Module 5

– Core Java

Introduction to Java

Theory:

1 . History of Java

### Ans. History of Java

Java is a powerful, versatile, and widely-used programming language developed by **Sun Microsystems** in the mid-1990s. Below is a chronological overview of Java's history:

**1. Origins and Creation (1991–1995)**

* **1991**: The Java project was initiated by **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** at Sun Microsystems. Originally, it was called **"Oak"**, named after an oak tree outside Gosling’s office.
* The goal was to develop a platform-independent language for embedded systems (like set-top boxes).
* Later, it was renamed **Java**, inspired by Java coffee (a type of coffee from Indonesia), to avoid trademark issues.

**2. Public Release (1995)**

* **May 23, 1995**: Java was officially launched by Sun Microsystems.
* **"Write Once, Run Anywhere"** became its core principle—programs written in Java can run on any device that has the **Java Virtual Machine (JVM)**.
* Java 1.0 included the **Java Development Kit (JDK)** and was used mainly for building interactive websites (applets).

**3. Growth and Popularity (Late 1990s - 2000s)**

* Java became popular for **enterprise applications** due to its security, stability, and cross-platform capabilities.
* Key editions were introduced:
  + **Java SE (Standard Edition)** – for desktop applications.
  + **Java EE (Enterprise Edition)** – for web and enterprise apps.
  + **Java ME (Micro Edition)** – for mobile and embedded systems.

**4. Acquisition by Oracle (2010)**

* In **2010**, **Oracle Corporation acquired Sun Microsystems**, and with it, Java.
* Oracle continued to develop Java and introduced licensing and structural changes, while also focusing on regular version releases.

**5. Modern Java (2011–Present)**

* Starting from **Java 7 (2011)** to **Java 17 (2021)** and beyond, Java introduced many modern programming features:
  + **Lambda expressions**, **Streams**, **Modules**, **Pattern Matching**, and more.
* A new **six-month release cycle** was adopted from Java 9 (2017), making Java evolve faster.
* **Java 21 (2023)** is a **Long-Term Support (LTS)** version with enhanced performance and modern features.

**Key Contributors**

* **James Gosling** is known as the “**Father of Java**.”
* Other key figures include **Bill Joy**, **Arthur van Hoff**, and **Jonathan Payne**.

**Conclusion**

Java has evolved from a simple language for embedded systems to one of the most widely used programming languages in the world. It powers web servers, Android apps, enterprise software, and more, maintaining its position through continuous innovation and strong community support.

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

Ans. Here’s a list of the **key features of Java** that make it one of the most popular programming languages in the world:

### **1. Platform Independent**

* Java programs are **compiled into bytecode** by the Java compiler.
* This bytecode can be run on any system with a **Java Virtual Machine (JVM)**.
* Slogan: **“Write Once, Run Anywhere.”**

### **2. Object-Oriented**

* Everything in Java is treated as an **object**, promoting modular and reusable code.
* Core principles: **Encapsulation, Inheritance, Polymorphism, and Abstraction**.

### **3. Simple and Easy to Learn**

* Java has a **clean and easy-to-read syntax**, similar to C/C++ but without complex features like pointers and operator overloading.

### **4. Secure**

* Java provides a **secure execution environment**:
  + Bytecode verification
  + Runtime security checks
  + No direct memory access (no pointers)

### **5. Robust**

* Java emphasizes **early error checking** and **runtime exception handling**.
* Features like **automatic garbage collection**, **strong memory management**, and **type checking** ensure robustness.

### **6. Architecture-Neutral**

* Java bytecode is not tied to any processor architecture.
* Ensures **long-term compatibility** with different hardware/software platforms.

### **7. Portable**

* Java programs can be moved easily from one system to another.
* Portability is achieved through:
  + Platform-independent bytecode
  + Standardized data sizes

### **8. Multithreaded**

* Java supports **multithreading**, allowing concurrent execution of two or more threads (small units of a process).
* Built-in classes like Thread and Runnable.

### **9. High Performance (Relatively)**

* Although slower than compiled languages like C++, Java uses **Just-In-Time (JIT) compiler** to improve performance.
* JVM optimizes frequently used code paths.

### **10. Distributed**

* Java supports building **distributed applications** using features like **Remote Method Invocation (RMI)** and **Enterprise JavaBeans (EJB)**.
* It integrates well with web-based technologies and services.

### **11. Dynamic**

* Java supports **runtime dynamic linking** of classes.
* Applications can dynamically load and execute new code.

3. Understanding JVM, JRE, and JDK

Ans. Understanding the differences and roles of **JVM**, **JRE**, and **JDK** is essential for any Java programmer. Here's a clear explanation:

### ****1. JVM (Java Virtual Machine)****

### ➤ ****What is it?****

* JVM is the **engine** that runs Java bytecode.
* It provides a **runtime environment** where Java programs are executed.

### ➤ ****Key Functions:****

* Converts **bytecode into machine code** (using Just-In-Time compiler).
* Handles **memory management**, **garbage collection**, and **security**.
* Ensures **platform independence** by allowing the same bytecode to run on different systems.

### ➤ ****Important Note:****

* JVM is **platform-specific** (different for Windows, Linux, etc.), but bytecode is **platform-independent**.

### **2. JRE (Java Runtime Environment)**

### ➤ ****What is it?****

* JRE is a **package** that contains everything needed to **run** Java applications.

### ➤ ****Includes:****

* JVM
* Java class libraries (core classes like java.lang, java.util, etc.)
* Supporting files (configuration files, property files)

### ➤ ****Purpose:****

* Meant for **end users** who want to **run Java applications**, but **not develop** them.

### **3. JDK (Java Development Kit)**

### ➤ ****What is it?****

* JDK is a **software development kit** used to **develop Java programs**.

### ➤ ****Includes:****

* JRE (which includes JVM)
* **Development tools** like:
  + javac (Java compiler)
  + java (launcher)
  + javadoc, jdb, and other utilities

### ➤ ****Purpose:****

* Used by **Java developers** to **write, compile, debug, and run** Java programs.

### Simple Analogy:

Imagine you're baking a cake:

* **JDK** is the full **kitchen** (you can bake, taste, and clean).
* **JRE** is just the **oven** (you can bake but not prepare ingredients).
* **JVM** is the **heat inside the oven** that actually bakes the cake (executes the bytecode).

4. Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ) o Java Program Structure (Packages, Classes, Methods)

Ans. Here’s a complete guide to:

### **Setting Up the Java Environment & IDE (e.g., Eclipse, IntelliJ)**

### ****Step 1: Install Java JDK****

1. **Download JDK:**
   * Visit
   * Download the latest version (LTS version recommended, e.g., Java 17 or 21).
2. **Install JDK:**
   * Follow the installation instructions for your OS (Windows/Mac/Linux).
   * After installation, set the **JAVA\_HOME** environment variable:
     + Example (Windows):
     + JAVA\_HOME=C:\Program Files\Java\jdk-21
     + PATH=%JAVA\_HOME%\bin
3. **Verify Installation:**  
   Open terminal/command prompt:
4. java -version
5. javac -version

### ****Step 2: Choose and Install an IDE****

### **Eclipse IDE**

* Download from
* Install the **Eclipse IDE for Java Developers**
* Key Features:
  + Free, lightweight, plugin-based
  + Good for beginners

### **IntelliJ IDEA**

* Download from
* Choose **Community Edition** (Free) or **Ultimate** (Paid)
* Features:
  + Smart code completion
  + Built-in version control
  + Fast and developer-friendly UI

### ****Step 3: Create Your First Java Project in IDE****

### In Eclipse:

* File → New → Java Project
* Name your project, e.g., MyFirstProject
* Create a new class: Right-click → New → Class
* Write and run your Java code using the green **Run** button

### In IntelliJ:

* File → New Project → Java
* Choose JDK
* Create a Java class: Right-click on src → New → Java Class
* Run with the **Run** icon.

## 

2. Data Types, Variables, and Operators

Theory:

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

Ans. Here’s a complete guide to the **Primitive Data Types in Java**:

### **What are Primitive Data Types?**

Java provides **8 primitive data types** that are the **building blocks of data manipulation**. They represent **simple values**, not objects, and are **stored directly in memory**.

## **Java’s 8 Primitive Data Types:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Data Type** | **Size** | **Default Value** | **Example** | **Description** | | --- | --- | --- | --- | --- | | byte | 1 byte | 0 | byte a = 100; | Smallest integer (-128 to 127) | | short | 2 bytes | 0 | short b = 1000; | Larger than byte (-32,768 to 32,767) | | int | 4 bytes | 0 | int c = 50000; | Most commonly used integer type | | long | 8 bytes | 0L | long d = 100000L; | Very large integer | | float | 4 bytes | 0.0f | float e = 3.14f; | Decimal numbers with less precision | | double | 8 bytes | 0.0d | double f = 3.14159; | Decimal numbers with higher precision | | char | 2 bytes | '\u0000' | char g = 'A'; | Stores a single 16-bit Unicode character | | boolean | 1 bit | false | boolean h = true; | Stores true or false only | |

### ****Quick Notes:****

* **int** is the default for whole numbers.
* **double** is the default for decimal values.
* **char** stores characters like 'A', '9', or symbols ('$', '%').
* **boolean** is useful for conditional checks (if, while, etc.).
* You must suffix L for long and f for float literals.
  1. **Variable Declaration and Initialization o Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise**

Ans. Here’s a clear explanation of:

### **2. Variable Declaration and Initialization in Java**

### ****Declaration****

To declare a variable, you specify its **data type** and **name**:

int age; // Declaration only

### ****Initialization****

To initialize, assign a value:

age = 25; // Initialization

### ****Declaration + Initialization Together:****

int age = 25; // Combined declaration & initialization

### **Operators in Java**

Operators perform operations on variables and values. Java includes:

### ****1. Arithmetic Operators****

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | | **Operator** | **Description** | **Example** | | --- | --- | --- | | + | Addition | a + b | | - | Subtraction | a - b | | \* | Multiplication | a \* b | | / | Division | a / b | | % | Modulus (remainder) | a % b | | |

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Operator** | **Description** | **Example** | | --- | --- | --- | | == | Equal to | a == b | | != | Not equal to | a != b | | > | Greater than | a > b | | < | Less than | a < b | | >= | Greater than or equal to | a >= b | | <= | Less than or equal to | a <= b | |

### ****3. Logical Operators****

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | | **Operator** | **Description** | **Example** | | --- | --- | --- | | && | Logical AND | a > 10 && b < 20 | | ` |  | ` | | ! | Logical NOT | !(a > b) | |  |  | | |

### ****4. Assignment Operators****

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Operator** | **Description** | **Example** | | --- | --- | --- | | = | Assign | a = 10 | | += | Add and assign | a += 5 → a = a + 5 | | -= | Subtract and assign | a -= 5 | | \*= | Multiply and assign | a \*= 5 | | /= | Divide and assign | a /= 5 | | %= | Modulus and assign | a %= 5 | |

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Operator** | **Description** | **Example** | | --- | --- | --- | | + | Unary plus (positive sign) | +a | | - | Unary minus (negative sign) | -a | | ++ | Increment | a++ or ++a | | -- | Decrement | a-- or --a | | ! | Logical NOT | !true → false | |

### ****6. Bitwise Operators**** (Work at bit level)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Operator** | **Description** | **Example** | | --- | --- | --- | | & | Bitwise AND | a & b | | ` | ` | Bitwise OR | | ^ | Bitwise XOR | a ^ b | | ~ | Bitwise Complement | ~a | | << | Left shift | a << 2 | | >> | Right shift | a >> 2 | |

* 1. Type Conversion and Type Casting

Ans. Sure! Here's a complete explanation of **Type Conversion and Type Casting** in Java:

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

### ➤ What is it?

* Automatically **converts a smaller data type into a larger one**.
* Done by the **compiler** without any explicit instruction.
* No data loss occurs.

### ➤ Allowed Widening Conversions:

byte → short → int → long → float → double

↑ ↑

char char

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

### ➤ What is it?

* Converts a **larger data type into a smaller one**.
* Must be **explicitly specified** by the programmer.
* Might result in **data loss**.

### **Comparison: Implicit vs Explicit**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Feature** | **Type Conversion (Widening)** | **Type Casting (Narrowing)** | | --- | --- | --- | | Conversion type | Smaller → Larger | Larger → Smaller | | Automatic? | Yes | No (needs manual cast) | | Data loss possible? | No | Yes | | Example | int → double | double → int | |

Control Flow Statements

Theory:

* + 1. **If-Else Statements**

Ans. Here’s a clear and simple explanation of **If-Else Statements in Java**:

### ****1. If-Else Statements in Java****

The if-else statement is a **conditional control structure** used to make **decisions** in a program based on whether a condition is true or false.

