**1. Introduction to Java**

**1.1) History of Java**

Java is a high-level, object-oriented programming language that was developed by James Gosling and his team at Sun Microsystems in 1991. Initially called "Oak," it was designed for embedded systems but was later renamed Java and officially released in 1995. Its key strength was the ability to run the same code on different platforms, summarized by the slogan "Write Once, Run Anywhere." Java quickly gained popularity, especially for web-based applications. In 1998, Java 2 introduced significant improvements, including the Collections Framework and a more robust graphical user interface with Swing. In 2006, Sun made Java open source, and in 2010, Oracle acquired Sun Microsystems, taking over Java's development. Java 8, released in 2014, marked a major evolution with the introduction of lambda expressions and the Stream API. Since 2017, Oracle has followed a six-month release cycle, steadily adding modern features such as modules, records, pattern matching, and virtual threads. Today, Java remains a foundational technology in enterprise software, Android app development, and large-scale systems.

**1.2) Features of Java (Platform Independent, Object-Oriented, etc.)**

1. **Platform Independent**: Java code is compiled into bytecode, which can run on any system with a Java Virtual Machine (JVM).
2. **Object-Oriented**: Everything in Java is treated as an object, making the code modular, reusable, and easier to maintain.
3. **Simple**: Java has a clean and easy-to-understand syntax, reducing complexity compared to languages like C++.
4. **Secure**: Java provides a secure runtime environment with features like bytecode verification and a security manager.
5. **Robust**: It has strong memory management, exception handling, and type checking to reduce errors.
6. **Multithreaded**: Java supports multithreading, allowing programs to perform multiple tasks simultaneously.
7. **Portable**: Java programs can run on any device or OS that supports the JVM, making it highly portable.
8. **High Performance**: Though interpreted, Java's performance is enhanced through Just-In-Time (JIT) compilation.
9. **Distributed**: Java has built-in networking capabilities, making it easy to build distributed applications.
10. **Dynamic**: Java supports dynamic loading of classes, making it adaptable to evolving environments.

**1.3) Understanding JVM, JRE, and JDK**

1. **JVM (Java Virtual Machine)**:  
   It runs Java bytecode and provides platform independence. JVM is responsible for executing Java programs, managing memory, and providing security.
2. **JRE (Java Runtime Environment)**:  
   It includes the JVM and core libraries needed to run Java applications. It does **not** include development tools like compilers.
3. **JDK (Java Development Kit)**:  
   It is a full-featured kit for Java development. It includes the JRE **plus** tools like the compiler (**javac**), debugger, and other utilities to develop, compile, and run Java programs.

**1.4) Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ).**

**1. Install Java Development Kit (JDK)**

**Step 1:**  
Go to the official website: <https://www.oracle.com/java/technologies/javase-downloads.html>

**Step 2:**  
Download the latest JDK for your OS (Windows/Mac/Linux).

**Step 3:**  
Run the installer and follow the instructions.

**Step 4:**  
Set the environment variable (for Windows):

* Go to **System Properties → Advanced → Environment Variables**
* Add a new **System Variable**:
  + Name: JAVA\_HOME
  + Value: path to your JDK folder (e.g., C:\Program Files\Java\jdk-21)
* Add ;%JAVA\_HOME%\bin to the **Path** variable.

**Step 5:**  
Verify the installation:  
Open Command Prompt and run:

java -version

javac -version

**2. Install an Eclipse IDE**

**Step 1:**  
Go to https://www.eclipse.org/downloads/

**Step 2:**  
Download the **Eclipse IDE for Java Developers**.

**Step 3:**  
Extract and run the installer → Select **Eclipse IDE for Java Developers** → Install.

**Step 4:**  
Launch Eclipse and select a workspace folder.

**1.5) Java Program Structure (Packages, Classes, Methods).**

**1. Package**

* A package is a namespace that organizes classes.
* Declared at the top of the file:
* package mypackage;

**2. Class**

* The basic building block of a Java program.
* Every Java program must have at least one class:
* public class MyClass {
* // Fields, constructors, methods
* }

