Module 1: Introduction to Java Fundamentals

Premanand S

Assistant Professor
School of Electronics Engineering
Vellore Institute of Technology
Chennai Campus

premanand.s@vit.ac.in

January 13, 2026

Module 1: Introduction to Java Fundamentals

- OOP Paradigm and Features of Java
- JVM, Bytecode, Java Program Structure
- Data Types, Variables, Naming Conventions
- Operators, Control and Looping Constructs
- One- and Multi-dimensional Arrays
- Enhanced for-loop
- Strings, StringBuffer, StringBuilder, Math Class
- Wrapper Classes

Arrays — What is an Array?

An **array** is a collection of values stored under a **single variable name**.

Arrays — What is an Array?

An **array** is a collection of values stored under a **single variable name**.

In simple words:

- Store many values
- Of the same data type
- In continuous memory locations

Arrays — What is an Array?

An **array** is a collection of values stored under a **single variable name**.

In simple words:

- Store many values
- Of the same data type
- In continuous memory locations

Why arrays?

- Avoid many separate variables
- Easy processing using loops

Arrays — Why Do We Need Them?

Imagine storing marks of 5 students:

Arrays — Why Do We Need Them?

Imagine storing marks of 5 students:

- Without array: `m1, m2, m3, m4, m5`

Arrays — Why Do We Need Them?

Imagine storing marks of 5 students:

- Without array: `m1, m2, m3, m4, m5`
- With array: `marks[0] to marks[4]`

Arrays — Why Do We Need Them?

Imagine storing marks of 5 students:

- Without array: `m1, m2, m3, m4, m5`
- With array: `marks[0] to marks[4]`

Benefit:

- Easy to process with loops
- Less code, more clarity

One-Dimensional Array

A **one-dimensional array** stores data in a single row.

One-Dimensional Array

A **one-dimensional array** stores data in a single row.
Visual Representation:

```
marks = [85, 90, 78, 92, 88]
```

One-Dimensional Array

A **one-dimensional array** stores data in a single row.
Visual Representation:

```
marks = [85, 90, 78, 92, 88]
```

Index positions:

```
0   1   2   3   4
```

Declaring an Array in Java

```
int[] marks;
```

Declaring an Array in Java

```
int[] marks;
```

Or

```
int marks[];
```

Declaring an Array in Java

```
int[] marks;
```

Or

```
int marks[];
```

Best practice:

- Use `int[] marks`

Creating an Array

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

Creating an Array

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

Meaning:

- Creates space for 5 integers
- Index range: 0 to 4

Declaring and Creating Together

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

Declaring and Creating Together

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

Shortcut initialization:

```
int[] marks = {85, 90, 78, 92, 88};
```

Accessing Array Elements

```
System.out.println(marks[0]); // first element
System.out.println(marks[4]); // last element
```

Accessing Array Elements

```
System.out.println(marks[0]); // first element
System.out.println(marks[4]); // last element
```

Important Rule:

- Index always starts from **0**

Array Length

```
int size = marks.length;
```

Array Length

```
int size = marks.length;
```

Key Point:

- `length` is a **property**, not a method
- So $\rightarrow$ no brackets: `length()`

