

# 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

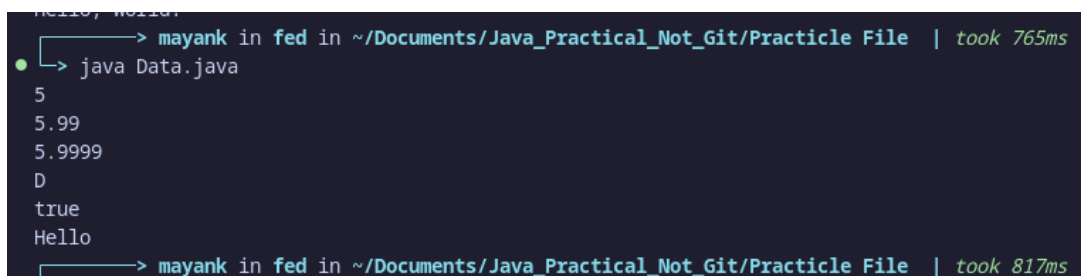
Java has two categories of data types: **Primitive Data Types** and **Reference/Object Data Types**.

#### # Primitive Data Types

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

> Code:

```
1 public class Data {
2     public static void main(String[] args) {
3         int myNum = 5;
4         float myFloatNum = 5.99f;
5         double myDoubNum = 5.9999d;
6         char myLetter = 'D';
7         boolean myBool = true;
8         String myText = "Hello";
9
10        System.out.println(myNum);
11        System.out.println(myFloatNum);
12        System.out.println(myDoubNum);
13        System.out.println(myLetter);
14        System.out.println(myBool);
15        System.out.println(myText);
16    }
17 }
```



```
hello, hello!
> 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:

```

1 public class NonPrimitive {
2     public static void main(String[] args) {
3         // String Data Type
4         String stringVar = "Hello, Java!";
5         System.out.println("\nString Data Type:");
6         System.out.println("String: " + stringVar);
7
8         // Array Data Type
9         int[] intArray = {1, 2, 3, 4, 5};
10        System.out.println("\nArray Data Type:");
11        System.out.print("intArray: ");
12        for (int num : intArray) {
13            System.out.print(num + " ");
14        }
15        System.out.println();
16    }
17 }

```

```

→ 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-Primitive 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

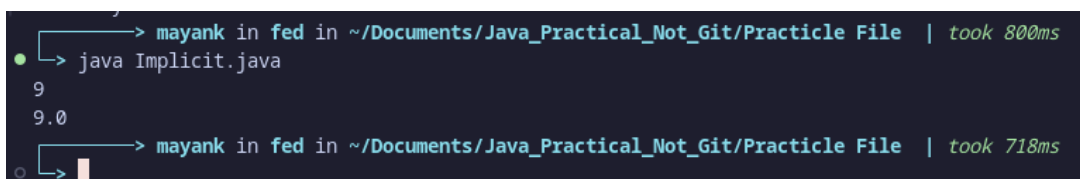
Implicit type casting is done automatically when passing a smaller size type to a larger size type.

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

> Code:

```
1 public class Implicit {
2     public static void main(String[] args) {
3         int myInt = 9;
4         double myDouble = myInt; // Automatic casting: int to double
5         System.out.println(myInt); // Outputs 9
6         System.out.println(myDouble); // Outputs 9.0
7     }
8 }
```

> Output:



```
> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 800ms
> java Implicit.java
9
9.0
> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 718ms
```

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.

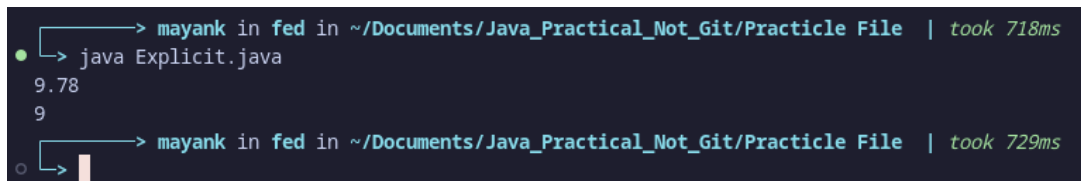
double → float → long → int → char → short → byte

```

1 public class Explicit {
2     public static void main(String[] args) {
3         double myDouble = 9.78d;
4         int myInt = (int) myDouble; // Manual casting: double to int
5
6         System.out.println(myDouble);    // Outputs 9.78
7         System.out.println(myInt);       // Outputs 9
8     }
9 }

```

> **Output:**



```

> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 718ms
● > java Explicit.java
9.78
9
○ > mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 729ms

```

Figure 4: Explicit Type Conversion

# Practical 3

## Array 1D and 2D

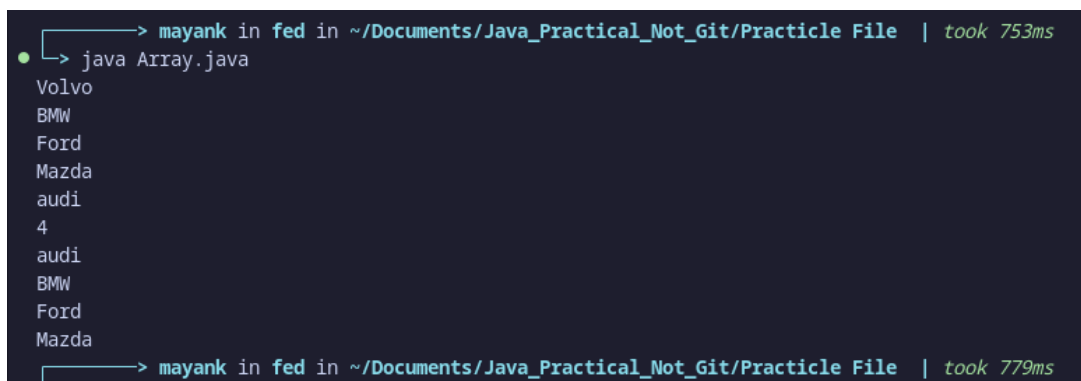
### 1 1-Dimensional Array

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

> Code:

```
1 public class Array {
2     public static void main(String[] args) {
3         String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
4         System.out.println(cars[0]);
5         System.out.println(cars[1]);
6         System.out.println(cars[2]);
7         System.out.println(cars[3]);
8         // Changing an element of an array
9         cars[0] = "audi";
10        System.out.println(cars[0]);
11        // Length of an array
12        System.out.println(cars.length);
13        // Loop through an array
14        for (String arr : cars) {
15            System.out.println(arr);
16        }
17    }
18 }
```

