Module : 1

### 1.Introduction to Java

* Theory :
* History of Java :
* Java is a programming language originally developed by James Gosling at Sun Microsystems.
* and released in 1995 as a core component of Sun Microsystems' Java platform.
* The language derives much of its syntax from C and C++.
* Java is currently one of the most popular programming languages in use, particularly for client server web applications.
* Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture.
* Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture.
* Java's motto was **"Write Once, Run Anywhere"** (WORA), highlighting its platform independence using the Java Virtual Machine (JVM).
* Oak was renamed to **Java** (the name was inspired by Java coffee).
* Summary :
* **Creator**: James Gosling and team at Sun Microsystems
* **First Released**: 1995
* **Current Owner**: Oracle Corporation
* **Key Features**: Platform independence, robustness, security, object-oriented
* **Modern Usage**: Continues to evolve with regular updates and broad community support.
* Features of Java :
* Simple
* Object – Oriented
* Portable
* Distribute
* High Performance
* Multithreaded
* Robust
* Dynamic
* Secure
* Object – Oriented :
* Java is fully Object-oriented programming language. & Object oriented concepts are following.
* Class : A blueprint for creating objects. It defines properties (fields) and behaviors (methods).
* Object : A real-world entity created from a class. It has a state and behavior.
* Inheritance : Bundling data and methods that operate on that data within a class, and hiding internal details from outside access.
* Encapsulation : A mechanism where one class acquires properties and behavior of another class.
* Polymorphism : Ability to take many forms. There are two types:   
   • **Compile-time** (method overloading)   
   • **Runtime** (method overriding)
* Platform Independent :
* **Platform independence** means that a Java program can run on any operating system or hardware platform **without modification**.
* Allows Java applications to run across different operating systems without rewriting code.
* **Source Code Compilation:**
* Java code (.java file) is compiled by the Java compiler (javac) into an intermediate format called **bytecode** (.class file).
* This bytecode is **not** platform-specific.
* **Java Virtual Machine (JVM):**
* The bytecode is executed by the **JVM**, which is platform-specific.
* JVM interprets the bytecode and runs it on the native machine.
* Understanding JVM, JRE, and JDK
* JVM : Java Virtual Machine
* It is a platform-independent execution environment that converts the Java bytecode into machine language and executes it.
* Most programming languages compile source code directly into machine code that is designed to run on a specific operating system, such as Windows or UNIX.
* JVMs are available for many hardware and software platforms.
* The use of the same bytecode for all JVMs on all platforms allows Java to be described as a "compile once, run anywhere" programming language, as opposed to "write once, compile anywhere", which describes crossplatform compiled languages.
* Thus, the JVM is a crucial component of the Java platform.
* JRE : Java Run-Time Environment
* Java Runtime Environment contains JVM, class libraries, and other supporting files.
* It does not contain any development tools such as compiler, debugger, etc.
* Actually JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE.
* . If you want to run any java program, you need to have JRE installed in the system.
* Is a part of the Java Development Kit (JDK) that provides the necessary components to run Java applications.
* It acts as a layer between the Java application and the underlying operating system, handling the execution of Java bytecode.
* JDK : Java Development Kit
* is a comprehensive software development environment used for developing Java applications and applets.
* Java Developer Kit contains tools needed to develop the Java programs
* **Appletviewer:-** (For viewing java applets) this tool can be used to run and debug Java applets without a web browser
* **Javac:-** Javac means Java Compiler, which converts source code into Java bytecode.
* **Java :-** ( java interpreter) the loader for Java applications. This tool is an interpreter and can interpret the class files generated by the javac compiler. Now a single launcher is used for both development and deployment.
* The old deployment launcher, jre, no longer comes with Sun JDK, and instead it has been replaced by this new java loader.
* **Javap :-** ( java disassembler) the class file disassemble.
* **Javah:-** (produce header files) the C header and stub generator, used to write native methods.
* **Java doc:-** (creating html document) the documentation generator, which automatically generates documentation from source code comments.
* **Jdb:-** java debugger.
* Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ)
* Step 1: Install Java Development Kit (JDK)
* **Download and Install:**
* Go to the official [Oracle JDK download page](https://www.oracle.com/java/technologies/javase-downloads.html) or use [OpenJDK](https://jdk.java.net/).
* Download the latest LTS (Long Term Support) version (e.g., Java 17 or Java 21).
* Install it and note the installation path.
* **Set Environment Variables (if not automatically set):**
* **On Windows :**
* Open System Properties > Environment Variables.
* Add a new system variable:
  + **Name:** JAVA\_HOME
  + **Value:** C:\Program Files\Java\jdk-XX
  + Edit the Path variable and add: %JAVA\_HOME%\bin
  + **Check installation:**
  + bash
  + java -version
  + javac -version
* **Step 2: Install an IDE**
  + Option A: Eclipse IDE
  + Download & Install:
  + Download from: https://www.eclipse.org/downloads/
  + Choose “Eclipse IDE for Java Developers” and install it.
  + First-Time Setup:
  + Launch Eclipse.
  + Choose a workspace folder (your projects will be stored here).
  + Install any updates it recommends.
  + Create a new Java project via:
  + File > New > Java Project
  + Option B: IntelliJ IDEA
  + Download & Install:
  + Download from: <https://www.jetbrains.com/idea/download/>
  + Choose the Community Edition (free) or Ultimate (paid).
  + First-Time Setup:
  + Run IntelliJ and complete the first-time setup wizard.
  + Create a new project:
  + New Project > Java > Select SDK
  + (You can add your installed JDK here.)
  + Start coding in the **src** folder.
  + IDEs support Maven or Gradle for managing dependencies.
* **Java Program Structure (Packages, Classes, Methods)**
* **1. Document Section :-** In Documentation section one can write author name, definition of class or a program , description of a program and how this algorithm works and other details, and this is OPTIONAL part and this part write in (/\* \*/)multiline comment.
* **2. Package Statement :-** In java program first statement one can write is a package statement. This statement declares a package name and tells the compiler that the classes included here belongs to this package; the package statement is optional in any java program For ex:-package first; .
* **3. Import Statement:-** Once a package is declared one can write any number of import statements. import statement can be write after package statement and before the class definition. For ex:-import java.util.\*; Import first.\*; .
* **4. Class Declaration :-** In java program there may be more than one class. In this class definition statement we have to provide keyword class along with the class name. A class can include variable as well as methods.
* **5. Main Method :-** In a java program there may be more than one class one class can have main method. This is the essential part of java program. In main method one can create objects of different classes and with the use of there.
* **Lab Exercise:**
* **Install JDK and set up environment variables.**
* **Write a simple "Hello World" Java program.**
* **Compile and run the program using command-line tools (javac, java).**
* **Program :**

**public** **class** hello {

**public** **static** **void** main(String[] args)

{

System.***out***.println("hello world");