Traversing an Array — Using `for`

```
int[] marks = {85, 90, 78, 92, 88};

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

Array — Common Mistake

Dangerous code:

Array — Common Mistake

Dangerous code:

- `marks[5]` when array size is 5

Array — Common Mistake

Dangerous code:

- `marks[5]` when array size is 5

Result:

- `ArrayIndexOutOfBoundsException`

Array — Common Mistake

Dangerous code:

- `marks[5]` when array size is 5

Result:

- `ArrayIndexOutOfBoundsException`

Rule:

- Always access from `0` to `length-1`

1D Array

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

  int[] marks = {85, 90, 78, 92, 88};

  System.out.println("Marks are:");

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

Arrays — Indexing Rule

In Java, array indexing always starts from **0**.

Arrays — Indexing Rule

In Java, array indexing always starts from **0**.
Example:

- `a[0]` $\rightarrow$ first element
- `a[length-1]` $\rightarrow$ last element

Arrays — Indexing Rule

In Java, array indexing always starts from **0**.

Example:

- `a[0]` → first element
- `a[length-1]` → last element

Common Trap:

- Using index = size → causes `ArrayIndexOutOfBoundsException`

Arrays — `length` Property

For arrays:

Arrays — `length` Property

For arrays:

- `length` is a **property**
- Not a method

Arrays — `length` Property

For arrays:

- `length` is a **property**
- Not a method

Correct:

- `arr.length`

Arrays — `length` Property

For arrays:

- `length` is a **property**
- Not a method

Correct:

- `arr.length`

Wrong:

- `arr.length()`

Arrays — Objects in Java

Even if an array stores primitives:

Arrays — Objects in Java

Even if an array stores primitives:

- The array itself is an **object**
- Stored in heap memory

Arrays — Objects in Java

Even if an array stores primitives:

- The array itself is an **object**
- Stored in heap memory

Result:

- Passed by reference
- Shared between variables

Arrays — Shared Between Variables

When we assign one array to another:

Arrays — Shared Between Variables

When we assign one array to another:

```
int[] a = {10, 20, 30};
int[] b = a;
```

Arrays — Shared Between Variables

When we assign one array to another:

```
int[] a = {10, 20, 30};
int[] b = a;
```

Memory view:

Arrays — Shared Between Variables

When we assign one array to another:

```
int[] a = {10, 20, 30};
int[] b = a;
```

Memory view:

$$\texttt{a} \longrightarrow [10 \quad 20 \quad 30]$$
$$\texttt{b} \longrightarrow [10 \quad 20 \quad 30]$$

Arrays — Shared Between Variables

When we assign one array to another:

```
int[] a = {10, 20, 30};
int[] b = a;
```

Memory view:

$$\texttt{a} \longrightarrow [10 \quad 20 \quad 30]$$
$$\texttt{b} \longrightarrow [10 \quad 20 \quad 30]$$

What this means:

- No new array is created

Arrays — Shared Between Variables

When we assign one array to another:

```
int[] a = {10, 20, 30};
int[] b = a;
```

Memory view:

$$\texttt{a} \longrightarrow [10 \quad 20 \quad 30]$$
$$\texttt{b} \longrightarrow [10 \quad 20 \quad 30]$$

What this means:

- No new array is created
- Both variables point to the **same memory**

Arrays — Shared Between Variables

When we assign one array to another:

