Arrays Class in Java

The **Arrays** class is a utility class in the java.util package that provides static methods to manipulate arrays (like searching, sorting, filling, comparing, etc.). It works with **normal arrays** (not collections like ArrayList).

Import Statement

java

```
import java.util.Arrays;
```

Key Characteristics

- **Static methods** - call directly using class name: Arrays.methodName()
- Works with **primitive arrays** (int[], double[], etc.) and **object arrays** (String[], Integer[], etc.)
- Does **not** work with ArrayList or other collections
- Very useful for common array operations

---

1. Arrays.sort() - Sort an Array

Sorts the array in **ascending order**.

Sorting Primitive Arrays

java

```
import java.util.Arrays;


public class ArraysSortDemo {
  public static void main(String[] args) {
    // Integer array
    int[] numbers = {50, 20, 80, 10, 40};


    System.out.println("Before sorting: " + Arrays.toString(numbers));
    // [50, 20, 80, 10, 40]
```

```
        Arrays.sort(numbers);

        System.out.println("After sorting: " + Arrays.toString(numbers));
        // [10, 20, 40, 50, 80] - SORTED!
    }
}
```

Sorting String Arrays

java

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

System.out.println("Before: " + Arrays.toString(names));
// [Diana, Alice, Charlie, Bob]

Arrays.sort(names);

System.out.println("After: " + Arrays.toString(names));
// [Alice, Bob, Charlie, Diana] - Alphabetically sorted!
```

Sorting in Descending Order

For object arrays (like Integer, String), use Collections.reverseOrder():

java

```
Integer[] numbers = {50, 20, 80, 10, 40};

// Sort in descending order
Arrays.sort(numbers, Collections.reverseOrder());

System.out.println(Arrays.toString(numbers));
// [80, 50, 40, 20, 10] - Descending order!
```

Note: This doesn't work with primitive arrays (int[], double[]). You must use wrapper classes (Integer[], Double[]).

---

2. Arrays.binarySearch() - Search for an Element

Searches for a specified value in the array using **binary search algorithm**.

⚠️ **IMPORTANT:** The array **MUST be sorted** before using binarySearch(), otherwise results are unpredictable!

Returns:

- **Index** of the element if found
- **Negative value** if not found (specifically: -(insertion_point) - 1)

Basic Usage

java

```java
import java.util.Arrays;

public class BinarySearchDemo {
  public static void main(String[] args) {
    int[] numbers = {10, 20, 30, 40, 50, 60, 70};

    // Array is already sorted - required for binarySearch!

    // Search for existing element
    int index = Arrays.binarySearch(numbers, 40);
    System.out.println("Index of 40: " + index); // 3

    // Search for another element
    index = Arrays.binarySearch(numbers, 70);
    System.out.println("Index of 70: " + index); // 6
```

```
        // Search for non-existent element
        index = Arrays.binarySearch(numbers, 35);
        System.out.println("Index of 35: " + index); // Negative value (not found)
    }
}
```

Example with Unsorted Array (WRONG!)

java

```
int[] unsorted = {50, 20, 80, 10, 40};

// DON'T DO THIS - array is not sorted!
int index = Arrays.binarySearch(unsorted, 20);
System.out.println(index); // Unpredictable result!

// CORRECT WAY - sort first!
Arrays.sort(unsorted);
index = Arrays.binarySearch(unsorted, 20);
System.out.println("Index of 20: " + index); // 1 (correct)
```

Complete Example with Sort + Search

java

```
import java.util.Arrays;

public class SortAndSearch {
    public static void main(String[] args) {
        int[] numbers = {50, 20, 80, 10, 40, 60, 30};

        System.out.println("Original: " + Arrays.toString(numbers));
        // [50, 20, 80, 10, 40, 60, 30]
        // Step 1: Sort the array (REQUIRED for binarySearch)
        Arrays.sort(numbers);
        System.out.println("Sorted: " + Arrays.toString(numbers));
        // [10, 20, 30, 40, 50, 60, 80]