### ****if-else-if Ladder****

Used when there are **multiple conditions** to check.

int marks = 85;

if (marks >= 90) {

System.out.println("Grade A");

} else if (marks >= 75) {

System.out.println("Grade B");

} else if (marks >= 60) {

System.out.println("Grade C");

} else {

System.out.println("Fail");

}

### ****Nested if Statement****

You can place one if inside another for **more complex logic**.

int age = 25;

boolean hasLicense = true;

if (age >= 18) {

if (hasLicense) {

System.out.println("You can drive");

} else {

System.out.println("You need a license");

}

} else {

System.out.println("You are too young to drive");

}

* + 1. Switch Case Statements

Ans. Here’s a clear explanation of **Switch Case Statements in Java**:

### ****Switch Case Statements in Java****

The switch statement is used as an **alternative to multiple if-else-if conditions**, especially when checking a **single variable against multiple constant values**.

### 

### ****Without break (Fall-through behavior):****

int num = 2;

switch (num) {

case 1:

System.out.println("One");

case 2:

System.out.println("Two");

case 3:

System.out.println("Three");

}

### ****Supported Types in**** switch****:****

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Java Version** | **Supported Types** | | --- | --- | | Java 1.0+ | byte, short, char, int | | Java 5+ | enum | | Java 7+ | String | | Java 14+ | switch expressions with arrow syntax (→) | |

### ****Use**** switch ****When:****

* You're comparing **one variable** to **many constant values**
* You want cleaner code than multiple if-else-if blocks
  + 1. **Loops (For, While, Do-While)**

Ans. Here's a complete and simple explanation of **Loops in Java**:

### ****3. Loops in Java (for, while, do-while)****

Loops allow you to **execute a block of code multiple times**, based on a condition.

### ****1.**** for ****Loop****

### ****2.**** while ****Loop****

### ****3.**** do-while ****Loop****

### **Loop Comparison Table:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Loop Type** | **Checks Condition** | **Runs At Least Once?** | **Best Use Case** | | --- | --- | --- | --- | | for | Before loop | No | When number of iterations is known | | while | Before loop | No | When condition-based looping is needed | | do-while | After loop | Yes | When loop should run at least once | |

* + 1. **Break and Continue Keywords**

Ans. Here's a clear explanation of the **break** and **continue** keywords in Java:

### ****4.**** break ****and**** continue ****Keywords in Java****

Both are **jump statements** used to control the flow of loops and switch statements.

### ****1.**** break ****Keyword****

### ****Purpose:****

* Used to **exit a loop or switch** statement **immediately**, even if the loop condition is still true.

### ****2****. continue Keyword

### ****Purpose:****

* Skips the **current iteration** of a loop and jumps to the **next iteration**.

### ****Using**** break ****and**** continue ****in**** while ****or**** do-while ****loops****

Classes and Objects

Theory:

* + 1. **Defining a Class and Object in Java**

Ans. Here’s a clear explanation of how to **define a class and create objects** in Java:

### ****1. Defining a Class and Object in Java****

### ****What is a Class?****

A **class** is a **blueprint** or template for creating **objects**.  
It defines **properties** (variables) and **behaviors** (methods) of an object.

### ****What is an Object?****

An **object** is an **instance** of a class.  
You create objects to **use the fields and methods** defined in a class.

* + 1. **Constructors and Overloading o Object Creation, Accessing Members of the Class**

Ans. Here's a complete explanation of **constructors, constructor overloading, object creation, and accessing members** in Java:

### ****2. Constructors and Overloading in Java****

### ****What is a Constructor?****

A **constructor** is a **special method** used to **initialize objects** of a class. It runs **automatically** when an object is created.

### ****Features of Constructors:****

* Name **must be the same** as the class.
* **No return type**, not even void.
* Automatically called when an object is created.
* Used to **set default or initial values**.

### ****Types of Constructors:****

1. **Default Constructor** – No parameters
2. **Parameterized Constructor** – Takes parameters to set initial values

### ****Constructor Overloading****

When a class has **multiple constructors with different parameters**, it’s called **constructor overloading**.

### ****Creating an Object:****

ClassName objectName = new ClassName();

### ****Accessing Fields & Methods:****

objectName.fieldName = value; // Setting a field

objectName.methodName(); // Calling a method

* + 1. **this Keyword**

### Ans. this Keyword in Java — Explained Simply

The **this keyword** in Java is a **reference to the current object** — the object whose method or constructor is being executed.

### Common Uses of this Keyword

### 1. ****Referring to Instance Variables****

Use this when local variables (e.g., constructor parameters) have the **same name** as instance variables.

### 2. ****Calling Another Constructor (Constructor Chaining)****

Use this() to call another constructor in the **same class**.

### 3. ****Passing Current Object as a Parameter****

You can pass this to another method or class that needs a reference to the current object.

### 4. ****Returning the Current Object****

Helpful in **method chaining**.

**Methods in Java**

**Theory:**

* 1. **Defining Methods**

### Ans. Defining Methods in Java

In Java, a **method** is a block of code that performs a specific task. It can be **called (invoked)** whenever needed to reuse code, avoid repetition, and organize programs better.

### Method Components

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Component** | **Description** | | --- | --- | | returnType | Data type returned (int, void, String, etc.) | | methodName | Any valid identifier (e.g., add, display) | | parameters | Optional input values (int a, int b) | | return | Sends a result back to the caller (if needed) | |

### Types of Methods

1. **Without return & without parameters**
2. **parameters & without return**
3. **With parameters & with return**
4. **Without parameters & with return**

### Good to Know:

* You can call methods from other methods within the same class.
* Methods improve **reusability**, **modularity**, and **readability**.
  1. **Method Parameters and Return Types**

### Ans. 2. Method Parameters and Return Types in Java

In Java, **methods** can accept **parameters** (inputs) and may return a **value**. Understanding these concepts is key to writing reusable, modular programs.

### ****What Are Method Parameters?****

**Parameters** are variables listed inside the method’s parentheses.  
They **allow data to be passed** into the method from the caller.

* String name is a **parameter** of type String.
* The method uses it to print a personalized message.

### ****What Are Return Types?****

The **return type** defines the **type of value** a method sends back to the caller.  
If a method **doesn’t return anything**, use void.

* 1. **Method Overloading**

### Ans. 3. Method Overloading in Java

**Method Overloading** means **defining multiple methods with the same name** in a class but with **different parameters** (type, number, or order).

### Why Use Method Overloading?

* Increases **readability** of the program.
* Allows you to **perform similar operations** with different kinds of input.
* Provides **flexibility** and code reuse.

### ****Rules of Method Overloading****

You can overload a method by changing:

1. Number of parameters
2. Type of parameters
3. Order of parameters (if types are different)

You **cannot overload** a method **by return type only**.

