

INSTITUTE :- TOPS TECHNOLOGIES (BARODA)

NAME :- SRUJAL SOLANKI M

COURSE :- JAVA FULL STACK DEVELOPER

1. Introduction to Java

• Theory:

o History of Java o Features of Java (Platform Independent, Object-Oriented, etc.)

Java, an object-oriented programming language developed by James Gosling at Sun Microsystems, emerged in 1991 as a platform-independent language called Oak before being renamed Java in 1995. Java's core features include platform independence, object orientation, and robust memory management.

History of Java:

* **1991:** James Gosling and his team at Sun Microsystems began developing the language, initially named "Oak," with the goal of creating a platform-independent language for consumer electronics.
* **1995:** The language was officially renamed "Java".
* **1996:** The first public release, JDK 1.0, was released.
* **Present:** Java has evolved through numerous versions and remains a widely used language for various applications.

Key Features of Java:

* **Platform Independence:** Java programs can run on any device with a Java Virtual Machine (JVM), making it highly portable.
* **Object-Oriented:** Java is a purely object-oriented language, supporting concepts like encapsulation, inheritance, and polymorphism.
* **Robust:** Java provides automatic memory management and exception handling, making it a robust and reliable language.
* **Security:** Java's security features, such as the sandboxing mechanism, protect against malicious code.
* **Multithreaded:** Java supports multithreading, allowing for parallel execution of code.
* **Dynamic:** Java's dynamic nature allows for runtime changes and flexibility.
* **Distributed:** Java facilitates distributed computing by integrating networking capabilities.

o Understanding JVM, JRE, and JDK o Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ)

| **Term** | **Full Form** | **What It Is** | **Role** |
| --- | --- | --- | --- |
| **JVM** | Java Virtual Machine | A virtual machine that runs Java bytecode | Converts .class files into machine-specific instructions |
| **JRE** | Java Runtime Environment | JVM + libraries + runtime components | Needed to run Java programs (but not develop them) |
| **JDK** | Java Development Kit | JRE + development tools (compiler, debugger) | Needed to **write**, **compile**, and **run** Java code |

**Option 1: Eclipse IDE**

* Download: https://www.eclipse.org/downloads/
* Choose "Eclipse IDE for Java Developers"
* Install and open Eclipse
* Create a new Java project: File > New > Java Project

o Java Program Structure (Packages, Classes, Methods)

• Lab Exercise:

o Install JDK and set up environment variables.

**Step 1: Download & Install JDK**

1. **Go to the official JDK download page**:  
   👉 <https://www.oracle.com/java/technologies/javase-downloads.html>
2. **Choose the latest Long-Term Support (LTS) version**, e.g., Java 17 or Java 21.
3. Download the installer for **Windows x64**.
4. **Run the installer** and follow the setup wizard.
   * Default install location:  
     C:\Program Files\Java\jdk-XX (where XX is the version number, like jdk-21)

o Write a simple "Hello World" Java program.

public class HelloWorld {

public static void main(String[] args) {

System.out.println(“hello World”);

}

o Compile and run the program using command-line tools (javac, java).

cd C:\JavaPrograms

javac HelloWorld.java

java HelloWorld

Hello, Java

1. Data Types, Variables, and Operators

• Theory:

o Primitive Data Types in Java (int, float, char, etc.)

| **Category** | **Data Type** | **Size** | **Example** |
| --- | --- | --- | --- |
| Integer | byte | 1 byte | byte a = 100; |
|  | short | 2 bytes | short b = 32000; |
|  | int | 4 bytes | int c = 100000; |
|  | long | 8 bytes | long d = 10000000000L; |
| Floating-point | float | 4 bytes | float e = 5.75f; |
|  | double | 8 bytes | double f = 19.99; |
| Character | char | 2 bytes (Unicode) | char g = 'A'; |
| Boolean | boolean | 1 bit (internally) | boolean h = true; |

o Variable Declaration and Initialization

Declaration :

int age;

float height;

char grade;

boolean isPassed;

Initialization:

age = 20;

height = 5.9f;

grade = 'A';

isPassed = true;

o Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise

**Arithmetic:**

| **Operator** | **Meaning** | **Example (int a = 10, b = 3)** | **Result** |
| --- | --- | --- | --- |
| + | Addition | a + b | 13 |
| - | Subtraction | a - b | 7 |
| \* | Multiplication | a \* b | 30 |
| / | Division | a / b | 3 |
| % | Modulus (remainder) | a % b | 1 |

**Relational operators :**

| **Operator** | **Meaning** | **Example (a = 10, b = 3)** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | a == b | false |
| != | Not equal to | a != b | true |
| > | Greater than | a > b | true |
| < | Less than | a < b | false |
| >= | Greater than or equal to | a >= b | true |
| <= | Less than or equal to | a <= b | false |
| | **Operator** | **Meaning** | **Example** | **Equivalent To** | | --- | --- | --- | --- | | = | Assign | x = 10 | - | | += | Add and assign | x += 5 | x = x + 5 | | -= | Subtract and assign | x -= 2 | x = x - 2 | | \*= | Multiply and assign | x \*= 3 | x = x \* 3 | | /= | Divide and assign | x /= 2 | x = x / 2 | | %= | Modulus and assign | x %= 3 | x = x % 3 | |  |  |  |

Unary operator :

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | Unary plus | +x | x |
| - | Unary minus | -x | Negative of x |
| ++ | Increment | ++x or x++ | Increase by 1 |
| -- | Decrement | --x or x-- | Decrease by 1 |
| ! | Logical NOT | !true | false |

| **Operator** | **Meaning** | **Example (a = 5, b = 3)** | **Result** |
| --- | --- | --- | --- |
| & | Bitwise AND | a & b (0101 & 0011) | 1 |
| ` | ` | Bitwise OR | `a |
| ^ | Bitwise XOR | a ^ b | 6 |
| ~ | Bitwise Complement | ~a | -6 |
| << | Left Shift | a << 1 | 10 |
| >> | Right Shift | a >> 1 | 2 |

o Type Conversion and Type Casting

**1. Type Conversion (Widening / Implicit Casting)**

* **Automatically done by Java**
* Converts **smaller type to larger type**
* **Safe**, no data loss

