Java Notes Part 4:

Constructor:

- It is a special type of non-static method, which will be executed automatically only at the time of creating an object.

Example:

> Demo d1 = new Demo();

- The meaning of the above statement is to create an object of the Demo class by executing the zero-argument constructor of the Demo class.

Syntax of a Constructor:

> **[Access_Modifier] Class_Name([Param_List])[throws Exception_List]**
> **{**
>   **//...body of the constructor...**
> **}**

Example:

```
Demo(){
    System.out.println("Constructor executed");
}
```

Note:- We can have a dot Java file of a class, without a constructor, but we can't have a dot class file of a class without a constructor.

At least the default constructor must be there inside the dot class file of a class.

- Inside our class, if we keep any constructor, then that constructor will be executed at the time of creating an object of our class, but if we don't keep any constructor explicitly

inside our class, then the **Java compiler** will provide a default constructor to the dot class file.

- **The default constructor given by the compiler will always be public and have zero arguments.**

Example: The default constructor inside the above Demo class will be like:

```
public Demo(){
        //zero body
}
```

- This default constructor given by the Java compiler seems like an empty constructor, but strictly speaking, there is a hidden statement there, as the first line of this default constructor. (It is a "**super();**" it is a call to the parent class constructor. We will discuss this statement inside the Inheritance concept.)

Example:

```
public Demo(){
        super();
}
```

- If we want to execute certain logic at the time of creating an object of a class, then we should place that logic inside the constructor of that class.

- For the object creation, the constructor execution must be there.

- Until and unless our constructor gets executed completely, our object will not be created completely; it will be in the creation process.

- The main utilization of constructors is to provide initializations for the instance variables.

Difference between a non-static method and a constructor:

1. The method name can be anything, but the name of the constructor must be the class name.

   Note:- We can take a method name as a class name also, but it is a bad practice.

2. A method must have a **return type**, at least **void**, whereas a constructor should not have any **return type.**

3. A method we can call on an object whenever we want to call. But a constructor will be called on an object **automatically** only one time at the time of creating that object.

4. A method can be **static**, where the static keyword does not apply to a constructor.

5. A method can be abstract, whereas an **abstract** keyword does not apply to the constructor.

6. The **final** keyword applies to the method, whereas it does not apply to the constructor.

Similarities between a non-static method and a constructor:

- Both are code blocks, and we can write multiple executable statements.

- We can overload a constructor also, and the rules of constructor overloading are similar to the rules of method overloading.

Note: if we keep any constructor in our dot java file, then the Java compiler won't provide a default constructor to our dot class file.

Example:

Demo.java

```java
package com.masai;
public class Demo{

        Demo(){
                System.out.println("Inside constructor of Demo");
        }

        public static void main(String[] args){
```

```
                Demo d1=new Demo();
        }
    }
```

Constructor overloading:

- We can have multiple overloaded constructors inside our Java class.

- If inside a Java class, we have 5 overloaded constructors, then we can create the object of that class in 5 ways.

Example:

```java
package com.masai;
public class Employee {

        String empId;
        String name;
        double salary;

        //zero argument constructor
        public Employee() {

                empId="Emp-01";
                name= "Ramesh";
                salary = 50000.00;
        }
        //overloaded parameterized constructor
        public Employee(String empId, String name, double salary) {
                this.empId = empId;
                this.name = name;
                this.salary = salary;
        }


        public void showDetails() {

                System.out.println("Employee Id :"+empId);
                System.out.println("Employee Name :"+name);
                System.out.println("Salary is :"+salary);
```

```
        }
        public static void main(String[] args) {

                Employee emp1=new Employee();
                emp1.showDetails();

                Employee emp2=new Employee("Emp-02", "Dinesh", 40000.00);
                emp2.showDetails();


        }
    }
```

- Each constructor provides a different way to create objects.

The this keyword in Java:

- In Java, "this" is a keyword that refers to the current object instance. It can be used to refer to the instance variables and methods of the current object.

- 'this' keyword points to the current object.

- Whenever it is required to point to an object from a method that is under execution because of that object, then we use the 'this' keyword.