* 1. **Static Methods and Variables**

### Ans. 4. Static Methods and Variables in Java

In Java, the keyword **static** is used for **class-level members**, meaning they **belong to the class itself** rather than to instances (objects) of the class.

### What is a Static Variable?

A **static variable** is shared among **all objects** of a class.

### What is a Static Method?

A **static method** can be called **without creating an object**.  
It can **access only static variables/methods** directly.

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

**Theory:**

* 1. **Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction**

Ans. Sure! Here's a simple and complete explanation of the **basics of Object-Oriented Programming (OOP)** in Java:

### ****1. Basics of OOP in Java****

Java is a **fully object-oriented programming language**, and it follows the four main **OOP principles**:

### 1. ****Encapsulation****

**Encapsulation** means **binding data (variables) and methods (functions)** together into a single unit — a class — and **hiding** the internal details from the outside world.

### 2. ****Inheritance****

**Inheritance** allows a class (subclass) to **inherit properties and methods** from another class (superclass).

### 3. ****Polymorphism****

**Polymorphism** means "**many forms**". It allows one interface to be used for **different types of actions**.

There are **two types**:

### a) ****Compile-Time Polymorphism (Method Overloading)****

### b) ****Run-Time Polymorphism (Method Overriding)****

### 4. ****Abstraction****

**Abstraction** means **hiding the implementation details** and showing **only the essential features**.

You can achieve abstraction using:

* **Abstract classes** (with abstract methods)
* **Interfaces**

1. **Inheritance: Single, Multilevel, Hierarchical**

Ans. Here's a clear explanation of **Inheritance Types in Java** with examples:

### ****2. Inheritance in Java: Single, Multilevel, Hierarchical****

**Inheritance** allows a class to acquire the properties and behaviors (fields and methods) of another class.

Java supports the following types of inheritance:

### 1. ****Single Inheritance****

**One subclass inherits from one superclass**

### 2. ****Multilevel Inheritance****

✅ **A class inherits from a class which itself inherits from another class**

### 3. ****Hierarchical Inheritance****

**Multiple subclasses inherit from a single superclass**

1. **Method Overriding and Dynamic Method Dispatch**

### Ans. 3. Method Overriding and Dynamic Method Dispatch in Java

### ****Method Overriding****

**Method overriding** occurs when a **subclass provides its own version** of a method that is already defined in its **superclass**.

### ****Rules for Method Overriding:****

1. Method name must be the **same**.
2. Parameters must be **exactly the same** (no change).
3. The return type must be **same or covariant** (i.e., a subclass of the return type).
4. Access modifier in the subclass method must be **same or more accessible**.
5. The overridden method **cannot be static, private, or final**.

### ****Dynamic Method Dispatch (Run-Time Polymorphism)****

**Dynamic Method Dispatch** is the process in which a **superclass reference** is used to **refer to a subclass object**, and the **overridden method is called at runtime** based on the actual object type.

This enables **runtime polymorphism**.

### What happens here?

* The **method call is resolved at runtime**, **not compile time**.
* This makes your code **more flexible and extensible**.

**Constructors and Destructors**

**Theory:**

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

### Ans. 1. Constructor Types in Java: Default & Parameterized

In Java, a **constructor** is a special method that is automatically called **when an object is created**. Its main purpose is to **initialize the object**.

### What is a Constructor?

* Has the **same name** as the class.
* **No return type**, not even void main.
* Called automatically when you create an object using new.

### Types of Constructors in Java

### 1. ****Default Constructor****

A **default constructor** is a constructor **with no parameters**.

If **no constructor** is defined in a class, Java provides one **automatically**.

### 2. ****Parameterized Constructor****

A **parameterized constructor** allows you to **pass values** at the time of object creation.

* You can have **multiple constructors** in a class → this is called **constructor overloading**.
* If you define any constructor yourself, Java **won’t provide the default constructor automatically**.

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

### Ans. 2. Copy Constructor in Java (Emulated)

Java **does not provide a built-in copy constructor** like C++, but you can **create your own copy constructor** to copy the **values of one object to another**.

### What is a Copy Constructor?

A **copy constructor** is a constructor that **creates a new object** by **copying the values** from an existing object.

### Why Use a Copy Constructor?

* To **duplicate an object’s state** into a new object.
* Helps avoid writing repeated code for assigning values one by one.
* Especially useful for **immutable objects** or **custom cloning**.

* 1. **Constructor Overloading**

### Ans. 1. Constructor Overloading in Java

**Constructor Overloading** means having **multiple constructors** in the **same class**, but with **different parameter lists**.

Just like method overloading, constructor overloading helps to create objects in **different ways** based on the given input.

### Why Use Constructor Overloading?

* To **initialize objects with different data**.
* Provides **flexibility** in object creation.
* Helps to avoid code duplication.

### ****Rules:****

You can overload a constructor by changing:

* Number of parameters
* Type of parameters
* Order of parameter.

### Benefits of Constructor Overloading

* Simplifies object initialization.
* Allows setting **default values** when full data isn't available.
* Makes code more readable and maintainable.

1. **Object Life Cycle and Garbage Collection**

### Ans. 4. Object Life Cycle and Garbage Collection in Java

In Java, **objects are created, used, and then automatically destroyed** when they’re no longer needed. This process is known as the **object life cycle**, and **garbage collection** is responsible for reclaiming unused memory.

### ****Object Life Cycle in Java****

### 1️⃣ ****Object Creation****

* An object is created using the new keyword.
* The **constructor** is called to initialize the object.

Student s = new Student(); // Object is created

### 2️⃣ ****Object in Use****

* The object is used by calling its methods or accessing its fields.

s.setName("Alice");

s.display();

### 3️⃣ ****Object Becomes Unreachable****

* When there are **no more references** pointing to an object, it becomes **eligible for garbage collection**.

s = null; // Now the object has no reference

### ****Garbage Collection in Java****

Java has **automatic memory management** through the **Garbage Collector (GC)**.  
It runs in the background and **frees memory** by destroying objects that are **no longer reachable**.

### ****Key Features:****

* **No need to manually delete objects** (unlike C/C++).
* Improves **memory efficiency** and **application performance**.
* Based on **reference checking** — if an object has **zero references**, it's considered garbage.

### ****Requesting Garbage Collection (optional):****

You can suggest GC, but it's **not guaranteed** to run immediately.

### ****Finalize Method (Deprecated):****

Java used to provide a method called finalize(), which was called **before an object was destroyed**.

