

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

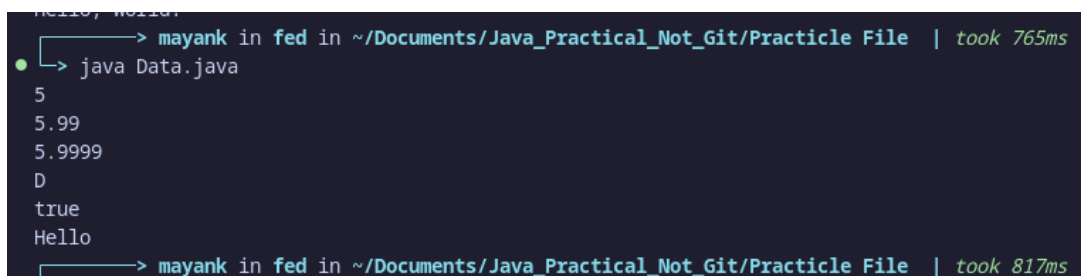
Java has two categories of data types: **Primitive Data Types** and **Reference/Object Data Types**.

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

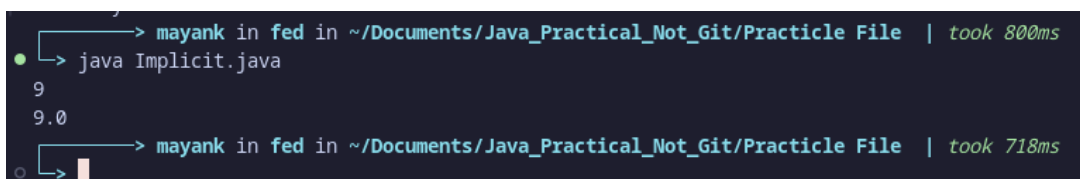
Implicit type casting is done automatically when passing a smaller size type to a larger size type.

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.

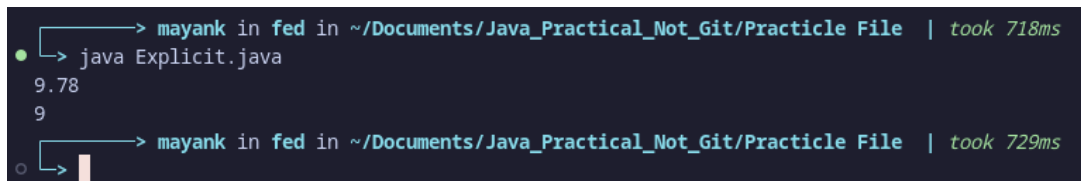
double → float → long → int → char → short → byte

```

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

Practical 3: Array 1D and 2D

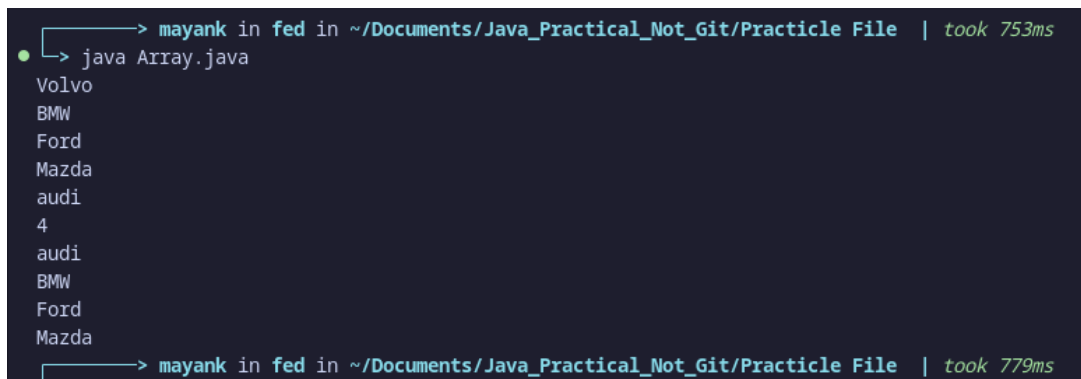
1 1-Dimensional Array

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

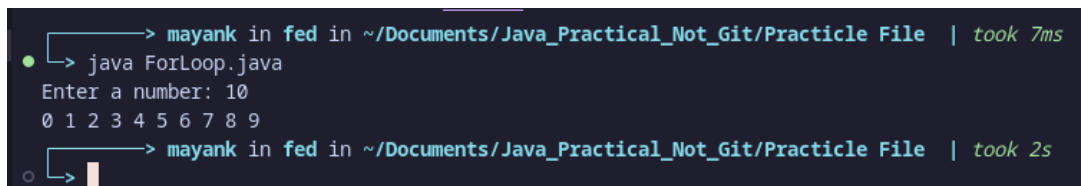
1 For loop

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
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7         System.out.print("Enter a number: ");
8         int n = sc.nextInt();
9         for (int i = 0; i < n; i++) {
10             System.out.print(i + " ");
11         }
12         System.out.println();
13     }
14 }
```

