**Arrays**

**Definition of Arrays**

* An array is a collection of elements stored at contiguous memory locations.
* It can store elements of the same data type (primitive or objects).
* Arrays in Java are zero-indexed, meaning the first element is at index 0.
* Arrays are stored in heap memory. When declared, they are allocated contiguous memory blocks. **Heap vs. Stack Memory**: Arrays are stored in the heap, but references are on the stack. **Garbage Collection**: Setting large arrays to null can help free up memory.

**2. Types of Arrays**

* **Single-Dimensional Array:** int[] arr = new int[5];
* **Multi-Dimensional Array:** int[][] arr = new int[3][4];
* **Jagged Array:** Arrays with rows of varying lengths.

**3. Array Declaration & Initialization**

// Declaration

int[] arr;

// Initialization

arr = new int[5];

// Declaration and Initialization in one step

int[] arr = {1, 2, 3, 4, 5};

// Multi-dimensional array

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

**4. Array Limitations**

* **Fixed Size:** Once declared, the size cannot be changed.
* **Homogeneous Data:** Can store only elements of the same type.
* **Memory Waste:** Can lead to memory wastage if the array size is overestimated.

**5. When to Use Arrays in DSA?**

* **Static Data Requirements:** When the data size is known in advance.
* **Fast Access:** When quick access to elements using indices is required (O(1) time complexity).
* **Memory Efficiency:** When memory allocation needs to be compact and contiguous.

**6. Good Practices with Arrays**

* Always check array bounds to avoid ArrayIndexOutOfBoundsException.
* Initialize arrays properly to avoid NullPointerException.
* Prefer ArrayList when a dynamic array is needed.

**Properties of Arrays in Java**

Arrays in Java are objects that store multiple values of the same data type. They offer several important properties and characteristics:

**1. Fixed Size**

* Once an array is created, its size is fixed and cannot be changed.
* Example: An array of size 5 can only hold 5 elements.

int[] arr = new int[5]; // An array of size 5

**2. Homogeneous Elements**

* Arrays can only store elements of the same data type (e.g., int, String, Object).
* Example: An array of integers (int[]) can only contain integers.

String[] names = {"Alice", "Bob", "Charlie"};

**3. Zero-Based Indexing**

* Elements in an array are stored in contiguous memory locations and can be accessed using indices.
* The first element is accessed with index 0, the second with index 1, and so on (e.g., array[0] for the first element).
* Allows for fast access using the index (O(1) time complexity).
* Example:

int[] arr = {10, 20, 30};

System.out.println(arr[0]); // Output: 10

**4. Contiguous Memory Allocation**

* Array elements are stored in consecutive memory locations, allowing fast index-based access (O(1) complexity).

**5. Mutable Elements**

* Elements of an array can be modified.
* Example:

int[] arr = {1, 2, 3};

arr[1] = 5; // Now the array is {1, 5, 3}

**6. Default Values**

* Arrays in Java are automatically initialized with default values:
  + int, byte, short, long → 0
  + float, double → 0.0
  + char → '\u0000'
  + boolean → false
  + Object references → null

int[] arr = new int[3];

System.out.println(arr[0]); // Output: 0 (default value)

**7. Supports Multidimensional Arrays**

* Java supports multidimensional arrays, often used as matrices.
* Example:

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

System.out.println(matrix[1][1]); // Output: 4

**8. Length Property**

* The length of an array can be accessed using the .length attribute.

int[] arr = {1, 2, 3, 4};

System.out.println(arr.length); // Output: 4

**9. Arrays are Objects**

* In Java, arrays are treated as objects and stored on the heap.

int[] arr = new int[3];

System.out.println(arr.getClass().getName()); // Output: [I

**10. Array Initialization Techniques**

* **Static Initialization:** When elements are directly assigned.

int[] arr = {1, 2, 3, 4, 5};

* **Dynamic Initialization:** When size is given, but elements are added later.

int[] arr = new int[5];

arr[0] = 10;

**11. Traversal Techniques**

Arrays can be easily traversed using loops (e.g., for, while, forEach).