**3. Method**

* A block of code that performs an action.
* The main method is the entry point of the program:
* public static void main(String[] args) {
* System.out.println("Hello, World!");
* }

**Example:**

package mypackage;

public class HelloWorld {

public static void main(String[] args) {

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

}

}

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

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

* **byte (1 byte)** – Stores small whole numbers from -128 to 127. Useful when working with streams or saving memory in large arrays.  
  *Example:* byte b = 100;
* **short (2 bytes)** – Stores whole numbers from -32,768 to 32,767. Used when memory savings are important and int is too large.  
  *Example:* short s = 1000;
* **int (4 bytes)** – The most commonly used integer type. Stores numbers from -2,147,483,648 to 2,147,483,647.  
  *Example:* int x = 12345;
* **long (8 bytes)** – Used for very large whole numbers. Requires an L at the end of the number.  
  *Example:* long l = 1234567890L;
* **float (4 bytes)** – Stores decimal numbers with 6–7 digits of precision. Requires an f at the end. Suitable for performance-sensitive applications.  
  *Example:* float f = 3.14f;
* **double (8 bytes)** – Default for decimal numbers, with 15–16 digits of precision. More accurate than float.  
  *Example:* double d = 2.718281828;
* **char (2 bytes)** – Stores a single character using Unicode (0 to 65,535). Can store letters, digits, or symbols.  
  *Example:* char c = 'A';
* **boolean (1 bit)** – Represents only two values: true or false. Used for conditional logic.  
  *Example:* boolean flag = true;

**2.2) Variable Declaration and Initialization.**

**What is a Variable?**

A **variable** in Java is a container that holds data during program execution. Each variable must be **declared with a type**, and can optionally be **initialized** with a value.

**Declaration**

Declaring a variable means telling the compiler the **name** and **type** of data it will store.

int number; // declares an integer variable

**Initialization**

Initializing means **assigning a value** to the variable:

number = 10; // assigns the value 10 to the variable

**Declaration + Initialization**

Most commonly, both steps are done together:

int number = 10;

**2.3) Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise.**

**1.Arithmetic Operators**

Used for basic mathematical operations:  
+ (add), - (subtract), \* (multiply), / (divide), % (modulus)  
**Example:** int sum = a + b;

**2**.**Relational (Comparison) Operators**

Used to compare two values:  
== (equal), != (not equal), > (greater), < (less), >=, <=  
**Example:** if (a > b) { ... }

**3.Logical Operators**

Used to combine boolean expressions:  
&& (AND), || (OR), ! (NOT)  
**Example:** if (a > 0 && b > 0) { ... }

**4.Assignment Operators**

Used to assign values:  
=, +=, -=, \*=, /=, %=  
**Example:** x += 5; (means x = x + 5)

**5.** **Unary Operators**

Work with a single operand:  
+, -, ++ (increment), -- (decrement), ! (logical NOT)  
**Example:** i++; increases i by 1

**6.** **Bitwise Operators**

Operate on bits (binary level):  
& (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift)  
**Example:** a & b; performs bitwise AND

**2.4) Type Conversion and Type Casting.**

**Type Conversion (Implicit/Widening)**

Java automatically converts a **smaller data type to a larger one** when no data loss happens. Safe and automatic.

**Example:**

int a = 10;

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

**Type Casting (Explicit/Narrowing)**

Manually converting a **larger data type to a smaller one**, which may lose data. Requires casting syntax.

**Example:**

double x = 10.5;

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

**3) Control Flow Statements**

**3.1) If-Else Statements.**

Used for **decision-making**. The program executes different blocks of code depending on whether a condition is true or false.

**Example (C/C++/Java):**

int age = 20;

if (age >= 18) {

printf("You are an adult.\n");

} else {

printf("You are a minor.\n");

}

* **If** checks the condition.
* **Else** runs if the condition is false.
* Can also use **else if** for multiple conditions.

**3.2) Switch Case Statements.**

Used when you want to compare a variable with multiple constant values.

