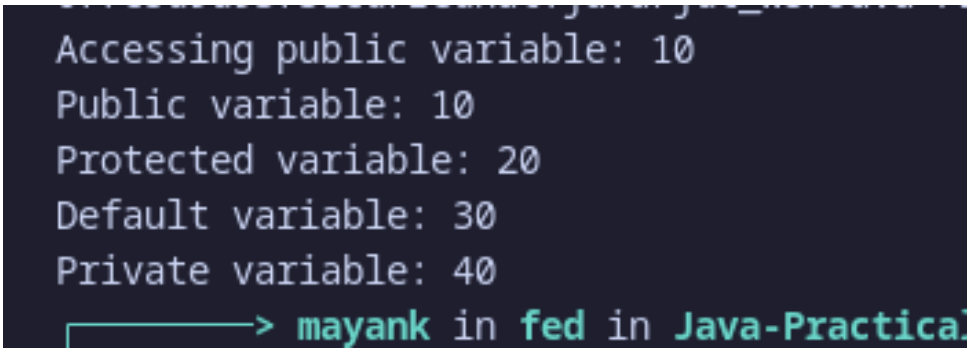

Figure 29: Output demonstrating access control when importing from a user-defined package

Practical 17

Interfaces, nested interfaces and use of extending interfaces

In Java, an interface is a reference type that is similar to a class but is used to specify a set of abstract methods (methods without a body). Interfaces define what a class must do but not how it does it. Any class that implements an interface must provide an implementation for all of its abstract methods.

Code:

```
interface Printable {
    // Defining an interface
    void print(); // Abstract method
}

class Document implements Printable {

    @Override
    public void print() {
        System.out.println("Printing a document ...");
    }
}

class Image implements Printable {

    @Override
    public void print() {
        System.out.println("Printing an image...");
    }
}

public class InterfaceExample {

    public static void main(String[] args) {
        // Creating objects of classes that implement the Printable interfac
        Printable doc = new Document();
        Printable img = new Image();
        // Calling the print method using interface references
        doc.print();
        img.print();
    }
}
```

> **Output:**

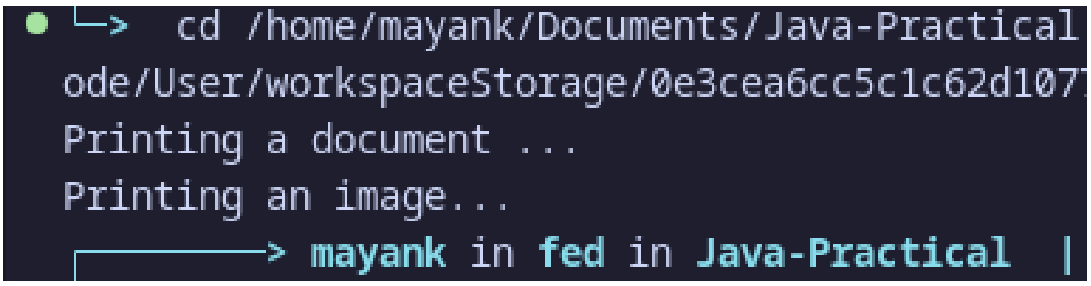

Figure 30: ouput of interface

1 Nested Interfaces

A nested interface in Java is an interface that is declared within another interface or class. This allows better organization of code, especially when the interface is only relevant in the context of the enclosing class or interface.

```
1
2       class Vehicle { // Outer class
3
4       interface Engine { // Nested
5
6           void start();
7
8           void stop();
9       }
10
11      // Method to demonstrate the nested interface
12      public void useEngine() {
13          // Implementing the nested interface
14          Engine engine = new Engine() {
15              @Override
16              public void start() {
17                  System.out.println("Engine is starting...");
18              }
19
20              @Override
21              public void stop() {
22                  System.out.println("Engine is stopping...");
23              }
24
25          };
26          engine.start();
27          engine.stop();
28      }
29  }
30
31  public class Main {
32
```

```
33      public static void main(String[] args) {
34          System.out.println("Nested Interface (Interface within Class)");
35          Vehicle vehicle = new Vehicle();
36          // Using the nested interface
37          vehicle.useEngine();
38      }
39  }
```

```
077c5d9d93731cd/redhat.java/jdt_ws/Java-Practical_1
Nested Interface (Interface within Class)
Engine is starting...
Engine is stopping...
╭──────────> mayank in fed in Java-Practical  | took
```