**🔸 Example:**

java

CopyEdit

int a = 10;

double b = a; // int to double (implicit)

System.out.println(b); // Output: 10.0

**🔹 Conversion Order (small → big):**

byte → short → int → long → float → double

**2. Type Casting (Narrowing / Explicit Casting)**

* **Manually done by the programmer**
* Converts **larger type to smaller type**
* **May cause data loss**

**🔸 Example:**

java

CopyEdit

double x = 9.78;

int y = (int) x; // double to int (explicit)

System.out.println(y); // Output: 9

• Lab Exercise:

o Write a program to demonstrate the use of different data types.

public class DataTypesDemo {

public static void main(String[] args) {

// Integer types

byte myByte = 100;

short myShort = 32000;

int myInt = 100000;

long myLong = 10000000000L;

// Floating-point types

float myFloat = 5.75f;

double myDouble = 19.99;

// Character type

char myChar = 'J';

// Boolean type

boolean isJavaFun = true;

// Printing all variables

System.out.println("byte: " + myByte);

System.out.println("short: " + myShort);

System.out.println("int: " + myInt);

System.out.println("long: " + myLong);

System.out.println("float: " + myFloat);

System.out.println("double: " + myDouble);

System.out.println("char: " + myChar);

System.out.println("boolean: " + isJavaFun);

}

}

o Create a calculator using arithmetic and relational operators.

import java.util.Scanner;

public class Calculator {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

// Input two numbers

System.out.print("Enter first number: ");

double num1 = sc.nextDouble();

System.out.print("Enter second number: ");

double num2 = sc.nextDouble();

// Arithmetic operations

System.out.println("\nArithmetic Operations:");

System.out.println("Addition: " + (num1 + num2));

System.out.println("Subtraction: " + (num1 - num2));

System.out.println("Multiplication: " + (num1 \* num2));

System.out.println("Division: " + (num1 / num2));

System.out.println("Modulus: " + (num1 % num2));

// Relational operations

System.out.println("\nRelational Operations:");

System.out.println(num1 + " == " + num2 + " : " + (num1 == num2));

System.out.println(num1 + " != " + num2 + " : " + (num1 != num2));

System.out.println(num1 + " > " + num2 + " : " + (num1 > num2));

System.out.println(num1 + " < " + num2 + " : " + (num1 < num2));

System.out.println(num1 + " >= " + num2 + " : " + (num1 >= num2));

System.out.println(num1 + " <= " + num2 + " : " + (num1 <= num2));

sc.close();

}

}

o Demonstrate type casting (explicit and implicit).

public class TypeCastingDemo {

public static void main(String[] args) {

System.out.println("Implicit Type Casting (Widening):");

int intVal = 100;

long longVal = intVal; // int to long

float floatVal = longVal; // long to float

System.out.println("int value: " + intVal);

System.out.println("long value (from int): " + longVal);

System.out.println("float value (from long): " + floatVal);

System.out.println("\nExplicit Type Casting (Narrowing):");

double doubleVal = 99.99;

int newIntVal = (int) doubleVal; // double to int (decimal will be lost)

System.out.println("double value: " + doubleVal);

System.out.println("int value (from double): " + newIntVal);

// Overflow example

int bigInt = 130;

byte smallByte = (byte) bigInt; // Overflow will happen here

System.out.println("\nOverflow in explicit casting:");

System.out.println("int value: " + bigInt);

System.out.println("byte value (from int): " + smallByte);

}

}

1. Control Flow Statements

• Theory:

o If-Else Statements

**1. If-Else Statements**

* Used for **decision making**.
* Executes certain blocks of code based on whether a condition is **true** or **false**.

Syntax:

if(condition) {

// Executes when condition is true

} else {

// Executes when condition is false

}

* You can have multiple conditions using else if.

**2. Switch Case Statements**

* An alternative to multiple if-else when checking the same variable for many possible values.
* It **compares a variable** against **constant values**.
* Each matching case executes its block.
* break keyword is used to **exit** the switch.
* default case runs if no other case matches.

**3. Loops**

* Loops are used to **repeat** a block of code multiple times.

**Types:**

* **For loop**: Repeats code for a known number of times.
* **While loop**: Repeats as long as a condition is true.
* **Do-while loop**: Similar to while, but runs the code **at least once** before checking condition.

**4. Break and Continue**

* **Break**: Immediately **exits** the nearest loop or switch statement.
* **Continue**: Skips the rest of the current loop iteration and **jumps to the next iteration**.

• Lab Exercise:

o Write a program to find if a number is even or odd using an if-else statement

import java.util.Scanner;

public class OddEvenCheck {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter an integer: ");

int number = sc.nextInt();

if (number % 2 == 0) {

System.out.println(number + " is Even.");

} else {

System.out.println(number + " is Odd.");

}

sc.close();

}

}.

o Implement a simple menu-driven program using a switch-case.

import java.util.Scanner;

public class MenuDrivenCalculator {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("=== Simple Calculator Menu ===");

System.out.println("1. Add");

System.out.println("2. Subtract");

System.out.println("3. Multiply");

System.out.println("4. Divide");

System.out.print("Choose an option (1-4): ");

int choice = sc.nextInt();

System.out.print("Enter first number: ");

double num1 = sc.nextDouble();

System.out.print("Enter second number: ");

double num2 = sc.nextDouble();

switch (choice) {

case 1:

System.out.println("Result: " + (num1 + num2));

break;

case 2:

System.out.println("Result: " + (num1 - num2));

break;

case 3:

System.out.println("Result: " + (num1 \* num2));

break;

case 4:

if (num2 != 0)

System.out.println("Result: " + (num1 / num2));

else

System.out.println("Error: Division by zero not allowed.");

break;

default:

System.out.println("Invalid choice!");

}

sc.close();

}

}

o Write a program to display the Fibonacci series using a loop.

public class SimpleFibonacci {

public static void main(String[] args) {

int n = 10; // Total 10 terms

int a = 0, b = 1;

System.out.print("Fibonacci Series: ");

for (int i = 1; i <= n; i++) {

System.out.print(a + " ");

int next = a + b;

a = b;

b = next;

}

}

}

1. Classes and Objects

• Theory:

o Defining a Class and Object in Java

**What is an Object?**

* An **Object** is an **instance** of a class.
* When you create an object, memory is allocated, and you can access the properties and behaviors defined in the class.