The following 3 main jobs of this keyword:

1. Points to the current class object.

   System.out.println(this);

2. Differentiate between the local and the instance variable inside the non-static context.

   this.x = x;

3. Calling a constructor of a class from another constructor of the same class.

   this(10);

Example:

Demo.java

```java
package com.masai;
public class Demo
{
        //instance variable
        int x=100;

        void fun1(){
                //local variable
                int x=500;
                System.out.println("inside fun1() of Demo");
                System.out.println(this); //Demo@232323 current class obj
                System.out.println(this.x); // 100
                System.out.println(x); //500

        }
        public static void main(String[] args)
        {
                Demo d1=new Demo();
                System.out.println(d1); // Demo@232323
                d1.fun1();

                //System.out.println(this); //Compilation Error

        }
}
```

Note: The 'this' keyword can not be used inside the static area.

Constructor for Initialization Example:

Student.java

```java
public class Student {

    int roll;
    String name;
    int marks;
```

```java
        Student(int roll, String name, int marks){
            this.roll = roll;
            this.name = name;
            this.marks = marks;
        }

        void show(){
            System.out.println(roll + " " + name + " "+ marks);
        }

    }
```

Uses:

```java
        Student s = new Student(1,"Ram",90);
```

Constructor Chaining: (Calling a constructor explicitly):

- A constructor will be called automatically whenever we create an object of a class.

- We can call a constructor of a class explicitly, but that call must be :

    1. From another constructor of the same class (using the **'this'** keyword)

    2. From the constructor of its child class. (using the **'super'** keyword)

Note:- if we call a constructor explicitly from another constructor, then that call must be the first line.

Example:

Demo.java:

```java
        package com.masai;
        public class Demo
        {
```

```java
    public Demo(){
        this(10);
        System.out.println("inside zero argument constructor Demo()");
    }

    public Demo(int x){
        this(100, 200);
        System.out.println("inside one argument constructor Demo(int x)");
    }

    public Demo(int x,int y){
        this("Hello");
        System.out.println("inside two argument constructor Demo(int x,int y)");
    }

    public Demo(String s){
        //this(); //it will become recursive call
System.out.println("inside one(String) argument constructor Demo(String s)");
    }

    public static void main(String[] args){


            Demo d1=new Demo();
    }
}
```

Pure Encapsulation

Encapsulation means:

- Hide data

- Provide controlled access

Steps

1. Make variables private

2. Provide getters and setters

.

Java Bean Class Rules

- A Java Bean is a reusable component.

    Rules:

    1. Class must be **public**.

    2. Variables must be **private**.

    3. Provide getters/setters.

    4. Must have a zero-argument constructor.

    5. A parameterized constructor is **optional**.

Example:

Student.java:

```java
package com.masai;
public class Student {

        private int roll;
        private String name;
        private int marks;

        public Student() {

        }
        public Student(int roll, String name, int marks) {
                super();
                this.roll = roll;
                this.name = name;
                this.marks = marks;
        }
```

```java
        public int getRoll() {
                return roll;
        }
        public void setRoll(int roll) {
                this.roll = roll;
        }


        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }

        public int getMarks() {
                return marks;
        }

        public void setMarks(int marks) {
                this.marks = marks;
        }
    }
```

Student Task:

Activity1: Predict the Output

```java
class Test {

        int x = 10;

        Test() {
                x = 20;
        }

        public static void main(String[] args) {
```

```java
            Test t = new Test();
            System.out.println(t.x);
        }
    }
```

Activity2: Predict the Output

```java
class Test {

    int x = 10;

    void show() {
        int x = 30;
        System.out.println(this.x);
    }

    public static void main(String[] args) {
        new Test().show();
    }
}
```

Activity3: Predict the Output

```java
class Test {

    Test() {
        this(10);
        System.out.println("A");
    }

    Test(int x) {
        System.out.println("B");
    }

    public static void main(String[] args) {
        new Test();
    }
}
```

Activity4: Predict the Output

```java
class Test {

    int x;

    Test(int x) {
        x = x;
    }

    public static void main(String[] args) {
        Test t = new Test(5);
        System.out.println(t.x);
    }
}
```

Activity5: Predict the Output

```java
class Test {

    Test() {
        System.out.println(this);
    }

    public static void main(String[] args) {
        Test t = new Test();
        System.out.println(t);
    }
}
```

Activity6: Inventory Item Tracker

Create a class Item with:

- String itemName
- int quantity
- double price
- The constructor takes all parameters.
- Use this for assigning instance variables.
- Method calculateTotalValue() returns quantity * price.
- Display item info and total value for 2–3 objects in main().

