JAVA CODING QUESTIONS WITH EXPLANATIONS

ARRAYS

1. Largest number in an Array

2. **Write code to print only the even numbers from an array.**

3. Duplicate Elements in an Array: Finding and Printing Duplicates

4. Write a program to find the second highest integer in an array

5. Write Java code to print all the array elements that appear at least 2 times.

6. **Write Java code to remove duplicate elements from an array without using HashMap**

7. **Initialize the array and find the missing letters (10, 9, 2, 1) and print:**

8. Move all zeros in an array to the end

9. **Move all odd numbers to the front and even numbers to the end in an array.**

10. Reverse an array in subsets of size N.

11. Count Odd & Even Numbers in an Array

12. Remove Duplicates in an Array using Hashset

13. **Remove Duplicates from ArrayList**

14. Search an Element in an Array

15. **Sort an Array**

1. .Largest number in an Array

Main.java

```java
import java.util.Scanner;
public class Main
{
  public static void main(String args[])
  {
      int arr[] = {12, 13, 1, 10, 34, 10};
      int max = arr[0];
      for(int i=0; i<arr.length; i++)
      {
        if(max < arr[i])
        {
          max = arr[i];
        }
      }
    System.out.print(max);
  }
}
```

Code Structure:

- **Import Statement:**
  - `import java.util.Scanner;`: The `Scanner` class is imported but unused in this program. It can be removed.
- **Class and Main Method:**
  - `public class Main`: Defines the class `Main`.
  - `public static void main(String args[])`: The starting point of the program execution.

Steps in the Program:

1. **Array Declaration:**
   - `int arr[] = {12, 13, 1, 10, 34, 10};`:
     - An integer array `arr` is initialized with the values `{12, 13, 1, 10, 34, 10}`.
2. **Initialize `max`:**
   - `int max = arr[0];`:
     - The first element of the array (`12`) is stored in the variable `max`. This variable will hold the largest number as the program executes.
3. **Iterate Over the Array:**
   - `for (int i = 0; i < arr.length; i++)`:
     - A loop starts from the first index (`i = 0`) and continues until the last index (`i = arr.length - 1`).
4. **Compare Current Element with `max`:**
   - `if (max < arr[i])`:
     - Checks if the current array element `arr[i]` is greater than the current value of `max`.
     - If true, update `max` to hold the value of `arr[i]`.

5. **Update `max`:**
   - `max = arr[i];`:
     - Assigns the value of the current element (`arr[i]`) to `max` if the condition in the `if` statement is true.
6. **Print the Result:**
   - `System.out.print(max);`:
     - Prints the largest value stored in `max` after the loop finishes.

Execution Example:

- **Input Array:** `{12, 13, 1, 10, 34, 10}`
- **Execution:**
  - **Initial `max`:** `12`
  - **Loop Iterations:**
    1. Compare `12` (current `max`) with `12`: No change.
    2. Compare `13` with `12`: Update `max` to `13`.
    3. Compare `1` with `13`: No change.
    4. Compare `10` with `13`: No change.
    5. Compare `34` with `13`: Update `max` to `34`.
    6. Compare `10` with `34`: No change.
- **Final `max`:** `34`.

Output:

- Prints the largest number: `34`.

Key Characteristics:

- **Logic:** Compares each element with the current `max` and updates `max` if the element is larger.
- **Time Complexity:** $O(n)$, as the array is traversed once.
- **Space Complexity:** $O(1)$, as no additional space is used except for the variable `max`.

Edge Cases:

- Single-element array: The largest number is the only element.
- Array with all identical numbers: The largest number is any of the identical values.

2. Write code to print only the even numbers from an array.

Main.java

```java
public class EvenNumbersFromArray {
    public static void main(String[] args) {
        int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        System.out.println("Even numbers:");
        for (int num : numbers) {
            if (num % 2 == 0) {
                System.out.println(num);
            }
        }
    }
}
```

- **Class Declaration:**
  - `public class EvenNumbersFromArray`: Defines the main class of the program.
- **Main Method:**
  - `public static void main(String[] args)`: The entry point of the program.