}

}

### 2. Data Types, Variables, and Operators

* **Theory :**
* **Primitive Data Types in Java (int, float, char, etc.)**
* The primitive data types are predefined data types, which always hold the value of the same data type, and the values of a primitive data type don't share the state with other primitive values.
* These data types are named by a reserved keyword in Java programming language.
* There are eight primitive data types supported by Java programming language:
* **byte :**
* The byte data type is an 8-bit signed two's complement integer. It ranges from -128 to127 (inclusive).
* This type of data type is useful to save memory in large arrays.
* We can also use byte instead of int to increase the limit of the code.
* Syntax : byte b = 5;
* **short:**
* The short data type is a 16-bit signed two's complement integer.
* It ranges from -32,768 to 32,767.short is used to save memory in large arrays.
* **Syntax :** short s = 2;
* **int :** The int data type is used to store the integer values not the fraction values.
* It is a 32-bit signed two's complement integer data type.
* It ranges from -2,147,483,648 to 2,147,483,647 that are more enough to store large number in your program.
* However for wider range of values use long.
* **float :**
* The float data type is a single-precision 32-bit IEEE 754 floating point.
* It ranges from 1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative).
* Use a float (instead of double) to save memory in large arrays.
* **double :**
* This data type is a double-precision 64-bit IEEE 754 floating point.
* It ranges from 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative).
* This data type is generally the default choice for decimal values.
* **char :**
* The char data type is a single 16-bit, unsigned Unicode character.
* It ranges from 0 to 65,535.
* They are not integral data type like int, short etc. i.e. the char data type can't hold the numeric values.
* **Boolean :**
* The boolean data type represents only two values: true and false and occupy is 1-bit in the memory.
* These values are keywords in Java and represent the two boolean states: on or off, yes or no.
* We use boolean data type for specifying conditional statements as if, while, do, for.
* **Variable Declaration and Initialization**
* A **variable** is a name that refers to a memory location used to store data.
* **Instance Variables (Non-static fields):**
* In object oriented programming, objects store their individual states in the "non-static fields" that is declared without the static keyword.
* **Class Variables (Static fields):** These are collectively related to a class and none of the object can claim them its soleproprietor.
* The variables defined with static keyword are shared by all objects.
* **Local Variables:** The variables defined in a method or block of code is called local variables.
* **Parameters:** Parameters or arguments are variables used in method declarations.
* **Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise**
* **Operators**
* + Additive operator (also used for String concatenation)
* - Subtraction operator
* \* Multiplication operator
* / Division operator
* % Remainder operator
* **Unary Operators**
* ++ Increment operator; increments a value by 1
* -- Decrement operator; decrements a value by 1
* ! Logical complement operator
* **Relational Operator**
* = = Equal to
* != Not equal to
* > Greater than
* > = Greater than or equal to
* < Less than
* < = Less than or equal to
* **Conditional Operators**
* && Conditional-AND
* || Conditional-OR
* ?: Ternary (shorthand for if-then-else statement)
* **Bitwise and Bit Shift Operators**
* ~ Unary bitwise complement
* << Signed left shift
* >> Signed right shift
* >>> Unsigned right shift
* & Bitwise AND
* ^ Bitwise exclusive OR
* | Bitwise inclusive OR
* **Type Conversion and Type Casting**
* (Widening / Implicit Casting)
* In Java, **Type Conversion** and **Type Casting** are processes used to convert one data type into another. Though often used interchangeably, they refer to slightly different concepts depending on whether the conversion is done **automatically** or **manually**.
* This is the automatic conversion of a smaller data type to a larger data type. It happens when there is no risk of data loss.
* **Also known as:** Widening Conversion or Implicit Casting
* When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
* The two types are compatible.
* The destination type is larger than the source type.
* Although the automatic type conversions are helpful, they will not fulfill all needs.
*  Done automatically by the compiler
*  No data loss
*  Safe and preferred when moving from smaller to larger data types
* Type Casting(Narrowing / Explicit Casting)
* This is the manual conversion of a larger data type to a smaller data type. It must be explicitly done by the programmer because it may result in data loss.
* **Also known as:** Narrowing Conversion or Explicit Casting
* **Ex.:**
* double d = 10.5;
* int num = (int) d;
* **Key Points:**
* Requires explicit syntax: (targetType) value
* Can lead to data loss (e.g., truncating decimal values)
* Must be used carefully
* **Lab Exercise:**
* **Write a program to demonstrate the use of different data types.**

**public** **class** DatatypeDemo {

**public** **static** **void** main(String[] args)

{

**byte** b= 5;

**short** s = 2;

**int** i = 10;

**float** f = 12.1f;

**double** d = 6677.60;

**char** ch = 'c';

**boolean** bl = **false**;

System.**out**.println("This byte : "+b);

System.**out**.println("This short : "+s);

System.**out**.println("This integer : "+i);

System.**out**.println("This float : "+f);

System.**out**.println("This double : "+d);

System.**out**.println("This char : "+ch);

System.**out**.println("This boolean : "+b);

}

}

* **Create a calculator using arithmetic and relational operators.**

package basicJava;

import java.util.Scanner;

public class SimpleCalculator {

public static void main(String[] args) {

int a,b;

double5 result;

char operator;

Scanner sc = new Scanner(System.in);

System.out.print("Enter A :");

a=sc.nextInt();

System.out.print("Enter B :");

b=sc.nextInt();

System.out.print("choose the operator (+,-,\*,/) :");

operator=sc.next().charAt(0);

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

switch(operator)

{

case '+':

result=a+b;

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

break;

case '-':

result=a-b;

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

break;

case '\*':

result=a\*b;

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

break;

case '/':

if(b!=0)

{

result=a/b;

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

}

else

{

System.out.println("This number can not divide by Zero");

}

break;

default:

System.out.println("Invalid operator..");

}

}

}

* **Demonstrate type casting (explicit and implicit).**

class Conversion {

public static void main(String args[]) {

byte b;

int i = 257;

double d = 323.142;

System.out.println("\nConversion of int to byte.");

b = (byte) i;

System.out.println("i and b " + i + " " + b);

System.out.println("\nConversion of double to int.");

i = (int) d;

System.out.println("d and i " + d + " " + i);

System.out.println("\nConversion of double to byte.");

b = (byte) d;

System.out.println("d and b " + d + " " + b);

}

}

Output :- Conversion of int to byte.

i and b 257 1

Conversion of double to int.

d and i 323.142 323

Conversion of double to byte.

d and b 323.142 67

### 3. Control Flow Statements

* **Theory:**
* If – Else Statement :
* A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program.
* A common programming construct that is based upon a sequence of nested ifs is the ifelse-if ladder.
* Syntax:
* if(<Boolean\_expression>)

<Statement\_or\_block>

elseif(<Boolean\_expression>)

<Statement\_or\_block>

elseif(<Boolean\_expression>)

<Statement\_or\_block>

else

<Statement\_or\_block>

* Switch Case Statements
* This is an easier implementation to the if-else statements.
* The keyword "switch" is followed by an expression that should evaluates to byte, short, char or int primitive data types , only. In a switch block there can be one or more labeled cases.
* Syntax:
* switch ( expression) {

case : <constant1>:

<statement\_or\_block>\* ;;;;;;;;;;;;;;;;;;;;;

[break;]

case : <constant2>:

<statement\_or\_block>\*

[break;]

default:

<statement\_or\_block>\*

[break;]

* Loops (For, While, Do-While)
* In Java, loops are control flow statements that repeatedly execute a block of code as long as a specified condition is true.
* Loops are fundamental for performing repetitive tasks efficiently.
* **For Loop :**
* **Use case**: When the number of iterations is known beforehand.
* Syntax :
* For(initialization;condition;iteration)
* {
* //body
* }
* When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, condition is evaluated. This must be a Boolean expression.
* **while Statements**
* The while loop is Java’s most fundamental looping statement.
* It repeats a statement or block while its controlling expression is true.
* While(condition)
* {
* //body
* }
* The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true.
* **Do-While statement**
* The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.
* **Break and Continue Keywords**
* In Java, the break statement has three uses.
* First, as you have seen, it terminates a statement sequence in a switch statement.
* Second, it can be used to exit a loop.
* Third, it can be used as a “civilized” form of go to.
* By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
* When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.
* **Continue Statement :**
* Sometimes it is useful to force an early iteration of a loop.
* That is, you might want to continue running the loop, but stop processing the remainder of the code in its body for this particular iteration.
* **Lab Exercise:**
* Write a program to find if a number is even or odd using an if-else statement.

