Cheat Sheet on Arrays in Java

Visualize your JAVA CODE
https://cscircles.cemc.uwaterloo.ca/java_visualize/

Comparison of Sorting Algorithms
https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Searching and Sorting
https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/visualize/

1. Introduction to Arrays

Arrays in Java are one of the most fundamental data structures used to store multiple values of the same type in a single container. Arrays provide an efficient mechanism for handling large datasets and are essential in algorithm design and implementation.

- Arrays allow indexed access, making element retrieval and updates fast (`O(1)` time complexity).
- Fixed in size: Once declared, their length cannot change during runtime.

2. Definition and Importance

Definition: An array is a collection of variables of the same type stored at contiguous memory locations. Each element is accessed using an index.

Importance:

- Enables compact storage of data.
- Provides a base for more complex data structures like matrices, heaps, and hash tables.
- Efficient traversal and manipulation of data sets.

3. Types of Arrays

Java supports:

- **Single-Dimensional Arrays:** A linear list of elements.
- `int[] numbers = new int[5];`
- **Multi-Dimensional Arrays (mostly 2D):** Arrays of arrays.
- `int[][] matrix = new int[3][4];  // 3 rows, 4 columns`

- **Jagged Arrays:** An array where each sub-array can have a different length.

- `int[][] jagged = new int[3][];`
- `jagged[0] = new int[2];`
- `jagged[1] = new int[4];`

4. Syntax and Declaration

There are two standard forms:

```
// Declaration and instantiation
int[] arr = new int[5]; // Default values assigned
int[] arr = {1, 2, 3, 4, 5}; // Array initializer

// Accessing elements
int firstElement = arr[0];     // Access first element
arr[1] = 10;                   // Modify second element
```

Key Points:

- Index starts from 0.
- Accessing an index out of bounds results in `ArrayIndexOutOfBoundsException`.

5. Arrays Class Methods (java.util.Arrays)

Java provides utility methods via the `Arrays` class for common operations:

```
import java.util.Arrays;

int[] a = {3, 1, 4, 1, 5};
Arrays.sort(a);                     // Sort array
Arrays.fill(a, 0);                  // Fill array with value
int index = Arrays.binarySearch(a, 4);   // Binary search
boolean equal = Arrays.equals(a, b);     // Compare arrays
int[] copy = Arrays.copyOf(a, 10);       // Copy with new length
```

Common Methods Summary:

Method	Description
`sort(array)`	Sorts the array in ascending order
`fill(array, value)`	Fills entire array with value
`binarySearch(array, key)`	Searches for key (sorted array)
`equals(array1, array2)`	Compares contents of two arrays

Method	Description
`copyOf(array, newLength)`	Copies to new array with specified length
`toString(array)`	Returns string representation

6. The Arrays Class in Java

Java provides a utility class `java.util.Arrays` that simplifies array operations. It's part of the standard Java library and provides several static methods for sorting, searching, copying, and manipulating arrays.

Key Methods

- **sort()** – Sorts elements in ascending order.
- **copyOf()** – Copies original array to a new array.
- **equals()** – Compares two arrays.
- **fill()** – Assigns a single value to all elements.
- **toString()** – Converts array into a readable string.
- **binarySearch()** – Performs binary search on a sorted array.
- **copyOfRange()** – Copies a specific range from the original array.
- **deepEquals()** – Checks deep equality for multi-dimensional arrays.

Example:

```
int[] arr = {3, 1, 4, 1, 5};
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
int[] copy = Arrays.copyOf(arr, 7);
Arrays.fill(copy, 5, 7, 0);
System.out.println(Arrays.toString(copy));
System.out.println(Arrays.binarySearch(copy, 4));
```

These methods reduce boilerplate and increase code readability and efficiency when working with arrays.

6.1 Comparable and Comparator Interfaces in Arrays

Sorting custom objects in an array requires a way to define the logic for comparison. Java provides two interfaces to handle this:

`Comparable` Interface

The `Comparable` interface allows a class to define its *natural ordering* by implementing the `compareTo()` method. It is part of the `java.lang` package.

Syntax:

```
public class Student implements Comparable<Student> {
    String name;
    int marks;

    Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }

    @Override
    public int compareTo(Student other) {
        return this.marks - other.marks;   // Ascending order by
marks
    }
}
```

Usage:

```
Student[] students = {
    new Student("Alice", 85),
    new Student("Bob", 90),
    new Student("Charlie", 80)
};

Arrays.sort(students);
```

`Comparator` Interface

Use `Comparator` when you want to sort objects in different ways without changing the class. It's part of `java.util`.