> 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



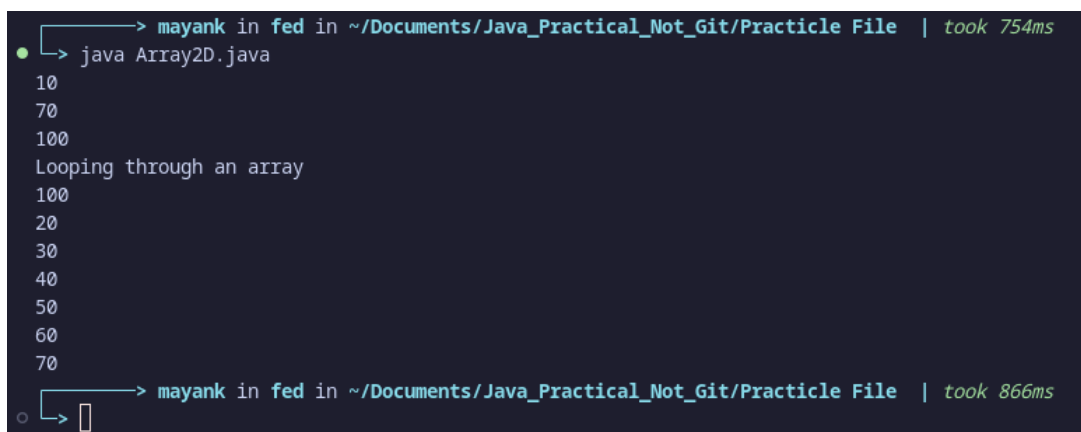
## 2 Multi Dimensional Array

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:

```
1      // Program for Multi-Dimensional Array
2
3  public class Array2D {
4      public static void main(String[] args) {
5          int[][] my2DArr = {{10, 20, 30, 40}, {50, 60, 70}};
6          // Accessing Elements of array
7          System.out.println(my2DArr[0][0]); // 10
8          System.out.println(my2DArr[1][2]); // 70
9          // change element of array
10         my2DArr[0][0] = 100;
11         System.out.println(my2DArr[0][0]); //100
12
13         // Loop through a multi dimensional array
14         System.out.println("Looping through an array");
15         for (int[] row : my2DArr) {
16             for(int i : row) {
17                 System.out.println(i);
18             }
19         }
20     }
21 }
```

> 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 Structures

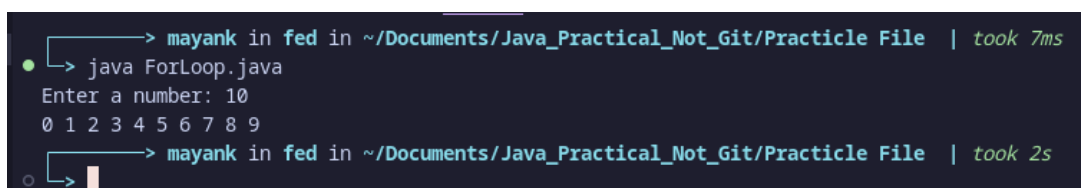
### 1 For loop

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:

```
1 import java.util.Scanner;
2
3 public class ForLoop {
4     public static void main(String[] args) {
5         Scanner sc = new Scanner(System.in);
6         System.out.print("Enter a number: ");
7         int n = sc.nextInt();
8         for (int i = 0; i < n; i++) {
9             System.out.print(i + " ");
10        }
11        System.out.println();
12    }
13 }
```

> 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:

```

1 import java.util.Scanner;
2
3 public class WhileLoop {
4
5     public static void main(String[] args) {
6         // Sum of first n numbers
7         Scanner sc = new Scanner(System.in);
8         System.out.print("Enter a number: ");
9         int n = sc.nextInt();
10        int sum = 0;
11        while (n > 0) {
12            sum += n--;
13        }
14        System.out.println(sum);
15    }
16 }

```

> Output:

```

> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 2s
● → java WhileLoop.java
Enter a number: 10
55
○ → mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 2s

```

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.

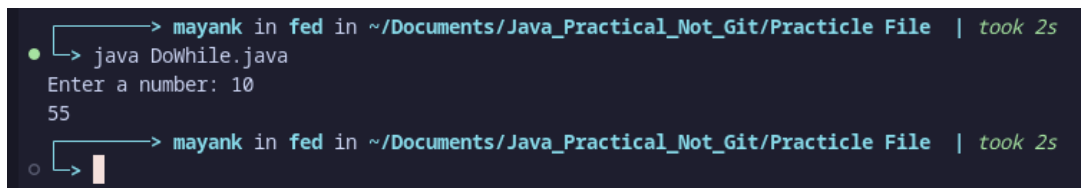
> Code:

```

1 import java.util.Scanner;
2
3 public class DoWhile {
4     public static void main(String[] args) {
5         // Sum of first n numbers
6         Scanner sc = new Scanner(System.in);
7         System.out.print("Enter a number: ");
8         int n = sc.nextInt();
9         int sum = 0;
10        do {
11            sum += n--;
12        } while (n > 0);
13        System.out.println(sum);
14    }
15 }

```

> Output:



```
> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 2s
● > java Dowhile.java
  Enter a number: 10
  55
○ >
```

The image shows a terminal window with a dark background. The prompt is a green dot. The first line shows the command 'java Dowhile.java' being executed. The second line shows the prompt 'Enter a number:' followed by the input '10'. The third line shows the output '55'. The prompt changes to a green circle for the next line, which is empty.

Figure 9: output of do-while loop

# Practical 5

## Various Decision Structures

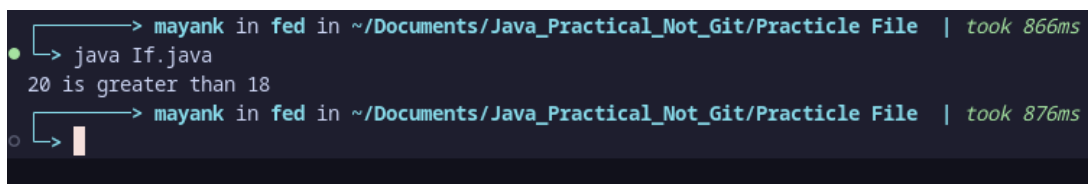
### 1 The IF statement

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

> Code:

```
1 public class If {
2     public static void main(String[] args) {
3         if (20 > 18) {
4             System.out.println("20 is greater than 18");
5         }
6     }
7 }
```

> Output:



```
> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 866ms
• -> java If.java
20 is greater than 18
○ -> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 876ms
```

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:

```
1 public class IfElse {
2     public static void main(String[] args) {
3         int time = 20;
4         if (time < 18) {
5             System.out.println("Good day.");
6         } else {
7             System.out.println("Good evening.");
8         }
9     }
10 }
```

> 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 conditions 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