        // Step 2: Now we can use binarySearch
        int searchValue = 40;
        int index = Arrays.binarySearch(numbers, searchValue);

        if (index >= 0) {
            System.out.println(searchValue + " found at index: " + index);
            // 40 found at index: 3
        } else {
            System.out.println(searchValue + " not found!");
        }

        // Search for non-existent element
        searchValue = 45;
        index = Arrays.binarySearch(numbers, searchValue);
        System.out.println("Search for " + searchValue + ": " + index);
        // Negative value (e.g., -5)
    }
}
```

Binary Search with Strings

java

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

// Array must be sorted (already is)
```

```
int index = Arrays.binarySearch(names, "Charlie");
System.out.println("Index of Charlie: " + index); // 2

index = Arrays.binarySearch(names, "Frank");
System.out.println("Index of Frank: " + index); // Negative (not found)
```

---

3. Arrays.fill() - Fill Array with a Value

Fills the array with the specified value.

Fill Entire Array

java

```
import java.util.Arrays;

public class FillDemo {
  public static void main(String[] args) {
    // Create array
    int[] numbers = new int[5];

    System.out.println("Before fill: " + Arrays.toString(numbers));
    // [0, 0, 0, 0, 0] - default values

    // Fill entire array with value 10
    Arrays.fill(numbers, 10);

    System.out.println("After fill: " + Arrays.toString(numbers));
    // [10, 10, 10, 10, 10]
  }
}
```

Fill with Different Values

java

```
// String array
String[] colors = new String[6];
Arrays.fill(colors, "Red");
System.out.println(Arrays.toString(colors));
// [Red, Red, Red, Red, Red, Red]

// Double array
double[] prices = new double[4];
Arrays.fill(prices, 99.99);
System.out.println(Arrays.toString(prices));
// [99.99, 99.99, 99.99, 99.99]

// Boolean array
boolean[] flags = new boolean[5];
Arrays.fill(flags, true);
System.out.println(Arrays.toString(flags));
// [true, true, true, true, true]
```

Practical Use Cases for fill()

java

```
// Initialize array with specific value
int[] scores = new int[10];
Arrays.fill(scores, 100); // All students get 100 initially
System.out.println(Arrays.toString(scores));
// [100, 100, 100, 100, 100, 100, 100, 100, 100, 100]

// Reset array
int[] data = {5, 10, 15, 20};
```

```
Arrays.fill(data, 0); // Reset all to 0

System.out.println(Arrays.toString(data)); // [0, 0, 0, 0]
```

Additional Useful Arrays Methods

4. Arrays.toString() - Convert Array to String

Converts array to readable string format (we've been using this already!):

java

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

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

// [10, 20, 30, 40, 50]


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

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

// [Alice, Bob, Charlie]
```

Without Arrays.toString():

java

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

System.out.println(arr); // [I@15db9742 - Memory address! Not readable!
```

5. Arrays.equals() - Compare Two Arrays

Checks if two arrays are equal (same elements in same order):

java

```
int[] arr1 = {10, 20, 30};

int[] arr2 = {10, 20, 30};

int[] arr3 = {10, 30, 20};


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

System.out.println(Arrays.equals(arr1, arr3)); // false (different order)
```

6. Arrays.copyOf() - Copy Array

Creates a copy of the array:

java

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

// Copy entire array
int[] copy = Arrays.copyOf(original, original.length);
System.out.println(Arrays.toString(copy));
// [10, 20, 30, 40, 50]

// Copy first 3 elements
int[] partial = Arrays.copyOf(original, 3);
System.out.println(Arrays.toString(partial));
// [10, 20, 30]
```

7. Arrays.asList() - Convert Array to List

Converts array to a List (useful for using collection methods):

java

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

List<String> nameList = Arrays.asList(names);
System.out.println(nameList);
// [Alice, Bob, Charlie]

// Can use List methods now
System.out.println(nameList.contains("Bob")); // true
```