**8. Arrays and Strings**

**Theory:**

* 1. **One-Dimensional and Multidimensional Arrays**

### Ans. 1. One-Dimensional and Multidimensional Arrays in Java

Arrays in Java are used to **store multiple values of the same data type** in a single variable. They are **fixed in size** and indexed starting from 0.

### ****One-Dimensional Array (1D Array)****

A **1D array** is like a **list** — a single row of elements.

### ****Multidimensional Arrays****

A **multidimensional array** is an array of arrays — typically used to represent **matrices** or **tables**.

### 2D Array (Most common form)

* 1. **String Handling in Java: String Class, String Buffer, String Builder**

### Ans. 2. String Handling in Java

Java provides **three main classes** to handle strings:

* String (immutable)
* String Buffer (mutable, thread-safe)
* String Builder (mutable, not thread-safe but faster)

### 1. String Class

* **Immutable**: Once created, the content cannot be changed.
* Stored in the **String pool**.
* Most commonly used for basic string operations.

### 2. String Buffer Class

* **Mutable**: Content can be changed.
* **Thread-safe** (synchronized).
* Slower than String Builder.

### 3. String Builder Class

* Also **mutable** like String Buffer.
* **Not thread-safe**, but **faster** (no synchronization overhead).
* Ideal for single-threaded environments.
  1. **Array of Objects**

### Ans. 3. Array of Objects in Java

In Java, you can create an **array that stores objects** just like you can store integers or strings. This is called an **Array of Objects**.

It is useful when you need to manage **multiple instances of a class** in a structured collection.

### Useful Patterns

You can also use:

* **Enhanced for loop**:
* **Dynamic input with Scanner** for real-world data entry.

### Why Use Array of Objects?

* When you need to manage a **group of similar objects** (e.g., students, employees, products).
* Helps with data organization and makes **loop-based processing easy**.
  1. **String Methods (length, char , substring, etc.)**

### Ans. 4. Common String Methods in Java

The String class in Java provides many **built-in methods** to work with text data. Below are the **most frequently used** string methods with examples:

### 1. length()

Returns the **number of characters** in the string.

### 2. char At(int index)

Returns the **character at a specific index** (0-based).

### 3. substring(int begin Index)

Returns a **substring** starting from the specified index to the end.

### 4. substring(int begin Index, int end Index)

Returns a substring from begin Index to end Index - 1.

**5. to Lower Case() / to Upper Case()**

Converts the string to **lowercase** or **uppercase**.

### 6. equals(String s2)

Checks if two strings are **exactly equal** (case-sensitive).

7. equals Ignore Case(String s2)

Checks if two strings are equal **ignoring case**.

### 8. contains(String sequence)

Checks if the string contains a certain substring.

### 9. index Of(char) / last Index Of(char)

Returns the index of the **first** or **last occurrence** of a character or substring.

### 10. replace(char old, char new)

Replaces **all occurrences** of a character with another.

### 11. trim()

Removes **leading and trailing spaces**.

**9**. **Inheritance and Polymorphism**

**Theory:**

* 1. **Inheritance Types and Benefits**

### Ans. ****1. Inheritance Types and Benefits in OOP (Java)****

### ****What is Inheritance?****

**Inheritance** is a key feature of Object-Oriented Programming (OOP) that allows a **class to acquire the properties and methods** of another class.

* The **class that inherits** is called the **subclass** (child class).
* The **class being inherited from** is the **superclass** (parent class).

### ****Types of Inheritance in Java****

Java supports the following types of inheritance:

### 1. ****Single Inheritance****

### 2. ****Multilevel Inheritance****

### 3. ****Hierarchical Inheritance****

* Multiple classes inherit from a single superclass.

### 4. ****Multiple Inheritance (via Interfaces in Java)****

* A class implements multiple interfaces (Java doesn’t support multiple inheritance with classes).

### 5. ****Hybrid Inheritance****

* A combination of two or more types of inheritance.
* Achieved in Java using **interfaces**, not classes, to avoid ambiguity (diamond problem).
  1. **Method Overriding**

### Ans. Method Overriding in Java – Explanation

**What is Method Overriding?**

**Method Overriding** occurs when a **subclass** provides a **specific implementation** of a method that is already defined in its **superclass**.

* The method in the subclass must have the **same name**, **return type**, and **parameter list** as in the superclass.
* Used to achieve **runtime polymorphism** (dynamic method dispatch).

**Key Rules of Method Overriding:**

1. Method name, return type, and parameters **must be identical** to the superclass method.
2. The method **must be inherited** from the superclass.
3. Access modifier **cannot be more restrictive** (e.g., public in superclass, cannot be protected in subclass).
4. Only **instance methods** can be overridden (not static, final, or private).
5. The **@Override** annotation is optional but recommended for clarity and error checking.

**Why Use Method Overriding?**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Purpose** | **Description** | | --- | --- | | **Customize behaviour** | Subclass provides a specific version of a method. | | **Runtime Polymorphism** | Helps decide method call based on the object, not reference type. | | **Code Flexibility** | Promotes reusable and extendable design. | |

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

### Ans. Dynamic Binding (Run-Time Polymorphism) in Java – Explained

**What is Dynamic Binding?**

**Dynamic Binding** (also called **Late Binding**) is the process where the method call is **resolved at runtime** rather than at compile time.  
It is a key concept in **Run-Time Polymorphism**, allowing Java to decide **which method implementation to execute based on the actual object type**, not the reference type.

**How It Works**

* It occurs **when a superclass reference refers to a subclass object**.
* The method that gets executed is determined **at runtime** using **method overriding**.

* 1. **Super Keyword and Method Hiding**

### Ans. ****Super Keyword and Method Hiding in Java****

### ****1.**** super ****Keyword in Java:****

The super keyword in Java is used in the context of inheritance to refer to the immediate parent class of a subclass. It helps in:

### ****2. Method Hiding in Java:****

Method hiding occurs when a **static method** in a subclass has the **same signature** as a **static method** in its parent class.

* Unlike method overriding (which applies to instance methods), method hiding applies only to **static methods**.
* The method resolution is done at **compile time**, not runtime.

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

**Theory:**

* 1. **Abstract Classes and Methods**

### Ans. ****Abstract Classes and Methods in Java****

### ****1. What is an Abstract Class?****

An **abstract class** in Java is a class that **cannot be instantiated** (i.e., you can't create objects of it) and is meant to be **inherited by other classes**. It can contain:

* Abstract methods (methods without a body)
* Non-abstract methods (with a body)
* Fields (variables)
* Constructors

### ****2. What is an Abstract Method?****

An **abstract method** is a method that **has no body** (implementation) and is meant to be **overridden** in subclasses.