Syntax:

```
Comparator<Student> byName = new Comparator<Student>() {
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name); // Sort by name
    }
};
```

Java 8 Lambda Version:

```
Comparator<Student> byMarksDescending = (s1, s2) -> s2.marks -
s1.marks;
```

Usage:

```
Arrays.sort(students, byName);
Arrays.sort(students, byMarksDescending);
```

Summary

Feature	Comparable	Comparator
Package	`java.lang`	`java.util`
Method	`compareTo(Object o)`	`compare(Object o1, Object o2)`
Use Case	Natural order	Multiple sorting criteria
Affects Class	Yes (modifies class)	No (external to class)

7. Arrays in Stream API

Java 8 introduced the Stream API, which allows developers to perform functional-style operations on arrays and collections. You can convert arrays to streams and perform operations like map, filter, reduce, etc.

Create Stream from Array

```
int[] numbers = {10, 20, 30};
IntStream stream = Arrays.stream(numbers);
```

Common Stream Operations

```
Arrays.stream(numbers).forEach(System.out::println);   //   Print
each element

int[] squares = Arrays.stream(numbers)
                      .map(n -> n * n)
                      .toArray();

int[] evens = Arrays.stream(numbers)
                    .filter(n -> n % 2 == 0)
                    .toArray();

int sum = Arrays.stream(numbers).sum();

OptionalDouble avg = Arrays.stream(numbers).average();
```

```
long count = Arrays.stream(numbers).count();

int max = Arrays.stream(numbers).max().orElse(Integer.MIN_VALUE);

int min = Arrays.stream(numbers).min().orElse(Integer.MAX_VALUE);
```

Advanced Stream Use Cases

- **Sorting and Distinct Elements:**

```
int[] sorted = Arrays.stream(numbers).sorted().toArray();
int[] unique = Arrays.stream(numbers).distinct().toArray();
```

- **Collecting to Other Types:**

```
List<Integer>                              list                                =
Arrays.stream(numbers).boxed().collect(Collectors.toList());
```

- **Flat Mapping 2D Arrays:**

```
int[][] matrix = {{1, 2}, {3, 4}};
int total = Arrays.stream(matrix)
                  .flatMapToInt(Arrays::stream)
                  .sum();
```

The Stream API allows clean, declarative operations on arrays, significantly improving expressiveness and reducing verbosity.

8. Practical Examples

Example 1: Find Maximum

```
int[] data = {1, 5, 9, 3};
int max = data[0];
for (int i = 1; i < data.length; i++) {
    if (data[i] > max) max = data[i];
}
System.out.println("Max: " + max);
```

Example 2: Reverse Array

```
int[] data = {1, 2, 3, 4};
for (int i = 0; i < data.length / 2; i++) {
    int temp = data[i];
    data[i] = data[data.length - 1 - i];
    data[data.length - 1 - i] = temp;
}
```

```
System.out.println(Arrays.toString(data));
```

Example 3: Frequency Count

```
int[] data = {1, 1, 2, 3, 2, 1};
int target = 1, count = 0;
for (int num : data) {
    if (num == target) count++;
}
System.out.println("Occurrences of " + target + ": " + count);
```

9. Performance Considerations

- Arrays offer **constant-time** access (O(1)) to elements using indices.
- Insertion or deletion in the middle requires shifting elements (O(n)).
- Arrays are not resizable; use `ArrayList` for dynamic resizing.
- Prefer primitive arrays over wrapper classes for performance-critical code.

10. Memory Management in Arrays

- Arrays are stored in the heap memory.
- Reference to the array is stored in the stack.
- Arrays of objects store references, not actual objects.
- Java's garbage collector reclaims memory when no reference to an array remains.

11. Common Pitfalls and Errors

- Accessing elements out of bounds:

```
int[] arr = new int[3];
System.out.println(arr[3]);               //                Throws
ArrayIndexOutOfBoundsException
```

- Forgetting array initialization.
- Using == instead of `Arrays.equals()` to compare arrays.