Figure 31: ouput of nested interface

2 Extended Interfaces

```
1  interface Animal { // Outer interface
2  void sound();
3  interface Behavior { // Nested interface
4      void eat();
5  }
6  }
7
8  class Dog implements Animal, Animal.Behavior {
9
10 @Override
11 public void sound() {
12     System.out.println("Dog barks");
13 }
14
15 // Implementing the eat method from the nested interface
16 @Override
17 public void eat() {
18     System.out.println("Dog eats dog food");
19 }
20 }
21
22 public class Inter {
23
24 public static void main(String[] args) {
25     System.out.println("Nested Interface(Interface within Interface )");
26     Dog dog = new Dog();
27     dog.sound(); // Call the soundmethod dog
28     dog.eat(); // Call the eat
29
```

```
30  }
31  }
```

```
> mayank in fed in ~/Documents/Java-Lab  | took
> java Inter.java
Nested Interface(Interface within Interface )
Dog barks
Dog eats dog food
> mayank in fed in ~/Documents/Java-Lab  | took
```

Figure 32: ouput of Extended interface

Practical 18

Exception Handling - Using Predefined Exception

1 Exceptions

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Out of bound
- Null reference
- Type mismatch
- Opening an unavailable file
- Database errors
- Arithmetic errors

2 Errors

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

3 Exception Handling

Exception handling in Java is a mechanism used to handle runtime errors, allowing the normal flow of the application to be maintained. Java provides a set of predefined exceptions in the `java.lang` package, which can handle common runtime errors, such as `ArithmeticException`, `NullPointerException` and more. These exceptions are part of Java's standard library and extend the `Exception` class.

4 Example Code for Exception Handling Using Predefined Exception

The following code demonstrates handling an `ArithmeticException` using a `try-catch` block.

Code:

```java
public class ExceptionHandling {
    public static void main(String[] args) {
        int a = 10;
        int b = 0;

        try {
            int result = a / b; // This will cause an ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: Division by zero is not allowe
            System.out.println("Error: " + e);
        }

        System.out.println("Program continues after handling the exception.");
    }
}
```

Output:

```
Exception caught: Division by zero is not allowed.
Error: java.lang.ArithmeticException: / by zero
Program continues after handling the exception.
╭──────> mayank in fed in Java-Practical  | took 45ms
```

Figure 33: Output demonstrating exception handling for ArithmeticException in Java

Practical 19

Exception Handling - Creating User-defined Exceptions

1 User Defined Exceptions

In Java, we can create custom exceptions (user-defined exceptions) by extending the `Exception` class. User-defined exceptions allow developers to create exceptions specific to the application's needs, providing more meaningful error messages and handling unique error conditions.

A user-defined exception is created by defining a new class that extends `Exception` or any of its subclasses. By overriding the `Exception` class's constructors, we can customize the exception message and implement specific behaviors for our custom exception.

2 Throw Keyword

Java's exception-handling mechanism includes the throw keyword, which is used to explicitly throw an exception. Exceptions can be thrown in two ways:

- **Implicit Throwing**: This occurs when the Java runtime system automatically throws an exception in response to common errors (e.g., `NullPointerException` or `ArithmeticException`).
- **Explicit Throwing**: Using the throw keyword, developers can explicitly throw an exception when a specific condition is met. This is often used with user-defined exceptions to control error handling based on custom logic.

For example, we can create an exception that will be thrown if a user enters an invalid input, such as an age below a certain threshold. The throw keyword enables us to control precisely when and where an exception occurs, ensuring more robust error handling.

3 Example Code for User-defined Exception

The following code demonstrates how to define and use a custom exception in Java.

Code:

```
1 public class UserDefinedException {
2     public static void main(String[] args) {
3         try {
4             checkAge(15); // This will throw an InvalidAgeException
5         } catch (InvalidAgeException e) {
6             System.out.println("Exception caught: " + e.getMessage());
7         }
```

```
8
9           System.out.println("Program continues after handling the exception.");
10      }
11
12      // Method to check age
13      static void checkAge(int age) throws InvalidAgeException {
14          if (age < 18) {
15              throw new InvalidAgeException("Age must be 18 or above.");
16          }
17          System.out.println("Age is valid.");
18      }
19  }
20
21  // Custom exception class
22  class InvalidAgeException extends Exception {
23      public InvalidAgeException(String message) {
24          super(message);
25      }
26  }
```

Output:

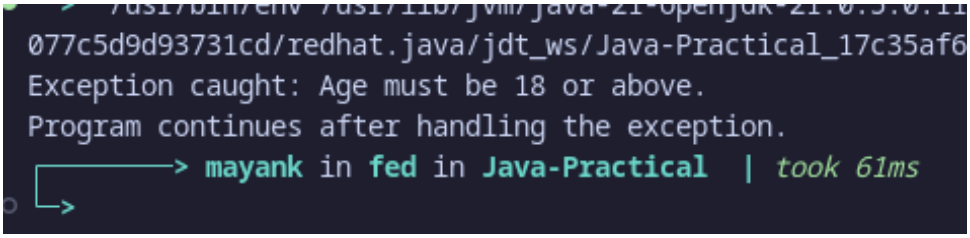

Figure 34: Output demonstrating a user-defined exception in Java

Practical 20

Multithreading by extending Thread Class

1 Program:

```
class EvenThread extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
                System.err.println("Even Thread");
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
        }
    }
}

class OddThread extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
                System.err.println("Odd Thread");
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

        }
    }
}
public class MultiThread {
    public static void main(String[] args) {
        Thread odd = new OddThread();
        Thread even = new EvenThread();

        even.start();
        odd.start();
    }

}
```

Ouptut:

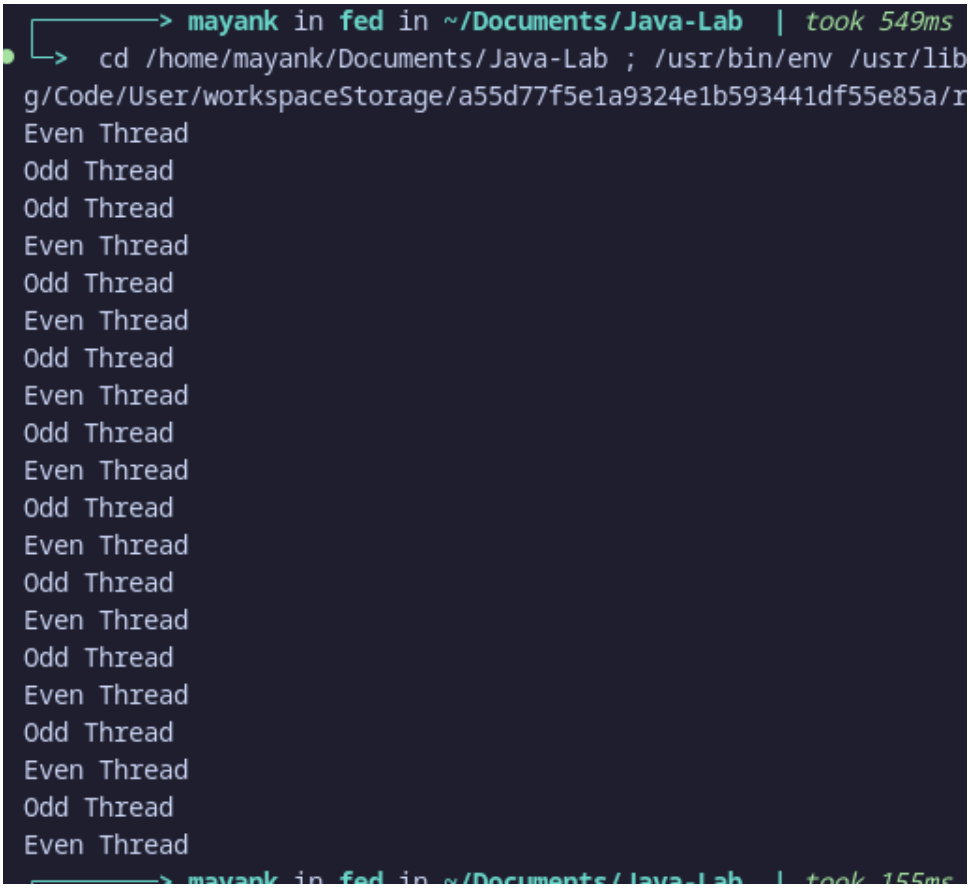

Figure 35: Multithreading using thread class

Practical 21

Multithreading by implementing Runnable Interface

1 Program:

```
1
2       public class LambdaThread {
3
4       public static void main(String[] args) {
5           Runnable obj1 = () -> {
6               for (int i = 0; i < 6; i++) {
7                   System.err.println("Thread 1");
8                   try {
9                       Thread.sleep(10);
10                  } catch (InterruptedException e) {
11                      e.printStackTrace();
12                  }
13              }
14          };
15          Runnable obj2 = () -> {
16              for (int i = 0; i < 6; i++) {
17                  System.err.println("Thread 2");
18                  try {
19                      Thread.sleep(10);
20                  } catch (InterruptedException e) {
21                      e.printStackTrace();
22                  }
23              }
24          };
25
26          Thread t1 = new Thread(obj1);
27          Thread t2 = new Thread(obj2);
28
29          t1.start();
30          t2.start();
31      }
32  }
```

