Java Day9

📦 What Is an Array?

An **array** is a collection of elements of the same data type stored in a **contiguous block of memory**. It allows you to store and access multiple values using a single variable name and index positions.

✅ Key Features

- Fixed size (declared at creation)
- Indexed access (starts from 0)
- Homogeneous elements (same type)

🧪 Declaring and Initializing Arrays

🔷 Declaration

java

`int[] numbers; // declares an array of integers`

🔷 Initialization

java

`numbers = new int[5]; // allocates memory for 5 integers`

🔷 Combined Declaration + Initialization

java

`int[] numbers = {10, 20, 30, 40, 50};`

🔍 Accessing Array Elements

Use index positions to read or update values:

java

```
System.out.println(numbers[2]); // prints 30
numbers[4] = 100; // updates last element
```

🧠 Memory Layout

- Arrays are stored in **heap memory**
- The reference (variable name) is stored in the **stack**
- Each element is placed in a continuous memory block

Example Diagram:

Code

```
Stack:        Heap:
numbers ───────▶ [10][20][30][40][50]
```

🔁 Looping Through Arrays

🔷 Using for loop

java

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

🔷 Using enhanced for loop

java

```java
for (int num : numbers) {
  System.out.println(num);
}
```

⚠️ Common Mistakes

- Accessing out-of-bound index → `ArrayIndexOutOfBoundsException`
- Forgetting `.length` gives array size, not a method → `numbers.length`, not `numbers.length()`

🧠 What Is Time Complexity?

Time complexity measures how the runtime of an algorithm grows as the input size increases. It helps us predict performance and choose the best algorithm for a given task.

- Input size is usually represented as **n**
- Time complexity is written using **Big O notation**, which describes the upper bound of runtime growth

🔢 Why Time Complexity Matters

- Helps compare algorithms before coding
- Predicts how code will perform with large inputs
- Avoids slow or inefficient solutions in real-world applications

⏱️ Common Time Complexities (Big O Notation)

Big O	Meaning	Example Use Case
$O(1)$	Constant time	Accessing an array element
$O(n)$	Linear time	Looping through an array
$O(n^2)$	Quadratic time	Nested loops (e.g., Bubble Sort)
$O(\log n)$	Logarithmic time	Binary Search
$O(n \log n)$	Linearithmic time	Merge Sort, Quick Sort
$O(2^n)$	Exponential time	Recursive Fibonacci
$O(n!)$	Factorial time	Brute-force permutations

🔍 How to Analyze Time Complexity

1. **Count the number of operations**

- Focus on loops, recursive calls, and input-dependent steps

2. **Ignore constants**

- $O(2n)$ becomes $O(n)$, $O(3n + 5)$ becomes $O(n)$

3. **Keep the dominant term**

- $O(n^2 + n)$ becomes $O(n^2)$

4. **Look at worst-case scenario**

- Time complexity usually refers to the worst case unless stated otherwise

📊 Examples with Code

🔷 Example 1: Constant Time — O(1)

java

```
int x = arr[5]; // Direct access
```

🔷 Example 2: Linear Time — O(n)

java

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

🔷 Example 3: Quadratic Time — O(n²)

java

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    System.out.println(i + j);
  }
}
```

🔷 Example 4: Logarithmic Time — O(log n)

java

```
int binarySearch(int[] arr, int target) {
  int low = 0, high = arr.length - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    if (arr[mid] == target) return mid;
    else if (arr[mid] < target) low = mid + 1;
    else high = mid - 1;
  }
  return -1;
}
```

🔍 1. Searching Techniques

🔷 Linear Search — O(n)

Search each element one by one until the target is found.

```
for (int i = 0; i < arr.length; i++) {
   if (arr[i] == target) {
      return i;
   }
}
```

🔷 Binary Search — O(log n)

Works only on **sorted arrays**. Divides the array in half repeatedly.

```
int low = 0, high = arr.length - 1;
while (low <= high) {
    int mid = (low + high) / 2;
    if (arr[mid] == target) return mid;
    else if (arr[mid] < target) low = mid + 1;
    else high = mid - 1;
}
```

🔄 2. Rotation Techniques

🔷 Right Rotation by 3

- Brute force approach
- Reversal approach

Rotation of an Array using brute force approach

```
package DSA.Array.Day9;

import java.util.Arrays;

public class RightRotation {
    public static void main(String[] args) {
        int[] nums={1,2,3,4,5,6,7};
        int k=2;
        for(int i=0;i<k;i++){
            int j;
            int temp=nums[nums.length-1];
            for(j=nums.length-1;j>0;j--){
                nums[j]=nums[j-1];
            }
            nums[0]=temp;
        }
        System.out.println(Arrays.toString(nums));
```

```
    }
}
```

Rotation of an Array using Reversal Technique

```
package DSA.Array.Day9;

import java.util.Arrays;

public class RightRotationUsingReversal {
    public static void main(String[] args) {
        int[] nums={1,2,3,4,5,6,7};
        int k=3;
        rotate(nums,k);
        System.out.println(Arrays.toString(nums));
    }
    public static void rotate(int[] nums,int k){
        int n=nums.length;
        k=k%n;
        reverse(nums,0,n-1);
        reverse(nums,0,k-1);
        reverse(nums,k,n-1);
    }
    public static void reverse(int[] nums,int start,int end){
        while(start<end){
            int temp=nums[start];
            nums[start]=nums[end];
            nums[end]=temp;
            start++;
            end--;
        }
    }
}
```

🔷 Left Rotation by 3

- Brute force approach

- Reversal approach

Rotation of an Array using brute force approach

```
package DSA.Array.Day9;

import java.util.Arrays;

public class LeftRotation {
    public static void main(String[] args) {
        int[] nums={1,2,3,4,5,6,7};
        int k=2;
        for(int i=0;i<k;i++){
            int j;
            int temp=nums[0];
            for(j=0;j<nums.length-1;j--){
                nums[j]=nums[j+1];
            }
            nums[nums.length-1]=temp;
        }
        System.out.println(Arrays.toString(nums));
    }
}
```

Rotation of an Array using Reversal Technique

```
package DSA.Array.Day9;

import java.util.Arrays;

public class LeftRotationUsingReversal {
    public static void main(String[] args) {
        int[] nums={1,2,3,4,5,6,7};
        int k=3;
        rotate(nums,k);
        System.out.println(Arrays.toString(nums));
    }
```

```
    public static void rotate(int[] nums,int k){
        int n=nums.length;
        k=k%n;
        reverse(nums,0,k-1);
        reverse(nums,k,n-1);
        reverse(nums,0,n-1);
    }
    public static void reverse(int[] nums,int start,int end){
        while(start<end){
            int temp=nums[start];
            nums[start]=nums[end];
            nums[end]=temp;
            start++;
            end--;
        }
    }
}
```