Complete Example - All Methods Together

java

```
import java.util.Arrays;
import java.util.Collections;

public class CompleteArraysDemo {
  public static void main(String[] args) {
    // 1. Original array
    int[] numbers = {50, 20, 80, 10, 40, 60, 30};
    System.out.println("=== Original Array ===");
    System.out.println(Arrays.toString(numbers));

    // 2. Sorting
    System.out.println("\n=== After Sorting ===");
    Arrays.sort(numbers);
    System.out.println(Arrays.toString(numbers));
    // [10, 20, 30, 40, 50, 60, 80]

    // 3. Binary Search (array must be sorted!)
    System.out.println("\n=== Binary Search ===");
    int index = Arrays.binarySearch(numbers, 40);
    System.out.println("Index of 40: " + index); // 3

    index = Arrays.binarySearch(numbers, 45);
    System.out.println("Index of 45 (not found): " + index); // Negative

    // 4. Fill array
    System.out.println("\n=== Filling Array ===");
    int[] fillArray = new int[7];
    Arrays.fill(fillArray, 100);
    System.out.println("Filled with 100: " + Arrays.toString(fillArray));
```

```
// [100, 100, 100, 100, 100, 100, 100]

// 5. Equals comparison
System.out.println("\n=== Array Comparison ===");
int[] arr1 = {10, 20, 30};
int[] arr2 = {10, 20, 30};
System.out.println("Arrays equal? " + Arrays.equals(arr1, arr2)); // true

// 6. Copy array
System.out.println("\n=== Copying Array ===");
int[] copy = Arrays.copyOf(numbers, numbers.length);
System.out.println("Copy: " + Arrays.toString(copy));

// 7. Descending order (with wrapper class)
System.out.println("\n=== Descending Order ===");
Integer[] nums = {50, 20, 80, 10, 40};
Arrays.sort(nums, Collections.reverseOrder());
System.out.println("Descending: " + Arrays.toString(nums));
// [80, 50, 40, 20, 10]

// 8. String array operations
System.out.println("\n=== String Array ===");
String[] names = {"Diana", "Alice", "Charlie", "Bob"};
System.out.println("Original: " + Arrays.toString(names));

Arrays.sort(names);
System.out.println("Sorted: " + Arrays.toString(names));
// [Alice, Bob, Charlie, Diana]
```

```
        int nameIndex = Arrays.binarySearch(names, "Charlie");
        System.out.println("Index of Charlie: " + nameIndex); // 2
    }
}
```

Collections Class in Java

The **Collections** class is a utility class in the java.util package that provides static methods to manipulate collections (like ArrayList, LinkedList, etc.). It's different from the **Collection interface**.

Import Statement

java

```
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;
```

Key Characteristics

- **Static methods** - call directly using class name: Collections.methodName()
- Works with **Collection objects** (ArrayList, LinkedList, etc.)
- Does **not** work with normal arrays or Map

---

1. Collections.sort() - Sort a List

Sorts the list in **ascending order**.

Sorting Integer List

java

```
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;


public class SortDemo {
  public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<>();
    numbers.add(50);
    numbers.add(20);
    numbers.add(80);
    numbers.add(10);
    numbers.add(40);


    System.out.println("Before sorting: " + numbers);
    // [50, 20, 80, 10, 40]


    Collections.sort(numbers);


    System.out.println("After sorting: " + numbers);
    // [10, 20, 40, 50, 80] - SORTED!
  }
}
```

Sorting String List

java

```
List<String> names = new ArrayList<>();
names.add("Diana");
names.add("Alice");
names.add("Charlie");
names.add("Bob");
```

```
System.out.println("Before: " + names);
// [Diana, Alice, Charlie, Bob]

Collections.sort(names);

System.out.println("After: " + names);
// [Alice, Bob, Charlie, Diana] - Alphabetically sorted!
```