* **Using For Loop:**

int[] arr = {10, 20, 30};

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

System.out.println(arr[i]);

}

* **Using Enhanced For Loop:**

for (int num : arr) {

System.out.println(num);

}

* **Using Streams (Java 8+):**

Arrays.stream(arr).forEach(System.out::println);

**12. Cloning Arrays**

* Java allows shallow cloning of arrays using the .clone() method.

int[] original = {1, 2, 3};

int[] clone = original.clone();

System.out.println(Arrays.equals(original, clone)); // Output: true

**13. Common Methods in java.util.Arrays Class**

import java.util.Arrays;

int[] arr = {3, 1, 2};

Arrays.sort(arr); // Sorts the array

System.out.println(Arrays.toString(arr)); // Output: [1, 2, 3]

int index = Arrays.binarySearch(arr, 2); // Searches for element 2

System.out.println(index); // Output: 1

int[] filledArray = new int[5];

Arrays.fill(filledArray, 10); // Fills array with 10

System.out.println(Arrays.toString(filledArray)); // Output: [10, 10, 10, 10, 10]

**14. Boundary Checking**

* Java performs automatic boundary checking.
* Accessing an invalid index throws an ArrayIndexOutOfBoundsException.

int[] arr = {1, 2, 3};

System.out.println(arr[5]); // Throws Exception

**15. Memory Efficiency**

* Arrays provide memory efficiency, especially for primitive data types.

**16. Limitation of Fixed Size**

* Once created, the size of the array cannot be increased or decreased.
* For dynamic resizing, ArrayList or other collection classes are recommended.

**🔥 Conclusion**

Arrays in Java are powerful and efficient for storing a fixed-size collection of homogeneous elements. They provide fast access to elements via indices and support both primitive and object data types. However, their fixed size can sometimes be a limitation, making dynamic collections like ArrayList a better choice in certain scenarios.

The **java.util.Arrays** class in Java provides a variety of static utility methods to manipulate arrays. These methods are helpful for sorting, searching, comparing, filling, and converting arrays, among other tasks. Below are some of the most commonly used methods:

**1. Sorting Arrays**

**👉 Arrays.sort()**

* **Description:** Sorts the elements of an array in ascending order.
* **Supported Types:** int[], double[], String[], and more.
* **Time Complexity:** O(n log n) for primitive types, O(n log n) for objects using Comparable or Comparator.

**parallelSort(array)**: Sorts the array **in parallel** (faster for large arrays).

import java.util.Arrays;

int[] numbers = {5, 3, 8, 1};

Arrays.sort(numbers);

System.out.println(Arrays.toString(numbers)); // Output: [1, 3, 5, 8]

String[] names = {"Charlie", "Alice", "Bob"};

Arrays.sort(names);

System.out.println(Arrays.toString(names)); // Output: [Alice, Bob, Charlie]

// Parallel sort (Java 8+)

int[] moreNumbers = {10, 7, 2, 6};

Arrays.parallelSort(moreNumbers);

System.out.println(Arrays.toString(moreNumbers)); // [2, 6, 7, 10]

**2. Searching Elements**

**👉 Arrays.binarySearch()**

**binarySearch(array, key)**

* **Description:** Searches for an element in a **sorted array** using the binary search algorithm.
* **Returns:** The index of the element if found; otherwise, (-insertionPoint - 1).

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

int index = Arrays.binarySearch(arr, 30);

System.out.println(index); // Output: 2

int notFound = Arrays.binarySearch(arr, 25);

System.out.println(notFound); // Output: -3 (insertion point is 2)

**3. Filling Arrays**

**👉 Arrays.fill()**

**fill(array, value)**

* **Description:** Fills all elements of an array with the specified value.
* **Overloaded Methods:** Fill a specific range of the array.