Public class IfElase

{

Public Static void main(String[] args)

{

int i;

System.out.println(“enter number :”);

i=sc.nextInt();

if(i%2==0)

{

System.out.println(i+“number is even”);

}

Else

{

System.out.println(i+“number is odd”);

}

}

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

int day = 5;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday");

break;

case 4:

System.out.println("Thursday");

break;

case 5:

System.out.println("Friday");

break;

case 6:

System.out.println("Saturday");

break;

case 7:

System.out.println("Sunday");

break;

default:

System.out.println("Invalid entry");

break;

}

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

public class FibonacciSeries {

public static void main(String[] args) {

int n;

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of terms for the Fibonacci series: ");

i = scanner.nextInt();

int first = 0, second = 1;

System.out.println("Fibonacci Series up to " + n + " n:");

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

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

// Calculate next term

int next = first + second;

first = second;

second = next;

}

}

}

### 4. Classes and Objects

* **Theory:**
* Defining a Class and Object in Java
* A class is a blueprint or prototype from which objects are created.
* This section defines a class that models the state and behavior of a real-world object.
* Syntax :
* <Modifier>\*class<class\_name>
* <attribute\_declaration>
* <constructor\_declaration>
* <method\_declaration>
* **Object :**
* An object is a instance of class. We can also say that object is blue print of the class.
* We use the all member of the class. Using object name But we can not use private member of class.
* Classname objectname = new classname();
* Constructors and Overloading.
* **Constructor**
* It is a special member function whose task is to initialization the objects of it’s classes.
* A Constructor initializes an object immediately upon creation.
* **Constructor**
* It has the same name as the class in which it resides and is syntactically similar to a method.
* They do not have any return type.
* They are invoked automatically when object are created.
* **Types**
* 1. Default(which have no arguments, arg provided by java automatically)
* 2. Parameterized(which have arguments)
* 3. Copy constructor(which have object as arguments)
* **Constructor overloading**
* When there is more than one constructor in a single class there is called constructor overloading.
* Must have different number or types of parameters.
* Helps to create objects in different ways.
* Object Creation, Accessing Members of the Class
* An object is an instance of a class. When you create an object, Java allocates memory and allows you to access the class’s methods and variables.
* **Accessing Members of the Class**
* Once you create an object, you can access the members (variables and methods) of the class using the dot (.)operator.
* **Lab Exercise:**
* Create a class Student with attributes (name, age) and a method to display the details.

class student

{

String name;

int age;

void set(String n,int a){

name =n;

age = a;

}

Void display()

{

System.out.println(“student name :”+n);

System.out.println(“student age is :”+age);

}

public static void main(String[] args)

{

Student s1=new student();

S1.set(“bansari”,20);

S1.display();

}

}

* Create multiple constructors in a class and demonstrate constructor overloading.

class Student {

String name;

int age;

Student() {

name = "Unknown";

age = 0;

}

Student(String name) {

this.name = name;

age = 18;

}

Student(String name, int age) {

this.name = name;

this.age = age;

}

void display() {

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

}

}

public class Main {

public static void main(String[] args) {

// Using default constructor

Student s1 = new Student();

s1.display();

// Using constructor with one parameter

Student s2 = new Student(“bansari”);

S2.display();

Student s3 = new Student(“banari”,20);

S3.display();

}

}

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

### 5. Methods in Java

* **Theory**
* Defining Methods
* a method is a block of code that performs a specific task. Methods help you organize and reuse code efficiently.
* Predefined Methods :Already available in Java
* (e.g., System.out.println(),ath.sqrt()).
* User-defined Methods :
* Created by the programmer to perform specific tasks.
* Method Parameters and Return Types
* Method parameters are variables passed to a method when it is called. They allow methods to work with input values.
* **Parameters** - Inputs passed to a method.
* **Return Type** - The data type of the value returned by the method.
* **Void** - Used when no value is returned.
* Method Overloading
* Method Overloading in Java means defining multiple methods in the same class with the same name but different parameters.
* Improves code readability and reusability.
* Allows the same method name to perform different tasks based on arguments.
* Static Methods and Variables
* It is use to declare variable, methods and block
* Allocate the common memory
* static variable - Shared by all objects of a class
* static method - Can be called without creating an object
* Limitation - Can access only static members directly
* **Lab Exercise:**
* Write a program to find the maximum of three numbers using a method.
* class MaximumFinder {

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

int max = a;

if (b > max) {

max = b;

}

if (c > max) {

max = c;

}

return max;

}

}

public class Main {

public static void main(String[] args) {

MaximumFinder mf = new MaximumFinder();

// Test values

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

int max = mf.findMax(num1, num2, num3);

System.out.println("The maximum of " + num1 + ", " + num2 + ", and " + num3 + " is: " + max);

}

}

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

package basicJava;

public class MethodOverLoding {

void test()

{

System.out.println("test with no argument");

}

void test(int a)

{

System.out.println("text with 1 argument");

}

void test(int a,int b)

{

System.out.println("test with 2 argument");

}

public static void main(String[] args) {

MethodOverLoding m=new MethodOverLoding();

m.test();

m.test(10);

m.test(10,20);

}

}

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

public class StaticMethod {

static int a=10;

static int b;

static void math(int x)

{

System.out.println("Math Method called..");

System.out.println("x :"+x);

System.out.println("A :"+a);

System.out.println("B :"+b);

}

static

{

System.out.println("Static block initialized");

b=a\*4;

}

public static void main(String[] args) {

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

math(12);

}

}

### 6. Object-Oriented Programming (OOPs) Concepts

* **Theory:**
* **Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction**
* **Encapsulation :** Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
* Encapsulation is the process of wrapping data (variables) and methods (functions) together as a single unit (class), and restricting direct access to some of the object's components.
* Use private variables.
* Use public getter and setter methods to access them.
* **Inheritance :**
* Inheritance allows one class to inherit fields and methods from another class. It supports code reuse.
* The object of one class can aquire the properties of object of another class.
* Creating a new class from an existing class is called inheritance.
* **Polymorphism:**
* Polymorphism means "many forms". It allows one name to be used for different behaviors.
* One name multiple form
* **Type of polymorphism :**
* Method Overloading (compile-time)
* Method Overriding (runtime)
* **Abstraction**
* Abstraction means hiding the internal details and showing only the essential features.
* There are situations in which you will want to define a super class that declares the structure of a given abstraction without providing a complete implementation of every method.
* **Inheritance: Single, Multilevel, Hierarchical**
* **Single Inheritance: -** One class can acquire properties of one super class.
* **Multiple Inheritance: -** One class can acquire properties of more than one classes.
* **Multi-level Inheritance: -** In multilevel inheritance, the ladder of single inheritance increases.
* **Method Overriding and Dynamic Method Dispatch**
* **Method Overriding :** when there is same method prototypes in your both base class and derived class and if you called that method using the object of derived class than only derived class method will be called and say that method of derived class overrides the method of base class.
* Method Overriding is a feature that allows a subclass (child class) to provide a specific implementation of a method that is already defined in its superclass (parent class).
* The method must have the same name, return type, and parameters.
* The method must be inherited from the superclass.
* **Dynamic Method Dispatch :** Dynamic Method Dispatch is the mechanism by which a call to an overridden method is resolved at runtime rather than compile time. It is also known as runtime polymorphism.
* When there is more than one in a single class having the same name but with different number of argument and their datatype there it is called method overloading.
* **Lab Exercise:**
* Write a program demonstrating single inheritance.

class Person {

String name;

int age;

void displayPersonDetails() {

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

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

}

}

class Student extends Person {

String course;

void displayStudentDetails() {

System.out.println("Course: " + course);

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student();

s1.name = "Alice";

s1.age = 20;

s1.course = "Computer Science";

s1.displayPersonDetails();

s1.displayStudentDetails();

}

* Create a class hierarchy and demonstrate multilevel inheritance.

class Animal {

void eat() {

System.out.println("Animal eats food");

}

}

class Dog extends Animal {

void bark() {

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

}

}

class Puppy extends Dog {

void weep() {

System.out.println("Puppy weeps");

}

}

public class Main {

public static void main(String[] args) {

// Create an object of Puppy

Puppy p = new Puppy();

// Call methods from all levels of inheritance

p.eat(); // Inherited from Animal

p.bark(); // Inherited from Dog

p.weep(); // Defined in Puppy

}

}

* Implement method overriding to show polymorphism in action.