```
int[] a = {10, 20, 30};
int[] b = a;
```

Memory view:

$$\begin{aligned} \texttt{a} &\longrightarrow [10 \quad 20 \quad 30] \\ \texttt{b} &\longrightarrow [10 \quad 20 \quad 30] \end{aligned}$$

What this means:

- No new array is created
- Both variables point to the **same memory**

Result:

- Change using `b` affects `a`
- Change using `a` affects `b`

Arrays — Default Values

When you create an array:

Arrays — Default Values

When you create an array:

- Java fills it with default values

Arrays — Default Values

When you create an array:

- Java fills it with default values

Examples:

- `int[]` $\rightarrow$ 0
- `double[]` $\rightarrow$ 0.0
- `boolean[]` $\rightarrow$ false
- `Object[]` $\rightarrow$ null

Arrays — Printing Trap

Printing an array directly:

Arrays — Printing Trap

Printing an array directly:

- `System.out.println(arr);`

Arrays — Printing Trap

Printing an array directly:

- `System.out.println(arr);`

Output:

- Memory reference, not elements

Arrays — Printing Trap

Printing an array directly:

- `System.out.println(arr);`

Output:

- Memory reference, not elements

Correct Way:

- `Arrays.toString(arr);`

Arrays — Loop Boundary Mistake

Wrong loop:

Arrays — Loop Boundary Mistake

Wrong loop:

- `for(int i=0; i<=arr.length; i++)`

Arrays — Loop Boundary Mistake

Wrong loop:

- `for(int i=0; i<=arr.length; i++)`

Correct loop:

- `for(int i=0; i<arr.length; i++)`

Arrays — Loop Boundary Mistake

Wrong loop:

- `for(int i=0; i<=arr.length; i++)`

Correct loop:

- `for(int i=0; i<arr.length; i++)`

Reason:

- Last valid index = `length-1`

Arrays — Fixed Size

Once created, array size cannot change.

Arrays — Fixed Size

Once created, array size cannot change.

Example:

- `new int[5]` $\rightarrow$ always size 5

Arrays — Fixed Size

Once created, array size cannot change.

Example:

- `new int[5]` $\rightarrow$ always size 5

If you need dynamic size:

- Use `ArrayList`

Arrays — Enhanced `for` Loop

Enhanced loop:

Arrays — Enhanced `for` Loop

Enhanced loop:

- `for(int x : arr)`

Arrays — Enhanced `for` Loop

Enhanced loop:

- `for(int x : arr)`

Important:

- `x` is a copy of each element
- Changing `x` does NOT change the array

Arrays — Passing to Methods

When an array is passed to a method:

Arrays — Passing to Methods

When an array is passed to a method:

- The reference is passed
- Not a copy of the data

Arrays — Passing to Methods

When an array is passed to a method:

- The reference is passed
- Not a copy of the data

Result:

- Changes inside method affect original array

Arrays — Shallow vs Deep Copy

Shallow Copy:

Arrays — Shallow vs Deep Copy

Shallow Copy:

- `int[] b = a;`

Arrays — Shallow vs Deep Copy

Shallow Copy:

- `int[] b = a;`

Deep Copy:

Arrays — Shallow vs Deep Copy

Shallow Copy:

- `int[] b = a;`

Deep Copy:

- `Arrays.copyOf(a, a.length);`

Arrays — Shallow vs Deep Copy

Shallow Copy:

- `int[] b = a;`

Deep Copy:

- `Arrays.copyOf(a, a.length);`

Difference:

- Shallow → same memory
- Deep → new memory

Arrays — Common Exceptions

- `ArrayIndexOutOfBoundsException`

Arrays — Common Exceptions

- `ArrayIndexOutOfBoundsException`
- `NullPointerException`

Arrays — Common Exceptions

- `ArrayIndexOutOfBoundsException`
- `NullPointerException`
- `ArrayStoreException`

Arrays — Common Exceptions

- `ArrayIndexOutOfBoundsException`
- `NullPointerException`
- `ArrayStoreException`

Tip: Most array errors come from wrong indexing.

Multi-Dimensional Arrays — Introduction

A **multi-dimensional array** stores data in **rows and columns**.

Multi-Dimensional Arrays — Introduction

A **multi-dimensional array** stores data in **rows and columns**.
In simple words:

- Array inside another array
- Looks like a table or matrix

Multi-Dimensional Arrays — Introduction

A **multi-dimensional array** stores data in **rows and columns**.
In simple words:

- Array inside another array
- Looks like a table or matrix

Most common type:

- Two-dimensional array (2D array)

2D Array — Real-Life Analogy

Think of a **classroom marks table**:

2D Array — Real-Life Analogy

Think of a **classroom marks table**:

- Rows $\rightarrow$ Students
- Columns $\rightarrow$ Subjects

2D Array — Real-Life Analogy

Think of a **classroom marks table**:

- Rows → Students
- Columns → Subjects

Example:

`marks[student][subject]`

2D Array — Real-Life Analogy

Think of a **classroom marks table**:

- Rows → Students
- Columns → Subjects

Example:

`marks[student][subject]`

So:

- `marks[2][1]` → Student 3, Subject 2

Declaring a 2D Array

```
int[][] marks;
```

Declaring a 2D Array

```
int[][] marks;
```

Or

```
int marks[][];
```

Declaring a 2D Array

```
int[][] marks;
```

Or

```
int marks[][];
```

Best Practice:

- Use `int[][] marks;`

Creating a 2D Array

```
marks = new int[3][4];
```

Creating a 2D Array

```
marks = new int[3][4];
```

Meaning:

- 3 rows
- 4 columns

Creating a 2D Array

```
marks = new int[3][4];
```

Meaning:

- 3 rows
- 4 columns

Index range:

- Rows $\rightarrow$ 0 to 2
- Columns $\rightarrow$ 0 to 3

Declaring and Creating Together