**Example:**

int day = 3;

switch(day) {

case 1:

printf("Monday");

break;

case 2:

printf("Tuesday");

break;

case 3:

printf("Wednesday");

break;

default:

printf("Invalid day");

}

* Each case checks for a match.
* break exits the switch; without it, execution "falls through" to the next case.
* default runs if no cases match.

**3.3) Loops (For, While, Do-While).**

Loops repeat a block of code while a condition is true.

**a.** **For Loop**

Used when the number of iterations is **known**.

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

printf("%d\n", i);

}

**b. While Loop**

Used when the condition must be checked **before** running the loop.

int i = 1;

while (i <= 5) {

printf("%d\n", i);

i++;

}

**c**. **Do-While Loop**

Executes the loop body **at least once**, then checks the condition.

int i = 1;

do {

printf("%d\n", i);

i++;

} while (i <= 5);

**3.4) Break and Continue Keywords.**

**a. Break**

Exits from a loop or switch immediately.

for (int i = 1; i <= 10; i++) {

if (i == 5)

break; // exits the loop

printf("%d\n", i);

}

**b. Continue**

Skips the current iteration and continues with the next one.

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

if (i == 3)

continue; // skips 3

printf("%d\n", i);

}

**4.) Classes and Objects**

**4.1) Defining a Class and Object in Java.**

A **class** is a blueprint for creating objects. It defines variables (fields) and methods (functions).

An **object** is an instance of a class, created in memory.

**Example:**

class Car {

String color;

void drive() {

System.out.println("The car is driving.");

}

}

Creating an object:

Car myCar = new Car(); // Object creation

myCar.color = "Red"; // Accessing member variable

myCar.drive(); // Accessing method

**4.2) Constructors and Overloading.**

A **constructor** is a special method used to initialize objects.

It has the same name as the class and no return type.

Java provides a **default constructor** if none is defined.

**Example:**

class Car {

String color;

// Constructor

Car(String c) {

color = c;

}

**Constructor Overloading:**

**Creating multiple constructors with different parameters.**

class Car {

String color;

int speed;

Car(String c) {

color = c;

}

Car(String c, int s) {

color = c;

speed = s;

}

}

**4.3) Object Creation, Accessing Members of the Class.**

Car car1 = new Car("Blue", 120); // Object creation using constructor

System.out.println(car1.color); // Access variable

car1.drive(); // Call method

**4.4) this Keyword.**

Refers to the **current object**.

Used to **resolve naming conflicts** or to **pass the current object**.

**Example:**

class Car {

String color;

Car(String color) {

this.color = color; // distinguishes class variable from parameter

}

}

**5.) Methods in Java**

**5.1) Defining Methods.**

A **method** is a block of code that performs a specific task. It is declared inside a class.

Syntax:

returnType methodName(parameters) {

// method body

}

Example:

void sayHello() {

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

}

**5.2) Method Parameters and Return Types.**

**Parameters**: Allow you to pass data into a method.

**Return Type**: Specifies what the method returns (e.g., int, String, void if nothing).

**Example:**

int add(int a, int b) {

return a + b;

}

**Calling the method:**

int result = add(5, 3); // result = 8

**5.3) Method Overloading.**

**Method overloading** means defining multiple methods with the **same name** but **different parameters**.

Helps improve code readability and reusability.

**Example:**

class Calculator {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

**5.4) Static Methods and Variables.**

**Static methods/variables** belong to the class, **not to objects**.

Can be accessed **without creating an object**.

Commonly used for utility or shared data.

**Example:**

class MathUtils {

static int count = 0; // static variable

static int square(int x) { // static method

return x \* x;

}

}

**Accessing statics:**

int result = MathUtils.square(5); // returns 25

System.out.println(MathUtils.count); // access static variable

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

**6.1) Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction.**

**Encapsulation**

Wrapping data (variables) and methods into a single unit (class).

Fields are usually private, and accessed via getters and setters.

class Student {

private int age;

public void setAge(int a) { age = a; }

public int getAge() { return age; }

}

**Inheritance**

A class (child/subclass) inherits fields and methods from another class (parent/superclass).