Steps in the Program:

1. **Array Initialization:**
   - `int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };`:
     - A predefined array `numbers` is created with the values `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`.
2. **Header Print:**
   - `System.out.println("Even numbers:");`:
     - Prints a header message, "Even numbers:", to label the output.
3. **For-Each Loop:**
   - `for (int num : numbers)`:
     - Loops through each element (`num`) in the `numbers` array.
4. **Check Even Numbers:**
   - `if (num % 2 == 0)`:
     - Uses the modulo operator (`%`) to check if `num` is divisible by `2` without a remainder.
     - If true, the number is even.

4. **Check Even Numbers:**
   - `if (num % 2 == 0)`:
     - Uses the modulo operator (`%`) to check if `num` is divisible by `2` without a remainder.
     - If true, the number is even.
5. **Print Even Numbers:**
   - `System.out.println(num);`:
     - Prints the current number (`num`) if it satisfies the condition for being even.

Example Execution for `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`:

1. **Iteration 1:** `num = 1` → Not even → Not printed.
2. **Iteration 2:** `num = 2` → Even → Printed: `2`.
3. **Iteration 3:** `num = 3` → Not even → Not printed.
4. **Iteration 4:** `num = 4` → Even → Printed: `4`.
5. **Iteration 5:** `num = 5` → Not even → Not printed.
6. **Iteration 6:** `num = 6` → Even → Printed: `6`.
7. **Iteration 7:** `num = 7` → Not even → Not printed.
8. **Iteration 8:** `num = 8` → Even → Printed: `8`.
9. **Iteration 9:** `num = 9` → Not even → Not printed.
10. **Iteration 10:** `num = 10` → Even → Printed: `10`.

Output:

yaml Copy code

```yaml
Even numbers:
2
4
6
8
10
```

Key Points:

- **Logic:**
  - The program identifies even numbers by checking if a number leaves a remainder of `0` when divided by `2`.
- **Time Complexity:**
  - $O(n)$, where n is the size of the array. The loop processes each element once.
- **Space Complexity:**
  - $O(1)$, as no additional memory is used apart from the loop variable.
- **Advantages:**
  - Clear and concise code.
  - Easy to understand and modify for different conditions (e.g., finding odd numbers).

3. Duplicate Elements in an Array: Finding and Printing Duplicates

```java
import java.util.HashSet;

public class DuplicateElements {
    public static void findDuplicates(int[] arr) {
        HashSet<Integer> seen = new HashSet<>();
        System.out.print("Duplicates: ");
        for (int num : arr) {
            if (!seen.add(num)) {
                System.out.print(num + " ");
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 5, 1};
        findDuplicates(arr);
    }
}
```

Code Structure:

1. **Import Statement:**
    - `import java.util.HashSet;` :
        - Imports the `HashSet` class from the Java library. `HashSet` is a data structure used to store unique elements.
2. **Class Declaration:**
    - `public class DuplicateElements` :
        - Defines the main class named `DuplicateElements` .
3. **Method:** `findDuplicates` :
    - **Parameters:**
        - Accepts an integer array (`int[] arr`) as input.
    - **Logic:**
        1. `HashSet<Integer> seen = new HashSet<>();` :
            - Creates an empty `HashSet` named `seen` to store unique elements encountered during iteration.
        2. **Loop Through Array:**
            - `for (int num : arr)` :
                - Iterates through each element (`num`) in the array.
            - `if (!seen.add(num))` :
                - Tries to add `num` to the `HashSet` .
                - If `num` is already in the `HashSet` (`add()` returns `false`), it is identified as a duplicate.

duplicate.

- `System.out.print(num + " ");`:
  - Prints the duplicate number.

4. **Main Method**:
   - `public static void main(String[] args)`:
     - The entry point of the program.
   - **Input Array**:
     - `int[] arr = {1, 2, 3, 2, 4, 5, 1};`: Defines the array to check for duplicates.
   - **Call to `findDuplicates`**:
     - Passes the array `arr` to the `findDuplicates` method.