**What is an Object?**

* An **Object** is an **instance** of a class.
* When you create an object, memory is allocated, and you can access the properties and behaviors defined in the class.

o Constructors and Overloading

**What is a Constructor?**

* A **Constructor** is a special method in a class used to **initialize objects**.
* It has the **same name as the class** and **no return type** (not even void).
* Automatically called when you create an object using new.
* Used to set initial values for object attributes.

**Constructor Overloading**

* You can have **more than one constructor** in a class with **different parameter lists**.
* This is called **constructor overloading**.
* Allows creating objects in different ways.

o Object Creation, Accessing Members of the Class

**Object Creation**

* To create an **object** (an instance of a class), use the **new** keyword followed by the class constructor.
* Syntax:

ClassName objectName = new ClassName();

* This allocates memory for the object and calls the constructor to initialize it.

**Accessing Members of a Class**

* Once an object is created, you can access its **variables (fields)** and **methods** using the **dot (.) operator**.
* Example: objectName.variableName or objectName.methodName()

o this Keyword

**What is this Keyword?**

* this is a **reference variable** that refers to the **current object** — the object whose method or constructor is being called.
* It is mainly used to **resolve naming conflicts** between instance variables and parameters, or to refer to the current object explicitly.

• Lab Exercise:

o Create a class Student with attributes (name, age) and a method to display the details. Create multiple constructors in a class and demonstrate constructor overloading.

class Student {

String name;

int age;

// Constructor 1: No-argument constructor

Student() {

name = "Unknown";

age = 0;

}

// Constructor 2: Constructor with name only

Student(String name) {

this.name = name;

age = 0;

}

// Constructor 3: Constructor with name and age

Student(String name, int age) {

this.name = name;

this.age = age;

}

// Method to display student details

void displayDetails() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

public class Main {

public static void main(String[] args) {

// Create Student objects using different constructors

Student s1 = new Student();

Student s2 = new Student("srujal");

Student s3 = new Student("sam", 20);

// Display details

s1.displayDetails();

s2.displayDetails();

s3.displayDetails();

}

}

o Implement a simple class with getters and setters for encapsulation.

class Student {

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

if (age > 0) {

this.age = age;

} else {

System.out.println("Age must be positive.");

}

}

}

public class Main {

public static void main(String[] args) {

Student s = new Student();

s.setName("Srujal");

s.setAge(21);

System.out.println("Name: " + s.getName());

System.out.println("Age: " + s.getAge());

s.setAge(-5);

}

}

1. Methods in Java

• Theory:

o Defining Methods

**What is a Method?**

* A **method** is a block of code that performs a specific task.
* It helps to **reuse code**, **organize programs**, and **make them modular**.
* Methods can take input parameters, perform operations, and optionally return a result.

o Method Parameters and Return Types

**Method Parameters**

* **Parameters** are variables declared in a method’s definition.
* They allow passing values into methods when you call them.
* You can have zero or more parameters separated by commas.
* Parameters have a **type** and a **name**.

o Method Overloading

**What is Method Overloading?**

* **Method Overloading** means having **multiple methods with the same name** but **different parameter lists** (different type, number, or both).
* It allows a class to perform a similar operation in different ways depending on the input.
* Return type alone **cannot** distinguish overloaded methods; parameter types/count must differ.

o Static Methods and Variables

**🔹 What is static in Java?**

* The static keyword is used to indicate that **a method or variable belongs to the class itself**, rather than to instances (objects) of the class.
* So, **you don’t need to create an object** to access static members.

• Lab Exercise:

o Write a program to find the maximum of three numbers using a method.

public class MaxOfThree {

// Method to find maximum of three numbers

static int findMax(int a, int b, int c) {

int max;

if (a >= b && a >= c) {

max = a;

} else if (b >= a && b >= c) {

max = b;

} else {

max = c;

}

return max;

}

public static void main(String[] args) {

int num1 = 25, num2 = 42, num3 = 17;

int maximum = findMax(num1, num2, num3);

System.out.println("Maximum number is: " + maximum);

}

}

o Implement method overloading by creating methods for different data types.

public class OverloadDemo {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

String add(String a, String b) {

return a + b;

}

public static void main(String[] args) {

OverloadDemo obj = new OverloadDemo();

System.out.println("Integer Addition: " + obj.add(10, 20));

System.out.println("Double Addition: " + obj.add(5.5, 6.7));

System.out.println("String Concatenation: " + obj.add("Hello, ", "Srujal!"));

}

}

o Create a class with static variables and methods to demonstrate their use.

public class College {

static String collegeName = "ABC Institute"; // static variable

String studentName;

int rollNo;

College(String studentName, int rollNo) {

this.studentName = studentName;

this.rollNo = rollNo;

}

void display() {

System.out.println("Name: " + studentName);

System.out.println("Roll No: " + rollNo);

System.out.println("College: " + collegeName);

}

static void changeCollegeName(String newName) {

collegeName = newName;

}

public static void main(String[] args) {

College.changeCollegeName("XYZ University");

College s1 = new College("Srujal", 1);

College s2 = new College("Ankit", 2);

s1.display();

System.out.println("--------------------");

s2.display();

}

}

1. Object-Oriented Programming (OOPs) Concepts

• Theory:

o Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction

**1. Encapsulation**

**Definition**: Encapsulation is the technique of **bundling data** (variables) and the **methods** that operate on that data into a single unit (class), and **restricting direct access** to some of the object's components.