class Animal {

void makeSound() {

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

}

}

class Dog extends Animal {

void makeSound() {

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

}

}

class Cat extends Animal {

void makeSound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal a;

a = new Dog(); // a refers to Dog

a.makeSound(); // Output: Dog barks

a = new Cat(); // a refers to Cat

a.makeSound(); // Output: Cat meows

}

}

### 7. Constructors and Destructors

* **Theory:**
* **Constructor Types (Default, Parameterized)**
* A Constructor initializes an object immediately upon creation.
* It has the same name as the class in which it resides and is syntactically similar to a method.
* Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
* Constructors look a little strange because they have no return type, not even void.
* **Default :** There is always at least one constructor in every class.
* A default constructor is a constructor with no parameters.
* **Parameterized:**
* A parameterized constructor is a constructor that accepts arguments to initialize an object with specific values.
* **Copy Constructor (Emulated in Java)**
* A copy constructor is a special constructor used to create a new object as a copy of an existing object.
* Emulating a Copy Constructor in Java :
* class Student {

String name;

int age;

Student(String name, int age) {

this.name = name;

this.age = age;

}

Student(Student s) {

this.name = s.name;

this.age = s.age;

}

void display() {

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

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student("Alice", 20); // original object

Student s2 = new Student(s1); // copy using copy constructor

s1.display();

s2.display();

}

}

* **Constructor Overloading**
* Constructor overloading in Java means having more than one constructor in a class with different parameter lists. This allows an object to be initialized in multiple ways.
* Constructors must have the same name as the class
* Each overloaded constructor must differ in the number or type of parameters.
* It improves code flexibility and readability.

* **Object Life Cycle and Garbage Collection**
* Java is an object-oriented programming language that manages memory automatically through its Garbage Collector (GC).
* The object life cycle refers to the phases an object goes through from creation to destruction:

1. Object Creation

* An object is created using the new keyword.
* The constructor of the class is invoked.
* Memory is allocated on the heap.
* Example:  
  Student s = new Student();

2. Object Usage

* The object’s methods and properties are used as long as a reference exists.
* Example:  
  s.display();

3. Object Becomes Unreachable

* When no references to the object exist, it becomes unreachable and is eligible for garbage collection.
* Example:  
  s = null;

4. Garbage Collection

* The Java Garbage Collector automatically deallocates memory by destroying unreachable objects.
* This helps avoid memory leaks.
* **Lab Exercise:**
* Write a program to create and initialize an object using a parameterized constructor.

Class Student

{

String name;

int age;

}

Public class Student(String n, int a)

{

Name = n;

Age = a;

}

Void display()

{

System.out.println(“name is :”+n) ;

System.out.println(“age is :”+a);

}

Public static void main(String[] arg)

{

Student s=new Student(“bansi”,20);

s.display();

}

}

* Demonstrate constructor overloading by passing different types of parameters.

Class Box

{

double width,height,depth;

Box()

{

System.out.println("Default Constructor called ");

}

Box(double w,double h,double d)

{

System.out.println("Parameterized Constructor called");

width=w;

height=h;

depth=d;

}

Box(Box b)

{

System.out.println("Copy Contructor called ");

width=b.width;

height=b.height;

depth=b.depth;

}

void volume()

{

System.out.println("Volume is :"+(width\*height\*depth));

}

}

public class ConstructorDemo {

public static void main(String[] args) {

Box b1=new Box();

Box b2=new Box(10,10,10);

b2.volume();

Box b3=new Box(b2);

b3.volume();

}

}

8.Arrays and Strings

* **Theory:**
* **One-Dimensional and Multidimensional Arrays**
* An array is a container object that holds a fixed number of values of a single type.
* The length of an array is established when the array is created. After creation, its length is fixed.
* A 1D array is a linear structure that holds a fixed number of elements in a single row.
* Syntax: dataType arrayname[]=new datatype[size];
* **Multidimensional Arrays :**
* A 2D array is like a table with rows and columns.
* Syntax: dataType arrayname[][] = new datatype[][];
* Example: int twoD[][] = new int[4][5];
* This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int.
* **String Handling in Java: String Class, StringBuffer, StringBuilder**
* **Array of Objects**
* An array of objects is just like an array of primitive data types (int, float, etc.), but instead of storing values, it stores references to objects.
* This is useful when you want to manage a group of similar objects, like multiple students, employees, or books.
* This is the object just like an array of primitive and data types.

public Point[] createArray() {

Point[] p;

p = new Point[10];

for ( int i=0; i<=10;i++)

{

P[i]=new point(i,i+1);

}

return p;

* **String Methods (length, charAt, substring, etc.)**
* **Lab Exercise:**
* **Write a program to perform matrix addition and subtraction using 2D arrays.**

import java.util.Scanner;

public class TowDArray {

public static void main(String[] args) {

int a[][]=new int[3][3];

int i,j;

Scanner sc=new Scanner(System.in);

for(i=0;i<a.length;i++)

{

for(j=0;j<a.length;j++)

{

System.out.print("Emter "+i+" Row & " +j+ " column :");

a[i][j]=sc.nextInt();

}

}

System.out.println("2 D Array Is");

for(i=0;i<a.length;i++)

{

for(j=0;j<a.length;j++)

{

System.out.print(" A["+i+"]["+j+"] = " +a[i][j]);

}

System.out.println();

}

}

}

* **Create a program to reverse a string and check for palindromes.**
* import java.util.Scanner;
* public class PalindromeCheck {
* public static void main(String[] args) {
* Scanner scanner = new Scanner(System.in);
* System.out.print("Enter a string: ");
* String original = scanner.nextLine();
* String reversed = reverseString(original);
* System.out.println("Reversed string: " + reversed);
* if (original.equalsIgnoreCase(reversed)) {
* System.out.println("The string is a palindrome.");
* } else {
* System.out.println("The string is not a palindrome.");
* }
* scanner.close();
* }
* public static String reverseString(String str) {
* StringBuilder sb = new StringBuilder(str);
* return sb.reverse().toString();
* }
* }
* **Implement string comparison using equals() and compareTo() methods.**
* import java.util.Scanner;
* public class StringComparison {
* public static void main(String[] args) {
* Scanner scanner = new Scanner(System.in);
* System.out.print("Enter first string: ");
* String str1 = scanner.nextLine();
* System.out.print("Enter second string: ");
* String str2 = scanner.nextLine();
* if (str1.equals(str2)) {
* System.out.println("Using equals(): The strings are equal.");
* } else {
* System.out.println("Using equals(): The strings are not equal.");
* }
* int result = str1.compareTo(str2);
* if (result == 0) {
* System.out.println("Using compareTo(): The strings are equal.");
* } else if (result < 0) {
* System.out.println("Using compareTo(): First string is less than the second.");
* } else {
* System.out.println("Using compareTo(): First string is greater than the second.");
* }
* scanner.close();
* }
* }

### 9.Inheritance and Polymorphism

* **Theory:**
* **Inheritance Types and Benefits**
* Inheritance is one of the four fundamental of oops.
* It allows a class to inherit the properties and behaviors (fields and methods) of another class.
* The class that inherits is called the subclass (child).
* The class being inherited from is called the superclass (parent).
* Types of Inheritance in Java :
* Single Inheritance : One subclass inherits from one superclass.
* Multilevel Inheritance : A class inherits from a class that itself inherits from another class.
* Hierarchical inheritance : Multiple subclass inherits from a single superclass.