**fill(array, fromIndex, toIndex, value) :** Fills a **range** of the array.

int[] arr = new int[5];

Arrays.fill(arr, 7);

System.out.println(Arrays.toString(arr)); // Output: [7, 7, 7, 7, 7]

int[] rangeArr = {1, 2, 3, 4, 5};

Arrays.fill(rangeArr, 1, 4, 9); // Fills indices 1 to 3 with 9

System.out.println(Arrays.toString(rangeArr)); // Output: [1, 9, 9, 9, 5]

**4. Comparing Arrays**

**👉 Arrays.equals()**

**equals(array1, array2)**

* **Description:** Compares two arrays for equality. Returns true if both arrays have the same elements in the same order.
* **Supports:** int[], Object[], String[], etc.

int[] arr1 = {1, 2, 3};

int[] arr2 = {1, 2, 3};

System.out.println(Arrays.equals(arr1, arr2)); // Output: true

int[] arr3 = {3, 2, 1};

System.out.println(Arrays.equals(arr1, arr3)); // Output: false

**👉 Arrays.deepEquals()**

**deepEquals(array1, array2)**

* **Description:** Used for comparing multidimensional arrays (e.g., Object[][]).

Integer[][] arr1 = {{1, 2}, {3, 4}};

Integer[][] arr2 = {{1, 2}, {3, 4}};

System.out.println(Arrays.deepEquals(arr1, arr2)); // Output: true

**5. Copying Arrays**

**👉 Arrays.copyOf()**

**copyOf(original, newLength)**

* **Description:** Copies the specified array, truncating or padding with default values if necessary.

int[] original = {1, 2, 3};

int[] copy = Arrays.copyOf(original, 5);

System.out.println(Arrays.toString(copy)); // Output: [1, 2, 3, 0, 0]

**👉 Arrays.copyOfRange()**

* **Description:** Copies a specified range of the array.

copyOfRange(original, from, to)

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

int[] rangeCopy = Arrays.copyOfRange(original, 1, 4);

System.out.println(Arrays.toString(rangeCopy)); // Output: [20, 30, 40]

**6. Converting Arrays to Strings**

**👉 Arrays.toString()**

**toString(array)**

* **Description:** Converts an array to a String representation (for one-dimensional arrays).

int[] arr = {1, 2, 3};

System.out.println(Arrays.toString(arr)); // Output: [1, 2, 3]

**👉 Arrays.deepToString()**

**deepToString(array)**

* **Description:** Converts a multidimensional array to a String representation.

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

System.out.println(Arrays.deepToString(matrix)); // Output: [[1, 2], [3, 4]]

**7. Parallel Sorting (Java 8+)**

**👉 Arrays.parallelSort()**

* **Description:** Sorts the array using parallel sorting (multi-threaded sorting for better performance with large arrays).

int[] arr = {5, 2, 8, 3, 1};

Arrays.parallelSort(arr);

System.out.println(Arrays.toString(arr)); // Output: [1, 2, 3, 5, 8]

**8. Creating Streams (Java 8+)**

**👉 Arrays.stream()**

* **Description:** Converts an array into a Stream for functional-style operations.

int[] arr = {1, 2, 3, 4, 5};

Arrays.stream(arr)

.filter(x -> x % 2 == 0)

.forEach(System.out::println); // Output: 2 4

**9. Array Mismatch**

**👉 Arrays.mismatch() (Java 9+)**

**mismatch(array1, array2)**

* **Description:** Returns the index of the first mismatch between two arrays, or -1 if no mismatch is found.

int[] arr1 = {1, 2, 3};

int[] arr2 = {1, 4, 3};

System.out.println(Arrays.mismatch(arr1, arr2)); // Output: 1

**10. Set All Elements**

**👉 Arrays.setAll()**

* **Description:** Sets all elements of the array using a generator function.

int[] arr = new int[5];

Arrays.setAll(arr, i -> i \* i);

System.out.println(Arrays.toString(arr)); // Output: [0, 1, 4, 9, 16]

**11. Array Hash Code**

**👉 Arrays.hashCode()**

* **Description:** Returns a hash code for the array.

int[] arr = {1, 2, 3};

System.out.println(Arrays.hashCode(arr)); // Output: Hash code (integer)

**12. Array Spliterator (Java 8+)**

**👉 Arrays.spliterator()**

* **Description:** Returns a Spliterator for array traversal, useful in parallel processing.

int[] arr = {10, 20, 30};

Arrays.spliterator(arr).forEachRemaining(System.out::println);

// Output: 10 20 30

**Other Utility Methods**

* **hashCode(array)**: Returns a **hash code** of the array.
* **setAll(array, generator)**: Populates the array using a **lambda expression**.
* **spliterator(array)**: Returns a **spliterator** for parallel processing.
* **stream(array)**: Converts an array into a **Stream** (Java 8+).

