

# Java Arrays

## 1. Declaring an Array

```
int[] numbers;
```

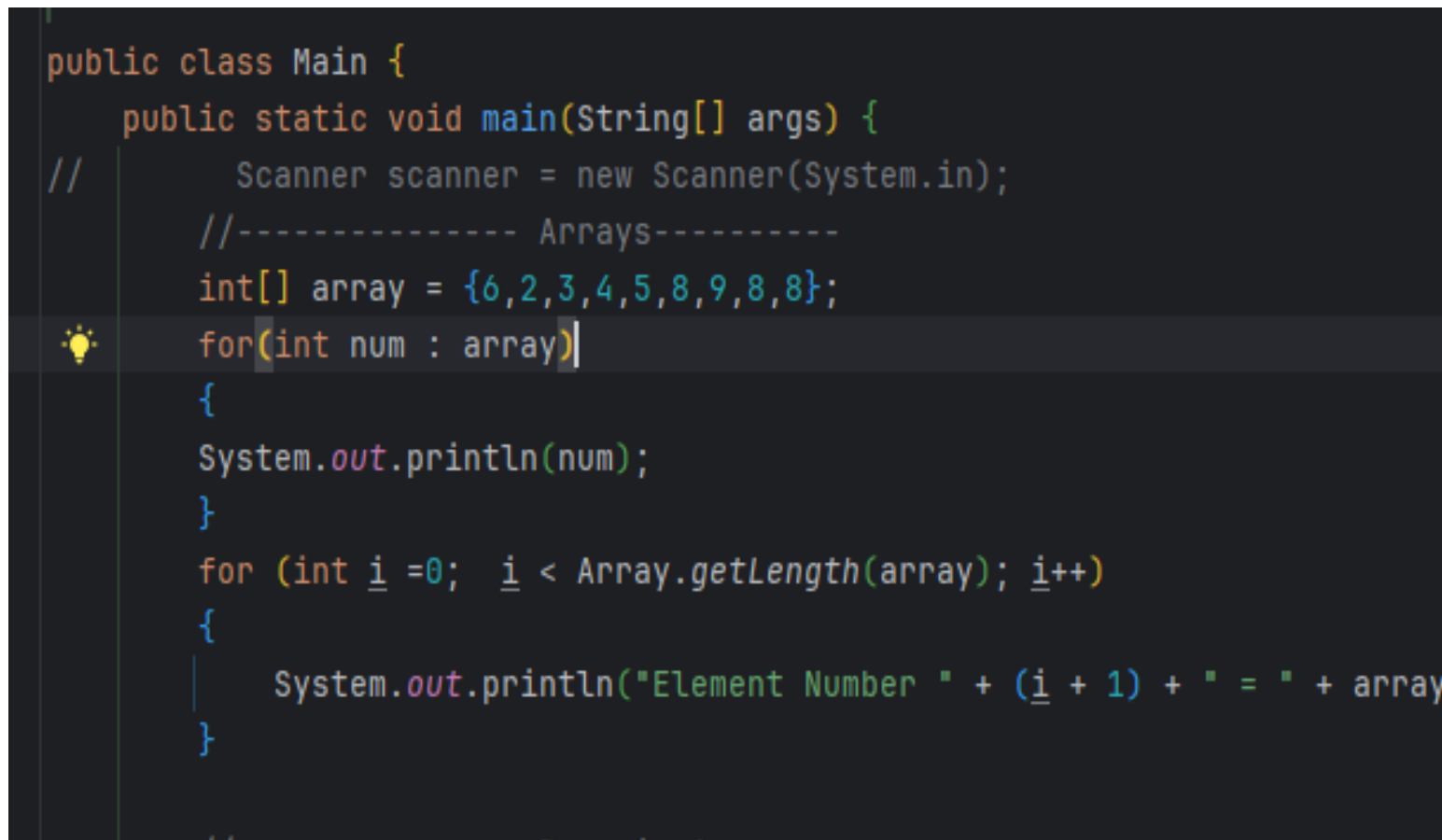
## 2. Initializing an Array

### *a. Specify the size*

```
int[] numbers = new int[5]; // Creates an array of size 5 with default values (0 for int)
```