— *Oracle Java Documentation*

**2. Inheritance**

**Definition**: Inheritance is a mechanism wherein a new class **acquires the properties and behaviors** (fields and methods) of an existing class. It supports **code reusability** and establishes a **parent-child relationship** between classes.

— *GeeksforGeeks / Oracle Java Tutorials*

**3. Polymorphism**

**Definition**: Polymorphism allows objects to be **treated as instances of their parent class**, even though they may behave differently. It refers to the **ability of a variable, function, or object to take on multiple forms**.

— *W3Schools / JavaTPoint*

**4. Abstraction**

**Definition**: Abstraction is the process of **hiding the internal implementation details** and showing only the **functionality** to the user. It lets you focus on **what** an object does instead of **how** it does it.

— *Oracle Java Documentation*

o Inheritance: Single, Multilevel, Hierarchical o Method Overriding and Dynamic Method Dispatch

**Single Inheritance**

**Definition**:  
When a child class inherits the properties and behavior of a single parent class.

**Syntax**:

class Parent {

// parent class code

}

class Child extends Parent {

// child class code

}

**Multilevel Inheritance**

**Definition**:  
When a class inherits from a child class, which itself inherits from another class.

**Syntax**:

class Grandparent {

// code

}

class Parent extends Grandparent {

// code

}

class Child extends Parent {

// code

}

**Hierarchical Inheritance**

**Definition**:  
When multiple classes inherit from a single parent class.

**Syntax**:

class Parent {

// code

}

class Child1 extends Parent {

// code

}

class Child2 extends Parent {

// code

}

**Method Overriding**

**Definition**:  
When a subclass provides its own version of a method that is already defined in its superclass.

**Syntax**:

class Parent {

void show() {

System.out.println("Parent class");

}

}

class Child extends Parent {

@Override

void show() {

System.out.println("Child class");

}

}

**Dynamic Method Dispatch (Runtime Polymorphism)**

**Definition**:  
The process where a call to an overridden method is resolved at runtime using the parent class reference pointing to a child class object.

**Syntax**:

Parent obj;

obj = new Child();

obj.show(); // Calls Child class method at runtime

• Lab Exercise:

o Write a program demonstrating single inheritance.

class Animal {

void eat() {

System.out.println("Animal is eating");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog is barking");

}

}

public class Main {

public static void main(String[] args) {

Dog d = new Dog();

d.eat();

d.bark();

}

}

o Create a class hierarchy and demonstrate multilevel inheritance.

class Animal {

void eat() {

System.out.println("Animal is eating");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Dog is barking");

}

}

class Puppy extends Dog {

void weep() {

System.out.println("Puppy is weeping");

}

}

public class Main {

public static void main(String[] args) {

Puppy p = new Puppy();

p.eat();

p.bark();

p.weep();

}

}

o Implement method overriding to show polymorphism in action.

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Animal a = new Animal();

a.sound();

Animal d = new Dog();

d.sound();

}

}

1. Constructors and Destructors

• Theory:

**Constructor Types**

**1. Default Constructor**

* A no-argument constructor automatically provided by Java if no constructor is defined by the user.
* It initializes objects with default values.

**2. Parameterized Constructor**

* A constructor that takes parameters to initialize an object with specific values.

**Copy Constructor (Emulated in Java)**

* Java does not provide a built-in copy constructor like C++.
* You can emulate it by creating a constructor that takes an object of the same class and copies its fields manually.

**Constructor Overloading**

* Defining multiple constructors in the same class with different parameter lists.
* Allows creating objects in different ways depending on the arguments passed.

**Object Life Cycle and Garbage Collection**

* **Object Life Cycle**:
  1. **Creation** — Object is created using new.
  2. **Usage** — Object is used in the program.
  3. **Eligibility for Garbage Collection** — When there are no references to the object.
  4. **Garbage Collection** — JVM automatically frees memory by destroying unreachable objects.
* **Garbage Collection**:
  1. Automatic process by JVM to reclaim memory.
  2. Programmers do not explicitly free memory.
  3. The finalize() method can be overridden but is deprecated in recent Java versions.

• Lab Exercise:

o Write a program to create and initialize an object using a parameterized constructor.

class Student {

String name;

int age;

Student(String n, int a) {

name = n;

age = a;

}

void display() {

System.out.println("Name: " + name);

System.out.println("Age: " + age);

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student("Srujal", 21);

s1.display();

}

}

o Demonstrate constructor overloading by passing different types of parameters.

class Calculator {

int result;

Calculator() {

result = 0;

}

Calculator(int a, int b) {

result = a + b;

}

Calculator(double a, double b) {

result = (int)(a \* b);

}

void display() {

System.out.println("Result = " + result);

}

}

public class Main {

public static void main(String[] args) {

Calculator c1 = new Calculator();

c1.display();

Calculator c2 = new Calculator(10, 20);

c2.display();

Calculator c3 = new Calculator(5.5, 4.5);

c3.display();

}

}

8. Arrays and Strings

• Theory:

**One-Dimensional and Multidimensional Arrays**

* **One-Dimensional Array:**  
  A linear collection of elements of the same data type, accessed by a single index.  
  Example: int[] arr = new int[5];
* **Multidimensional Array:**  
  An array of arrays, commonly used as a matrix with rows and columns.  
  Example: int[][] matrix = new int[3][4];

**String Handling in Java**

* **String Class:**  
  Immutable sequence of characters. Once created, contents cannot be changed.  
  Example: String str = "Hello";
* **StringBuffer Class:**  
  Mutable sequence of characters. Thread-safe and synchronized. Suitable for use in multithreaded environment.  
  Example: StringBuffer sb = new StringBuffer("Hello");
* **StringBuilder Class:**  
  Mutable sequence of characters. Not synchronized, faster than StringBuffer but not thread-safe.  
  Example: StringBuilder sb = new StringBuilder("Hello");

**Array of Objects**

* An array that holds references to objects instead of primitive data types.  
  Example:

java

CopyEdit

Student[] students = new Student[3];

students[0] = new Student("Srujal");