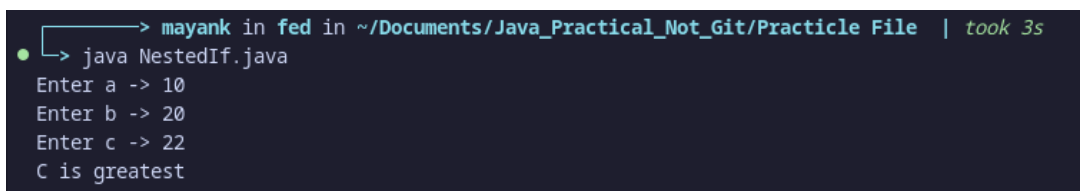
### 4 Nested If-Else

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

> Code:

```
1 import java.util.Scanner;
2
3 public class NestedIf {
4     public static void main(String[] args) {
5         Scanner sc = new Scanner(System.in);
6         System.out.print("Enter a -> ");
7         int a = sc.nextInt();
8         System.out.print("Enter b -> ");
9         int b = sc.nextInt();
10        System.out.print("Enter c -> ");
11        int c = sc.nextInt();
12        if (a > b) {
13            if (a > c) {
14                System.out.println("A is greatest");
15            } else {
16                System.out.println("C is greatest");
17            }
18        } else {
19            if (b > c) {
20                System.out.println("B is greatest");
21            } else {
22                System.out.println("C is greatest");
23            }
24        }
25    }
26 }
```

> Output:



```
> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 3s
• java NestedIf.java
Enter a -> 10
Enter b -> 20
Enter c -> 22
C is greatest
```

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:

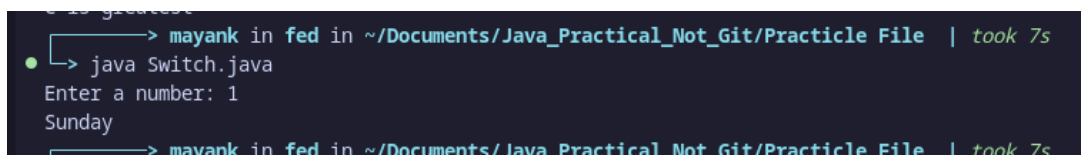
```
1 import java.util.Scanner;
2
```

```

3  public class Switch {
4      public static void main(String[] args) {
5          Scanner sc = new Scanner(System.in);
6          System.out.print("Enter a number: ");
7          int day = sc.nextInt();
8          switch (day) {
9              case 1:
10                 System.out.println("Sunday");
11                 break;
12             case 2:
13                 System.out.println("Monday");
14                 break;
15             case 3:
16                 System.out.println("Tuesday");
17                 break;
18             case 4:
19                 System.out.println("Wednesday");
20                 break;
21             case 5:
22                 System.out.println("Thursday");
23                 break;
24             case 6:
25                 System.out.println("Friday");
26                 break;
27             case 7:
28                 System.out.println("Saturday");
29                 break;
30             default:
31                 System.out.println("Invalid day");
32         }
33     }
34 }

```

> Output:



```

> mayank in fed in ~/Documents/Java_Practical_Not_Git/Practicle File | took 7s
• → java Switch.java
Enter a number: 1
Sunday
→ mayank in fed in ~/Documents/Java Practical Not Git/Practicle File | took 7s

```

Figure 14: output of switch statement



# Practical 6

## Recursion

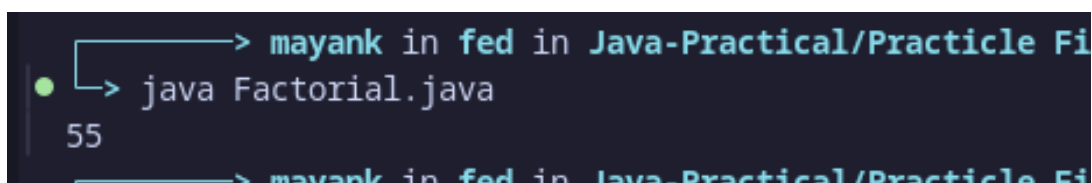
Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

### 1 Example of Recursion

# Code:

```
1 public class Recursion {
2     public static void main(String[] args) {
3         int result = sum(10);
4         System.out.println(result);
5     }
6     public static int sum(int k) {
7         if (k > 0) {
8             return k + sum(k - 1);
9         } else {
10            return 0;
11        }
12    }
13 }
```

> Output:



```
> mayank in fed in Java-Practical/Practicle Fi
> java Factorial.java
55
> mayank in fed in Java-Practical/Practicle Fi
```

Figure 15: Recursion Example

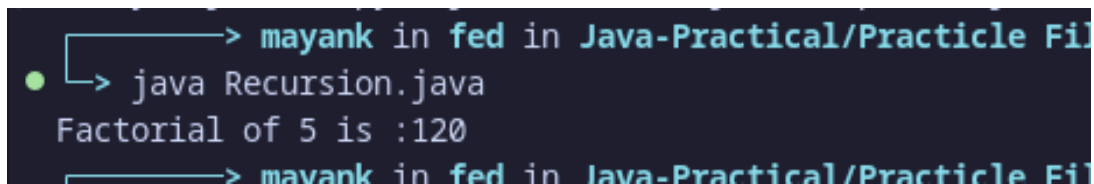
### 2 Factorial of a number using Recursion

# Code:

```
1 public class Factorial {
2     static int factorial(int n) {
3         if (n == 0 || n == 1)
4             return 1;
5         return n * factorial(n - 1);
6     }
7 }
```

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

> Output:

A terminal window with a dark background and light blue text. The prompt is '> mayank in fed in Java-Practical/Practicle Fi'. The user enters 'java Recursion.java'. The output is 'Factorial of 5 is :120'. The prompt is partially visible at the bottom.

```
> mayank in fed in Java-Practical/Practicle Fi  
● > java Recursion.java  
Factorial of 5 is :120  
> mayank in fed in Java-Practical/Practicle Fi
```

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.