**Benefits of Inheritance**

* **Code Reusability :** Reuse existing code from the parent class.
* **Method Overriding :** Allows subclass to provide a specific implementation of a method already defined in its parent.
* **Extensibility :** Easy to add new features by extending existing classes.
* **Runtime Polymorphism :**Enables dynamic method dispatch using parent class references.
* **Organized Code :**Helps in better modularization and understanding of hierarchical relationships.
* **Method Overriding**
* Method Overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.
* Methods must have the **same name** but **different parameters** (type, number, or order).
* Return type **can be different**, but it **does not** contribute to overloading (it’s not enough on its own).
* Makes the program more readable.
* Method Overloading in Java is a feature that allows a class to have more than one method with the same name, as long as their parameter lists are different. This is a type of **compile-time polymorphism**.
* **Dynamic Binding (Run-Time Polymorphism)**
* Dynamic Binding, also known as Run-Time Polymorphism, is a concept in Java where the method that is to be executed is determined at runtime, not at compile time.
* The method to be executed is determined by the **object type**, not the reference type.
* Happens when a **subclass overrides a method** of its superclass.

|  |  |  |
| --- | --- | --- |
| Feature | Static Bindin(Compile-Time) | Dynamic Binding (Run-Time) |
| Method Type | Overloading | Overriding |
| Binding Time | Compile-Time | Run-Time |
| Performance | Faster | Slightly slower (due to lookup) |

* **Super Keyword and Method Hiding**
* Super keyword is used in derived class to access properties from your immediate base class, like variables, methods & constructor.
* **Lab Exercise:**
* **Write a program that demonstrates inheritance using extends keyword.**

package basicJava;

import java.util.Scanner;

class A

{

int a;

void getA()

{

Scanner sc=new Scanner(System.in);

System.out.print("Enter A :");

a=sc.nextInt();

}

void putA()

{

System.out.println("A is :"+a);

}

}

class B extends A

{

int b;

void getB()

{

Scanner sc=new Scanner(System.in);

System.out.print("enter B :");

b=sc.nextInt();

}

void putB()

{

System.out.println("B is :"+b);

}

}

public class InheritanceDemo {

public static void main(String[] args) {

B b1=new B();

b1.getA();

b1.getB();

b1.putA();

b1.putB();

}

}

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

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

void sound() {

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

}

}

class Cat extends Animal {

void sound() {

System.out.println("Cat meows");

}

}

public class TestPolymorphism {

public static void main(String[] args) {

Animal a;

a = new Dog();

a.sound();

a = new Cat();

a.sound();

}

}

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

class Animal {

Animal() {

System.out.println("Animal constructor called");

}

void display() {

System.out.println("Animal display method");

}

}

class Dog extends Animal {

Dog() {

super();

System.out.println("Dog constructor called");

}

void display() {

super.display();

System.out.println("Dog display method");

}

}

public class Super {

public static void main(String[] args) {

Dog d = new Dog();

d.display();

}

}

**10.Interfaces and Abstract Classes**

* **Theory:**
* **Abstract Classes and Methods**
* Abstract is declared with abstract keyword.
* You can’t create object of abstract class.
* You can create normal method in abstract class as well as abstract method using abstract keyword.
* You can’t define abstract method.
* There are situations in which you will want to define a super class that declares the structure of a given abstraction without providing a complete implementation of every method.
* **Interfaces: Multiple Inheritance in Java**
* Java does not support multiple inheritance with classes (to avoid ambiguity), but it does support multiple inheritance with interfaces.
* Java allows a class to **implement multiple interfaces.**
* This provides a way to **achieve multiple inheritance**.

interface A {

void methodA();

}

interface B {

void methodB();

}

class MyClass implements A, B {

public void methodA() {

System.out.println("Method A from interface A");

}

public void methodB() {

System.out.println("Method B from interface B");

}

}

* **Implementing Multiple Interfaces**
* A class to implement multiple interfaces, which is how it achieves multiple inheritance safely (unlike classes, where multiple inheritance can cause ambiguity).
* You **must implement** all methods from each interface.
* If two interfaces contain a method with the **same signature**, only **one implementation** is needed.
* Interfaces can also **extend other interfaces**.
* Interfaces can also **extend other interfaces**.
* Example :

interface Ifsc1 {

void method1();

}

interface Ifsc2 {

void method2();

}

class Interface implements Ifsc1, Ifsc2 {

public void method1() {

System.out.println("Method 1 from Ifsc1");

}

public void method2() {

System.out.println("Method 2 from Ifsc2");

}

}

* **Lab Exercise:**
* **Create an abstract class and implement its methods in a subclass.**

abstract class Abs1

{

void math1()

{

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

}

abstract void math2();

}

class Abs2 extends Abs1

{

void math2()

{

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

}

}

public class Abstractclass {

public static void main(String[] args) {

Abs2 a=new Abs2();

a.math1();

a.math2();

}

}

* **Write a program that implements multiple interfaces in a single class.**
* interface test{

public void callme();

public void document();

}

interface demo {

public void hello();

public void welcome();

}

class demo{

}

class testing extends demo2 implements test, demo {

public void hello() {

System.out.println("this os hello method");

}

public void welcome()

{ System.out.println("this is welcome method");

}

public void callme()

{ System.out.println("this is callme method");

}

public void document() {

System.out.println("this is document method");

}

}

public class multipleinterface {

public static void main(String args[]) {

testing t1=new testing();

t1.callme();

t1.document();

t1.welcome();

t1.hello();

}

}

* **Implement an interface for a real-world example, such as a payment gateway.**
* interface PaymentGateway {
* void makePayment(double amount);
* void refund(double amount);
* }
* class CreditCardPayment implements PaymentGateway {
* public void makePayment(double amount) {
* System.out.println("Paid" + amount + " using Credit Card.");
* }
* public void refund(double amount) {
* System.out.println("Refunded " + amount + " to Credit Card.");
* }
* }
* class PayPalPayment implements PaymentGateway {
* public void makePayment(double amount) {
* System.out.println("Paid" + amount + " using PayPal.");
* }
* public void refund(double amount) {
* System.out.println("Refunded " + amount + " to PayPal.");
* }
* }
* public class PaymentApp {
* public static void main(String[] args) {
* PaymentGateway payment;
* payment = new CreditCardPayment();
* payment.makePayment(1500);
* payment.refund(500);
* payment = new PayPalPayment();
* payment.makePayment(3000);
* payment.refund(1200);
* }
* }

**11.Packages and Access Modifier**

* **Theory:**
* Java Packages: Built-in and User-Defined Packages
* A **package** in Java is a **namespace** that organizes related classes and interfaces. Packages help:
* Avoid class name conflicts
* Improve modularity
* Provide access protection
* Organize code logically
* **Built-in Packages :**
* java.lang - Core classes like String, Math, Object
* java.util - Collections, Date, Scanner, etc.
* java.io - Input/output (File, Streams)
* java.net - Networking (Socket, URL)
* java.sql - Database connectivity (JDBC)
* **User-Defined Packages :**
* You can create your own packages to organize code.
* Save as MyClass.java inside folder "mypackage".
* **Access Modifiers: Private, Default, Protected, Public**
* Access modifiers in Java control the visibility (access level) of classes, methods, and variables. Java has four main access modifiers:
* **1. private**
* Accessible **only within the same class**.
* **Not accessible** from other classes
* **2. default (no modifier)**
* Accessible **within the same package only**.
* Not accessible from classes in other packages.
* **3.Protected**
* Accessible Within the **same package**.
* In **subclasses (even in different packages)** using inheritance.
* **4. Public**
* Accessible **from anywhere**.
* No restrictions.
* **Importing Packages and Classpath**
* The import keyword is used to access classes from other packages.
* In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the import statement:
* import pkg1[.pkg2].(classname|\*);
* Built-in packages: Provided by Java (e.g., java.util, java.io, java.lang)
* User-defined packages: Created by the programmer
* **Classpath :**
* The classpath is a parameter that tells the Java compiler and the Java Virtual Machine (JVM) where to look for .class files (compiled bytecode) and packages.
* To **locate classes and packages** used in a program.
* classpath tells the compiler and JVM **where to find those classes**.
* **Lab Exercise:**
* Create a user-defined package and import it into another program.
* **Mypackage.java**
* package mypackage;

public class Message {

public void show() {

System.out.println("user-defined package!");

}

}

* **main.java**
* import mypackage.Message;
* public class Main {
* public static void main(String[] args) {
* Message msg = new Message();
* msg.show();
* }
* }
* Demonstrate the use of different access modifiers within the same package andacross different packages.

package pack1;

public class A {

private int a = 10;

int b = 20; // default

protected int c = 30;

public int d = 40;

public void show() {

System.out.println("Private a: " + a);

System.out.println("Default b: " + b);