12. Deep Comparison of Arrays

Use `Arrays.deepEquals()` for multi-dimensional arrays:

```
int[][] a = {{1, 2}, {3, 4}};
int[][] b = {{1, 2}, {3, 4}};
System.out.println(Arrays.deepEquals(a, b)); // true
```

13. Advanced Stream Operations

- Sorting and filtering:

```
int[] sorted = Arrays.stream(data).sorted().toArray();
int[] unique = Arrays.stream(data).distinct().toArray();
```

- Summary statistics:

```
IntSummaryStatistics                    stats                    =
Arrays.stream(data).summaryStatistics();
System.out.println("Average: " + stats.getAverage());
```

14. Array Conversion Utilities

- Convert array to list:

```
String[] fruits = {"Apple", "Banana"};
List<String> list = Arrays.asList(fruits);
```

- Convert list to array:

```
List<String> list = new ArrayList<>();
list.add("Mango");
String[] fruitArray = list.toArray(new String[0]);
```

15. ArrayList in Java

While arrays are useful for fixed-size collections, Java offers `ArrayList` as a resizable alternative. It is a part of the Java Collection Framework and resides in `java.util` package.

15.1 What is an ArrayList?

An `ArrayList` is a class in Java that implements the `List` interface. Unlike arrays, `ArrayList` can grow or shrink in size dynamically. It maintains the order of insertion and allows duplicate elements.

Key Characteristics:

- Dynamic resizing
- Zero-based indexing
- Supports all object types (cannot hold primitives directly)
- Allows null values

15.2 Declaring an ArrayList

```java
import java.util.ArrayList;

ArrayList<String> names = new ArrayList<>();
```

You can also specify an initial capacity:

```java
ArrayList<Integer> numbers = new ArrayList<>(20);
```

15.3 Adding Elements

```java
names.add("Alice");
names.add("Bob");
names.add(1, "Charlie");  // Insert at specific index
```

15.4 Accessing Elements

```java
System.out.println(names.get(0));  // Output: Alice
```

15.5 Modifying Elements

```java
names.set(1, "David");  // Replace element at index 1
```

15.6 Removing Elements

```java
names.remove("Alice");  // Remove by value
names.remove(0);        // Remove by index
```

15.7 Iterating Over an ArrayList

```java
for (String name : names) {
    System.out.println(name);
}

names.forEach(System.out::println);
```

15.8 Useful Methods

Method	Description
`add(E e)`	Appends element to the end
`add(int index, E e)`	Inserts element at index
`remove(Object o)`	Removes first occurrence
`remove(int index)`	Removes element at index
`get(int index)`	Returns element at index
`set(int index, E e)`	Replaces element at index
`contains(Object o)`	Checks if element exists
`indexOf(Object o)`	Returns index of element
`clear()`	Removes all elements
`isEmpty()`	Checks if list is empty
`size()`	Returns number of elements

15.9 Conversion Between Array and ArrayList

Array to ArrayList:

```
String[] fruits = {"Apple", "Banana"};
List<String> fruitList = new ArrayList<>(Arrays.asList(fruits));
```

ArrayList to Array:

```
String[] fruitArray = fruitList.toArray(new String[0]);
```

15.10 ArrayList of Custom Objects

```
class Student {
    String name;
    int age;
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

ArrayList<Student> students = new ArrayList<>();
students.add(new Student("Alice", 20));
students.add(new Student("Bob", 21));
```

15.11 Sorting ArrayList

Sorting strings or integers:

```
Collections.sort(names);  // Alphabetical order
Collections.reverse(names);  // Reverse order
```

Sorting custom objects:

```
Collections.sort(students, Comparator.comparing(s -> s.name));
```

15.12 ArrayList vs Array

Feature	**Array**	**ArrayList**
Size	Fixed	Dynamic
Type	Can store primitives	Stores only objects
Performance	Slightly faster	Slightly slower
Flexibility	Less flexible	More flexible
Utility Methods	Few (`Arrays` class)	Rich (`ArrayList` methods)

16. Arrays of Objects

```
class Student {
    String name;
    int id;

    Student(String name, int id) {
        this.name = name;
        this.id = id;
    }
}

Student[] students = new Student[2];
students[0] = new Student("Alice", 101);
students[1] = new Student("Bob", 102);
```

17. Logic-Building Questions Using ArrayList

Q1. Remove All Duplicates from a List

Problem: Given a list with duplicate integers, remove all duplicates and return a list with only unique elements.

```
public static List<Integer> removeDuplicates(List<Integer> list)
{
    return new ArrayList<>(new LinkedHashSet<>(list));     //
maintains order
}
```