Ouptut:

Figure 36: Mutli threading using runnable interface

Practical 22

Thread life cycle

1 Thread States

- New State
- Runnable State
- Blocked State
- Waiting State
- Terminated State

```
    // Java program to demonstrate thread states
class thread implements Runnable {
    public void run()
    {
        // moving thread2 to timed waiting state
        try {
            Thread.sleep(1500);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(
            "State of thread1 while it called join() method on thread2 -"
            + Test.thread1.getState());
        try {
            Thread.sleep(200);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class Test implements Runnable {
    public static Thread thread1;
    public static Test obj;

    public static void main(String[] args)
    {
        obj = new Test();
        thread1 = new Thread(obj);

```

```
34          // thread1 created and is currently in the NEW
35          // state.
36          System.out.println(
37              "State of thread1 after creating it - "
38              + thread1.getState());
39          thread1.start();
40
41          // thread1 moved to Runnable state
42          System.out.println(
43              "State of thread1 after calling .start() method on it - "
44              + thread1.getState());
45      }
46
47      public void run()
48      {
49          thread myThread = new thread();
50          Thread thread2 = new Thread(myThread);
51
52          // thread2 created and is currently in the NEW
53          // state.
54          System.out.println(
55              "State of thread2 after creating it - "
56              + thread2.getState());
57          thread2.start();
58
59          // thread2 moved to Runnable state
60          System.out.println(
61              "State of thread2 after calling .start() method on it - "
62              + thread2.getState());
63
64          // moving thread2 to timed waiting state
65          try {
66              // moving thread2 to timed waiting state
67              Thread.sleep(200);
68          }
69          catch (InterruptedException e) {
70              e.printStackTrace();
71          }
72          System.out.println(
73              "State of thread2 after calling .sleep() method on it - "
74              + thread2.getState());
75
76          try {
77              // waiting for thread2 to die
78              thread2.join();
79          }
80          catch (InterruptedException e) {
81              e.printStackTrace();
82          }
83          System.out.println(
84              "State of thread2 when it has finished it's execution - "
```

```
85                 + thread2.getState());
86         }
87 }
```

Ouptut:

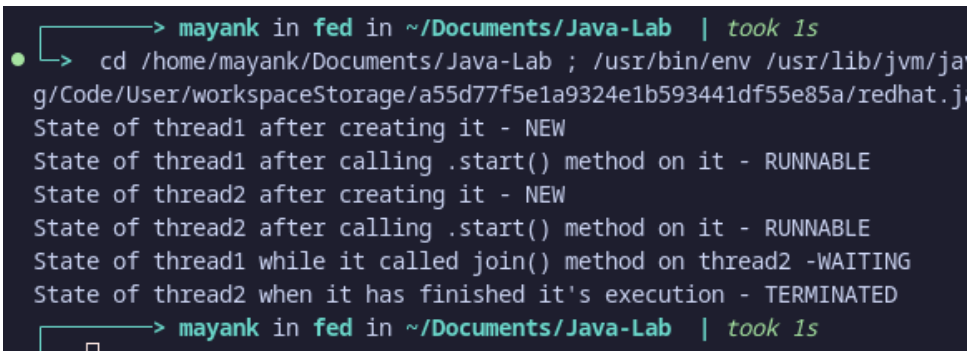

Figure 37: Thread Life Cycle

Practical 23

StringBuffer class and its methods

StringBuffer is a class in Java that represents a mutable sequence of characters. It provides an alternative to the immutable String class, allowing you to modify the contents of a string without creating a new object every time.