* It is declared using the abstract keyword.
* It **must be defined inside an abstract class**.
* All subclasses must **provide implementation** for the abstract method, unless the subclass is also declared abstract.

### ****3. Key Features of Abstract Classes:****

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Feature** | **Abstract Class** | | --- | --- | | Object Creation | Not allowed | | Method Types | Can have both abstract and normal methods | | Constructor | Yes, abstract classes can have constructors | | Access Modifiers | Can use any (public, protected, etc.) | | Fields | Can have instance variables | | Inheritance | Used as a **base class** | |

### ****When to Use Abstract Classes and Methods?****

* When you want to **provide a base class** with some common functionality, but **force subclasses** to implement specific methods.
* To represent **incomplete classes** or **generic concepts**, such as Animal, Shape, Vehicle, etc.
  1. **Interfaces: Multiple Inheritance in Java**

### Ans. ****Interfaces and Multiple Inheritance in Java****

### ****1. What is an Interface?****

An **interface** in Java is a **contract** that defines a set of **abstract methods** (and optionally static and default methods). Classes **implement** interfaces to provide behavior.

A class uses the implements keyword to use an interface:

### ****2. Why Use Interfaces?****

* To **achieve abstraction** without using abstract classes.
* To **simulate multiple inheritance** (Java does not support multiple inheritance with classes but allows it with interfaces).
* To define **common behaviour**  that unrelated classes can implement (e.g., Comparable, Serializable).

### ****3. Multiple Inheritance Using Interfaces****

Java does **not support multiple inheritance** with **classes** to avoid ambiguity (like the **Diamond Problem**), but it **supports multiple inheritance with interfaces**.

The Demo class implements **both interfaces** and provides method definitions for all abstract methods.

### ****4. Default and Static Methods in Interfaces (Java 8+)****

Java 8 introduced:

* **default methods** – methods with a body in the interface
* **static methods** – can be called using the interface name

### ****5. Diamond Problem in Interfaces****

If two interfaces have the same default method, the implementing class must **override it** to resolve ambiguity.

* 1. **Implementing Multiple Interfaces**

### Ans. Implementing Multiple Interfaces in Java

**What Does It Mean to Implement Multiple Interfaces?**

In Java, a class **can implement more than one interface**. This allows the class to inherit behaviours (method signatures) from multiple sources, enabling **multiple inheritance** of type.

**2. Key Rules:**

* A class **can implement multiple interfaces**.
* It must **override all abstract methods** from each interface.
* If interfaces have **default methods with the same signature**, the class must **override** the method to resolve conflict.

11. **Packages and Access Modifiers**

**Theory:**

* 1. **Java Packages: Built-in and User-Defined Packages**

### Ans. ****Java Packages: Built-in and User-Defined****

### ****What is a Package in Java?****

A **package** in Java is a **namespace** that organizes a set of related classes and interfaces. It helps:

* Avoid **name conflicts**
* Control **access**
* Group related types logically
* Simplify **code management**

### ****1. Built-in Packages in Java****

Java provides many **predefined (built-in)** packages as part of the Java Standard Library.

### ****2. User-Defined Packages****

You can create your own packages to organize your project’s code better.

* 1. **Access Modifiers: Private, Default, Protected, Public**

### Ans. ****Access Modifiers in Java:**** private****,**** default****,**** protected****,**** public

### ****What Are Access Modifiers?****

Access modifiers in Java **control the visibility** or **access level** of classes, methods, and variables. Java provides **four types** of access levels:

### ****1.**** private ****Access Modifier****

* **Scope:** Within the **same class only**
* **Not accessible** from outside the class.
* Often used for **encapsulation** (e.g., private fields with public getters/setters).

### ****2.**** Default ****(Package-Private) Access Modifier****

* **Scope:** Accessible within the **same package**.
* No keyword is used — just omit any access modifier.
* Cannot be accessed from outside its package.

Accessible within the same package only.

### ****3.**** protected ****Access Modifier****

* **Scope:** Accessible in:
  + The **same package**
  + **Subclasses (even in different packages)** using inheritance
* Used to allow **limited access** to child classes.

### ****4.**** public ****Access Modifier****

* **Scope:** Accessible from **anywhere** (any class, any package).
* Most open level of access.
  1. **Importing Packages and Class path**

### Ans. ****Importing Packages and Class path in Java****

### ****1. Importing Packages in Java****

To use **classes or interfaces from other packages**, you must **import** them using the import keyword.

### ****Types of Imports:****

### ****a. Single Type Import****

### ****b. On-Demand (Wildcard) Import****

### ****c. Static Import (Java 5+)****

Allows access to **static members** without class name prefix.

### ****2. What is Class path in Java?****

**Class path** tells the Java compiler (java) and JVM (java) **where to find** your .class files, libraries, and packages.

It can include:

* **Directories**
* **JAR files**
* **ZIP archives**

### ****Default Behaviour:****

By default, class path is the **current directory** (.).

### ****Setting Class path Temporarily****

### a. While Compiling****:****

### ****Setting Class path Permanently****

You can set it in your environment variables:

**12.** **Exception Handling**

**Theory:**

* 1. **Types of Exceptions: Checked and Unchecked**

### Ans. ****Types of Exceptions in Java: Checked and Unchecked****

In Java, **exceptions** are events that disrupt the normal flow of a program. They are objects that represent errors during execution.

### ****1. Checked Exceptions****

* **Checked at Compile-Time**
* The compiler **requires handling** (using try-catch or throws).
* Typically used for **recoverable errors** like file not found, invalid input, etc.

### ****2. Unchecked Exceptions****

* **Checked at Runtime** (not at compile time)
* The compiler **does not force** handling.
* Usually result from **programming logic errors** (e.g., null references, division by zero).
  1. **try, catch, finally, throw, throws**

### Ans. ****Exception Handling Keywords in Java:**** try****,**** catch****,**** finally****,**** throw****,**** throws

Java provides **structured exception handling** using five main keywords:

### ****1.**** try ****Block****

* Used to **wrap code** that might throw an exception.
* Must be followed by either catch or finally.

### ****2.**** catch ****Block****

* Handles exceptions thrown inside the try block.
* Can be multiple catch blocks to handle different exceptions.

### ****3.**** finally ****Block****

* Always **executes after try and catch**, whether an exception is thrown or not.
* Used to **release resources** (e.g., closing files, connections).

### ****4.**** throw ****Keyword****

* Used to **explicitly throw** an exception (object of Throw able class or subclass).