---

2. Collections.sort() with Comparator.reverseOrder() - Descending Order

Sorts the list in **descending order**.

java

```
List<Integer> numbers = new ArrayList<>();
numbers.add(50);
numbers.add(20);
numbers.add(80);
numbers.add(10);
numbers.add(40);

System.out.println("Before: " + numbers);
// [50, 20, 80, 10, 40]

Collections.sort(numbers, Collections.reverseOrder());

System.out.println("After (descending): " + numbers);
// [80, 50, 40, 20, 10]
```

String List in Reverse Order

java

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

```
names.add("Diana");
names.add("Alice");
names.add("Charlie");
names.add("Bob");

Collections.sort(names, Collections.reverseOrder());

System.out.println(names);
// [Diana, Charlie, Bob, Alice] - Reverse alphabetical!
```

---

3. Collections.min() - Find Minimum Element

Returns the minimum element in the collection.

java

```
List<Integer> numbers = new ArrayList<>();
numbers.add(50);
numbers.add(20);
numbers.add(80);
numbers.add(10);
numbers.add(40);

int min = Collections.min(numbers);
System.out.println("Minimum: " + min); // 10
```

With Strings

java

```
List<String> names = new ArrayList<>();
names.add("Diana");
names.add("Alice");
names.add("Charlie");
```

```
String minName = Collections.min(names);
System.out.println("Min name (alphabetically): " + minName); // Alice
```

---

4. Collections.max() - Find Maximum Element

Returns the maximum element in the collection.

java

```
List<Integer> numbers = new ArrayList<>();
numbers.add(50);
numbers.add(20);
numbers.add(80);
numbers.add(10);
numbers.add(40);

int max = Collections.max(numbers);
System.out.println("Maximum: " + max); // 80
```

With Strings

java

```
List<String> names = new ArrayList<>();
names.add("Diana");
names.add("Alice");
names.add("Charlie");

String maxName = Collections.max(names);
System.out.println("Max name (alphabetically): " + maxName); // Diana
```

---

5. Collections.frequency() - Count Occurrences

Returns the number of times a specified element appears in the collection.

java

```
List<Integer> numbers = new ArrayList<>();
numbers.add(10);
numbers.add(20);
numbers.add(10);
numbers.add(30);
numbers.add(10);
numbers.add(20);

int freq10 = Collections.frequency(numbers, 10);
int freq20 = Collections.frequency(numbers, 20);
int freq30 = Collections.frequency(numbers, 30);

System.out.println("Frequency of 10: " + freq10); // 3
System.out.println("Frequency of 20: " + freq20); // 2
System.out.println("Frequency of 30: " + freq30); // 1
```

With Strings

java

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
names.add("Alice");
names.add("Charlie");
names.add("Alice");
names.add("Bob");

int aliceCount = Collections.frequency(names, "Alice");
System.out.println("Alice appears: " + aliceCount + " times"); // 3
```

Complete Example - Basic Collections Methods

java

```
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;

public class CollectionsDemo {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(50);
        numbers.add(20);
        numbers.add(80);
        numbers.add(10);
        numbers.add(40);
        numbers.add(20);
        numbers.add(80);

        System.out.println("Original List: " + numbers);

        // Sort - Ascending
        Collections.sort(numbers);
        System.out.println("Sorted (ascending): " + numbers);

        // Sort - Descending
        Collections.sort(numbers, Collections.reverseOrder());
        System.out.println("Sorted (descending): " + numbers);
        // Min and Max
        System.out.println("Minimum: " + Collections.min(numbers));
        System.out.println("Maximum: " + Collections.max(numbers));