**Common String Methods**

| **Method** | **Description** |
| --- | --- |
| length() | Returns the length of the string |
| charAt(int i) | Returns the character at index i |
| substring(int beginIndex) | Returns substring from beginIndex to end |
| substring(int begin, int end) | Returns substring between begin and end-1 |
| equals(String s) | Compares two strings for equality |
| toUpperCase() | Converts string to uppercase |
| toLowerCase() | Converts string to lowercase |
| trim() | Removes leading and trailing whitespace |

• Lab Exercise:

o Write a program to perform matrix addition and subtraction using 2D arrays.

public class MatrixOperations {

public static void main(String[] args) {

int[][] matrix1 = {{1, 2, 3}, {4, 5, 6}};

int[][] matrix2 = {{6, 5, 4}, {3, 2, 1}};

int rows = matrix1.length;

int cols = matrix1[0].length;

int[][] sum = new int[rows][cols];

int[][] diff = new int[rows][cols];

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

sum[i][j] = matrix1[i][j] + matrix2[i][j];

diff[i][j] = matrix1[i][j] - matrix2[i][j];

}

}

System.out.println("Sum of matrices:");

printMatrix(sum);

System.out.println("Difference of matrices:");

printMatrix(diff);

}

static void printMatrix(int[][] matrix) {

for (int[] row : matrix) {

for (int val : row) {

System.out.print(val + " ");

}

System.out.println();

}

}

}

o Create a program to reverse a string and check for palindromes.

public class StringPalindrome {

public static void main(String[] args) {

String str = "madam";

String reversed = "";

for (int i = str.length() - 1; i >= 0; i--) {

reversed += str.charAt(i);

}

System.out.println("Original String: " + str);

System.out.println("Reversed String: " + reversed);

if (str.equals(reversed)) {

System.out.println("It is a palindrome");

} else {

System.out.println("It is not a palindrome");

}

}

}

o Implement string comparison using equals() and compareTo() methods.

public class StringComparison {

public static void main(String[] args) {

String s1 = "Apple";

String s2 = "apple";

String s3 = "Apple";

System.out.println("s1.equals(s2): " + s1.equals(s2sensitive

System.out.println("s1.equals(s3): " + s1.equals(s3));

System.out.println("s1.compareTo(s2): " + s1.compareTo(s2));

System.out.println("s1.compareTo(s3): " + s1.compareTo(s3));

}

}

1. Inheritance and Polymorphism

• Theory:

**Inheritance Types and Benefits**

**Types of Inheritance in Java:**

* **Single Inheritance:** One class inherits from one superclass.
* **Multilevel Inheritance:** Class inherits from a derived class, forming a chain.
* **Hierarchical Inheritance:** Multiple classes inherit from a single superclass.
* **Hybrid Inheritance:** Combination of two or more inheritance types (using interfaces in Java).

**Benefits:**

* **Code Reusability:** Inherit existing code, avoid duplication.
* **Method Overriding:** Modify inherited behavior.
* **Improved Maintainability:** Changes in superclass reflect in subclasses.
* **Polymorphism:** Ability to process objects differently based on their class.

**Method Overriding**

* Occurs when a subclass provides a specific implementation of a method already defined in its superclass.
* Signature (method name, parameters) must be the same.
* Enables runtime polymorphism.

**Dynamic Binding (Run-Time Polymorphism)**

* Method call is resolved at runtime rather than compile time.
* JVM determines which method to execute based on the object type being referred to, not the reference type.
* Achieved through method overriding.

**Super Keyword**

* Refers to the immediate parent class object.
* Used to:
  + Call parent class constructor.
  + Access parent class methods and variables.

Example:

super.methodName();

super.variableName;

super();

**Method Hiding**

* When a subclass defines a static method with the same signature as a static method in the superclass.
* Unlike method overriding, static methods are resolved at compile time.
* The subclass method hides the superclass method.

• Lab Exercise:

o Write a program that demonstrates inheritance using extends keyword.

class Vehicle {

String brand;

Vehicle(String brand) {

this.brand = brand;

}

void display() {

System.out.println("Vehicle brand: " + brand);

}

}

class Car extends Vehicle {

String model;

Car(String brand, String model) {

super(brand); // call parent constructor

this.model = model;

}

@Override

void display() {

super.display(); // call parent method

System.out.println("Car model: " + model);

}

}

public class Main {

public static void main(String[] args) {

Vehicle v = new Vehicle("GenericBrand");

v.display();

Vehicle c = new Car("Toyota", "Corolla");

c.display(); // runtime polymorphism - calls overridden method

}

}

o Implement runtime polymorphism by overriding methods in the child class.

class Vehicle {

String brand;

Vehicle(String brand) {

this.brand = brand;

}

void display() {

System.out.println("Vehicle brand: " + brand);

}

}

class Car extends Vehicle {

String model;

Car(String brand, String model) {

super(brand); // call parent constructor

this.model = model;

}

@Override

void display() {

super.display(); // call parent method

System.out.println("Car model: " + model);

}

}

public class Main {

public static void main(String[] args) {

Vehicle v = new Vehicle("GenericBrand");

v.display();

Vehicle c = new Car("Toyota", "Corolla");

c.display(); // runtime polymorphism - calls overridden method

}

}

o Use the super keyword to call the parent class constructor and methods.

class Person {

String name;

Person(String name) {

this.name = name;

}

void display() {

System.out.println("Name: " + name);

}

}

class Employee extends Person {

int employeeId;

Employee(String name, int employeeId) {

super(name); // call parent class constructor

this.employeeId = employeeId;

}

@Override

void display() {

super.display(); // call parent class method

System.out.println("Employee ID: " + employeeId);

}

}

public class Main {

public static void main(String[] args) {

Employee emp = new Employee("Srujal", 101);

emp.display();

}

}

1. Interfaces and Abstract Classes

• Theory:

o Abstract Classes and Methods

**Abstract Classes and Methods**

* **Abstract Class:**  
  A class declared with the keyword abstract.  
  It cannot be instantiated directly.  
  Can contain both abstract (without body) and concrete methods.  
  Used to provide a base class with some common functionality and some methods to be implemented by subclasses.
* **Abstract Method:**  
  A method declared without an implementation (no method body) using the keyword abstract.  
  Must be implemented by subclasses.

o Interfaces: Multiple Inheritance in Java o Implementing Multiple Interfaces

**Interface:**  
A reference type in Java, similar to a class, that can contain only abstract methods (before Java 8) and default/static methods (from Java 8 onward).  
Defines a contract that implementing classes must follow.  
Supports multiple inheritance, i.e., a class can implement multiple interfaces.

