Java Learning Journey

Chapter 7 - Java Single-Dimensional Arrays

✍ **Prepared by:** Ahmed Elsifi

📌 Why Use Arrays?

Arrays allow you to store multiple values of the same type under one variable name. They provide an efficient way to manage and process large collections of data without declaring many individual variables.

📌 How to Declare an Array in Java

Method 1 (Preferred):

```
elementType[] arrayRefVar;
```

Method 2 (Allowed but not preferred):

```
elementType arrayRefVar[];
```

Example:

```
double[] myList;  // Preferred
double myList[];  // Allowed
```

📌 When to Initialize the Array Size

- In Java, you can declare an array variable without initializing it:

  ```
  int[] arr; // declaration only
  ```

- You can initialize it later:

  ```
  arr = new int[10];
  ```

- Or declare and initialize in one line:

  ```
  int[] arr = new int[10];
  ```

🔍 **Comparison with C/C++**:

- In C/C++, you can declare an array with a fixed size without dynamic allocation.
- In Java, all arrays are dynamically allocated using `new`.

📌 Array Indexing

- Array indices start at **0**.
- The last index is `arrayRefVar.length - 1`.
- Access elements using:

```
arrayRefVar[index]
```

⚠ Accessing an invalid index throws `ArrayIndexOutOfBoundsException`.

📌 Common Array Operations/Methods

Operation	Code Example
Get length	`int len = arr.length;`
Initialize with values	`int[] arr = {1, 2, 3};`
Loop through	`for (int i = 0; i < arr.length; i++)`
Sum elements	`total += arr[i];`
Find max/min	Compare each element with a running max/min
Shuffle	Swap elements using random indices
Shift elements	Use a temporary variable and loop

📌 Default Values for Array Elements

Data Type	Default Value
`int`, `byte`, `short`, `long`	`0`
`float`, `double`	`0.0`
`char`	`\u0000`
`boolean`	`false`
Reference types	`null`

📌 For-Each Loop (Enhanced For Loop)

```
for (elementType element : arrayRefVar) {
    // Process element
}
```

Example:

```
for (double num : myList) {
    System.out.println(num);
}
```

- Useful for reading elements (no index access).
- Cannot modify elements using this loop.

📌 Copying Arrays

❌ Wrong Way (Copying Reference):

```
int[] list1 = {1, 2, 3};
int[] list2 = list1; // Both point to the same array
```

☑ Correct Ways:

1. Using a Loop:

```
int[] target = new int[source.length];
for (int i = 0; i < source.length; i++) {
    target[i] = source[i];
}
```

2. Using `System.arraycopy()`:

```
System.arraycopy(sourceArray, sourceStartIndex, destinationArray,
destinationStartIndex, numberOfElementsToCopy);
```

3. Using `clone()`:

```
int[] target = source.clone();
```

Here's a more detailed **Markdown explanation** expanding on **Stack vs Heap in Java**:

📌 Stack vs Heap in Java

Java divides memory into two main regions: **Stack** and **Heap**. They work together to manage data efficiently when a program runs.

🗂️ Stack Memory

- Stores:
  - **Primitive variables** (e.g., `int`, `char`, `boolean`, etc.).
  - **References (pointers)** to objects in the heap.
  - **Method calls** (each method gets its own stack frame).
- **LIFO (Last In, First Out)** structure.
- Memory is automatically freed when a method ends.
- **Faster access** than heap.
- Size is limited → `StackOverflowError` if it grows too big (e.g., deep recursion).

☑ Example:

```
public class Main {
    public static void main(String[] args) {
        int x = 10;               // x stored in stack
        String str = "Hello";     // reference in stack, object in string pool
(heap)
    }
}
```

Heap Memory

- Stores : all objects, arrays, and their contents (including instance variables).
- Accessed via references stored in stack.
- Managed by **Garbage Collector (GC)**.
- Larger in size than stack, but **slower access**.
- Memory remains until no reference points to it.

☑ Example:

```
public class Main {
    public static void main(String[] args) {
        int[] arr = new int[5];   // 'arr' reference in stack, array object in
heap
        arr[0] = 42;              // stored inside heap object
    }
}
```

Key Differences

Feature	Stack	Heap
Stores	Primitives + object references	Objects + instance variables
Access speed	Very fast	Slower
Size	Small, limited	Large, depends on JVM settings
Lifetime	Ends with method execution	Until GC removes unused objects
Management	Automatic (method scope)	Managed by Garbage Collector
Error	StackOverflowError	OutOfMemoryError

Visualization

Stack (method calls + primitives + references)	Heap (objects)
int x = 10; (value 10)	[Object1]
String str → (ref) ---------->	"Hello"
int[] arr → (ref) ---------->	[0, 0, 0, 0, 0]

👉 This separation helps Java balance **speed (stack)** with **flexibility and memory management (heap)**.

📌 Returning Arrays from Methods

Yes, methods can return arrays. The reference to the array is returned.

Example:

```
public static int[] reverse(int[] list) {
    int[] result = new int[list.length];
    // ... copy in reverse
    return result;
}
```

📌 Variable-Length Argument Lists

Use `...` to indicate variable arguments:

```
public static void printMax(double... numbers) {
    // numbers is treated as double[]
}
```

Call with:

```
printMax(1.0, 2.0, 3.0);
printMax(new double[]{1.0, 2.0, 3.0});
```

📌 Searching Algorithms

Linear Search:

- Checks each element one by one.
- Works on sorted and unsorted arrays.
- Time complexity: O(n)

Binary Search:

- Requires a sorted array.
- Divides the array in half each time.
- Time complexity: O(log n)

Other Search Methods:

- Hashing, tree-based searches (covered later).

📌 Sorting Algorithms

Selection Sort:

- Finds the smallest element and swaps it with the first, then repeats.
- Time complexity: $O(n^2)$

Bubble Sort:

- Compares adjacent elements and swaps if necessary.
- Time complexity: $O(n^2)$

Other Sorting Methods:

- `Arrays.sort()`: Uses optimized algorithm (e.g., TimSort).
- `Arrays.parallelSort()`: For multi-core systems.

📌 `Arrays.toString()` Method

Converts an array to a readable string:

```
int[] arr = {1, 2, 3};
System.out.println(Arrays.toString(arr)); // Output: [1, 2, 3]
```

📌 Using `main(String[] args)`

- Command-line arguments are passed as an array of strings.
- Example:

  ```
  java MyProgram arg1 arg2
  ```

- In code:

  ```
  public static void main(String[] args) {
      for (String arg : args) {
          System.out.println(arg);
      }
  }
  ```