> 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
○ →
```

Figure 7: output of for loop

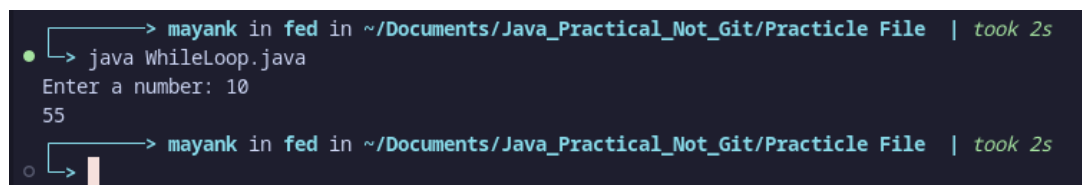
2 While loop

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
○ ->
```

Figure 8: output of while loop

3 Do-While loop

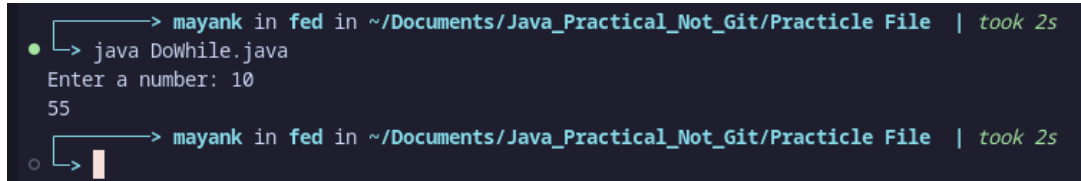
Do-While loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of Exit Control Loop.

> Code:

```
1 import java.util.Scanner;
2
3 public class DoWhile {
4
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        do {
12            sum += n--;
```

```
13         } while (n > 0);  
14         System.out.println(sum);  
15     }  
16 }
```