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

void bark() {

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

}

}

**Polymorphism**

**One name, many forms.** A method behaves differently based on the context.

Achieved through **overloading** and **overriding**.

// Overloading - Compile-time

int add(int a, int b) { return a + b; }

double add(double a, double b) { return a + b; }

// Overriding - Runtime (in child class)

**Abstraction**

Hiding complex implementation details and showing only essential features.

Achieved via **abstract classes** or **interfaces**.

abstract class Shape {

abstract void draw(); // no body

}

**6.2) Inheritance: Single, Multilevel, Hierarchical.**

**Single Inheritance**

One subclass inherits from one superclass.

class A { }

class B extends A { }

**Multilevel Inheritance**

A class inherits from a derived class.

class A { }

class B extends A { }

class C extends B { }

**Hierarchical Inheritance**

Multiple classes inherit from the same parent.

class A { }

class B extends A { }

class C extends A { }

**Note:** Java does **not support multiple inheritance** with classes (only via interfaces).

**6.3) Method Overriding and Dynamic Method Dispatch.**

**Method Overriding**

Subclass provides a specific implementation of a method already defined in its parent class.

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

@Override

void sound() {

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

}

}

**Dynamic Method Dispatch**

Runtime decision about which overridden method to call.

Done through a **parent class reference** pointing to a **child class object**.

Animal a = new Dog(); // parent ref → child object

a.sound(); // calls Dog's sound() (runtime decision)

**7.) Constructors and Destructors.**

**7.1) Constructor Types (Default, Parameterized).**

**Constructor**

Special method used to **initialize objects**.

Same name as the class, **no return type**.

**a) Default Constructor**

No parameters.

Provided automatically if no constructor is defined.

class Student {

Student() {

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

}

}

**b) Parameterized Constructor**

Takes arguments to initialize fields.

class Student {

String name;

Student(String n) {

name = n;

}

}

**7.2) Copy Constructor (Emulated in Java).**

Java does **not** have a built-in copy constructor like C++.

You can **manually define one** to copy values from another object.

class Student {

String name;

Student(String n) {

name = n;

}

// Copy constructor

Student(Student s) {

this.name = s.name;

}

}

**7.3) Constructor Overloading.**

Multiple constructors with different parameter lists.

Enables object initialization in multiple ways.

class Student {

String name;

int age;

Student() { } // Default

Student(String n) { name = n; } // One parameter

Student(String n, int a) { name = n; age = a; } // Two parameters

}

**7.4) Object Life Cycle and Garbage Collection.**

**Object Life Cycle:**

1. **Object Created** → using new keyword (calls constructor)
2. **Object Used** → data is accessed/manipulated
3. **Object Unreferenced** → no reference pointing to it
4. **Garbage Collected** → memory reclaimed by JVM

**Garbage Collection in Java**

Java **automatically destroys** unreferenced objects using the **Garbage Collector**.

You can define a finalize() method (not recommended in modern Java).

protected void finalize() {

System.out.println("Object is garbage collected");

}

**8.) Arrays and Strings.**

**8.1) One-Dimensional and Multidimensional Arrays.**

**One-Dimensional Array**

A list of elements of the same type.

int[] nums = {10, 20, 30};

System.out.println(nums[1]); // Output: 20

**Multidimensional Array**

An array of arrays (e.g., matrix).

int[][] matrix = {

{1, 2},

{3, 4}

};

System.out.println(matrix[1][0]); // Output: 3

**8.2) String Handling in Java: String Class, StringBuffer, StringBuilder.**

**String Class**

**Immutable** (cannot be changed after creation).

String s = "Hello";

String s2 = s + " World"; // creates a new string

**StringBuffer**

**Mutable**, thread-safe (synchronized).

Slower but safe for multithreading.

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World");

System.out.println(sb); // Hello World

**StringBuilder**

**Mutable**, not thread-safe.

Faster, used in single-threaded programs.