### ****5.**** throws ****Keyword****

* Used in method declarations to **declare** which exceptions a method might throw.
* Helps in **propagating exceptions** to the caller.
  1. **Custom Exception Classes**

### Ans. Custom Exception Classes in Java

**What is a Custom Exception?**

A **custom exception** is a user-defined class that **extends** the Java Exception or Runtime Exception class, allowing developers to define **specific error conditions** that aren't covered by built-in exceptions.

**Why Use Custom Exceptions?**

* To represent **domain-specific errors** (e.g., Insufficient Balance Exception, Invalid Age Exception)
* To make error handling **more meaningful and readable**
* To follow **clean code and better abstraction**

**13. Multithreading**

**Theory:**

### Ans. ****Introduction to Threads in Java****

### ****What is a Thread?****

A **thread** is a lightweight **sub-process** or the smallest unit of a program's execution. In Java, threads allow **concurrent execution**, meaning multiple tasks can run in **parallel**, improving efficiency and responsiveness.

### ****Why Use Threads?****

* To perform **multiple tasks simultaneously**
* To make applications **faster** and more **responsive**
* Essential for tasks like:
  + Background downloading
  + Animations in GUI
  + Parallel computations
  + Real-time data processing

### ****Creating Threads in Java****

There are **two ways** to create a thread:

**Runnable is preferred** when your class extends another class, since Java doesn’t support multiple inheritance with classes.

1. Creating Threads by Extending Thread Class or Implementing Runnable Interface

### Ans. ****Creating Threads in Java****

### ➤ Using Thread Class vs Runnable Interface

Java supports **multithreading** by providing two standard ways to create and run threads:

### ****1. Extending the**** Thread ****Class****

* Subclass the Thread class.
* Override its run() method.
* Call start() to begin execution.

### ****2. Implementing the**** Runnable ****Interface****

* Create a class that implements the Runnable interface.
* Implement the run() method.
* Pass an object of this class to a Thread constructor.

1. Thread Life Cycle

### Ans.

A thread in Java goes through **various states** during its execution. These states are defined in the Thread. State and reflect the **life cycle of a thread**.

**1. New**

* A thread object is **created**, but not yet started.

Thread t = new Thread(); // New state

**2. Runnable**

* After start() is called, the thread enters the **runnable pool** and is ready to run.

t.start(); // Now Runnable

The thread is **not yet running**, just eligible for CPU execution.

**3. Running**

* The thread is **selected by the thread scheduler** and is executing its run() method.

**4. Blocked**

* The thread is **waiting to acquire a lock** on an object that another thread holds.

**5. Waiting**

* The thread is waiting **indefinitely** for another thread to perform an action (like calling notify()).

wait(); // goes to waiting

**6. Timed Waiting**

* The thread waits for a **specified amount of time**.

**7. Terminated (Dead)**

* The thread has **finished execution** or is **abruptly terminated** due to an exception.

**Methods Related to Life Cycle:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Method** | **Purpose** | | --- | --- | | start() | Moves thread from **New → Runnable** | | run() | Called when thread gets CPU | | sleep(ms) | Timed Waiting | | wait() | Waiting state | | notify()/notify All() | Wakes up waiting threads | | join() | Waits for another thread to finish | |

### ****4. Synchronization and Inter-thread Communication in Java****

### ****Synchronization in Java****

### ****What is Synchronization?****

Synchronization is a process of **controlling access** to shared resources (like variables or methods) by multiple threads to **prevent data inconsistency** or **race conditions**.

### ****1. Why Synchronization?****

When multiple threads access the same object **without synchronization**, it can cause:

* Inconsistent data
* Unexpected behaviour
* Thread interference

### ****2. Synchronized Methods****

A synchronized method allows **only one thread** to execute it at a time for a particular object.

package com.thread;

class Callback

{

    public void call(String msg)

    {

        System.out.print("["+msg);

        try {

            Thread.sleep(5000);

        } catch (Exception e) {

            e.printStackTrace();

        }

        System.out.println("]");

    }

}

class Caller implements Runnable

{

    Thread t;

    String msg;

    Callback c;

    public Caller(String msg,Callback c) {

        this.msg=msg;

        this.c=c;

        t=new Thread(this);

        t.start();

    }

    public void run() {

        synchronized (c) {

            c.call(msg);

        }

    }

}

public class ThreadSynchronization {

    public static void main(String[] args) {

        Callback c=new Callback();

        Caller c1=new Caller("Hello",c);

        Caller c2=new Caller("welcome",c);

        Caller c3=new Caller("synchrozation",c);

    }

}

### ****3. Synchronized Blocks****

You can synchronize only a part of a method using a synchronized block.

class Printer {

void print(String msg) {

synchronized(this) {

System.out.print("[ " + msg);

try { Thread.sleep(100); } catch (Exception e) {}

System.out.println(" ]");

}

}

}

### ****4. Locking Mechanism****

* Every object in Java has an **intrinsic lock (monitor)**.
* A thread must **acquire the lock** before executing any synchronized method or block.
* Only **one thread** can hold the lock at a time.

### ****Inter-thread Communication****

Inter-thread communication in Java is a technique that allows **threads to communicate with each other** to ensure **coordinated execution**.

14. File Handling

Theory:

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

### Ans. 1. Introduction to File I/O in Java (java.io Package)

**What is File I/O?**

**File I/O (Input/Output)** in Java refers to **reading from** and **writing to** files on the disk. Java provides the **java.io package** to handle file operations like:

* Creating files
* Reading from files
* Writing to files
* Managing file streams

**1. Using File Class**

package assingment;

import java.io.File;

public class creatingfile {

    public static void main(String[] args) {

        try {

            File myFile = new File("myNewFile.txt");

            if (myFile.createNewFile()) {

                System.out.println("File created:" + myFile.getName());

            }else {

                System.out.println("File already exist.");

            }

        } catch (Exception e) {

            System.out.println("An error occured during file creation.");

            e.printStackTrace();

        }

    }

}

**2. Writing to a File using File Writer**

package com.file;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.FileWriter;

import java.io.IOException;

public class filewriteread {

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

        FileWriter fw=new FileWriter("tops2.txt");

        String s="this is file Writer/Reader Demo";

        fw.write(s);

        fw.flush();

        fw.close();

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

        int i;

        while ((i=fr.read())!=1)

        {

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

        }

        fr.close();

    }

}

**3. Reading from a File using FileReader**

package assingment;

import java.io.Closeable;

public class FileReader {

    public static void main(String[] args) {

            char[] array = new char[100];

        try {

            FileReader i = new FileReader();

            i.read(array);

            System.out.println("Data in the file");

            System.out.println(array);

             i.clone();

        } catch (Exception e) {

            e.getStackTrace();

        }

    }

    private void read(char[] array) {

    }

}

1. File Reader and FileWriter Classes

### Ans. ****2.**** FileReader ****and**** FileWriter ****Classes in Java****

### ****Overview****

FileReader and FileWriter are part of the **java.io package** and are used for **character-based file handling** in Java.