System.out.println("Protected c: " + c);

System.out.println("Public d: " + d);

}

}

package pack2;

import pack1.A;

public class B extends A {

public void display() {

// System.out.println(a); //private not accessible

// System.out.println(b); //default not accessible

System.out.println("Protected c: " + c);

System.out.println("Public d: " + d);

}

}

**12. Exception Handling**

* **Theory :**
* **Types of Exceptions: Checked and Unchecked**
* **Checked Exceptions :**
* Checked exceptions are checked at compile-time. The compiler ensures that you handle these exceptions, either with a try-catch block or by declaring them using the throws keyword.
* Example :-
* IOException
* SQLException
* FileNotFoundException
* ClassNotFoundException
* **Unchecked**
* Unchecked exceptions are **not checked at compile-time**. They occur at **runtime** and can crash your program if not handled.
* These exceptions are a subclass of RuntimeException.
* Example :
* ArithmeticException
* NullPointerException
* ArrayIndexOutOfBoundsException
* NumberFormatException
* **try, catch, finally, throw, throws**
* **try :** The try block contains code that might throw an exception.
* It’s used to wrap risky code (e.g., division by zero, file reading, etc.)
* **catch :**The catch block **handles the exception** that occurs in the try block.
* It must follow the try block.
* You can use multiple catch blocks for different exceptions.
* **finally :**The finally block **always executes**, whether an exception occurs or not.
* Commonly used to close resources (like files, database connections).
* **Throw :** The throw keyword is used to manually throw an exception.
* **Throws :** The throws keyword is used to **declare an exception** in the method signature.
* It tells the **caller of the method** that the method might throw an exception.
* **Custom Exception Classes**
* To make your code more meaningful and readable.
* To **clearly signal** specific error conditions relevant to your business logic.
* To **separate technical errors** from business-rule violations.
* Exception - for **checked** exceptions.
* RuntimeException - for **unchecked** exceptions.
* **Lab experience**
* **Implement multiple catch blocks for different types of exceptions.**
* Public static void main()
* {
* System.out.println(“Start code”);
* int a,b,c;
* Scanner sc=new Scanner(System.in);
* try{
* System.out.println(“Enter A :”);
* a=sc.nextInt();
* System.out.println(“Enter B :”);
* b=sc.nextInt();
* c=a/b;
* System.out.println(“division :”+c);
* Int arr[]={11,12,13,14,15};
* System.out.println(“Enter Index to print array element :”);
* Int index=sc.nextInt();
* System.out.println(“element ”+index + “is :”+arr[index]);
* }
* Catch(ArithmeticException e)
* {
* System.out.println(“exception caught :”+e);
* }catch(InputMismtchException e){
* System.out.println(“Exception cought :”+e);
* }catch(ArrayIndexBoundException e){
* System.out.println(“Exception Cought :”+e);
* }
* }
* **Write a program to demonstrate exception handling using try-catch-finally.**
* public class ExceptionExample {
* public static void main(String[] args) {
* int[] numbers = {10, 20, 30};
* try {
* System.out.println("Value: " + numbers[5]);
* } catch (ArrayIndexOutOfBoundsException e) {
* System.out.println("Exception caught: " + e.getMessage());
* } finally {
* System.out.println("Finally block executed: Program continues...");
* }
* System.out.println("Rest of the code executes normally.");
* }
* }
* **Create a custom exception class and use it in your program.**

**-** class ageexception extends Exception

{

int age; String msg="";

public ageexception()

{

}

public ageexception(String str) {

super(str);

}

public String toString() {

if(age60) {

msg="invalid age";

}

return msg;

}

}

public class userageexception {

public static void main(String args[]) {

userageexception uae=new userageexception();

uae.test();

} public void test() {

int i=25;

try {

if(i60) {

throw new ageexception();

}

else {

System.out.println("valid age");

}

} catch(ageexception e) {

System.out.println(e);

}

}

}

# 13. Multithreading

* **Theory:**
* **Introduction to Threads**
* The smallest individual unit of your program is called thread.
* You can create multiple thread but at a time only one thread can excecute.
* You can create Thread by two way :
* 1. By implementing runnable interface.
* 2. By extending Thread class.
* Thread is the feature of mostly languages including Java. Threads allow the program to perform multiple tasks simultaneously.
* Process speed can be increased by using threads because the thread can stop or suspend a specific running process and start or resume the suspended processes.
* **Creating Threads by Extending Thread Class or Implementing Runnable Interface**
* A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
* **Extending Thread:**
* public class MyThread extends Thread {
* public void run() {
* System.out.println("Thread executed!");
* } public static void main(String[] args) {
* Thread thread = new MyThread();
* thread.start();
* }
* }
* **Implementing the Runnable interface:**
* public class MyRunnable implements Runnable {
* public void run() {
* System.out.println("Thread executed!");
* } public static void main(String[] args) {
* Thread thread = new Thread(new MyRunnable());
* thread.start();
* **Thread Life Cycle**

**Dead**

**New**

**Blocked**

**Event Blocked**

**unblocked**

**Runnable**

**Running**

**Start()**

**Run() Completed**

**Sheduler**

* **New state:** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
* **Runnable (Ready-to-run) state:** A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
* **Running state:** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in the scheduler select a thread from runnable pool.
* **Dead state:** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
* Blocked - A thread can enter in this state because of waiting the resources that are hold by another thread.
* **Synchronization and Inter-thread Communication**
* When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
* Key to synchronization is the concept of the monitor (also called a semaphore).
* Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor.
* **This program uses a synchronized block.**
* class Callme {
* void call(String msg) {
* System.out.print("[" + msg);
* try {
* Thread.sleep(1000);
* } catch (InterruptedException e) { System.out.println("Interrupted");
* } System.out.println("]");
* }
* }
* class Caller implements Runnable {
* String msg;
* Callme target;
* Thread t;
* public Caller(Callme targ, String s) {
* target = targ;
* msg = s;
* t = new Thread(this);
* t.start();
* }
* public void run() {
* synchronized(target) {
* target.call(msg);
* }
* }
* class Synch1 {
* public static void main(String args[]) {
* Callme target = new Callme();
* Caller ob1 = new Caller(target, "Hello");
* Caller ob2 = new Caller(target, "Synchronized");
* Caller ob3 = new Caller(target, "World");
* try {
* ob1.t.join();
* ob2.t.join();
* ob3.t.join();
* }
* catch(InterruptedException e) {
* System.out.println("Interrupted");
* }
* }
* **Lab Exercise:**
* **Write a program to create and run multiple threads using the Thread class.**
* class MyThread extends Thread {
* private String threadName;
* public MyThread(String name) {
* this.threadName = name;
* }
* public void run() {
* for (int i = 1; i <= 5; i++) {
* System.out.println(threadName + " - Count: " + i);
* try {
* Thread.sleep(500);
* } catch (InterruptedException e) {
* System.out.println(threadName + " interrupted.");
* }
* }
* System.out.println(threadName + " finished.");
* }
* }
* public class MultiThreadExample {
* public static void main(String[] args) {
* MyThread t1 = new MyThread("Thread-1");
* MyThread t2 = new MyThread("Thread-2");
* MyThread t3 = new MyThread("Thread-3");
* t1.start();
* t2.start();
* t3.start();
* }
* }
* **Implement thread synchronization using synchronized blocks or methods.**
* class Counter {
* int count = 0;
* synchronized void increment() {
* count++;
* }
* }
* class MyThread extends Thread {
* Counter counter;
* MyThread(Counter c) {
* this.counter = c;
* }
* public void run() {
* for (int i = 0; i < 1000; i++) {
* counter.increment();
* }
* }
* }
* public class SyncExample {
* public static void main(String[] args) {
* Counter c = new Counter();
* MyThread t1 = new MyThread(c);
* MyThread t2 = new MyThread(c);
* t1.start();
* t2.start();
* try {
* t1.join();
* t2.join();
* } catch (Exception e) {
* System.out.println(e);
* }
* System.out.println("Final Count: " + c.count);
* }
* }
* **Use inter-thread communication methods like wait(), notify(), and notifyAll().**
* class SharedData {
* boolean available = false;
* synchronized void produce() {
* if (available) {
* try {
* wait();
* } catch (InterruptedException e) {
* e.printStackTrace();
* }
* }
* System.out.println("Producing data...");
* available = true;
* notify();
* }
* synchronized void consume() {
* if (!available) {
* try {
* wait();
* } catch (InterruptedException e) {
* e.printStackTrace();
* }
* }
* System.out.println("Consuming data...");
* available = false;
* notify();
* }
* }
* public class InterThreadExample {
* public static void main(String[] args) {
* SharedData data = new SharedData();
* Thread producer = new Thread(() -> {
* for (int i = 0; i < 5; i++) {
* data.produce();
* try { Thread.sleep(500); } catch (Exception e) {}
* }
* });
* Thread consumer = new Thread(() -> {
* for (int i = 0; i < 5; i++) {
* data.consume();
* try { Thread.sleep(500); } catch (Exception e) {}
* }
* });
* producer.start();
* consumer.start();
* }
* }