**🔥 Conclusion**

The java.util.Arrays class is a powerful utility for array manipulation in Java. It simplifies tasks like sorting, searching, filling, and copying arrays, providing robust and efficient methods. By leveraging these built-in methods, developers can write cleaner and more maintainable code.

**Java Streams and ArrayList Methods**

**1. Stream Methods**

**Stream Creation Methods**

* Stream.of(T... values) - Creates a stream from values.
* Stream.ofNullable(T t) - Returns a stream containing a single element, or an empty stream if the element is null.
* Arrays.stream(T[] array) - Creates a stream from an array.
* Collection.stream() - Returns a sequential stream from a collection.
* Collection.parallelStream() - Returns a parallel stream from a collection.
* Stream.generate(Supplier<T> s) - Creates an infinite sequential unordered stream.
* Stream.iterate(T seed, UnaryOperator<T> f) - Generates an infinite stream using a function.

**Intermediate Operations**

* filter(Predicate<T> predicate) - Filters elements based on a condition.
* map(Function<T, R> mapper) - Transforms elements.
* flatMap(Function<T, Stream<R>> mapper) - Flattens nested streams.
* distinct() - Removes duplicate elements.
* sorted() - Sorts elements naturally.
* sorted(Comparator<T> comparator) - Sorts elements using a custom comparator.
* limit(long maxSize) - Truncates the stream to a given size.
* skip(long n) - Skips the first n elements.
* peek(Consumer<T> action) - Applies an action to each element without modifying the stream.

**Terminal Operations**

* collect(Collector<T, A, R> collector) - Converts the stream to a collection.
* forEach(Consumer<T> action) - Iterates over elements.
* forEachOrdered(Consumer<T> action) - Maintains encounter order for parallel streams.
* toArray() - Converts the stream to an array.
* reduce(BinaryOperator<T> accumulator) - Combines elements to produce a single result.
* reduce(T identity, BinaryOperator<T> accumulator) - Reduces elements with an initial value.
* findFirst() - Returns the first element, if present.
* findAny() - Returns any element (useful in parallel streams).
* allMatch(Predicate<T> predicate) - Checks if all elements match the condition.
* anyMatch(Predicate<T> predicate) - Checks if any element matches the condition.
* noneMatch(Predicate<T> predicate) - Checks if no elements match the condition.
* count() - Returns the count of elements.
* min(Comparator<T> comparator) - Returns the minimum element.
* max(Comparator<T> comparator) - Returns the maximum element.

**Specialized Streams**

* IntStream, LongStream, DoubleStream - Primitive streams with additional methods.

**2. ArrayList Methods**

**Creation and Initialization**

* ArrayList<E> list = new ArrayList<>() - Creates an empty list.
* ArrayList<E> list = new ArrayList<>(Collection<? extends E> c) - Creates a list from a collection.
* ArrayList<E> list = new ArrayList<>(int initialCapacity) - Specifies initial capacity.

**Basic Operations**

* add(E e) - Adds an element to the list.
* add(int index, E element) - Adds an element at a specific index.
* addAll(Collection<? extends E> c) - Adds all elements from a collection.
* addAll(int index, Collection<? extends E> c) - Adds elements at a specific index.
* get(int index) - Retrieves an element by index.
* set(int index, E element) - Replaces an element at the specified index.
* remove(int index) - Removes the element at the specified index.
* remove(Object o) - Removes the first occurrence of the specified element.
* removeAll(Collection<?> c) - Removes elements present in the specified collection.
* clear() - Removes all elements.
* contains(Object o) - Checks if the list contains a specified element.
* isEmpty() - Checks if the list is empty.
* size() - Returns the number of elements in the list.
* indexOf(Object o) - Returns the index of the first occurrence of the specified element.
* lastIndexOf(Object o) - Returns the index of the last occurrence of the specified element.