* FileReader: Reads characters from a file
* FileWriter: Writes characters to a file

They are ideal for working with **text files**.

### ****1. FileWriter Class****

* Used to **write characters** to a file.
* Can **overwrite** or **append** to the file.

### ****2. File Reader Class****

* Used to **read characters** from a file.
* Reads the file **character by character**.

15. Collections Framework

Theory:

1. Introduction to Collections Framework

### Ans. 1. Introduction to Collections Framework in Java

**What is the Collections Framework?**

The **Java Collections Framework (JCF)** is a **unified architecture** for storing, retrieving, and manipulating groups of data (objects). It provides:

* **Interfaces** (like List, Set, Map)
* **Implementations** (like Array List, Hash Set, Hash Map)
* **Algorithms** (like sorting, searching)

**Why Use Collections?**

Before JCF, Java used **arrays and custom data structures** which were:

* Fixed in size
* Difficult to manage
* Not consistent

JCF solves this with **resizable**, **type-safe**, and **efficient** data structures.

**Key Interfaces in the Collection Hierarchy**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Interface** | **Description** | | --- | --- | | Collection | Root interface for all collections | | List | Ordered collection, allows duplicates | | Set | Unordered collection, **no duplicates** | | Queue | Supports FIFO (First In First Out) operations | | Map | Stores key-value pairs (not a true Collection) | |

**Benefits of Using Collections Framework**

* **Reusable and efficient** data structures
* **Polymorphic behaviour**: switch implementations easily
* **Built-in algorithms**: sorting, searching, reversing
* **Thread-safe** options (Vector, Collections.Synchronized List())

1. List, Set, Map, and Queue Interfaces

### Ans. ****2. List, Set, Map, and Queue Interfaces in Java****

The Java **Collections Framework** provides four core interfaces that represent different ways of storing and manipulating groups of objects.

### ****1. List Interface****

* **Ordered** collection (elements are stored in insertion order)
* **Allows duplicates**
* Elements can be accessed by **index**

### ****2. Set Interface****

* **Unordered** collection
* **Does not allow duplicates**
* Automatically removes duplicate entries

### ****3. Map Interface****

* Not part of the Collection hierarchy
* Stores **key-value pairs**
* **Unique keys**, values can be duplicated
* Useful for **associating keys with values**

### Common Implementations:

* Hash Map – No order
* Linked Hash Map – Maintains insertion order
* Tree Map – Sorted by keys

### ****4. Queue Interface****

* Designed for **FIFO (First-In-First-Out)** data structure
* Used for **holding elements before processing**

1. Array List, Linked List, Hash Set, Tree Set, Hash Map, Tree Map

### Ans. ****3. Java Collection Classes:**** Array List****,**** Linked List****,**** Hash Set****,**** Tree Set****,**** Hash Map****,**** Tree Map

These are the **most commonly used classes** in the Java Collections Framework. Each serves different use-cases based on performance, ordering, and uniqueness.

### ****1****. Array List

### Implements: List

* **Ordered** collection
* **Allows duplicates**
* **Fast access**, slow insert/delete (shifting needed)

### ****2. Linked List****

* Implements: List, Deque
* Doubly linked list
* **Allows duplicates**
* **Fast insert/delete**, slower access (no index-based memory access)
* Implements: Set
* **No duplicates**
* **No order guaranteed**
* Based on **hash table**
* Allows **one null element**
* **Sorted (natural order or comparator)**
* **No duplicates**
* Based on **Red-Black tree**
* Implements: Map
* **Stores key-value pairs**
* **Keys must be unique**, values can be duplicated
* **No guaranteed order**
* Allows **one null key** and **multiple null values**
* Implements: Navigable Map
* **Sorted by keys (ascending order by default)**
* **Unique keys**
* No null keys (throws Null Pointer Exception)
* Based on **Red-Black tree**

1. Iterators and List Iterators.

### Ans. ****4. Iterators and List Iterators in Java****

In Java, **Iterators** and **List Iterators** are used to **traverse** (loop through) collections such as Array List, Hash Set, and Linked List.

* Can be used with **any Collection** (List, Set, Queue, etc.)
* Provides **forward-only** traversal
* Allows **element removal** during iteration
* Part of java.util package
* **Only for List types** (Array List, Linked List, etc.)
* Supports **bidirectional traversal** (forward and backward)
* Can **add, remove, or replace** elements
* Extends Iterator

16. Java Input / Output (I/O)

Theory:

* 1. Streams in Java (Input Stream, Output Stream)

### Ans. ****1. Streams in Java (Input Stream, Output Stream)****

Java provides the **java.io package** to perform **input and output (I/O)** through **streams**. Streams represent an ordered sequence of bytes and are used to read from or write to **files, memory, network connections, etc.**

### ****Types of Streams in Java****

Java I/O streams are categorized into two main types:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Stream Type** | **Description** | **Base Class** | | --- | --- | --- | | **Input Stream** | Used to **read** data (from source to program) | Input Stream | | **Output Stream** | Used to **write** data (from program to destination) | Output Stream | |

These are **byte streams**, meaning they handle data in the form of bytes (8-bit data).

* Superclass for all byte input streams
* Provides methods to read bytes from a data source
  1. Reading and Writing Data Using Streams

### Ans. ****2. Reading and Writing Data Using Streams in Java****

In Java, **streams** from the java.io package are used to **read** data from an input source (like a file or console) and **write** data to an output destination (like a file or screen). Streams handle data as a continuous flow of bytes or characters.

### ****Types of Streams for Reading/Writing:****

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Type** | **Class Pairs Used** | **Data Type** | | --- | --- | --- | | **Byte Streams** | Input Stream / Output Stream | Binary data | | **Character Streams** | Reader / Writer | Character data (Unicode) | |

* Always **close streams** to free resources.
* Use **Buffered Reader/Writer** for reading/writing lines of text.
* Use **File Input Stream/File Output Stream** for binary files (images, audio, etc.).
  1. Handling File I/O Operations

### Ans. ****3. Handling File I/O Operations in Java****

File I/O (Input/Output) operations in Java allow reading from and writing to files using various classes in the java.io and java packages. These operations are essential for persistent data storage in desktop and enterprise applications.