```
int[][] marks = new int[3][4];
```

Declaring and Creating Together

```
int[][] marks = new int[3][4];
```

Shortcut initialization:

```
int[][] marks = {
 {85, 90, 78, 92},
 {88, 76, 91, 84},
 {70, 82, 89, 95}
};
```

Accessing 2D Array Elements

```
System.out.println(marks[0][0]); // first R, first C
System.out.println(marks[2][3]); // third R, fourth C
```

Accessing 2D Array Elements

```
System.out.println(marks[0][0]); // first R, first C
System.out.println(marks[2][3]); // third R, fourth C
```

Rule:

- First index $\rightarrow$ row
- Second index $\rightarrow$ column

Traversing a 2D Array

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

2D Arrays — Understanding `length`

2D Arrays — Understanding `length`

- `marks.length` $\rightarrow$ number of rows

2D Arrays — Understanding `length`

- `marks.length` $\rightarrow$ number of rows
- `marks[i].length` $\rightarrow$ columns in row `i`

2D Arrays — Understanding `length`

- `marks.length` → number of rows
- `marks[i].length` → columns in row `i`

Important:

- Each row can have different length

2D Arrays — Jagged Arrays

In Java, 2D arrays can be **jagged**.

2D Arrays — Jagged Arrays

In Java, 2D arrays can be **jagged**.
Meaning:

- Each row can have different number of columns

2D Arrays — Jagged Arrays

In Java, 2D arrays can be **jagged**.

Meaning:

- Each row can have different number of columns

Example:

- Row 1 $\rightarrow$ 3 elements
- Row 2 $\rightarrow$ 5 elements
- Row 3 $\rightarrow$ 2 elements

Jagged Array — Example

```
int[][] arr = {
 {1, 2, 3},
 {4, 5, 6, 7, 8},
 {9, 10}
};
```

2D Arrays — Common Mistakes

- Using wrong index order

2D Arrays — Common Mistakes

- Using wrong index order
- Assuming all rows have same length

2D Arrays — Common Mistakes

- Using wrong index order
- Assuming all rows have same length
- Using `arr[0].length` for all rows

2D Arrays — Common Mistakes

- Using wrong index order
- Assuming all rows have same length
- Using `arr[0].length` for all rows
- Looping with wrong boundaries

2D Array

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

  int[][] marks = {
   {85, 90, 78},
   {88, 76, 91},
   {70, 82, 89}
  };

  System.out.println("Marks Table:");

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

2D Arrays — Tricky Understandings

- 2D arrays are actually **arrays of arrays**

2D Arrays — Tricky Understandings

- 2D arrays are actually **arrays of arrays**
- Rows can have different sizes

2D Arrays — Tricky Understandings

- 2D arrays are actually **arrays of arrays**
- Rows can have different sizes
- Access always uses two indices

2D Arrays — Tricky Understandings

- 2D arrays are actually **arrays of arrays**
- Rows can have different sizes
- Access always uses two indices
- Nested loops are natural for traversal

Enhanced `for` Loop — Introduction

The **enhanced** `for` loop is used to **traverse elements** of a collection easily.

Enhanced `for` Loop — Introduction

The **enhanced** `for` loop is used to **traverse elements** of a collection easily.

Also called:

- **for-each loop**

Enhanced `for` Loop — Introduction

The **enhanced** `for` loop is used to **traverse elements** of a collection easily.

Also called:

- **for-each loop**

Used mainly with:

- Arrays
- Collections (List, Set, etc.)

Enhanced `for` — Why Use It?

Traditional loop:

Enhanced `for` — Why Use It?

Traditional loop:

- Needs index
- Risk of boundary errors

Enhanced `for` — Why Use It?

Traditional loop:

- Needs index
- Risk of boundary errors

Enhanced loop:

- No index needed
- Cleaner and safer

Enhanced `for` — Why Use It?

Traditional loop:

- Needs index
- Risk of boundary errors

Enhanced loop:

- No index needed
- Cleaner and safer

Goal:

- Focus on **values**, not **positions**

Enhanced `for` — Syntax

```
for (dataType variable : collection) {
 // use variable
}
```

Enhanced `for` — Syntax