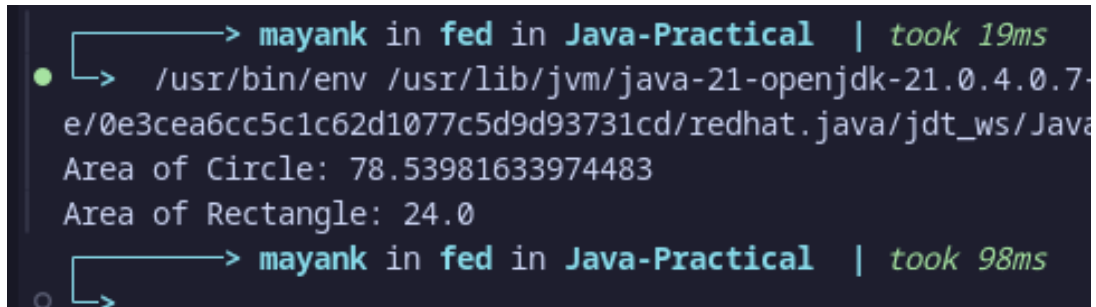
### 1 Example of Method Overloading by passing objects as arguments

# Code:

```
1  class Circle {
2      double radius;
3      Circle(double radius) {
4          this.radius = radius;
5      }
6  }
7
8  class Rectangle {
9      double length, width;
10     Rectangle(double length, double width) {
11         this.length = length;
12         this.width = width;
13     }
14 }
15
16 class AreaCalculator {
17     double calculateArea(Circle circle) {
18         return Math.PI * circle.radius * circle.radius;
19     }
20     double calculateArea(Rectangle rectangle) {
21         return rectangle.length * rectangle.width;
22     }
23 }
24
25 public class MOPOAA {
26     public static void main(String[] args) {
27         Circle circle = new Circle(5);
28         Rectangle rectangle = new Rectangle(4, 6);
29         AreaCalculator calculator = new AreaCalculator();
30         System.out.println
31         ("Area of Circle: "+calculator.calculateArea(circle));
32         System.out.println
```

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

> **Output:**



A terminal window with a dark background and light-colored text. The prompt is a green dot. The first command is a shell command to run a Java program. The output shows the area of a circle and a rectangle. The second command is a shell command to run the same Java program again, showing the same output.

```
> mayank in fed in Java-Practical | took 19ms  
● > /usr/bin/env /usr/lib/jvm/java-21-openjdk-21.0.4.0.7-  
e/0e3cea6cc5c1c62d1077c5d9d93731cd/redhat.java/jdt_ws/Java  
Area of Circle: 78.53981633974483  
Area of Rectangle: 24.0  
○ > mayank in fed in Java-Practical | took 98ms  
○ >
```

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:

```
1  class Book {
2      String title;
3      String author;
4      int pages;
5      // Constructor 1: No arguments
6      Book() {
7          this.title = "Unknown";
8          this.author = "Unknown";
9          this.pages = 0;
10     }
11     // Constructor 2: Passing title, author, and pages
12     Book(String title, String author, int pages) {
13         this.title = title;
14         this.author = author;
15         this.pages = pages;
16     }
17     /*
18     Constructor 3: Passing an existing Book object
19     (copy constructor)
20     */
21     Book(Book existingBook) {
22         this.title = existingBook.title;
23         this.author = existingBook.author;
24         this.pages = existingBook.pages;
25     }
26     void displayDetails() {
27         System.out.println("Title: " + title);
28         System.out.println("Author: " + author);
29         System.out.println("Pages: " + pages);
30     }
31 }
32
```

```

33 public class COBPOAA {
34     public static void main(String[] args) {
35         // Using Constructor 1
36         Book book1 = new Book();
37         System.out.println("Book 1 details:");
38         book1.displayDetails();
39
40         // Using Constructor 2
41         Book book2 = new Book("1984", "George Orwell", 328);
42         System.out.println("\nBook 2 details:");
43         book2.displayDetails();
44
45         // Using Constructor 3 (Copy Constructor)
46         Book book3 = new Book(book2);
47         System.out.println
48             ("\nBook 3 details (copied from Book 2):");
49         book3.displayDetails();
50     }
51 }

```

> Output:

```

> mayank in fed in Java-Practical | took 21ms
➤ /usr/bin/env /usr/lib/jvm/java-21-openjdk-21.0.4.0.7
e/0e3cea6cc5c1c62d1077c5d9d93731cd/redhat.java/jdt_ws/Jav
Book 1 details:
Title: Unknown
Author: Unknown
Pages: 0

Book 2 details:
Title: 1984
Author: George Orwell
Pages: 328

Book 3 details (copied from Book 2):
Title: 1984
Author: George Orwell
Pages: 328
➤ mayank in fed in Java-Practical | took 122ms

```

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:

```
1 class Example {  
2     public int publicVar = 10;  
3     protected int protectedVar = 20;  
4     int defaultVar = 30;  
5     private int privateVar = 40;  
6 }
```

## 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:

```

1 class StaticExample {
2     static int count = 0;
3
4     // Static method
5     static void increment() {
6         count++;
7     }
8 }

```

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:

```

1 final class FinalClass {
2     final int MAX_VALUE = 100;
3
4     final void display() {
5         System.out.println("This is a final method.");
6     }
7 }
8
9 // Uncommenting the below class will result in a compile-time error
10 // 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:



```

1 class FinalizeExample {
2     @Override
3     protected void finalize() throws Throwable {
4         System.out.println("Finalize method called.");
5     }
6
7     public static void main(String[] args) {
8         FinalizeExample obj = new FinalizeExample();
9         obj = null; // Make the object eligible for garbage collection
10        System.gc(); // Request garbage collection
11    }
12 }

```

## 5 Example

```

1 class AccessDemo {
2     public int publicVar = 10;
3     private int privateVar = 20;
4
5     static int staticVar = 30;
6
7     final int finalVar = 40;
8
9     public void show() {
10        System.out.println("Public Variable: " + publicVar);
11        System.out.println("Private Variable: " + privateVar);
12        System.out.println("Static Variable: " + staticVar);
13        System.out.println("Final Variable: " + finalVar);
14    }
15
16    @Override
17    protected void finalize() throws Throwable {
18        System.out.println("Finalize method called.");
19    }
20
21    public static void main(String[] args) {
22        AccessDemo demo = new AccessDemo();
23        demo.show();
24
25        // Static variable can be accessed directly without creating an object
26        System.out.println("Accessing static variable without object: " + Access
27
28        // Making object eligible for garbage collection
29        demo = null;
30        System.gc(); // Request garbage collection
31    }
32 }
33 %

```

> Output:

```
Public Variable: 10  
Private Variable: 20  
Static Variable: 30  
Final Variable: 40  
Accessing static variable without object: 30  
Finalize method called.  
_____> mayank in fed in Java-Practical | took 103ms
```

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:

```
1 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 Arguments

# Code:

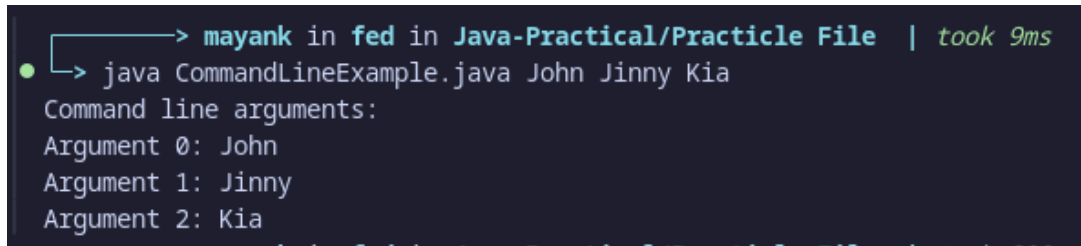
```
1 public class CommandLineExample {
2     public static void main(String[] args) {
3         // Check if any arguments were passed
4         if (args.length > 0) {
5             System.out.println("Command line arguments:");
6
7             // Iterate over the arguments and print each one
8             for (int i = 0; i < args.length; i++) {
9                 System.out.println
10                    ("Argument " + i + ": " + args[i]);
11            }
12        } else {
13            System.out.println
14                ("No command line arguments were passed.");
15        }
16    }
17 }
```

### 3 How to Run the Program with Command Line Arguments

# Run the program with following command:

```
1 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:

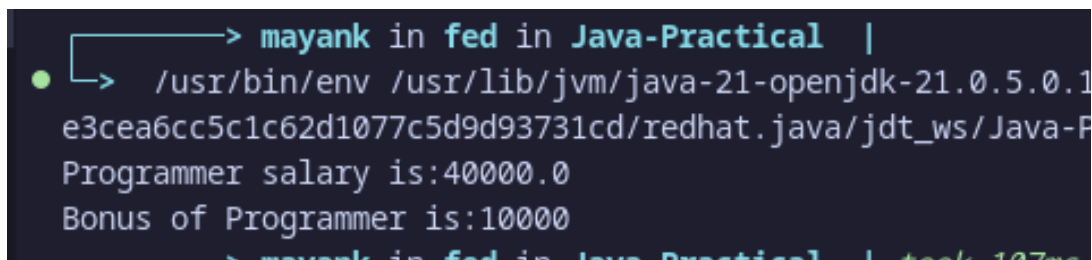
### 1 Single Inheritance:

A class inherits from a single superclass.

#### # Code:

```
1  class Employee {
2      float salary = 40000;
3  }
4
5  class Programmer extends Employee {
6      int bonus = 10000;
7      public static void main(String args[]) {
8          Programmer p = new Programmer();
9          System.out.println("Programmer salary is:" + p.salary);
10         System.out.println("Bonus of Programmer is:" + p.bonus);
11     }
12 }
```

#### > 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

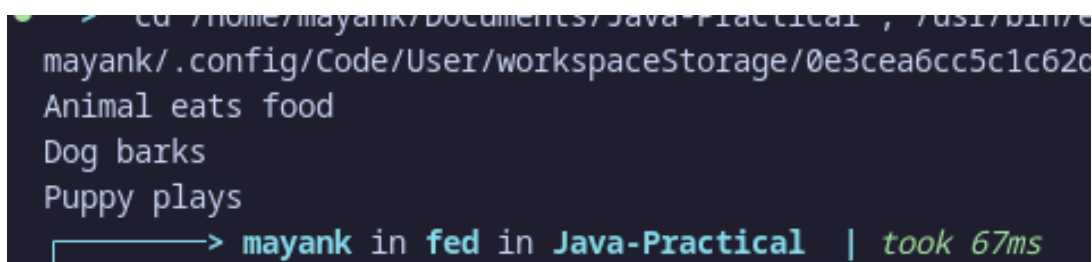
## 2 Multilevel Inheritance:

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

### # Code:

```
1 class animal {
2     public static void eat() {
3         System.out.println("Animal eats food");
4     }
5 }
6
7 class dog extends animal {
8     public static void bark() {
9         System.out.println("Dog barks");
10    }
11 }
12
13 class puppy extends dog {
14
15     public static void play() {
16         System.out.println("Puppy plays");
17     }
18
19     public static void main(String[] args) {
20         puppy obj = new puppy();
21         obj.eat();
22         obj.bark();
23         obj.play();
24     }
25 }
```

### > Output:



```
cd /home/mayank/Documents/Java-Practical ; /usr/bin/c
mayank/.config/Code/User/workspaceStorage/0e3cea6cc5c1c62c
Animal eats food
Dog barks
Puppy plays
> mayank in fed in Java-Practical | took 67ms
```

Figure 22: output of Multilevel Inheritance

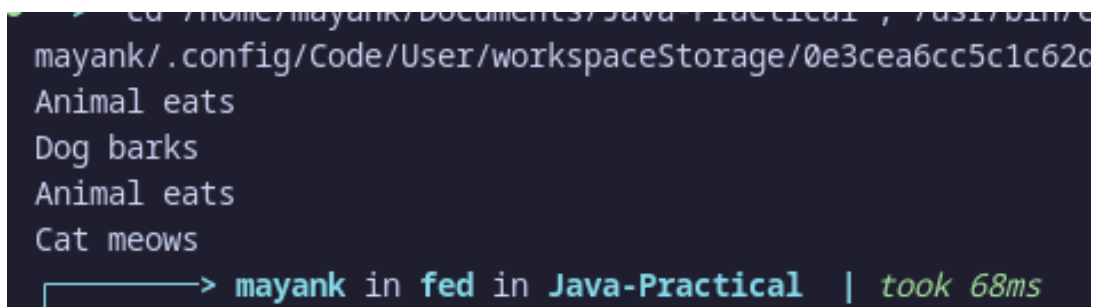
## 3 Hierarchical Inheritance:

Multiple classes inherit from a single superclass.

## # Code:

```
1 class Animal {
2     public void eat() {
3         System.out.println("Animal eats");
4     }
5 }
6
7 class Dog extends Animal {
8     public void bark() {
9         System.out.println("Dog barks");
10    }
11 }
12
13 class Cat extends Animal {
14     public void meow() {
15         System.out.println("Cat meows");
16     }
17 }
18
19 public class HInherit {
20     public static void main(String[] args) {
21         // Create Dog object and call methods
22         Dog dog = new Dog();
23         dog.eat(); // Inherited method from Animal
24         dog.bark(); // Specific method of Dog
25         // Create Cat object and call methods
26         Cat cat = new Cat();
27         cat.eat(); // Inherited method from Animal
28         cat.meow(); // Specific method of Cat
29     }
30 }
```

## > 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:

```
1  class Car {
2      public String bestModel() {
3          return "";
4      }
5  }
6
7  class BMW extends Car {
8      @Override
9      public String bestModel() {
10         return "M4 Competition";
11     }
12 }
13
14 class Audi extends Car {
15     @Override
16     public String bestModel() {
17         return "RS7";
18     }
19 }
20
21 class Porsche extends Car {
22     @Override
23     public String bestModel() {
24         return "911 gt3rs";
25     }
26 }
27
28 public class MetOvr {
29     public static void main(String[] args) {
30         BMW bmw = new BMW();
31         Audi audi = new Audi();
32         Porsche porsche = new Porsche();
33
34         System.out.println(bmw.bestModel());
```