### 14. File Handling

**Theory:**

* **Introduction to File I/O in Java (java.io package)**
* File I/O (Input/Output) in Java refers to the process of reading data from and writing data to files using Java classes.
* Java provides the java.io package to handle file operations such as creating, reading, writing, and deleting files.
* Provides low-level access to files and data streams.
* Supports both character and byte data processing.
* Helps with various file operations (creation, deletion, checking existence, etc.)
* Common File I/O Classes in java.io :
* File, FileReader, FileWriter, FileInputStream,FileOutputStream, BufferedReader,BufferedWriter, PrintWriter.
* **FileReader and FileWriter Classes**
* FileReader is a class in the java.io package used to read character data from a file. It reads data character by character and is suitable for reading text files.
* It reads characters (not bytes).
* Automatically handles character encoding (based on the default charset).
* Throws IOException if the file is not found or any I/O error occurs.
* **FileWriter :**
* FileWriter is used to write character data to a file. It writes characters one by one or in bulk to a text file.
* It writes character data, making it suitable for writing text.
* Can be used to append data or overwrite files.
* Throws IOException in case of write failures.
* Simple to use for character-based text file operations.
* Part of the standard Java I/O library.
* Can be combined with BufferedReader or BufferedWriter for efficiency.
* **BufferedReader and BufferedWriter**
* **BufferedReader :**
* for reading characters efficiently.
* These classes are part of the java.io package and are used to buffer input and output to reduce the number of interactions with the disk.
* Reader stream classes are used to read characters from the file. The reader class provides a functionality that is available for all character input stream.
* The purpose of read text efficiently.
* Is work with character input.
* The common use of reading large text files.
* **BufferedWriter :**
* Used to **write text to a character-output stream**, buffering characters to provide efficient writing of characters, arrays, and strings.
* Without buffering: File is accessed every time a read/write is made.
* With buffering: Data is first stored in memory buffer, and then written or read in bulk **reducing I/O operations** and increasing performance.
* Is work with Character output.
* The common use of writing larg text files.
* The purpose of write text efficiently.
* **Serialization and Deserialization**
* Serialization is the process of converting a Java object into a byte stream (sequence of bytes) so it can be saved to a file, sent over a network, or stored in memory for later use.
* Save an object’s state to a file (persistence).
* Send an object over a network (like RMI, sockets).
* The class must implement the Serializable interface.
* **Deserializable :**
* Deserialization is the process of converting a **byte stream** back into a **Java object**.
* Deserialization is the reverse process where the **byte stream is converted back** into the original **Java object**. It recreates the object with the same state it had when serialized.
* **Lab Exercise:**
* **Write a program to read and write content to a file using FileReader and FileWriter.**
* import java.io.FileReader;
* import java.io.FileWriter;
* import java.io.IOException;
* public class FileReadWriteExample {
* public static void main(String[] args) {
* String data = "This is the content written using FileWriter.";
* try {
* FileWriter writer = new FileWriter("example.txt");
* writer.write(data);
* writer.close();
* System.out.println("Data written to the file successfully.");
* } catch (IOException e) {
* System.out.println("An error occurred while writing to the file.");
* e.printStackTrace();
* }
* // Reading from the file
* try {
* FileReader reader = new FileReader("example.txt");
* int character;
* System.out.println("Data read from the file:");
* while ((character = reader.read()) != -1) {
* System.out.print((char) character);
* }
* reader.close();
* } catch (IOException e) {
* System.out.println("An error occurred while reading the file.");
* e.printStackTrace();
* }
* }
* }
* **Implement a program that reads a file line by line using BufferedReader.**
* import java.io.BufferedReader;
* import java.io.FileReader;
* import java.io.IOException;
* public class BufferedReaderExample {
* public static void main(String[] args) {
* try {
* BufferedReader reader = new BufferedReader(new FileReader("example.txt"));
* String line;
* System.out.println("Reading file line by line:");
* while ((line = reader.readLine()) != null) {
* System.out.println(line);
* }
* reader.close();
* } catch (IOException e) {
* System.out.println("An error occurred while reading the file.");
* e.printStackTrace();
* }
* }
* }
* **Create a program that demonstrates object serialization and deserialization.**
* import java.io.\*;
* class Student implements Serializable {
* private static final long serialVersionUID = 1L;
* int id;
* String name;
* public Student(int id, String name) {
* this.id = id;
* this.name = name;
* }
* void display() {
* System.out.println("ID: " + id + ", Name: " + name);
* }
* }
* public class SerializationDemo {
* public static void main(String[] args) {
* try {
* Student s1 = new Student(101, "Amit");
* FileOutputStream fos = new FileOutputStream("student.ser");
* ObjectOutputStream oos = new ObjectOutputStream(fos);
* oos.writeObject(s1);
* oos.close();
* fos.close();
* System.out.println("Object has been serialized.");
* } catch (IOException e) {
* System.out.println("Serialization error:");
* e.printStackTrace();
* }
* try {
* FileInputStream fis = new FileInputStream("student.ser");
* ObjectInputStream ois = new ObjectInputStream(fis);
* Student s2 = (Student) ois.readObject();
* ois.close();
* fis.close();
* System.out.println("Object has been deserialized.");
* s2.display();
* } catch (IOException | ClassNotFoundException e) {
* System.out.println("Deserialization error:");
* e.printStackTrace();
* }
* }
* }

### 15. Collections Framework

### **Theory:**

### **Introduction to Collections Framework**

* The Java Collections Framework is a standardized architecture that provides a set of interfaces and classes to store and manipulate groups of data (objects).
* It is part of the java.util package and is used to perform various operations such as searching, sorting, insertion, deletion, and updating on data efficiently.
* Before the collections framework, Java used **arrays** and custom data structures, which had several limitations: Fixed size,Lack of utility methods (like sorting, searching),Inconsistent APIs
* **Unified architecture** for representing and manipulating collections.
* **Dynamic memory management** (unlike fixed-size arrays).
* **Polymorphic algorithms** for searching, sorting, and shuffling.

### **List, Set, Map, and Queue Interfaces**

* **List Interface:**
* The List interface is a subinterface of Collection that represents an ordered collection (also known as a sequence).
* Lists can contain duplicate elements, and elements are accessed using zero-based index.
* **Set Interface:**
* The Set interface is a Collection that does not allow duplicate elements. It models the mathematical set abstraction.
* **Map Interface:**
* The Map interface does not extend Collection.
* It represents a collection of key-value pairs, where each key is unique, but values can be duplicated.
* **Queue Interface:**
* The Queue interface is used to hold multiple elements prior to processing, typically in First-In-First-Out (FIFO) order.