**Iterating and Sorting**

* forEach(Consumer<? super E> action) - Iterates over elements.
* sort(Comparator<? super E> c) - Sorts the list using a comparator.
* iterator() - Returns an iterator.
* listIterator() - Returns a list iterator.
* listIterator(int index) - Returns a list iterator starting at the specified index.

**Conversion**

* toArray() - Converts the list to an array.
* toArray(T[] a) - Converts the list to an array of the specified type.

**Bulk Operations**

* retainAll(Collection<?> c) - Retains only elements present in the specified collection.
* subList(int fromIndex, int toIndex) - Returns a view of the portion of the list.
* replaceAll(UnaryOperator<E> operator) - Replaces each element with the result of applying the operator.
* removeIf(Predicate<? super E> filter) - Removes elements based on a condition.

**Deep Dive into Arrays in Java**

**1. Advanced Array Operations**

**Searching Algorithms**

1. **Linear Search (O(n))**:

* Iterate through the array to find an element.
* Useful for unsorted arrays.

public static int linearSearch(int[] arr, int key) {

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

if (arr[i] == key) {

return i;

}

}

return -1;

}

1. **Binary Search (O(log n))**:

* Works only on sorted arrays.
* Repeatedly divides the array into halves to find the element.

public static int binarySearch(int[] arr, int key) {

int low = 0, high = arr.length - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

if (arr[mid] == key) return mid;

if (arr[mid] < key) low = mid + 1;

else high = mid - 1;

}

return -1;

}

**Sorting Algorithms**

1. **Bubble Sort (O(n^2))**:

**Quick Sort (O(n log n))**:

**Merge Sort (O(n log n))**:

* Repeatedly swaps adjacent elements if they are in the wrong order.

public static void bubbleSort(int[] arr) {

for (int i = 0; i < arr.length - 1; i++) {

for (int j = 0; j < arr.length - 1 - i; j++) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

1. **Selection Sort (O(n^2))**:

* Finds the minimum element and places it at the beginning.

public static void selectionSort(int[] arr) {

for (int i = 0; i < arr.length - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < arr.length; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

1. **Java Built-in Sorting (Arrays.sort())**:

* Uses Dual-Pivot Quicksort for primitives (O(n log n)).

import java.util.Arrays;

int[] arr = {3, 1, 4, 1, 5, 9};

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));

**Array Manipulation Techniques**

1. **Reversing an Array**:

public static void reverseArray(int[] arr) {

int start = 0, end = arr.length - 1;

while (start < end) {

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

1. **Rotating an Array (Cyclic Rotation)**:

public static void rotateArray(int[] arr, int k) {

int n = arr.length;

k = k % n;

reverse(arr, 0, n - 1);

reverse(arr, 0, k - 1);

reverse(arr, k, n - 1);

}

private static void reverse(int[] arr, int start, int end) {

while (start < end) {

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

**2. Common Array Interview Problems**

**1. Find the Maximum and Minimum Elements**

public static int findMax(int[] arr) {

int max = arr[0];

for (int num : arr) {

if (num > max) max = num;

}

return max;

}

public static int findMin(int[] arr) {

int min = arr[0];

for (int num : arr) {

if (num < min) min = num;

}

return min;

}

**2. Find Duplicates in an Array**

import java.util.HashSet;

import java.util.Set;

public static Set<Integer> findDuplicates(int[] arr) {

Set<Integer> duplicates = new HashSet<>();

Set<Integer> seen = new HashSet<>();

for (int num : arr) {

if (!seen.add(num)) {

duplicates.add(num);

}

}

return duplicates;

}

**3. Move Zeros to the End**

public static void moveZerosToEnd(int[] arr) {

int index = 0;

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

if (arr[i] != 0) {

arr[index++] = arr[i];

}

}

while (index < arr.length) {

arr[index++] = 0;