StringBuilder sb = new StringBuilder("Hi");

sb.append(" Java");

System.out.println(sb); // Hi Java

**8.3) Array of Objects.**

where each element is an object.

class Student {

String name;

Student(String n) {

name = n;

}

}

Student[] students = new Student[2];

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

students[1] = new Student("Bob");

System.out.println(students[1].name); // Bob

**8.4) String Methods (length, charAt, substring, etc.).**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| length() | Returns length of string | "Hello".length() → 5 |
| charAt(int index) | Returns character at index | "Hello".charAt(1) → 'e' |
| substring() | Returns part of string | "Hello".substring(1, 4) → "ell" |
| toUpperCase() | Converts to uppercase | "java".toUpperCase() → "JAVA" |
| toLowerCase() | Converts to lowercase | "JAVA".toLowerCase() → "java" |
| equals() | Compares values | "hi".equals("hi") → true |
| equalsIgnoreCase() | Case-insensitive comparison | "Hi".equalsIgnoreCase("hi") → true |
| indexOf() | Finds first index of a character or string | "Hello".indexOf('l') → 2 |
| replace() | Replaces characters or substrings | "Java".replace('a', 'o') → "Jovo" |
| trim() | Removes leading/trailing whitespace | " Java ".trim() → "Java" |

**9.) Inheritance and Polymorphism.**

**9.1) Inheritance Types and Benefits.**

**Inheritance** allows a class (called a subclass or child class) to inherit properties and behaviors (fields and methods) from another class (called a superclass or parent class).  
This promotes **code reusability**, helps organize code in a **hierarchical** manner, and makes it easier to maintain and extend.

Java supports these types of inheritance:

**Single Inheritance**: A subclass inherits from one superclass.

**Multilevel Inheritance**: A class inherits from a subclass, which in turn inherits from another superclass (like A → B → C).

**Hierarchical Inheritance**: Multiple subclasses inherit from the same superclass.

Java does **not support multiple inheritance** with classes (to avoid ambiguity), but you can achieve it using interfaces.

**9.2) Method Overriding.**

**Method overriding** occurs when a subclass provides its own version of a method that is already defined in its superclass.  
The method in the subclass must have the **same name**, **same return type**, and **same parameters** as in the superclass.

It allows Java to achieve **runtime polymorphism**, where the method that gets executed depends on the actual object (not the reference type).

Example:

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

void sound() {

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

}

}

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

**Dynamic binding** means that the method call is resolved at **runtime** instead of compile-time.  
It happens when a superclass reference refers to a subclass object, and the overridden method is called based on the actual object.

Example:

Animal a = new Dog(); // Parent reference to child object

a.sound(); // Calls Dog’s sound() at runtime

This is called **runtime polymorphism**, and it allows flexible and dynamic method execution.

**9.4) Super Keyword and Method Hiding.**

The **super keyword** in Java is used to refer to the **immediate parent class**. You can use it to:

Call a method from the superclass that is overridden in the subclass.

Access a field in the superclass that is hidden by a subclass field.

Call the parent class constructor.

Example:

class Animal {

void display() {

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

}

}

class Dog extends Animal {

void display() {

super.display(); // calls Animal's display()

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

}

}

**Method hiding** occurs when a subclass defines a **static method** with the same name as a static method in the superclass. In this case, it does not override the method—it hides it. Static methods are resolved at **compile time**, so the version called depends on the reference type, not the object.

**10.) Interfaces and Abstract Classes.**

**10.1) Abstract Classes and Methods.**

An **abstract class** is a class that **cannot be instantiated** (i.e., you can’t create objects of it).  
It can have **abstract methods** (methods without a body) as well as **concrete methods** (with implementation).

Use abstract classes when:

You want to **provide common behavior** but also require subclasses to implement specific methods.

Example:

abstract class Animal {

abstract void sound(); // abstract method

void eat() {

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

}

}

class Dog extends Animal {

void sound() {

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

}

}

**10.2) Interfaces: Multiple Inheritance in Java.**

An **interface** is a fully abstract type used to define **only method signatures** (no implementation), although Java 8+ allows default and static methods.