```

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

```

> **Output:**

```

➤ cd /home/mayank/Documents/Java-Practical ; /usr/bin/
orkspaceStorage/0e3cea6cc5c1c62d1077c5d9d93731cd/redhat.
M4 Competition
RS7
911 gt3rs
➤ mayank in fed in Java-Practical | took 25ms

```

Figure 24: ouput of Method overriding

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

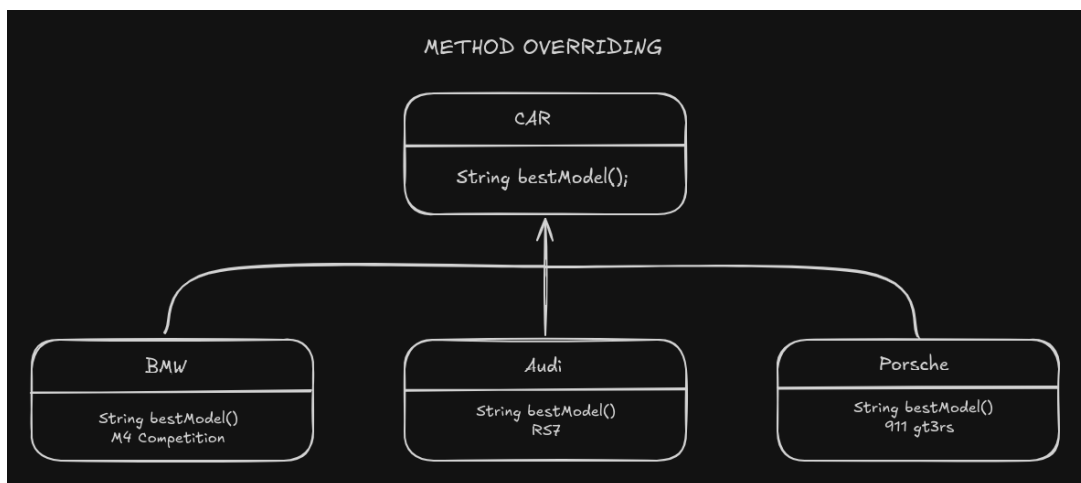


Figure 25: Example of Method overriding

# 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:

```
1  abstract class ClassName {
2      // Abstract method (no body)
3      public abstract void methodName();
4
5      // Concrete method (with body)
6      public void anotherMethod() {
7          System.out.println("This is a concrete method.");
8      }
9  }
```

### # Code:

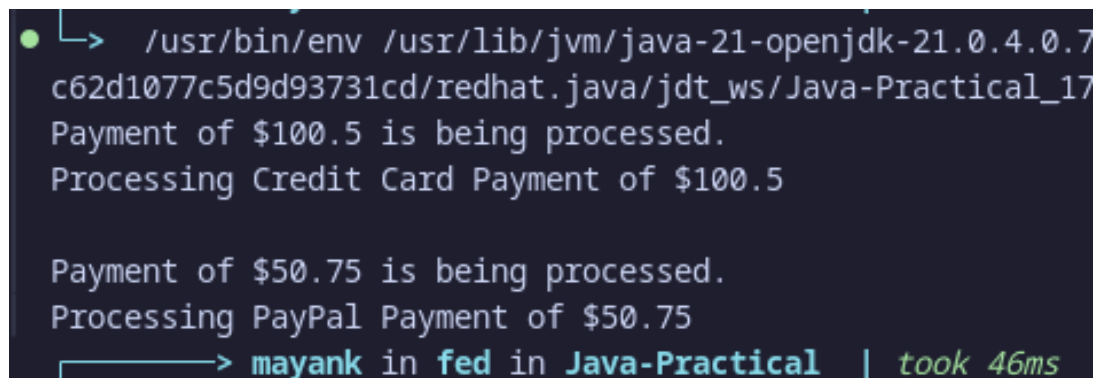
```
1  abstract class Payment {
2      public abstract void makePayment(double amount);
3
4      public void paymentDetails(double amount) {
5          System.out.println("Payment of $" + amount + " is being processed.");
6      }
7  }
8
9  class CreditCardPayment extends Payment {
10     @Override
11     public void makePayment(double amount) {
12         System.out.println("Processing Credit Card Payment of $" + amount);
13     }
14 }
15
16 class PayPalPayment extends Payment {
17     @Override
18     public void makePayment(double amount) {
19         System.out.println("Processing PayPal Payment of $" + amount);
20     }
21 }
22
23 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:**



```

➤ /usr/bin/env /usr/lib/jvm/java-21-openjdk-21.0.4.0.7
c62d1077c5d9d93731cd/redhat.java/jdt_ws/Java-Practical_17
Payment of $100.5 is being processed.
Processing Credit Card Payment of $100.5

Payment of $50.75 is being processed.
Processing PayPal Payment of $50.75
➤ mayank in fed in Java-Practical | took 46ms

```

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:

```
1  class outerClass {
2      static int outer_x = 34; // Static member
3      int outer_y = 102;
4      // Non-static member
5      private static int outerPrivate = 44; // Static member
6      static class innerClass {
7          // Static nested class
8          void display() {
9              outerClass outer = new outerClass();
10             System.out.println("Value of x: " + outer_x);
11             System.out.println("Value of y: " + outer.outer_y);
12             System.out.println("Value of private variable: " + outerPrivate);
13         }
14     }
15 }
16
17 public class Nest { // Separate class to run the main method
18     public static void main(String[] args) {
19         System.out.println("Static Nested Class");
20         outerClass.innerClass obj = new outerClass.innerClass();
21         obj.display(); // Call the display method
22     }
23 }
```

> Output:

```
cd /home/mayank/Documents/Java-Practical ; java -cp bin/c
mayank/.config/Code/User/workspaceStorage/0e3cea6cc5c1c62c
Static Nested Class
Value of x: 34
Value of y: 102
Value of private variable: 44
> mayank in fed in Java-Practical | took 132ms
```