**Multiple Inheritance:**  
Java does not support multiple inheritance of classes to avoid ambiguity, but interfaces allow a form of multiple inheritance.

• Lab Exercise:

o Create an abstract class and implement its methods in a subclass.

abstract class Animal {

abstract void sound(); // abstract method

void sleep() {

System.out.println("Animal is sleeping");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Dog d = new Dog();

d.sound();

d.sleep();

}

}

o Write a program that implements multiple interfaces in a single class.

interface Printable {

void print();

}

interface Showable {

void show();

}

class Demo implements Printable, Showable {

public void print() {

System.out.println("Print method called");

}

public void show() {

System.out.println("Show method called");

}

}

public class Main {

public static void main(String[] args) {

Demo obj = new Demo();

obj.print();

obj.show();

}

}

o Implement an interface for a real-world example,such as a payment gateway.

interface PaymentGateway {

void pay(double amount);

boolean refund(double amount);

}

class PayPal implements PaymentGateway {

@Override

public void pay(double amount) {

System.out.println("Paid $" + amount + " via PayPal.");

}

@Override

public boolean refund(double amount) {

System.out.println("Refunded $" + amount + " via PayPal.");

return true;

}

}

public class Main {

public static void main(String[] args) {

PaymentGateway payment = new PayPal();

payment.pay(150.75);

payment.refund(50.25);

}

}

10.Packages and Access Modifiers

• Theory:

**Java Packages**

* **Packages** group related classes and interfaces to organize code and avoid naming conflicts.
* **Built-in Packages:** Provided by Java API, e.g., java.util, java.io.
* **User-Defined Packages:** Created by developers to organize their own classes.

**Syntax to declare a package:**

java

CopyEdit

package mypackage;

**Access Modifiers**

| **Modifier** | **Within Class** | **Within Package** | **Subclass (Same Package)** | **Subclass (Different Package)** | **Everywhere** |
| --- | --- | --- | --- | --- | --- |
| private | Yes | No | No | No | No |
| **Default** (no modifier) | Yes | Yes | Yes | No | No |
| protected | Yes | Yes | Yes | Yes | No |
| public | Yes | Yes | Yes | Yes | Yes |

* **private:** Accessible only within the class.
* **default:** Accessible within the same package.
* **protected:** Accessible within the same package and subclasses.
* **public:** Accessible from everywhere.

**Importing Packages and Classpath**

* Use import statement to access classes from other packages.

Example:

java

CopyEdit

import java.util.Scanner;

* Use wildcard \* to import all classes in a package:

java

CopyEdit

import java.util.\*;

* **Classpath:** Specifies the location of user-defined classes and packages during compile and runtime.
* Can be set via command line or environment variables.

• Lab Exercise:

o Create a user-defined package and import it into another program.

// File: mypackage/Greeting.java

package mypackage;

public class Greeting {

public void sayHello() {

System.out.println("Hello from mypackage!");

}

}

o Demonstrate the use of different access modifiers within the same package

and across different packages.

// File: mypackage/AccessDemo.java

package mypackage;

public class AccessDemo {

private int privateVar = 1;

int defaultVar = 2; // default access

protected int protectedVar = 3;

public int publicVar = 4;

public void display() {

System.out.println("privateVar = " + privateVar);

System.out.println("defaultVar = " + defaultVar);

System.out.println("protectedVar = " + protectedVar);

System.out.println("publicVar = " + publicVar);

}

}

12. Exception Handling

• Theory:

**Types of Exceptions**

1. **Checked Exceptions:**
   * Checked at compile-time by the compiler.
   * Must be either handled using try-catch or declared with throws keyword.
   * Examples: IOException, SQLException.
2. **Unchecked Exceptions:**
   * Also called Runtime Exceptions.
   * Not checked at compile-time.
   * Occur due to programming errors like logic mistakes or improper use of APIs.
   * Examples: NullPointerException, ArithmeticException.

**try, catch, finally, throw, throws**

* **try:**  
  Block of code to monitor for exceptions.
* **catch:**  
  Block to handle exceptions thrown in the try block.
* **finally:**  
  Block that executes after try/catch regardless of exception occurrence. Typically used for cleanup.
* **throw:**  
  Used to explicitly throw an exception.
* **throws:**  
  Declares exceptions that a method might throw, passing responsibility to the caller.

**Syntax Example:**

try {

// code that may throw exception

} catch (ExceptionType e) {

// exception handler

} finally {

// always executed

}

void method() throws IOException {

// method that may throw IOException

}

throw new IOException("File not found");

**Custom Exception Classes**

* You can create your own exception by extending Exception (checked) or RuntimeException (unchecked).

**Example:**

java

CopyEdit

class MyException extends Exception {

MyException(String message) {

super(message);

}

}

• Lab Exercise:

o Write a program to demonstrate exception handling using try-catch-finally.

public class ExceptionDemo {

public static void main(String[] args) {

try {

int result = 10 / 0; // Causes ArithmeticException

System.out.println("Result: " + result);

} catch (ArithmeticException e) {

System.out.println("Error: Division by zero");

} finally {

System.out.println("Finally block always executes");

}

}

}

o Implement multiple catch blocks for different types of exceptions.

public class MultipleCatchDemo {

public static void main(String[] args) {

try {

int[] arr = new int[5];

arr[10] = 50; // ArrayIndexOutOfBoundsException

int result = 10 / 0; // ArithmeticException

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Error: Array index is out of bounds");

} catch (ArithmeticException e) {

System.out.println("Error: Division by zero");

}

}

}

o Create a custom exception class and use it in your program.

class MyException extends Exception {

MyException(String message) {

super(message);

}

}

public class CustomExceptionDemo {

static void validate(int age) throws MyException {

if (age < 18) {

throw new MyException("Age must be 18 or older");

} else {

System.out.println("Age is valid");

}

}

public static void main(String[] args) {

try {

validate(15);

} catch (MyException e) {

System.out.println("Caught custom exception: " + e.getMessage());

}

}

}

13. Multithreading

• Theory:

**Introduction to Threads**

* A **thread** is a lightweight sub-process, the smallest unit of execution within a program.
* Multithreading allows concurrent execution of two or more threads for maximum utilization of CPU.