### *b. Initialize with values*

```
int[] numbers = {10, 20, 30, 40, 50}; // below is a screenshot from my code
```

A screenshot of a code editor with a dark background. The code is in Java and shows a class named 'Main' with a 'main' method. Inside the 'main' method, there is a comment '// Scanner scanner = new Scanner(System.in);' followed by another comment '//----- Arrays-----'. Then, an array is declared and initialized: 'int[] array = {6,2,3,4,5,8,9,8,8};'. Below this, there is a 'for' loop using a range-based syntax: 'for(int num : array)'. Inside this loop, 'System.out.println(num);' is called. After the first loop, there is another 'for' loop using an index: 'for (int i =0; i < Array.getLength(array); i++)'. Inside this second loop, 'System.out.println("Element Number " + (i + 1) + " = " + array[i]);' is called. The code is color-coded: keywords are orange, comments are green, strings are pink, and identifiers/variables are white or light blue. A yellow lightbulb icon is visible in the left margin next to the first 'for' loop.

```
public class Main {  
    public static void main(String[] args) {  
        // Scanner scanner = new Scanner(System.in);  
        //----- Arrays-----  
        int[] array = {6,2,3,4,5,8,9,8,8};  
        for(int num : array)  
        {  
            System.out.println(num);  
        }  
        for (int i =0; i < Array.getLength(array); i++)  
        {  
            System.out.println("Element Number " + (i + 1) + " = " + array[i]);  
        }  
    }  
}
```

## Implementing Dynamic Arrays

A **dynamic array** is a resizable array that grows as needed when new elements are added. Unlike a fixed-size array, which has a set capacity, a dynamic array can expand or shrink during runtime.

### How It Works

#### 1. Start with a Fixed Array

- a. Create an array with an initial capacity.

#### 2. Resize When Full

- a. If the array is full and a new element is added, create a **new larger array** (usually double the size).
- b. Copy existing elements to the new array.
- c. Add the new element.

#### 3. Shrink When Necessary

- a. If too many elements are removed, the array can shrink to save memory.

### Implementation in Java

```
21
22 // ----- Dynamic Arrays -----
23 String userInput;
24 ArrayList<Integer> dynamicArray = new ArrayList<>();
25 while (true) {
26     System.out.println("Please Enter the array element value:");
27     int num = scanner.nextInt();
28     dynamicArray.add(num);
29
30     while (true) {
31         System.out.println("Would you like to add another element? (y/n)");
32         userInput = scanner.next().trim().toLowerCase();
33
34         if (userInput.equals("y") || userInput.equals("n")) {
35             break;
36         }
37         System.out.println("Invalid input. Please enter 'y' for Yes or 'n' for No.");
38     }
39
40     if (userInput.equals("n")) {
41         break;
42     }
43 }
44
45 System.out.println("Elements in the dynamic array:");
46 for (int number : dynamicArray) {
47     System.out.println(number);
48 }
49 scanner.close();
50
51 // ----- Linked List -----
```

## Implementing a Linked List

A **linked list** is a data structure that stores elements in **nodes**, where each node contains a value and a reference (or link) to the next node. Unlike arrays, linked lists do not use continuous memory and can grow dynamically.

### How It Works

#### 1. Nodes and Pointers

- a. Each node has two parts:
  - i. **Data** (stores the value)
  - ii. **Next** (points to the next node)

#### 2. Insertion

- a. To add a new node, adjust the links so the new node connects to the list.

#### 3. Deletion

- a. To remove a node, update the previous node's pointer to skip the deleted node.