Figure 27: ouput of nested class

# Practical 15

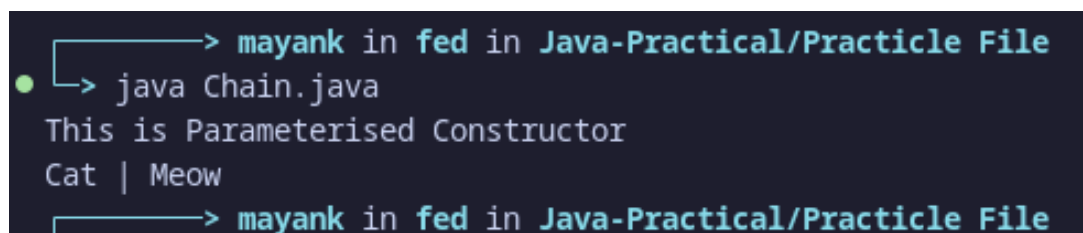
## Constructor chaining

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:

```
1  class Animal {
2      String name;
3      String speak;
4
5      Animal() {
6          this("Cat", "Meow");
7          System.out.println("This is Default Constructor");
8      }
9
10     Animal(String name, String speak) {
11         this.name = name;
12         this.speak = speak;
13         System.out.println("This is Parameterised Constructor");
14         System.out.println(name + " | " + speak);
15     }
16 }
17
18 public class Chain {
19     public static void main(String[] args) {
20         Animal obj1 = new Animal();
21     }
22 }
```

### > 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);
```

```

12         System.out.println("Default variable: " + defaultVar);
13         System.out.println("Private variable: " + privateVar);
14     }
15 }

```

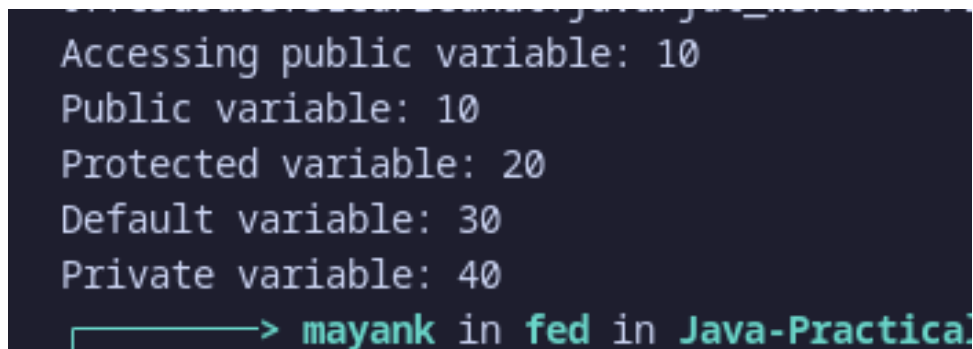
## # Code for Importing the Package (ImportPackage.java):

```

1  import mypackage.MyClass;
2
3  public class ImportPackage {
4      public static void main(String[] args) {
5          MyClass obj = new MyClass();
6
7          // Accessing variables with different access modifiers
8          System.out.println("Accessing public variable: " + obj.publicVar);
9          // System.out.println("Accessing protected variable: " + obj.protectedVar);
10         // System.out.println("Accessing default variable: " + obj.defaultVar);
11         // System.out.println("Accessing private variable: " + obj.privateVar);
12
13         obj.display(); // Method displays all variables within the same class
14     }
15 }

```

## # Output:



```

Accessing public variable: 10
Public variable: 10
Protected variable: 20
Default variable: 30
Private variable: 40
> mayank in fed in Java-Practical

```

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



# Practical 17

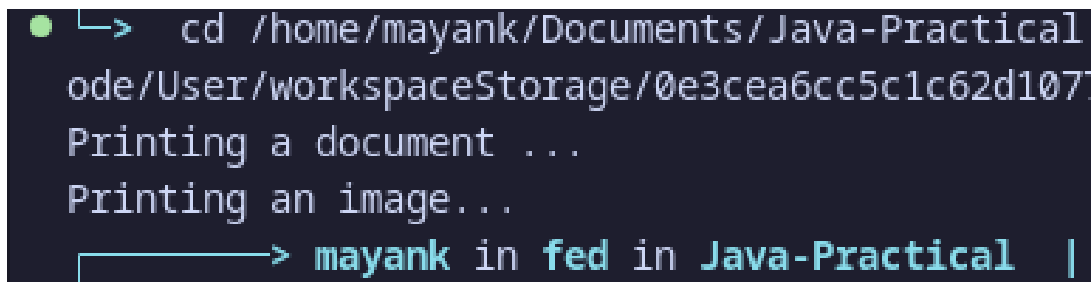
## Interfaces, nested interfaces and use of extending interfaces

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:

```
1  interface Printable {
2      // Defining an interface
3      void print(); // Abstract method
4  }
5
6  class Document implements Printable {
7
8      @Override
9      public void print() {
10         System.out.println("Printing a document ...");
11     }
12 }
13
14 class Image implements Printable {
15
16     @Override
17     public void print() {
18         System.out.println("Printing an image...");
19     }
20 }
21
22 public class InterfaceExample {
23
24     public static void main(String[] args) {
25         // Creating objects of classes that implement the Printable interface
26         Printable doc = new Document();
27         Printable img = new Image();
28         // Calling the print method using interface references
29         doc.print();
30         img.print();
31     }
32 }
```

> Output:

A terminal window with a dark background. The prompt is a green circle. The command is `cd /home/mayank/Documents/Java-Practical`. The output shows the directory path `ode/User/workspaceStorage/0e3cea6cc5c1c62d107`, followed by `Printing a document ...` and `Printing an image...`. The prompt is followed by `mayank in fed in Java-Practical |`.

```
➤ cd /home/mayank/Documents/Java-Practical
ode/User/workspaceStorage/0e3cea6cc5c1c62d107
Printing a document ...
Printing an image...
➤ mayank in fed in Java-Practical |
```

Figure 30: ouput of interface

## 1 Nested Interfaces

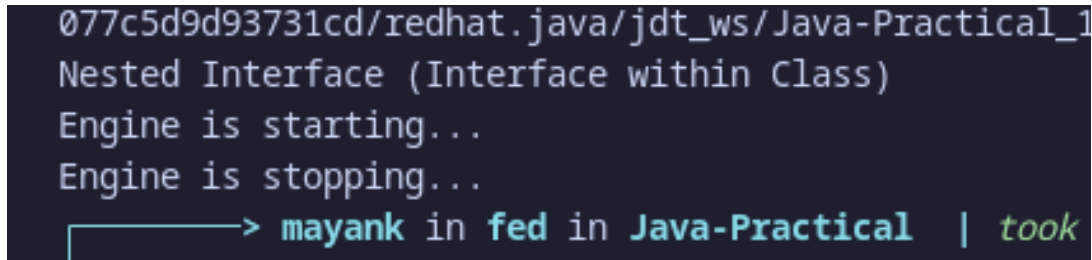
A nested interface in Java is an interface that is declared within another interface or class. This allows better organization of code, especially when the interface is only relevant in the context of the enclosing class or interface.