Q2. Find the Intersection of Two Lists

Problem: Return the common elements from two ArrayLists.

```
public static List<Integer> intersection(List<Integer> list1,
List<Integer> list2) {
    list1.retainAll(list2);
    return list1;
}
```

Q3. Merge and Sort Two Lists

Problem: Merge two lists of integers and return a sorted list.

```
public static List<Integer> mergeAndSort(List<Integer> list1,
List<Integer> list2) {
    List<Integer> merged = new ArrayList<>(list1);
    merged.addAll(list2);
    Collections.sort(merged);
    return merged;
}
```

Q4. Find the Frequency of an Element

Problem: Count how many times a specific element occurs in a list.

```
public static int frequencyOfElement(List<String> list, String
target) {
    return Collections.frequency(list, target);
}
```

Q5. Reverse an ArrayList

Problem: Reverse the elements of an `ArrayList`.

```
public static <T> void reverseList(List<T> list) {
    Collections.reverse(list);
}
```

Q6. Check if a List is a Palindrome

Problem: Determine if the elements in a list read the same forwards and backwards.

```
public static boolean isPalindrome(List<Integer> list) {
    int n = list.size();
    for (int i = 0; i < n / 2; i++) {
        if (!list.get(i).equals(list.get(n - 1 - i))) return
false;
    }
    return true;
}
```

Q7. Shift Elements Right by K Positions

Problem: Shift list elements to the right `k` times, wrapping around.

```
public static List<Integer> rotateRight(List<Integer> list, int k)
{
    int size = list.size();
    k = k % size;
    List<Integer> rotated = new ArrayList<>();
    rotated.addAll(list.subList(size - k, size));
    rotated.addAll(list.subList(0, size - k));
    return rotated;
}
```

Q8. Filter Prime Numbers from a List

Problem: Given a list of integers, return a list containing only prime numbers.

```
public static boolean isPrime(int n) {
    if (n < 2) return false;
    for (int i = 2; i <= Math.sqrt(n); i++)
        if (n % i == 0) return false;
    return true;
}

public static List<Integer> filterPrimes(List<Integer> list) {
    List<Integer> primes = new ArrayList<>();
    for (int n : list) {
        if (isPrime(n)) primes.add(n);
    }
    return primes;
}
```

Q9. Find the Longest String in a List

Problem: Return the string with the maximum length from an ArrayList.

```
public static String longestString(List<String> list) {
    String longest = "";
    for (String str : list) {
        if (str.length() > longest.length()) longest = str;
    }
    return longest;
}
```

Q10. Remove All Even Numbers

Problem: Remove all even numbers from an `ArrayList` of integers.

```
public static void removeEvens(List<Integer> list) {
    list.removeIf(n -> n % 2 == 0);
}
```

18. Best Practices

- Always initialize arrays properly.
- Use `length` property instead of hardcoding size.
- Prefer enhanced `for` loop when not modifying array.
- Use `Arrays` and `Streams` for concise and readable code.
- Validate array indices to avoid exceptions.

To enhance your understanding of Java concepts such as Arrays, the Arrays class, Stream API, and ArrayList, here are some engaging animated and visual tutorials:

Arrays in Java

1. **Arrays in Java**
   This video covers the basics of arrays, including their definition, syntax, and usage with examples.
   Watch Video
2. **Java Tutorial For Beginners: Arrays and Types**
   An animated tutorial explaining different types of arrays in Java.
   Watch Video

Stream API in Java

1. **How Java 8 Stream API Works?**
   An in-depth explanation of Java 8's Stream API, illustrating how it processes data.
   Watch Video
2. **Stream API in Java**
   A tutorial demonstrating the use of Stream API with practical examples.
   Watch Video

Arrays Class in Java

1. **The Arrays Class in Java**
   This video delves into the utility methods provided by the Arrays class in Java.
   Watch Video

ArrayList in Java

1. **Java ArrayList Tutorial**
   An introductory tutorial on ArrayList, explaining its features and usage.
   Watch Video
2. **How ArrayList Internally Works**
   A detailed explanation of the internal workings of ArrayList in Java.
   Watch Video

Visualize your JAVA CODE

https://cscircles.cemc.uwaterloo.ca/java_visualize/

Comparison Sorting Algorithms

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Searching and Sorting

https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/visualize/