> **Output:**



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

Figure 9: output of do-while loop

Practical 5

Various Decision Structures

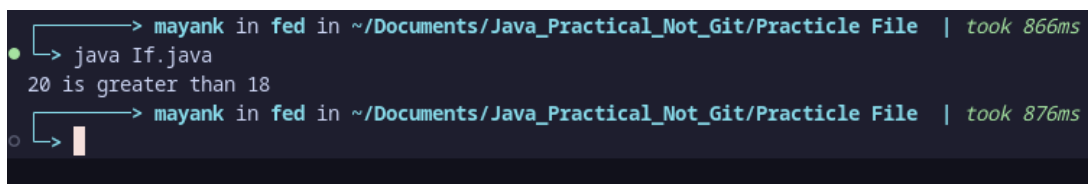
1 The IF statement

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

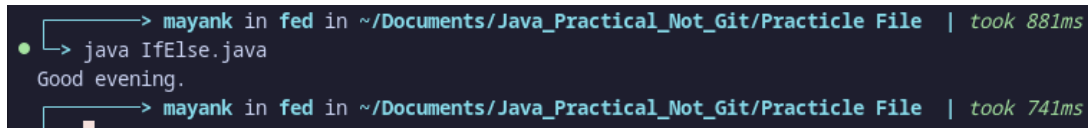
2 The IF-Else statement

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

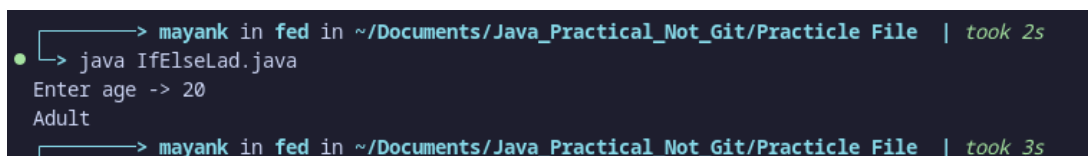
3 The IF-Else ladder

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
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7         System.out.print("Enter age -> ");
8         int age = sc.nextInt();
9         if (age < 12) {
10             System.out.println("Child");
11         } else if (age < 18) {
12             System.out.println("Teenager");
13         } else {
14             System.out.println("Adult");
15         }
16     }
17 }
```

> 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

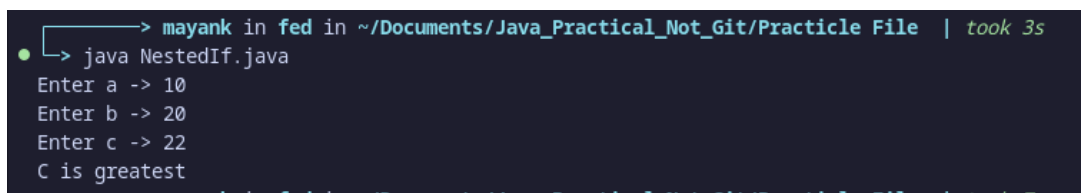
4 Nested If-Else

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
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7         System.out.print("Enter a -> ");
8         int a = sc.nextInt();
9         System.out.print("Enter b -> ");
10        int b = sc.nextInt();
11        System.out.print("Enter c -> ");
12        int c = sc.nextInt();
13        if (a > b) {
14            if (a > c) {
15                System.out.println("A is greatest");
16            } else {
17                System.out.println("C is greatest");
18            }
19        } else {
20            if (b > c) {
21                System.out.println("B is greatest");
22            } else {
23                System.out.println("C is greatest");
24            }
25        }
26    }
27 }
```

> 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.

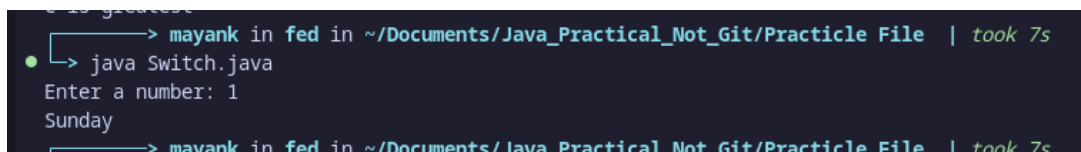
> Code:

```

1  import java.util.Scanner;
2
3  public class Switch {
4
5      public static void main(String[] args) {
6          Scanner sc = new Scanner(System.in);
7          System.out.print("Enter a number: ");
8          int day = sc.nextInt();
9          switch (day) {
10             case 1:
11                 System.out.println("Sunday");
12                 break;
13             case 2:
14                 System.out.println("Monday");
15                 break;
16             case 3:
17                 System.out.println("Tuesday");
18                 break;
19             case 4:
20                 System.out.println("Wednesday");
21                 break;
22             case 5:
23                 System.out.println("Thursday");
24                 break;
25             case 6:
26                 System.out.println("Friday");
27                 break;
28             case 7:
29                 System.out.println("Saturday");
30                 break;
31             default:
32                 System.out.println("Invalid day");
33         }
34     }
35 }

```

> 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

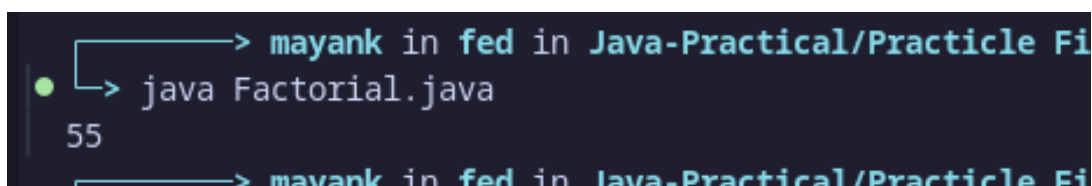
Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