**Creating Threads**

1. **Extending Thread class:**

java

CopyEdit

class MyThread extends Thread {

public void run() {

System.out.println("Thread running");

}

}

1. **Implementing Runnable interface:**

java

CopyEdit

class MyRunnable implements Runnable {

public void run() {

System.out.println("Runnable thread running");

}

}

* In both cases, override the run() method, which contains the code executed by the thread.

**Thread Life Cycle**

* **New:** Thread object created.
* **Runnable:** Thread ready to run and waiting for CPU.
* **Running:** Thread is executing.
* **Blocked/Waiting:** Thread waiting for resources or notification.
* **Terminated:** Thread finished execution or stopped.

**Synchronization and Inter-thread Communication**

* **Synchronization:**  
  Used to control access to shared resources by multiple threads to prevent data inconsistency.  
  Achieved using the synchronized keyword.
* **Inter-thread Communication:**  
  Allows threads to communicate and coordinate by methods like wait(), notify(), and notifyAll().

• Lab Exercise:

o Write a program to create and run multiple threads using the Thread class.

class MyThread extends Thread {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println(getName() + " - Count: " + i);

try {

Thread.sleep(500); // Pause for 0.5 seconds

} catch (InterruptedException e) {

System.out.println("Thread interrupted");

}

}

}

}

public class ThreadDemo {

public static void main(String[] args) {

MyThread t1 = new MyThread();

MyThread t2 = new MyThread();

t1.start();

t2.start();

}

}

o Implement thread synchronization using synchronized blocks or methods.

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

public int getCount() {

return count;

}

}

class MyThread extends Thread {

private Counter counter;

MyThread(Counter counter) {

this.counter = counter;

}

public void run() {

for (int i = 0; i < 1000; i++) {

counter.increment();

}

}

}

public class SyncDemo {

public static void main(String[] args) throws InterruptedException {

Counter counter = new Counter();

MyThread t1 = new MyThread(counter);

MyThread t2 = new MyThread(counter);

t1.start();

t2.start();

t1.join();

t2.join();

System.out.println("Final count: " + counter.getCount());

}

}

o Use inter-thread communication methods like wait(), notify(), and notifyAll().

class Message {

private String msg;

private boolean empty = true;

public synchronized void write(String msg) {

while (!empty) {

try {

wait();

} catch (InterruptedException e) {}

}

empty = false;

this.msg = msg;

notify();

}

public synchronized String read() {

while (empty) {

try {

wait();

} catch (InterruptedException e) {}

}

empty = true;

notify();

return msg;

}

}

class Writer extends Thread {

private Message message;

Writer(Message message) {

this.message = message;

}

public void run() {

String[] msgs = { "Hello", "How are you?", "Goodbye" };

for (String m : msgs) {

message.write(m);

System.out.println("Written: " + m);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {}

}

}

}

class Reader extends Thread {

private Message message;

Reader(Message message) {

this.message = message;

}

public void run() {

for (int i = 0; i < 3; i++) {

String msg = message.read();

System.out.println("Read: " + msg);

}

}

}

public class InterThreadCommDemo {

public static void main(String[] args) {

Message message = new Message();

new Writer(message).start();

new Reader(message).start();

}

}

14. File Handling

• Theory:

**Introduction to File I/O in Java**

* Java provides file input/output (I/O) operations through the java.io package.
* File I/O allows reading from and writing to files, enabling data persistence outside the program.

**FileReader and FileWriter Classes**

* **FileReader:** Used for reading character files.
* **FileWriter:** Used for writing character files.

**Example:**

FileReader fr = new FileReader("input.txt");

FileWriter fw = new FileWriter("output.txt");

**BufferedReader and BufferedWriter**

* **BufferedReader:** Wraps around FileReader to efficiently read text by buffering characters. Provides convenient methods like readLine().
* **BufferedWriter:** Wraps around FileWriter to write text efficiently with buffering and methods like newLine().

**Example:**

BufferedReader br = new BufferedReader(new FileReader("input.txt"));

BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"));

**Serialization and Deserialization**

* **Serialization:** Process of converting an object into a byte stream to save it to a file or send it over a network.
* **Deserialization:** Converting the byte stream back into a copy of the object.
* Classes must implement the Serializable interface to support serialization.

**Example:**

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("file.ser"));

oos.writeObject(object);

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("file.ser"));

Object object = ois.readObject();

• Lab Exercise:

o Write a program to read and write content to a file using FileReader and FileWriter.

package file\_io;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class Read\_write {

public static void main(String[] args) throws IOException {

FileInputStream fn=new FileInputStream("test.txt");

FileOutputStream fout=new FileOutputStream("test2.txt");

int i;

while((i=fn.read())!=-1) {

System.out.print((char)i);

fout.write(i);

}

}

}

o Implement a program that reads a file line by line using BufferedReader.

o Create a program that demonstrates object serialization and deserialization.

package file\_io;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

class person implements Serializable

{

int id;

String name;

public person(int id,String name) {

this.id=id;

this.name=name;

}

}

public class Serialize\_demo {

public static void main(String[] args) throws IOException, ClassNotFoundException {

person p=new person(1,"test");

// FileOutputStream fout=new FileOutputStream("ser\_demo.txt");

// ObjectOutputStream os=new ObjectOutputStream(fout);

// os.writeObject(p);

FileInputStream fin=new FileInputStream("ser\_demo.txt");

ObjectInputStream os=new ObjectInputStream(fin);

person p1=(person) os.readObject();

System.out.print(p1.id+" "+p1.name);

}

}

15. Collections Framework

• Theory:

**Introduction to Collections Framework**

* The **Collections Framework** in Java provides a set of interfaces and classes to store and manipulate groups of objects.
* It standardizes how collections are handled, including lists, sets, maps, and queues.