Example Output:

Item: Laptop, Quantity: 5, Price: 60000

Total Value: 300000

Activity7: Smart Device Initialization

Create a class SmartDevice with:

- ○ String name
- ○ String type
- ○ boolean isOn
- Constructor initializes all fields using this.
- Methods:
  - ● turnOn() → sets isOn = true
  - ● turnOff() → sets isOn = false
  - ● displayStatus() → prints name, type, and current status
  - In main(), create 2–3 devices and demonstrate turning them on/off.

Example Output:

Device: Living Room Light, Type: LED, Status: ON

Activity8: Vehicle Registration System with Constructor Chaining

Create a class Vehicle with the following attributes:
- ● String brand
- ● String model
- ● int year
- ● String registrationNumber
- Implement four constructors:
  - ● Default constructor → sets default brand/model/year/registration.
  - ● Constructor with brand → sets brand, other defaults.
  - ● Constructor with brand and model → sets brand and model, other defaults.
  - ● Constructor with brand, model, year, registration → initializes all fields.
  - ● Implement constructor chaining in the first three constructors one by one.
  - ● for constructor chaining:-

- brand = "Unknown-brand"
- model = "Unknown-model"
- year = 2023
- Registration = "Not-Assigned"

- Use constructor chaining (this()) to avoid duplicating initialization code.
- Implement a method displayInfo() to print all vehicle details.
- In main(), create 2–3 Vehicle objects using different constructors and display their info.

Scanner Class:

===============

Introduction

The **Scanner class** in Java is part of the java.util package is used to **take input from the user**.

It allows you to read **keyboard input**, files, and streams easily and flexibly.

Why it's important:

- Most programs require user interaction.
- Scanner makes it easy to read and parse input in Java.

Key Concepts and Definitions

1. **Scanner class**:
   - Part of java.util package.
   - Used to read **primitive data types** (int, double, etc.) and **strings**.
2. **Importing Scanner**:
   - Before using Scanner, you must import it:

     ```
     import java.util.Scanner;
     ```

3. **Creating a Scanner object**:

Scanner sc = new **Scanner**(System.in);

4. System. in tells Scanner to read input from the **keyboard**.
5. **Common Scanner methods**:

Method	Description
nextInt()	Reads an integer
nextDouble()	Reads a double
nextFloat()	Reads a float
nextLine()	Reads a full line of text
next()	Reads a single word
nextBoolean()	Reads a boolean (true/false)

Step-by-Step Explanation

1. Import and Create Scanner

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in); // Create Scanner object
    }
}
```

2. Reading Integer Input

```java
Scanner sc = new Scanner(System.in);

System.out.print("Enter your age: ");
int age = sc.nextInt();

System.out.println("Your age is: " + age);
```

Explanation:

- nextInt() reads the next integer typed by the user.

3. Reading Double Input

```java
System.out.print("Enter your salary: ");
double salary = sc.nextDouble();

System.out.println("Salary: " + salary);
```

4. Reading Single Word (String)

```java
System.out.print("Enter your first name: ");
String name = sc.next(); // Reads only the first word

System.out.println("Hello, " + name);
```

5. Reading Full Line (String)

```java
System.out.print("Enter your full name: ");
sc.nextLine(); // Consume leftover newline
String fullName = sc.nextLine();

System.out.println("Hello, " + fullName);
```

Important:

- After reading nextInt() or nextDouble(), a **newline remains in the buffer**.
- Use an extra nextLine() to consume it before reading full lines.

6. Reading Boolean

```java
System.out.print("Are you a student? (true/false): ");
```

```
boolean isStudent = sc.nextBoolean();