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

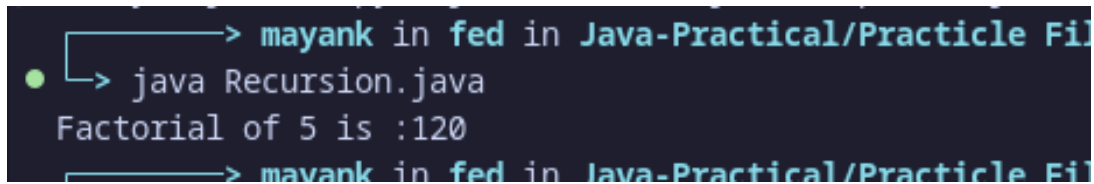
2 Factorial of a number using Recursion

Code:

```
1 public class Factorial {
2     static int factorial(int n)
3     {
4         if (n == 0 || n == 1)
5             return 1;
6         return n * factorial(n - 1);
7     }
8 }
```

```
7     }
8     public static void main(String[] args)
9     {
10         int ans = factorial(5);
11         System.out.println("Factorial of 5 is :" + ans);
12     }
13 }
```

> **Output:**

A terminal window with a dark background. The prompt is a green dot. The user enters a command to run a Java file. The output shows the factorial of 5 is 120.

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

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.

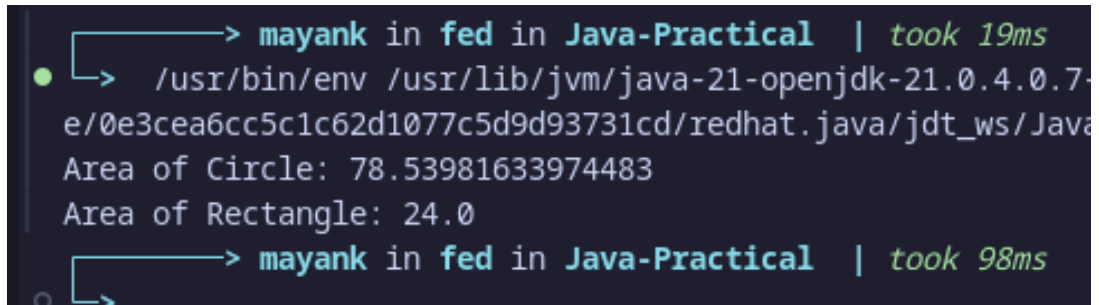
1 Example of Method Overloading by passing objects as arguments

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

```
33     }  
34 }
```

> **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 public class COBPOAA {
```

```

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

```

> 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

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:

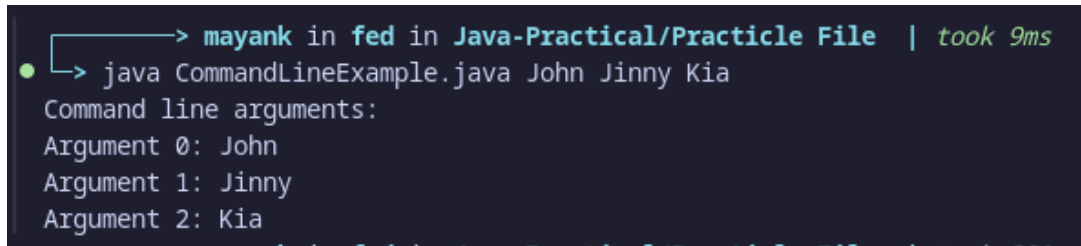
```
1 public static void main(String[] args) {  
2     // Check if any arguments were passed  
3     if (args.length > 0) {  
4         System.out.println("Command line arguments:");  
5  
6         // Iterate over the arguments and print each one  
7         for (int i = 0; i < args.length; i++) { 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

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 19: Command Line Arguments