```
for (dataType variable : collection) {
 // use variable
}
```

Meaning:

- Take each element from collection
- Store in variable
- Execute block

Enhanced for — 1D Array Example

```
int[] marks = {85, 90, 78, 92, 88};

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

Traditional vs Enhanced `for`

Traditional `for`

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

- Uses index
- More control

Enhanced `for`

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

- No index
- Cleaner code

Enhanced for — 2D Array

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

for (int[] row : matrix) {
 for (int val : row) {
  System.out.print(val + " ");
 }
 System.out.println();
}
```

When NOT to Use Enhanced `for`

Do **not** use enhanced `for` when:

When NOT to Use Enhanced `for`

Do **not** use enhanced `for` when:

- You need the index value

When NOT to Use Enhanced `for`

Do **not** use enhanced `for` when:

- You need the index value
- You want to modify array elements

When NOT to Use Enhanced `for`

Do **not** use enhanced `for` when:

- You need the index value
- You want to modify array elements
- You need reverse traversal

When NOT to Use Enhanced `for`

Do **not** use enhanced `for` when:

- You need the index value
- You want to modify array elements
- You need reverse traversal

Use traditional `for` instead.

Enhanced `for` — Read-Only Nature

This does NOT modify the array:

Enhanced `for` — Read-Only Nature

This does NOT modify the array:

- `for(int x : arr) { x = 10; }`

Enhanced `for` — Read-Only Nature

This does NOT modify the array:

- `for(int x : arr) { x = 10; }`

Why?

- `x` is only a **copy** of each element

Enhanced `for` — Read-Only Nature

This does NOT modify the array:

- `for(int x : arr) { x = 10; }`

Why?

- `x` is only a **copy** of each element

To modify elements:

- Use index-based loop

Enhanced `for` — Common Mistakes

- Trying to access index inside loop

Enhanced `for` — Common Mistakes

- Trying to access index inside loop
- Trying to change elements directly

Enhanced `for` — Common Mistakes

- Trying to access index inside loop
- Trying to change elements directly
- Using enhanced loop when order matters

Enhanced `for` — Common Mistakes

- Trying to access index inside loop
- Trying to change elements directly
- Using enhanced loop when order matters

Rule: Use enhanced `for` only for **reading/traversing**.

Enhanced for

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

  int[] nums = {10, 20, 30, 40};

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

  for (int n : nums) {
   System.out.println(n);
  }
 }
}
```

Tricky Question 1 — Can this modify the array?

Consider the code:

Tricky Question 1 — Can this modify the array?

Consider the code:

```
for(int x :  arr) { x = 10; }
```

Tricky Question 1 — Can this modify the array?

Consider the code:

```
for(int x :  arr) { x = 10; }
```

Will this change all elements of the array to 10?

- Yes
- No

Tricky Question 2 — Where is the index?

In an enhanced `for` loop:

Tricky Question 2 — Where is the index?

In an enhanced `for` loop:

```
for(int x :  arr)
```

How do you access the index of each element?

Tricky Question 2 — Where is the index?

In an enhanced `for` loop:

```
for(int x :  arr)
```

How do you access the index of each element?

- Using `x`
- Using a counter variable
- You cannot directly

Tricky Question 3 — Reverse Traversal

Can you traverse an array in **reverse order** using enhanced `for` loop?

Tricky Question 3 — Reverse Traversal

Can you traverse an array in **reverse order** using enhanced `for` loop?

- Yes
- No

Tricky Question 4 — Which loop is correct?

Which loop is better to **update** array elements?

Tricky Question 4 — Which loop is correct?

Which loop is better to **update** array elements?

- Enhanced `for`
- Traditional `for`

Tricky Question 5 — Works with which?

Enhanced `for` works with:

Tricky Question 5 — Works with which?

Enhanced `for` works with:

- Arrays
- Collections
- Both
- None

Thank You!

Stay Connected

Premanand S

Email: premanand.s@vit.ac.in
Phone: +91-7358679961

LinkedIn: linkedin.com/in/premsanand
Instagram: instagram.com/premsanand
WhatsApp Channel: anandsDataX

Google Scholar: Google Scholar Profile
GitHub: github.com/anandprems