System.out.println("Student status: " + isStudent);
```

7. Closing the Scanner

```
sc.close();
```

- Always close the scanner to **release system resources**.
- Avoid using Scanner on System. in multiple places after closing it.

Examples

1. **Simple Calculator**

```
Scanner sc = new Scanner(System.in);

System.out.print("Enter first number: ");
int a = sc.nextInt();

System.out.print("Enter second number: ");
int b = sc.nextInt();

int sum = a + b;
System.out.println("Sum = " + sum);

sc.close();
```

2. **Reading Mixed Input**

```
Scanner sc = new Scanner(System.in);

System.out.print("Enter your age: ");
int age = sc.nextInt();

sc.nextLine(); // Consume leftover newline
```

```
System.out.print("Enter your name: ");
String name = sc.nextLine();

System.out.println("Hello " + name + ", age " + age);

sc.close();
```

Scanner Class – hasNext() Methods

What is hasNext()?

- hasNext() is a **boolean method** in the Scanner class.
- It **checks whether there is another token/input available** before actually reading it.
- Returns **true** if there is more input; otherwise, **false**.

Why use it?

- To avoid runtime errors when input is missing or unexpected.
- Commonly used in **loops** when reading multiple inputs from a user or a file.

Types of hasNext() Methods

Method	Description
hasNext()	Checks if **any token** is available.
hasNextInt()	Checks if the **next token can be parsed as an int**.
hasNextDouble()	Checks if the **next token can be parsed as a double**.
hasNextBoolean()	Checks if the **next token can be parsed as a boolean**.
hasNextLine()	Checks if **another line of input** is available.

How hasNext() Works

1. Scanner **looks ahead** in the input stream without removing the token.
2. If input exists, it returns true; otherwise, it returns false.
3. Only when you call next(), nextInt(), or similar methods does Scanner actually **consume the input**.

 Analogy:

- Imagine you're **peeking at the next card** in a deck without drawing it. That's hasNext().
- Drawing the card is like next().

Examples

1. Using hasNext() to read words until input ends

```java
import java.util.Scanner;

public class Main {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter words (type 'exit' to stop):");

      while (sc.hasNext()) { // Check if there is another token
         String word = sc.next();
         if (word.equals("exit")) {
            break;
         }
         System.out.println("You typed: " + word);
      }
      System.out.println("Bye");

      sc.close();
   }
}
```

Explanation:

- hasNext() ensures the loop continues **only if there is input**.
- The program stops when the user types "exit".

2. Using hasNextInt() to safely read integers

```java
Scanner sc = new Scanner(System.in);
System.out.println("Enter numbers (type a non-number to stop):");

while (sc.hasNextInt()) { // Only true if next input is an integer
    int num = sc.nextInt();
    System.out.println("You entered: " + num);
}

System.out.println("Non-integer input detected, stopping.");
sc.close();
```

Explanation:

- hasNextInt() prevents runtime errors caused by trying to read a non-integer with nextInt().
- Only integers are consumed; invalid input stops the loop.

3. Using hasNextLine() to read lines safely

```java
Scanner sc = new Scanner(System.in);
System.out.println("Enter multiple lines (type 'quit' to stop):");

while (sc.hasNextLine()) {
    String line = sc.nextLine();
    if (line.equals("quit"))
            break;