        // Frequency
        System.out.println("Frequency of 20: " + Collections.frequency(numbers, 20));
        System.out.println("Frequency of 80: " + Collections.frequency(numbers, 80));
        System.out.println("Frequency of 100: " + Collections.frequency(numbers, 100));
    }
}
```

Sorting Custom Objects

To sort custom objects, you need to implement either the **Comparable interface** or use a **Comparator**.

---

6. Comparable Interface - Natural Ordering

The **Comparable interface** is used to define the **natural ordering** of objects. The class itself decides how its objects should be sorted.

Implementing Comparable

java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

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

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

    // compareTo method - defines natural ordering
    // Sort by marks (ascending)
    @Override
    public int compareTo(Student other) {
        return this.marks - other.marks;
        // If this.marks < other.marks → negative (this comes first)
        // If this.marks > other.marks → positive (other comes first)
        // If this.marks == other.marks → zero (equal)
    }

    @Override
    public String toString() {
        return name + "(" + marks + ")";
    }
}

public class ComparableDemo {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Alice", 85));
        students.add(new Student("Bob", 70));
        students.add(new Student("Charlie", 95));
        students.add(new Student("Diana", 80));
        System.out.println("Before sorting: " + students);
        // [Alice(85), Bob(70), Charlie(95), Diana(80)]

        Collections.sort(students); // Uses compareTo method

        System.out.println("After sorting (by marks): " + students);
        // [Bob(70), Diana(80), Alice(85), Charlie(95)] - Sorted by marks!
    }
}
```

Sorting by Name (String comparison)

java

```
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.name.compareTo(other.name); // Sort by name alphabetically
    }

    @Override
    public String toString() {
        return name + "(" + marks + ")";
```

```
  }
}
```

Descending Order using Comparable

java

```
@Override
public int compareTo(Student other) {
  return other.marks - this.marks; // Reverse for descending
  // or
  // return Integer.compare(other.marks, this.marks);
}
```

---

7. Comparator Interface - Custom Ordering

The **Comparator interface** allows you to define **multiple different ways** to sort objects without modifying the class.

Method 1: Creating a Separate Comparator Class

java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Student {
  String name;
  int marks;

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

    @Override
    public String toString() {
        return name + "(" + marks + ")";
    }
}

// Comparator to sort by marks
class SortByMarks implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.marks - s2.marks; // Ascending order
    }
}

// Comparator to sort by name
class SortByName implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name); // Alphabetical order
    }
}

public class ComparatorDemo {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Diana", 80));
        students.add(new Student("Alice", 85));
        students.add(new Student("Charlie", 95));
        students.add(new Student("Bob", 70));

        System.out.println("Original: " + students);

        // Sort by marks
        Collections.sort(students, new SortByMarks());
        System.out.println("Sorted by marks: " + students);
        // [Bob(70), Diana(80), Alice(85), Charlie(95)]

        // Sort by name
        Collections.sort(students, new SortByName());
        System.out.println("Sorted by name: " + students);
        // [Alice(85), Bob(70), Charlie(95), Diana(80)]
    }
}
```

Method 2: Anonymous Inner Class

java

```
List<Student> students = new ArrayList<>();
students.add(new Student("Diana", 80));
students.add(new Student("Alice", 85));
students.add(new Student("Bob", 70));

// Sort by marks using anonymous class
Collections.sort(students, new Comparator<Student>() {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.marks - s2.marks;
    }
});

System.out.println("Sorted by marks: " + students);
```

---

8. Using Lambda Expression - Modern Approach

Lambda expressions provide a cleaner, shorter syntax for Comparators.

Sort by Marks (Ascending)

java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Student {
    String name;
    int marks;

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

    @Override
    public String toString() {
        return name + "(" + marks + ")";
    }
}
```

```
public class LambdaComparatorDemo {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Diana", 80));
        students.add(new Student("Alice", 85));
        students.add(new Student("Charlie", 95));
        students.add(new Student("Bob", 70));

        System.out.println("Original: " + students);

        // Sort by marks (ascending) - Lambda
        Collections.sort(students, (s1, s2) -> s1.marks - s2.marks);
        System.out.println("Sorted by marks: " + students);
        // [Bob(70), Diana(80), Alice(85), Charlie(95)]
    }
}
```