In Java, **interfaces enable multiple inheritance** because a class can **implement multiple interfaces**.  
This overcomes Java's restriction on multiple inheritance with classes.

Example:

interface Flyable {

void fly();

}

interface Swimmable {

void swim();

}

class Bird implements Flyable, Swimmable {

public void fly() {

System.out.println("Bird flies");

}

public void swim() {

System.out.println("Bird swims");

}

}

**10.3) Implementing Multiple Interfaces.**

A class can **implement multiple interfaces** by separating them with commas. Each method from the interfaces must be **implemented in the class**.

Key points:

Interfaces support **multiple inheritance**.

A class can extend one class but implement **many interfaces**.

Interface methods are **public and abstract by default**.

Example:

interface A {

void show();

}

interface B {

void display();

}

class MyClass implements A, B {

public void show() {

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

}

public void display() {

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

}

}

**11.) Packages and Access Modifiers.**

**11.1) Java Packages: Built-in and User-Defined Packages.**

**Packages** in Java are used to group related classes, interfaces, and sub-packages. They help in:

**Organizing code**

**Avoiding class name conflicts**

**Providing access protection**

Built-in Packages

Java provides many built-in packages like:

* java.util (collections, dates)
* java.io (input/output)
* java.lang (core classes like String, Math, etc.)
* java.net (networking)

Example:

import java.util.Scanner;

User-Defined Packages

You can create your own packages to organize your code.

Example:

// File: mypackage/MyClass.java

package mypackage;

public class MyClass {

public void display() {

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

}

}

To use this:

import mypackage.MyClass;

class Test {

public static void main(String[] args) {

MyClass obj = new MyClass();

obj.display();

}

}

**11.2) Access Modifiers: Private, Default, Protected, Public.**

Access modifiers control **visibility** of classes, variables, methods, and constructors.

Private

Accessible **only within the same class**.

Not inherited.

default (no modifier)

Accessible **within the same package**.

Not accessible outside the package.

protected

Accessible in the **same package** and in **subclasses** (even if in different packages).

public

Accessible **from anywhere** (any class, any package).

Example:

public class Example {

private int a; // only within this class

int b; // default: within package

protected int c; // package + subclass

public int d; // accessible everywhere

}

**11.3) Importing Packages and Classpath.**

Importing Packages

Use the import keyword to use classes from another package.

import java.util.Scanner;

You can also import all classes from a package:

import java.util.\*;

Classpath

The **classpath** is the location where the Java compiler and JVM look for .class files.

You can set it using:

Environment variable

Command line using -cp or -classpath

Example:

javac -cp . mypackage/MyClass.java

**12.) Exception Handling.**

**12.1) Types of Exceptions: Checked and Unchecked.**

**Checked Exceptions**

Checked at **compile-time**.

Must be **handled** using try-catch or declared using throws.

Examples: IOException, SQLException, FileNotFoundException.

import java.io.\*;

public class Demo {

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

FileReader fr = new FileReader("file.txt"); // Checked exception

}

}

**Unchecked Exceptions**

Occur at **runtime**.

Not mandatory to handle.

Examples: NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException.

int a = 5 / 0; // ArithmeticException

**12.2) try, catch, finally, throw, throws.**

try

Block of code that might throw an exception.

catch

Block that handles the exception.

try {

int a = 10 / 0;

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero");

}

finally

Block that always executes (used for cleanup like closing files).

finally {

System.out.println("Always runs");

}

throw

Used to **manually throw** an exception.

throw new ArithmeticException("Manual error");

throws

Declares that a method may throw an exception.

public void readFile() throws IOException {

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

}

**12.3) Custom Exception Classes.**

You can create your own exceptions by **extending the Exception class** (for checked) or **RuntimeException** (for unchecked).

class MyException extends Exception {

public MyException(String msg) {

super(msg);

}

}

public class Test {

static void check(int age) throws MyException {

if (age < 18)

throw new MyException("Underage not allowed");

else

System.out.println("Welcome!");

}

public static void main(String[] args) {

try {

check(15);

} catch (MyException e) {

System.out.println(e.getMessage());

}

}

}

**13.) Multithreading.**

**13.1) Introduction to Threads.**

A **thread** is a lightweight subprocess — a separate path of execution.  
Java supports **multithreading**, allowing multiple threads to run concurrently to improve performance (especially in I/O or CPU-bound tasks).