    System.out.println("Line: " + line);
```

```
}

sc.close();
```

Explanation:

- hasNextLine() checks if another line exists before reading.

Student Task:-
============
Create a calculator

===============

- Create a Calculator class:-
- Implement four methods:
  - add(double a, double b)
  - subtract(double a, double b)
  - multiply(double a, double b)
  - divide(double a, double b)

- In main():
  - Display a menu to the user:
    1. Add
    2. Subtract
    3. Multiply
    4. Divide
    5. Exit
- Take choice input from the user using Scanner.
- Take two numbers as input.
- Call the corresponding method based on the user's choice using a switch-case statement.
- Loop until the user chooses Exit.

Java Arrays

Introduction:-
In programming, we often need to store **multiple values of the same type**.
Instead of creating many separate variables, Java provides **arrays**, which allow us to store multiple values in a **single variable**.

Example problem:
- What if we want to store the marks of 100 students?
- Creating 100 variables is impractical — arrays solve this problem efficiently.

What is an Array?
An **array** is a collection of **fixed-size**, **ordered** elements of the **same data type**.

Basic Syntax
int[] marks;

Creating an Array
int[] marks = new int[10];
📌 This creates an array that can store **10 integers**.

Key Concept (Simple Explanation)

If we want to store the names of 100 people:
String[] names = new String[100];
- names is a reference variable
- It points to an **array object** in memory
- The array can store **100 String values**

Important Note
- In Java, an array is a **special type of object**.
- Whenever the JVM encounters [], it creates an **array object in heap memory**.

Fundamentals of Arrays
====================
1. Object Nature
- Arrays are objects in Java.

- Stored in **heap memory**

2. Type Safety
- An array can store **only one data type**

```
int[] arr; // can store only integers
```

3. Indexing (Zero-Based)
- First element → index 0
- Last element → length - 1

4. Size Constraint
- Size must be an int or short
- long is not allowed

Characteristics of Arrays
=====================

1. Memory Address
```
int[] a = new int[5];
System.out.println(a);
```
➡ Prints the **reference (hash code)**, not values.

2. Ordered Collection
- Elements are stored in sequence
- Each element has a fixed index

3. Default Values
At creation time, all elements are initialized automatically:

Data Type	Default Value
int	0
double	0.0
char	'\u0000'
boolean	false

Object	null

```java
int[] marks = new int[10];
System.out.println(marks[0]); // 0
```

4. Array Length

Every array has a **non-static variable** called length.

```java
System.out.println(marks.length); // 10
```

5. Boundary Exception

Accessing an invalid index causes a runtime error:

```java
int[] marks = new int[5];
marks[10] = 50; // ArrayIndexOutOfBoundsException
```

Ways to Create an Array
=====================

Method 1: Using a new Keyword

```java
int[] marks = new int[5];

marks[0] = 100;
marks[1] = 120;
marks[2] = 150;
```

- Useful when the size is known but the values come later

Method 2: Using Curly Brackets

```java
int[] marks = {100, 120, 150, 180};
```

- Simple and commonly used
- Size is automatically calculated

Accessing and Iterating Elements
===========================

1. **Accessing by Index**

```java
int[] marks = {40, 40, 55, 25, 55};
System.out.println(marks[0]); // First element
System.out.println(marks[4]); // Last element
```

2. Using the Standard **for** Loop

```java
for (int i = 0; i < marks.length; i++) {
    System.out.println(marks[i]);
}
```

- Best when an index is needed

3. Using Enhanced **for-each** Loop

```java
for (int m: marks) {
    System.out.println(m);
}
```

- Simple and readable
- Index not available

Practical Examples
=================

Example 1: Sum and Average

```java
int[] numbers = {2, -9, 0, 5, 12, -25, 22, 9, 8, 12};
int sum = 0;
for (int number: numbers) {
    sum += number;
}

double average = (double) sum/numbers.length;

System.out.println("Sum = " + sum);
System.out.println("Average = " + average);
```

Example 2: Array of Objects
Arrays can store **reference types**.

```java
class Student {
    int rollNo;
    String name;

    Student(int rollNo, String name) {
        this.rollNo = rollNo;
        this.name = name;
    }
```

```java
    void printDetails() {
        System.out.println("Roll: " + rollNo + ", Name: " + name);
    }
}

Student[] students = new Student[3];

students[0] = new Student(10, "Ram");
students[1] = new Student(20, "Ramesh");
students[2] = new Student(40, "Amit");

for (Student s: students) {
    s.printDetails();
}
```

Memory Insight
- 1 array object

- 3 Student objects
  - ➡ Total **4 objects**

Multi-Dimensional Arrays (2D Arrays)

Definition
A **2D array** is an **array of arrays**.

Declaration
```java
int[][] matrix = new int[3][4];
```
- 3 rows and 4 columns

Direct Initialization
```java
int[][] a = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

Printing a 2D Array

```java
for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < a[i].length; j++) {
        System.out.print(a[i][j] + " ");
    }
    System.out.println();
}
```

Jagged Arrays

What is a Jagged Array?
A **jagged array** is a 2D array where **each row can have a different length**.
- Saves memory
- Useful for uneven data

Real-Life Analogy
🎓 **Classroom Seating**
- Row 1 → 3 students
- Row 2 → 5 students
- Row 3 → 2 students

- Using a normal 2D array wastes space.
- Jagged arrays fit perfectly.