```
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

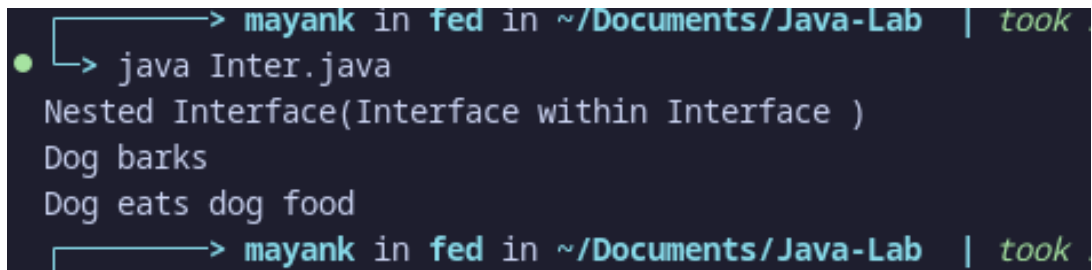
## 2 Extended Interfaces

```

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  }
```



A terminal window with a dark background and light-colored text. The prompt is `> mayank in fed in ~/Documents/Java-Lab | took`. The user enters `java Inter.java`. The output is `Nested Interface(Interface within Interface )`, `Dog barks`, and `Dog eats dog food`. The prompt is repeated at the bottom.

```
> 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.

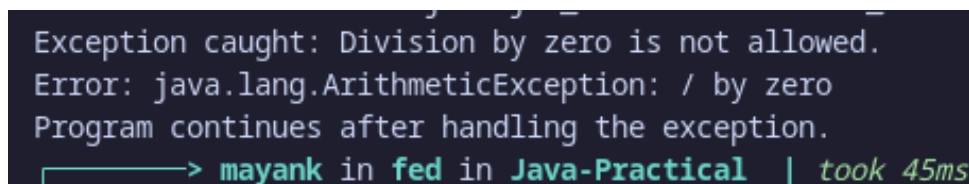
### 4 Example Code for Exception Handling Using Pre-defined Exception

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

#### # Code:

```
1 public class ExceptionHandling {
2     public static void main(String[] args) {
3         int a = 10;
4         int b = 0;
5
6         try {
7             int result = a / b; // This will cause an ArithmeticException
8             System.out.println("Result: " + result);
9         } catch (ArithmeticException e) {
10            System.out.println("Exception caught: Division by zero is not allowed");
11            System.out.println("Error: " + e);
12        }
13
14        System.out.println("Program continues after handling the exception.");
15    }
16 }
```

#### # 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:

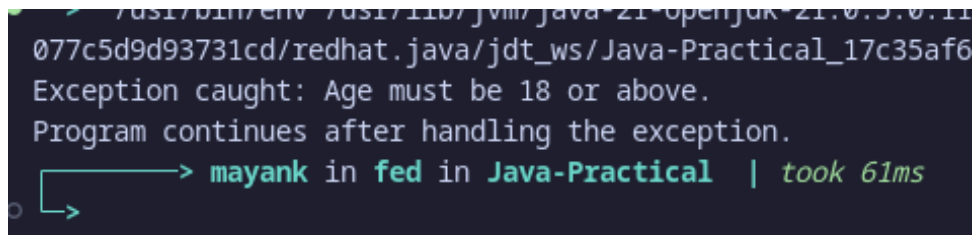
```
1 public class UserDefinedException {
2     public static void main(String[] args) {
3         try {
4             checkAge(15); // This will throw an InvalidAgeException
5         } catch (InvalidAgeException e) {
6             System.out.println("Exception caught: " + e.getMessage());
7         }
8     }
9 }
```

```

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:



```

077c5d9d93731cd/redhat.java/jdt_ws/Java-Practical_17c35af6
Exception caught: Age must be 18 or above.
Program continues after handling the exception.
> mayank in fed in Java-Practical | took 61ms

```

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



## Practical 20

### String class and its methods

# Practical 21

## StringBuffer class and its methods

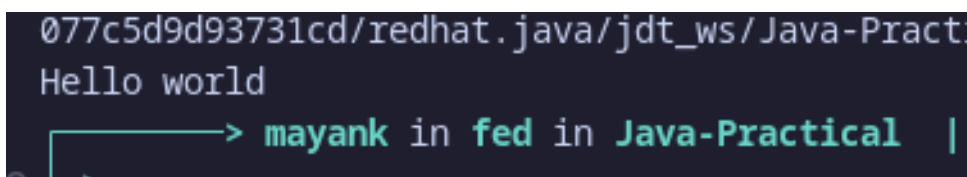
StringBuffer is a class in Java that represents a mutable sequence of characters. It provides an alternative to the immutable String class, allowing you to modify the contents of a string without creating a new object every time.

### 1 Append Method

# Code:

```
1 public class StringBufferAppend {
2     public static void main(String[] args)
3     {
4         // Append Method
5         StringBuffer sb = new StringBuffer();
6         sb.append("Hello");
7         sb.append(" ");
8         sb.append("world");
9         String message = sb.toString();
10        System.out.println(message);
11    }
12 }
```

> Output:



```
077c5d9d93731cd/redhat.java/jdt_ws/Java-Practical
Hello world
> mayank in fed in Java-Practical |
```

Figure 35: Output demonstrating a user-defined exception in Java

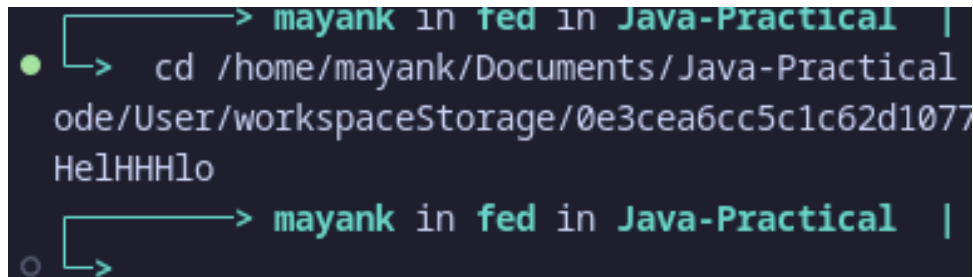
### 2 Insert Method

# Code:

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

```
7         System.out.println(message);  
8     }  
9 }
```

> **Output:**



A terminal window with a dark background and light green text. The prompt is '> mayank in fed in Java-Practical |'. The user enters 'cd /home/mayank/Documents/Java-Practical' followed by a newline. The prompt changes to 'ode/User/workspaceStorage/0e3cea6cc5c1c62d1077'. The user enters 'HelHHHlo' followed by a newline. The prompt changes to 'mayank in fed in Java-Practical |'. The user enters a character followed by a newline, indicated by a green circle at the end of the line.

Figure 36: Output demonstrating a user-defined exception in Java