1 Append Method

Code:

```
    public class StringBufferAppend {
        public static void main(String[] args)
        {
            // Append Method
            StringBuffer sb = new StringBuffer();
            sb.append("Hello");
            sb.append(" ");
            sb.append("world");
            String message = sb.toString();
            System.out.println(message);
        }
    }
```

> Output:

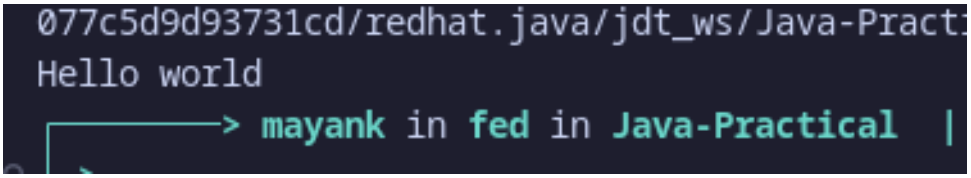

Figure 38: Append Method

2 Insert Method

Code:

```
public class StringBufferInsert {
    // Insert Method in String Buffer
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");
        sb.insert(3, "HHH");
        String message = sb.toString();
```

```
        System.out.println(message);
    }
}
```

> **Output:**

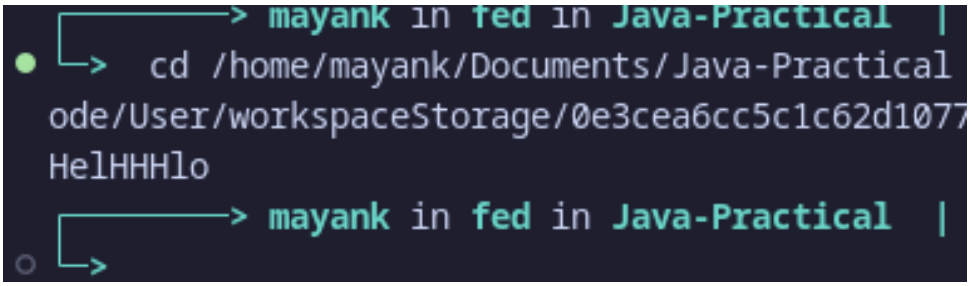

Figure 39: Insert Method

3 Replace Method

Code:

```
public class StringBufferReplace {
    // Replace Method in String Buffer
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello World");
        sb.replace(6, 11, "Java");
        String message = sb.toString();
        System.out.println(message);
    }
}
```

> **Output:**

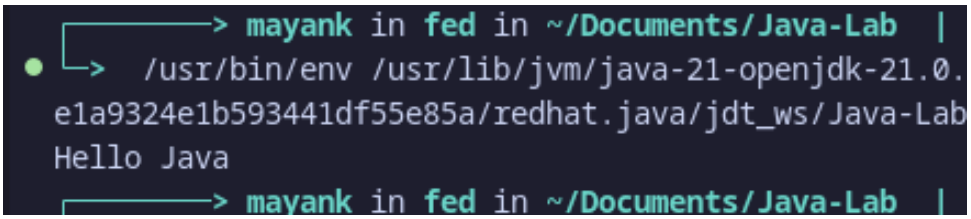

Figure 40: Delete Method

4 Delete Method

Code:

```java
public class StringBufferDelete {
    // Delete Method in String Buffer
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello World");
        sb.delete(5, 11);
        String message = sb.toString();
        System.out.println(message);
    }
}
```

> **Output:**

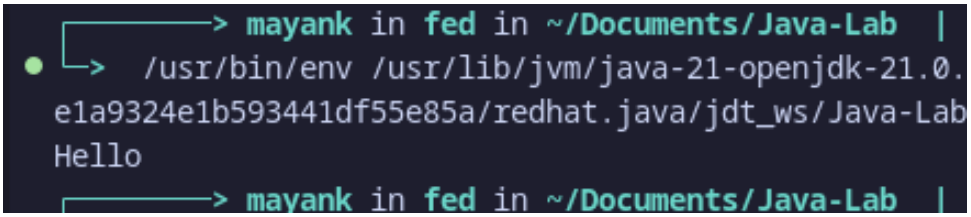

Figure 41: Delete Method

5 Reverse Method

Code:

```java
public class StringBufferReverse {
    // Reverse Method in String Buffer
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");
        sb.reverse();
        String message = sb.toString();
        System.out.println(message);
    }
}
```

> **Output:**

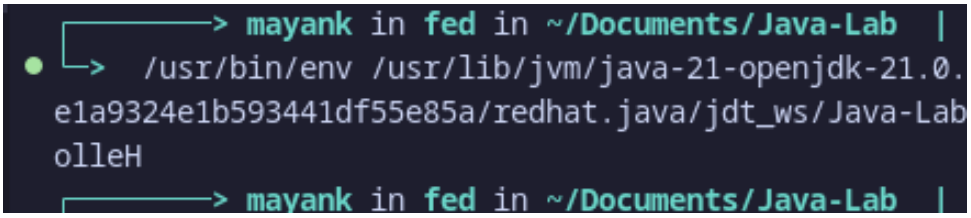

Figure 42: Reverse Method