### Implementation in Java

```
1 public class Node { 7 usages
2     int data; 4 usages
3     Node next; 13 usages
4     public Node(int data) 1 usage
5     {
6         this.data=data;
7         this.next=null;
8     }
9 }
```

نظرا لأن التعامل مع برنامج الورد طلع صعب بقيت الكود في الصفحة الثانية

```

1 public class LinkedList { 2 usages
2     private Node head; 9 usages
3     public void insert(int data) { 1 usage
4         Node newNode = new Node(data);
5         if (head == null) {
6             head = newNode;
7             return;
8         }
9         Node temp = head;
10        while (temp.next != null) {
11            temp = temp.next;
12        }
13        temp.next = newNode;
14    }
15    public void delete(int data){ 1 usage
16        if (head==null) return;
17        if(head.data == data){
18            head = head.next;
19            return;
20        }
21        Node temp = head;
22        while(temp.next != null && temp.next.data != data){
23            temp = temp.next;
24        }
25        if(temp.next != null){
26            temp.next= temp.next.next;
27        }
28    }
29    public void display() { 1 usage
30        Node temp = head;
31        while (temp != null) {
32            System.out.print(temp.data + " -> ");
33            temp = temp.next;
34        }
35        System.out.println("null");
36    }
37 }

```

## Implementing a Stack

A **stack** is a data structure that follows the **Last In, First Out (LIFO)** principle, meaning elements are added and removed from the same end (**top**).

### How It Works

1. **Push (Insertion)** – Add an element to the top.
2. **Pop (Removal)** – Remove the top element.
3. **Peek (Top Element)** – View the top element without removing it.

### Implementation

```
1  class Stack { no usages
2      private Node top; 6 usages
3
4  ✓  public void push(int data) { no usages
5      Node newNode = new Node(data);
6      newNode.next = top;
7      top = newNode;
8  }
9
10  ✓  public void pop() { no usages
11  ✓      if (top == null) {
12          System.out.println("Stack is empty");
13          return;
14      }
15      top = top.next;
16  }
17
18  ✓  public void display() { no usages
19      Node temp = top;
20  ✓      while (temp != null) {
21          System.out.print(temp.data + " -> ");
22          temp = temp.next;
23      }
24      System.out.println("null");
25  }
26  }
```

## Implementing a Queue

A **queue** is a data structure that follows the **First In, First Out (FIFO)** principle, meaning elements are added from one end (**rear**) and removed from the other (**front**).

### How It Works

1. **Enqueue (Insertion)** – Add an element to the rear.
2. **Dequeue (Removal)** – Remove an element from the front.
3. **Peek (Front Element)** – View the front element without removing it.

### Implementation

```
1  class Queue { no usages
2      private Node front, rear; 6 usages
3
4      public void enqueue(int data) { no usages
5          Node newNode = new Node(data);
6          if (rear == null) {
7              front = rear = newNode;
8              return;
9          }
10         rear.next = newNode;
11         rear = newNode;
12     }
13
14     public void dequeue() { no usages
15         if (front == null) {
16             System.out.println("Queue is empty");
17             return;
18         }
19         front = front.next;
20         if (front == null) rear = null;
21     }
22
23     public void display() { no usages
24         Node temp = front;
25         while (temp != null) {
26             System.out.print(temp.data + " <- ");
27             temp = temp.next;
28         }
29         System.out.println("null");
30     }
31 }
```

## Implementing a Map

A **map** is a data structure that stores key-value pairs, allowing efficient retrieval of values using unique keys. It is useful for scenarios where fast lookups, insertions, and deletions are needed.

### *How It Works*

- **Put (Insertion):** Add a key-value pair.
- **Get (Retrieval):** Retrieve a value using a key.
- **Remove (Deletion):** Delete a key-value pair.
- **Contains (Check):** Verify if a key exists.

### *Implementation in Java*

Using **HashMap**:

```
61 //----- Maps -----
62 HashMap<String, Integer> map = new HashMap<>();
63
64 // Inserting key-value
65 map.put("Noah", 25);
66 map.put("Mohamed", 30);
67 map.put("Ahmed", 22);
68
69 // Retrieving a value
70 System.out.println("Noah's age: " + map.get("Noah")); // Output: 25
71
72 // Removing a key-value pair
73 map.remove(key: "Ahmed");
74
75 // Checking if a key exists
76 System.out.println("Contains Mohamed? " + map.containsKey("Mohamed")); // Outp
77
78 // Iterating through the map
79 for (String key : map.keySet()) {
80     System.out.println(key + " -> " + map.get(key));
81 }
```