Each thread in Java runs independently and shares memory with other threads.

**13.2) Creating Threads by Extending Thread Class or Implementing Runnable Interface.**

**By Extending the Thread Class**

You create a new class that extends Thread and override the run() method.

class MyThread extends Thread {

public void run() {

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

}

}

public class Test {

public static void main(String[] args) {

MyThread t = new MyThread();

t.start(); // Starts the thread and calls run()

}

}

**By Implementing Runnable Interface**

You create a class that implements Runnable and pass it to a Thread object.

class MyRunnable implements Runnable {

public void run() {

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

}

}

public class Test {

public static void main(String[] args) {

Thread t = new Thread(new MyRunnable());

t.start();

}

}

**13.3) Thread Life Cycle.**

A thread goes through several states:

1. **New**: Thread object created but not started.
2. **Runnable**: After calling start(), it's ready to run.
3. **Running**: The thread is actually executing.
4. **Blocked/Waiting**: The thread is paused, waiting for resources or signal.
5. **Terminated**: The thread completes its execution.

You can control thread execution using methods like:

start(): Starts the thread.

sleep(ms): Pauses thread for milliseconds.

join(): Waits for another thread to finish.

yield(): Temporarily pauses current thread to let others run.

**13.4) Synchronization and Inter-thread Communication.**

**Synchronization**

Used to prevent **race conditions** when multiple threads access shared data.

class Counter {

int count = 0;

synchronized void increment() {

count++;

}

}

The synchronized keyword ensures **only one thread** can execute the method/block at a time.

**Inter-thread Communication**

Java provides wait(), notify(), and notifyAll() methods (used inside synchronized blocks) to allow threads to **cooperate** and **communicate**.

Example scenario: Producer-Consumer Problem

synchronized(obj) {

obj.wait(); // thread waits

obj.notify(); // wakes up waiting thread

}

**14.) File Handling.**

**14.1) Introduction to File I/O in Java (java.io package).**

Java provides the **java.io package** for file input and output (I/O) operations such as reading from and writing to files.  
File I/O is essential for storing data permanently and processing files.

Common classes:

File

FileReader / FileWriter

BufferedReader / BufferedWriter

ObjectOutputStream / ObjectInputStream (for serialization)

**14.2) FileReader and FileWriter Classes.**

FileReader

Used for reading character files (text files).

import java.io.\*;

FileReader

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

int i;

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

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

}

fr.close();

FileWriter

Used for writing characters to a file.

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

fw.write("Hello File!");

fw.close();

**14.3) BufferedReader and BufferedWriter.**

Buffered classes improve efficiency by using an internal buffer, reading or writing large chunks of data at once.

BufferedReader

Used for reading text line by line.

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

String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

br.close();

BufferedWriter

Used to write text efficiently.

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

bw.write("Buffered write example");

bw.newLine(); // Adds a new line

bw.close();

**14.4) Serialization and Deserialization.**

**Serialization**: Converting a Java object into a byte stream to save it to a file or send over a network.

**Deserialization**: Reconstructing the object from the byte stream.

Steps:

1. The class must implement Serializable interface.
2. Use ObjectOutputStream to serialize.
3. Use ObjectInputStream to deserialize.

import java.io.\*;

class Student implements Serializable {

int id;

String name;

Student(int id, String name) {

this.id = id; this.name = name;

}

}

public class Demo {

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

Student s = new Student(1, "John");

// Serialization

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

oos.writeObject(s);

oos.close();

// Deserialization

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

Student s2 = (Student) ois.readObject();

System.out.println(s2.id + " " + s2.name);

ois.close();

}

}

**15. Collections Framework**

**15.1) Introduction to Collections Framework.**

The **Java Collections Framework (JCF)** is a **set of classes and interfaces** in java.util used to **store, retrieve, and manipulate data efficiently**.