Sort by Marks (Descending)

java

```
// Descending order
Collections.sort(students, (s1, s2) -> s2.marks - s1.marks);
System.out.println("Sorted by marks (desc): " + students);
// [Charlie(95), Alice(85), Diana(80), Bob(70)]
```

Sort by Name (Alphabetically)

java

```
// Sort by name (ascending)
Collections.sort(students, (s1, s2) -> s1.name.compareTo(s2.name));
System.out.println("Sorted by name: " + students);
```

```
// [Alice(85), Bob(70), Charlie(95), Diana(80)]
```

Sort by Name (Reverse Alphabetically)

java

```
// Sort by name (descending)
Collections.sort(students, (s1, s2) -> s2.name.compareTo(s1.name));
System.out.println("Sorted by name (desc): " + students);
// [Diana(80), Charlie(95), Bob(70), Alice(85)]
```

Using Comparator.comparing() - Even Cleaner

java

```
import java.util.Comparator;

// Sort by marks
Collections.sort(students, Comparator.comparing(s -> s.marks));

// Sort by name
Collections.sort(students, Comparator.comparing(s -> s.name));

// Sort by marks (descending)
Collections.sort(students, Comparator.comparing((Student s) -> s.marks).reversed());

// Sort by name (descending)
Collections.sort(students, Comparator.comparing((Student s) -> s.name).reversed());
```

Complete Example - All Sorting Methods

java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
```

```java
import java.util.List;

class Student {
    String name;
    int marks;

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

    @Override
    public String toString() {
        return name + "(" + marks + ")";
    }
}

public class CompleteSortingDemo {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Diana", 80));
        students.add(new Student("Alice", 85));
        students.add(new Student("Charlie", 95));
        students.add(new Student("Bob", 70));

        System.out.println("=== Original List ===");
        System.out.println(students);
```

```
// 1. Lambda - Sort by marks (ascending)
System.out.println("\n=== Sort by Marks (Ascending) - Lambda ===");
Collections.sort(students, (s1, s2) -> s1.marks - s2.marks);
System.out.println(students);

// 2. Lambda - Sort by marks (descending)
System.out.println("\n=== Sort by Marks (Descending) - Lambda ===");
Collections.sort(students, (s1, s2) -> s2.marks - s1.marks);
System.out.println(students);

// 3. Lambda - Sort by name (ascending)
System.out.println("\n=== Sort by Name (Ascending) - Lambda ===");
Collections.sort(students, (s1, s2) -> s1.name.compareTo(s2.name));
System.out.println(students);

// 4. Lambda - Sort by name (descending)
System.out.println("\n=== Sort by Name (Descending) - Lambda ===");
Collections.sort(students, (s1, s2) -> s2.name.compareTo(s1.name));
System.out.println(students);

// 5. Comparator.comparing() - Sort by marks
System.out.println("\n=== Sort by Marks - Comparator.comparing() ===");
Collections.sort(students, Comparator.comparing(s -> s.marks));
System.out.println(students);

// 6. Comparator.comparing() - Sort by name (reversed)
System.out.println("\n=== Sort by Name (Reversed) - Comparator ===");
```

```
        Collections.sort(students, Comparator.comparing((Student s) -> s.name).reversed());
        System.out.println(students);


        // Min and Max with Comparator
        System.out.println("\n=== Min/Max with Custom Comparator ===");
        Student topStudent = Collections.max(students, (s1, s2) -> s1.marks - s2.marks);
        System.out.println("Top student (max marks): " + topStudent);


        Student bottomStudent = Collections.min(students, (s1, s2) -> s1.marks - s2.marks);
        System.out.println("Bottom student (min marks): " + bottomStudent);
    }
}
```