Jagged Array Example

```java
int[][] studentSteps = new int[3][];

studentSteps[0] = new int[2];
studentSteps[1] = new int[4];
studentSteps[2] = new int[3];

studentSteps[0][0] = 5000;
studentSteps[0][1] = 7000;

studentSteps[1][0] = 3000;
studentSteps[1][1] = 4500;
studentSteps[1][2] = 6000;
studentSteps[1][3] = 8000;

studentSteps[2][0] = 10000;
```

```
studentSteps[2][1] = 12000;
studentSteps[2][2] = 9000;
```

Displaying Jagged Array
```
for (int i = 0; i < studentSteps.length; i++) {
    System.out.print("Person " + (i + 1) + ": ");
    for (int j = 0; j < studentSteps[i].length; j++) {
        System.out.print(studentSteps[i][j] + " ");
    }
    System.out.println();
}
```

Common Mistakes to Avoid
- Accessing an invalid index
- Forgetting the array size is fixed
- Confusing length with length()
- Assuming arrays auto-resize (they don't)

Common Methods from Arrays Class

=============================

To use these methods, we must import:

```
import java.util.Arrays;
```

1. Arrays.toString()
Purpose:

Converts an array into a readable string format.

Without toString()
```
int[] a = {1, 2, 3};
System.out.println(a); // Prints memory reference
```

With toString()
```
System.out.println(Arrays.toString(a));
```

Output:

```
[1, 2, 3]
```

- Best for **printing arrays**

2. Arrays.sort()

Purpose:

Sorts the array in **ascending order**.

```
int[] numbers = {5, 2, 8, 1};
Arrays.sort(numbers);
System.out.println(Arrays.toString(numbers));
```

Output:

```
[1, 2, 5, 8]
```

- Uses **Dual-Pivot QuickSort** internally
- Modifies the original array

3. Arrays.equals()

Purpose:

Checks whether **two arrays are equal** (same size & same elements).

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};

System.out.println(Arrays.equals(a, b));
```

Output:

```
true
```

4. Arrays.fill()

Purpose:

Fills the entire array with a single value.

```java
int[] arr = new int[5];
Arrays.fill(arr, 10);
System.out.println(Arrays.toString(arr));
```

Output:

```
[10, 10, 10, 10, 10]
```

- Useful for initialization

5. Arrays.copyOf()

Purpose:

Creates a **new copy** of an array.

```java
int[] original = {1, 2, 3};
int[] copy = Arrays.copyOf(original, original.length);

System.out.println(Arrays.toString(copy));
```

- Changes to the copy won't affect the original.

6. Arrays.copyOfRange()

Purpose:

Copies a **portion** of an array.

```java
int[] a = {10, 20, 30, 40, 50};
int[] part = Arrays.copyOfRange(a, 1, 4);

System.out.println(Arrays.toString(part));
```

Output:

```
[20, 30, 40]
```

Start index → inclusive
End index → exclusive

7. Arrays.deepToString() (For 2D Arrays)

Purpose:

Prints multi-dimensional arrays.

```java
int[][] matrix = {
    {1, 2},
    {3, 4}
};

System.out.println(Arrays.deepToString(matrix));
```

Output:

```
[[1, 2], [3, 4]]
```

- Best for **2D & Jagged Arrays**

8. Arrays.deepEquals()

Purpose:

Compares **2D arrays** element by element.

```java
int[][] a = {{1, 2}, {3, 4}};
int[][] b = {{1, 2}, {3, 4}};

System.out.println(Arrays.deepEquals(a, b));
```

Output:

```
true
```