**Core Interfaces**

1. **List**
   * Ordered collection (allows duplicates).
   * Examples: ArrayList, LinkedList.
2. **Set**
   * Unordered collection (no duplicates allowed).
   * Examples: HashSet, TreeSet.
3. **Map**
   * Stores key-value pairs (keys unique).
   * Examples: HashMap, TreeMap.
4. **Queue**
   * Collection designed for holding elements prior to processing (FIFO order).
   * Examples: LinkedList (implements Queue), PriorityQueue.

**Common Implementations**

* **ArrayList:** Resizable-array implementation of List; fast random access.
* **LinkedList:** Doubly-linked list implementation of List and Queue; efficient insertions/deletions.
* **HashSet:** Backed by a hash table; fast access, no order guarantee.
* **TreeSet:** Sorted set based on a red-black tree; elements are sorted.
* **HashMap:** Hash table based implementation of Map; fast key-based lookup, no order guarantee.
* **TreeMap:** Sorted map based on red-black tree; keys are sorted.

**Iterators and ListIterators**

* **Iterator:** Provides methods (hasNext(), next(), remove()) to traverse collections like Set, List, etc.
* **ListIterator:** Extends Iterator for Lists; allows bi-directional traversal (hasPrevious(), previous()), and element modification.

• Lab Exercise:

o Write a program that demonstrates the use of an ArrayList and LinkedList.

import java.util.ArrayList;

import java.util.LinkedList;

public class ListDemo {

public static void main(String[] args) {

ArrayList<String> arrayList = new ArrayList<>();

arrayList.add("Apple");

arrayList.add("Banana");

arrayList.add("Cherry");

LinkedList<String> linkedList = new LinkedList<>();

linkedList.add("Dog");

linkedList.add("Elephant");

linkedList.add("Frog");

System.out.println("ArrayList elements:");

for (String fruit : arrayList) {

System.out.println(fruit);

}

System.out.println("\nLinkedList elements:");

for (String animal : linkedList) {

System.out.println(animal);

}

}

}

o Implement a program using HashSet to remove duplicate elements from a list.

import java.util.ArrayList;

import java.util.HashSet;

public class RemoveDuplicates {

public static void main(String[] args) {

ArrayList<String> listWithDuplicates = new ArrayList<>();

listWithDuplicates.add("Java");

listWithDuplicates.add("Python");

listWithDuplicates.add("Java");

listWithDuplicates.add("C++");

listWithDuplicates.add("Python");

System.out.println("Original List: " + listWithDuplicates);

HashSet<String> set = new HashSet<>(listWithDuplicates);

System.out.println("List after removing duplicates using HashSet: " + set);

}

}o Create a HashMap to store and retrieve key-value pairs.

import java.util.HashMap;

public class HashMapDemo {

public static void main(String[] args) {

HashMap<Integer, String> map = new HashMap<>();

map.put(1, "One");

map.put(2, "Two");

map.put(3, "Three");

System.out.println("Value for key 2: " + map.get(2));

System.out.println("All key-value pairs:");

for (Integer key : map.keySet()) {

System.out.println(key + " = " + map.get(key));

}

}

}

16. Java Input/Output (I/O)

• Theory:

**Streams in Java**

* Streams are sequences of data used to perform input and output (I/O) operations.
* They provide a way to read from and write to data sources like files, memory, or network connections.

**Types of Streams:**

* **InputStream:** Abstract class for reading byte streams (e.g., reading data from a file).
* **OutputStream:** Abstract class for writing byte streams (e.g., writing data to a file).

**Reading and Writing Data Using Streams**

* **InputStream** subclasses read bytes of data.
* **OutputStream** subclasses write bytes of data.
* Common subclasses:
  + FileInputStream / FileOutputStream — for file operations with raw bytes.
  + BufferedInputStream / BufferedOutputStream — add buffering for efficiency.

**Handling File I/O Operations**

* File streams allow reading and writing raw byte data to/from files.
* Typical steps for reading:
  1. Create a FileInputStream object linked to the file.
  2. Read bytes in a loop until end of file (EOF).
  3. Close the stream.
* Typical steps for writing:
  1. Create a FileOutputStream object linked to the file.
  2. Write bytes or byte arrays to the file.
  3. Close the stream.

• Lab Exercise:

o Write a program to read input from the console using Scanner.

import java.util.Scanner;

public class ConsoleInputDemo {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your name: ");

String name = scanner.nextLine();

System.out.print("Enter your age: ");

int age = scanner.nextInt();

System.out.println("Hello " + name + ", you are " + age + " years old.");

scanner.close();

}

}

o Implement a file copy program using FileInputStream and FileOutputStream.

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class FileCopyDemo {

public static void main(String[] args) {

try (FileInputStream fis = new FileInputStream("source.txt");

FileOutputStream fos = new FileOutputStream("destination.txt")) {

int byteData;

while ((byteData = fis.read()) != -1) {

fos.write(byteData);

}

System.out.println("File copied successfully.");

} catch (IOException e) {

System.out.println("Error: " + e.getMessage());

}

}

}

o Create a program that reads from one file and writes the content to another file

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class FileReadWriteDemo {

public static void main(String[] args) {

try (FileInputStream input = new FileInputStream("input.txt");

FileOutputStream output = new FileOutputStream("output.txt")) {

int data;

while ((data = input.read()) != -1) {

output.write(data);

}

System.out.println("Data copied from input.txt to output.txt");

} catch (IOException e) {

System.out.println("Exception: " + e.getMessage());

}

}

}