### **ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap**

* **ArrayList:**
* ArrayList is a resizable array implementation of the List interface. It maintains insertion order and allows duplicate elements.
* **LinkedList:**
* LinkedList is a doubly linked list implementation of both List and Deque interfaces. It allows sequential access and is preferred for frequent insertions and deletions.
* **HashSet:**
* HashSet implements the Set interface and stores unique elements in no particular order.
* **TreeSet:**
* TreeSet is a SortedSet implementation based on Red-Black Tree. It stores unique elements in sorted (natural or custom) order.
* **HashMap:**
* HashMap implements the Map interface and stores data in the form of key-value pairs.
* **TreeMap:**
* TreeMap is a SortedMap implementation that stores key-value pairs in sorted order of keys.

### **Iterators and ListIterators**

* An **Iterator** is an interface in Java used to **traverse** elements of a **Collection (List, Set, etc.)**
* It is part of the java.util package and is considered a **universal iterator** because it can be used with **any Collection**.
* Used to **iterate forward** through a collection.
* Can **remove elements** during iteration.
* Cannot **modify or add** elements.
* **Fail-fast behavior**: Throws ConcurrentModificationException if the collection is modified structurally after the iterator is created.
* **ListIterator:**
* ListIterator is a subinterface of Iterator, specifically designed for the List interface. It allows bi-directional traversal (forward and backward) and modification of list elements.
* Only applicable to classes that implement List (e.g., ArrayList, LinkedList).
* Can **traverse forward and backward**.
* Can **add, remove, and replace** elements during iteration.
* Provides **index position** of elements while traversing.
* **Lab Exercise:**
* **Write a program that demonstrates the use of an ArrayList and LinkedList.**
* import java.util.ArrayList;
* import java.util.LinkedList;
* import java.util.List;
* public class ListExample {
* public static void main(String[] args) {
* List<String> arrayList = new ArrayList<>();
* arrayList.add("Apple");
* arrayList.add("Banana");
* arrayList.add("Cherry");
* System.out.println("ArrayList elements:");
* for (String fruit : arrayList) {
* System.out.println(fruit);
* }
* System.out.println("\nElement at index 1 in ArrayList: " + arrayList.get(1));
* arrayList.remove(1);
* System.out.println("ArrayList after removing index 1: " + arrayList);
* List<String> linkedList = new LinkedList<>();
* linkedList.add("Dog");
* linkedList.add("Elephant");
* linkedList.add("Fox");
* System.out.println("\nLinkedList elements:");
* for (String animal : linkedList) {
* System.out.println(animal);
* }
* System.out.println("\nElement at index 0 in LinkedList: " + linkedList.get(0));
* linkedList.remove("Dog");
* System.out.println("LinkedList after removing 'Dog': " + linkedList);
* }
* }
* **Implement a program using HashSet to remove duplicate elements froma list.**
* import java.util.ArrayList;
* import java.util.HashSet;
* import java.util.List;
* import java.util.Set;
* public class RemoveDuplicates {
* public static void main(String[] args) {
* List<String> namesList = new ArrayList<>();
* namesList.add("Apple");
* namesList.add("Banana");
* namesList.add("Apple");
* namesList.add("Mango");
* namesList.add("Banana");
* System.out.println("Original List with duplicates:");
* System.out.println(namesList);
* Set<String> uniqueSet = new HashSet<>(namesList);
* List<String> uniqueList = new ArrayList<>(uniqueSet);
* System.out.println("\nList after removing duplicates:");
* System.out.println(uniqueList);
* }
* }
* **Create a HashMap to store and retrieve key-value pairs.**
* import java.util.HashMap;
* import java.util.Map;
* public class HashMapExample {
* public static void main(String[] args) {
* HashMap<String, Integer> studentMarks = new HashMap<>();
* studentMarks.put("ABC", 85);
* studentMarks.put("XYZ", 78);
* studentMarks.put("PQR", 85); // Duplicate value is allowed
* System.out.println("Marks of XYZ: " + studentMarks.get("XYZ"));
* System.out.println("\nAll students and their marks:");
* for (Map.Entry<String, Integer> entry : studentMarks.entrySet()) {
* System.out.println(entry.getKey() + " => " + entry.getValue());
* }
* if (studentMarks.containsKey("ABC")) {
* System.out.println("\nAlice's marks are stored in the map.");
* }
* if (studentMarks.containsValue(78)) {
* System.out.println("Some student scored 78 marks.");
* }
* }
* }

### 16. Java Input/Output (I/O)

* **Theory:**
* **Streams in Java (InputStream, OutputStream)**
* A stream is the ordered sequence of data that uses a common characteristics shared by the entire input/output device Stream is serried of byte that used to travel data from one place to another. It is common link between program and other device There are two types of stream 1) Byte Streams 2) Character Streams
* The **Input stream** class defines the functionality that is available for all byte input streams ¬ The method provided by the inputstream class are. ¬ The three basic read methods are:
* int read()
* int read(byte[] buffer)
* int read(byte[] buffer, int offset, int length)
* **Output Stream**
* The output stream class defines the functionality that is available for all byte output streams
* The method provided by the outputstream class are.
* The three basic write methods are:
* void write(int c)
* void write(byte[] buffer)
* void write(byte[] buffer, int offset, int length)
* **Reading and Writing Data Using Streams**
* **Reader**
* Reader stream classes are used to read characters from the file.
* The reader class provides a functionality that is available for all character input stream.
* The Reader class provides a following types of methods
* The three basic read methods are:
* int read()
* int read(char[] cbuf) int read(char[] cbuf, int offset, int length)

**Writer :**

* The Writer stream classes are used to perform all output operation on files the Writer class is an abstract class which acts as base class for all the other writer stream classes.
* The writer class provides following types of methods.
* The basic write methods are:
* void write(int c)
* void write(char[] cbuf)
* void write(char[] cbuf, int offset, int length)
* void write(String string)
* void write(String string, int offset, int length)
* **Handling File I/O Operations**
* File I/O (Input/Output) in Java refers to reading data from or writing data to a file. Java provides a powerful I/O API in the java.io and java.nio packages to handle file operations like:
* Creating a file
* Reading data from a file
* Writing data to a file
* Appending data
* Deleting a file
* **File creation -** File
* **Read file -** FileReader, BufferedReader, Scanner
* **Write to file -** FileWriter, BufferedWriter, PrintWriter
* **Append to file -** FileWriter(true)
* **Delete file -** File.delete()
* **Lab Exercise:**
* **Write a program to read input from the console using Scanner.**
* public class ConsoleInputExample {
* public static void main(String[] args) {
* String name;
* int age;
* Scanner sc =new Scanner(System.in);
* System.out.println(“enter name :”);
* name=sc.nextLine();
* System.out.prinln(“enter age :”);
* age=sc.
* **Implement a file copy program using FileInputStream and FileOutputStream.**
* import java.io.FileInputStream;
* import java.io.FileOutputStream;
* import java.io.IOException;
* public class FileCopy {
* public static void main(String[] args) {
* // Source and destination file paths
* String sourceFile = "source.txt";
* String destinationFile = "copy.txt";
* try (
* FileInputStream fis = new FileInputStream(sourceFile);
* FileOutputStream fos = new FileOutputStream(destinationFile);
* ) {
* int byteData;
* // Read byte-by-byte and write to destination
* while ((byteData = fis.read()) != -1) {
* fos.write(byteData);
* }
* System.out.println("File copied successfully!");
* } catch (IOException e) {
* System.out.println("An error occurred while copying the file: " + e.getMessage());
* }
* }
* }
* **Create a program that reads from one file and writes the content.**
* import java.io.FileInputStream;
* import java.io.FileOutputStream;
* import java.io.IOException;
* public class FileCopyExample {
* public static void main(String[] args) {
* String sourceFile = "input.txt";
* String destinationFile = "output.txt";
* try {
* FileInputStream inputStream = new FileInputStream(sourceFile);
* FileOutputStream outputStream = new FileOutputStream(destinationFile);
* int data;
* while ((data = inputStream.read()) != -1) {
* outputStream.write(data);
* }
* inputStream.close();
* outputStream.close();
* System.out.println("File copied successfully.");
* } catch (IOException e) {
* System.out.println("An error occurred: " + e.getMessage());
* }
* }
* }