Replaces older array-based structures.

Provides **dynamic data structures** like lists, sets, queues, and maps.

Supports **algorithms**, such as sorting and searching.

Includes **interfaces**, **implementations (classes)**, and **utility classes (e.g., Collections)**.

**15.2) List, Set, Map, and Queue Interfaces.**

List

* Ordered collection.
* Allows **duplicates**.
* Elements accessed by **index**.
* Examples: ArrayList, LinkedList

Set

* **No duplicate** elements.
* Unordered (except TreeSet).
* Examples: HashSet, TreeSet

Map

* **Key-value pairs**.
* Keys are unique; values can duplicate.
* Examples: HashMap, TreeMap

Queue

* Used for **FIFO** (First-In-First-Out) data handling.
* Examples: PriorityQueue, LinkedList (as Queue)

**15.3) ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap.**

ArrayList

* Dynamic array; fast **random access**.
* Slower insert/delete (shifting needed).

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

list.add("Apple");

list.add("Banana");

LinkedList

* Elements stored in **nodes**; fast insert/delete.
* Slower access (no indexing).

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

list.add("A");

list.add("B");

HashSet

* No duplicates, no order.
* Uses a **hash table** for fast lookups.

HashSet<Integer> set = new HashSet<>();

set.add(10);

set.add(20);

TreeSet

* No duplicates, **sorted** in natural order.
* Slower than HashSet.

TreeSet<String> ts = new TreeSet<>();

ts.add("Z");

ts.add("A");

HashMap

* Unordered, key-value pairs.
* Fast lookup by key.

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

map.put(1, "One");

map.put(2, "Two");

TreeMap

* Key-value pairs in **sorted order by keys**.

TreeMap<String, Integer> tm = new TreeMap<>();

tm.put("B", 2);

tm.put("A", 1);

**15.4) Iterators and ListIterators.**

Iterator

* Used to **traverse** any Collection.
* Methods: hasNext(), next(), remove()

Iterator<String> it = list.iterator();

while (it.hasNext()) {

System.out.println(it.next());

}

ListIterator

* Used only with **List**.
* Supports **forward and backward** traversal.
* Methods: hasPrevious(), previous(), add(), set()

ListIterator<String> lit = list.listIterator();

while (lit.hasNext()) {

System.out.println(lit.next());

}

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

**16.1) Streams in Java (InputStream, OutputStream).**

Java I/O is **stream-based**, meaning it reads and writes data as a continuous flow (stream) of bytes or characters.

Types of Streams:

* **Byte Streams**: Handle raw binary data.
  + Classes: InputStream, OutputStream and their subclasses (FileInputStream, FileOutputStream)
* **Character Streams**: Handle text (Unicode).
  + Classes: Reader, Writer and their subclasses (FileReader, FileWriter)

Core Classes:

* InputStream: Abstract class for reading bytes.
* OutputStream: Abstract class for writing bytes.

**16.2) Reading and Writing Data Using Streams.**

**Reading using** FileInputStream **(Byte Stream)**

import java.io.\*;

FileInputStream fis = new FileInputStream("data.txt");

int ch;

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

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

}

fis.close();

**Writing using** FileOutputStream

FileOutputStream fos = new FileOutputStream("output.txt");

String data = "Hello, Java I/O!";

fos.write(data.getBytes());

fos.close();

**Using Character Streams:** FileReader **and** FileWriter

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

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

int i;

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

fw.write(i);

}

fr.close();

fw.close();

**16.3) Handling File I/O Operations.**

Buffered Streams (Efficient I/O)

* BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream improve performance by reducing disk access.

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

String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

br.close();

File Class (For File Info and Management)

* The File class (from java.io) provides methods to **create**, **delete**, and **check file properties**.

File f = new File("example.txt");

if (f.exists()) {

System.out.println("File exists: " + f.getName());

}