}

}

**4. Two Sum Problem (O(n))**

import java.util.HashMap;

import java.util.Map;

public static int[] twoSum(int[] arr, int target) {

Map<Integer, Integer> map = new HashMap<>();

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

int complement = target - arr[i];

if (map.containsKey(complement)) {

return new int[]{map.get(complement), i};

}

map.put(arr[i], i);

}

return new int[]{-1, -1};

}

**5. Sliding Window Technique:**

* **Usage**: Finding the maximum sum of a subarray of size k, or the longest substring without repeating characters.

int maxSumSubarray(int[] arr, int k) {

int maxSum = 0, windowSum = 0;

for (int i = 0; i < k; i++) {

windowSum += arr[i];

}

maxSum = windowSum;

for (int i = k; i < arr.length; i++) {

windowSum += arr[i] - arr[i - k];

maxSum = Math.max(maxSum, windowSum);

}

return maxSum;

}

6. **Two-Pointer Technique**:

* **Examples**: Finding pairs with a specific sum, sorting colors (Dutch National Flag problem), etc.

java

CopyEdit

boolean hasPairWithSum(int[] arr, int target) {

Arrays.sort(arr);

int left = 0, right = arr.length - 1;

while (left < right) {

int sum = arr[left] + arr[right];

if (sum == target) return true;

else if (sum < target) left++;

else right--;

}

return false;

}

**3. Important Points to Remember**

* **Time Complexity:** Understand O(1), O(n), O(log n), O(n^2).

**O(1) (Constant Time)**: Accessing an element by index (arr[5]) is O(1), regardless of array size.

**O(n) (Linear Time)**: Iterating through an array (for loop) takes O(n), where n is the array length.

**O(log n) (Logarithmic Time)**: Binary search in a sorted array. The search space is halved with each step.

**O(n^2) (Quadratic Time)**: Nested loops, like in Bubble Sort or Selection Sort.

* **Space Complexity:** How much additional space is required.

**In-Place Operations (O(1) Space)**: When the algorithm uses a constant amount of extra space regardless of input size (e.g., swapping elements in an array).

**Additional Space (O(n))**: When auxiliary data structures (e.g., additional arrays, hashmaps) are used.

**Trade-Off**: Space vs. time complexity often requires balancing (e.g., using a hashmap to reduce time complexity at the cost of extra space).

* **Edge Cases:** Empty arrays, arrays with one element, large arrays, arrays with all identical elements.

**Empty Arrays**: Ensure algorithms handle arr.length == 0 gracefully.

**Single Element Arrays**: Algorithms like sorting and searching should manage this without errors.

**Large Arrays**: Be mindful of performance (O(n) vs. O(log n)) and memory usage.

**Identical Elements**: Search or sort operations should handle arrays where all elements are the same.

* **Mutability:** Arrays are mutable, meaning elements can be changed without creating a new array.
* **Fixed Size:** Arrays have a fixed size, unlike dynamic collections such as ArrayList.

**4. Best Practices with Arrays in DSA**

1. **Avoid Magic Numbers:**

**What Are Magic Numbers?**: Hard-coded values in code, e.g., arr[100].

**Why Avoid Them?**: Makes code difficult to maintain and understand.

Instead of using arr[100], declare a constant: final int SIZE = 100; int[] arr = new int[SIZE];.

1. **Always Check Bounds:** Avoid ArrayIndexOutOfBoundsException by checking indices.
2. **Use Enhanced for Loop Where Possible:** Improves readability.
3. **Prefer Built-in Methods:**

**Why?**: Java's built-in methods are optimized and tested. Utilize Arrays.sort(), Arrays.copyOf(), and Arrays.binarySearch() for optimized performance.

1. **Understand Primitive vs. Reference Arrays:** int[] vs. String[].

**Primitive Arrays (int[], char[])**:

* Stores actual values in contiguous memory.

Example:

int[] arr = {1, 2, 3}; // Stores 1, 2, 3 directly in memory

**Reference Arrays (String[], Object[])**:

* Stores references to objects, not the objects themselves.

Example:

String[] arr = {"Hello", "World"};

// Stores references to the String objects in the array