Execution:

- **Input Array**: `{1, 2, 3, 2, 4, 5, 1}`.
- **Step-by-Step Execution**:
  1. `HashSet` starts empty: `{}`.
  2. **Iteration 1**: `num = 1` → Added to `HashSet` → `HashSet: {1}`.
  3. **Iteration 2**: `num = 2` → Added to `HashSet` → `HashSet: {1, 2}`.
  4. **Iteration 3**: `num = 3` → Added to `HashSet` → `HashSet: {1, 2, 3}`.
  5. **Iteration 4**: `num = 2` → Duplicate (already in `HashSet`) → Printed: `2`.
  6. **Iteration 5**: `num = 4` → Added to `HashSet` → `HashSet: {1, 2, 3, 4}`.
  7. **Iteration 6**: `num = 5` → Added to `HashSet` → `HashSet: {1, 2, 3, 4, 5}`.
  8. **Iteration 7**: `num = 1` → Duplicate (already in `HashSet`) → Printed: `1`.

Output:

```makefile
Duplicates: 2 1
```

Key Points:

1. **Logic**:
   - `HashSet` ensures that only unique elements are stored.
   - If `add()` fails (returns `false`), the number is a duplicate.
2. **Efficiency**:
   - **Time Complexity**: $O(n)$, where n is the size of the array.
     - Adding elements to a `HashSet` and checking membership are $O(1)$ operations.
   - **Space Complexity**: $O(n)$, as the `HashSet` may store up to n unique elements.
3. **Advantages**:
   - Simple and efficient way to detect duplicates.
   - Works well for arrays of any size.
4. **Limitations**:
   - Does not maintain the order of duplicates.
   - Prints duplicates as soon as they are found. If all duplicates need to be collected first, additional storage would be needed.

How It Works:

- The `HashSet` acts as a record-keeper for numbers encountered.
- Duplicates are detected when `add()` fails, and they are immediately printed.

4. . Write a program to find the second highest integer in an array

java   Copy code

```java
import java.util.Arrays;

public class SecondLargest {
    public static int findSecondLargest(int[] arr) {
        if (arr.length < 2) throw new IllegalArgumentException("Array must have at least t
        Arrays.sort(arr);
        return arr[arr.length - 2]; // Second last element
    }

    public static void main(String[] args) {
        int[] arr = {5, 1, 8, 3, 10};
        System.out.println("Second Largest: " + findSecondLargest(arr));
    }
}
```

Code Structure:

1. **Import Statement:**
   - `import java.util.Arrays;`:
     - Imports the `Arrays` class, which provides utility methods for array operations, such as sorting.
2. **Class Declaration:**
   - `public class SecondLargest`:
     - Defines the class `SecondLargest`.
3. **Method:** `findSecondLargest`:
   - **Parameters:**
     - Accepts an integer array (`int[] arr`) as input.
   - **Steps:**
     1. **Validation:**
        - `if (arr.length < 2)`:
          - Checks if the array has fewer than 2 elements.
          - If true, throws an `IllegalArgumentException` with a relevant error message.
     2. **Sort the Array:**
        - `Arrays.sort(arr);`:
          - Sorts the array in ascending order.
     3. **Return Second Largest Element:**
        - `return arr[arr.length - 2];`:
          - Retrieves the second last element from the sorted array (which is the second largest).

4. **Main Method:**
   - `public static void main(String[] args)`:
     - The entry point of the program.
   - **Input Array:**
     - `int[] arr = {5, 1, 8, 3, 10};`:
       - Defines the input array.
   - **Call `findSecondLargest`:**
     - Passes the array `arr` to the `findSecondLargest` method.
   - **Print the Result:**
     - Prints the returned second largest number.

Execution:

1. **Input Array:** `{5, 1, 8, 3, 10}`
2. **Validation:**
   - The array has more than one element → Validation passed.
3. **Sort the Array:**
   - Sorted Array: `{1, 3, 5, 8, 10}`
4. **Find Second Largest:**
   - Second Last Element: `8`
5. **Output:**
   - Prints: `Second Largest: 8`

Output:

```sql
Second Largest: 8
```

Key Points:

1. **Logic:**
   - Sorting ensures the elements are in ascending order.
   - The second largest element is located at the second last position in the sorted array.
2. **Efficiency:**
   - **Time Complexity:**
     - Sorting the array takes $O(n \log n)$, where n is the size of the array.
   - **Space Complexity:**
     - $O(1)$, since sorting is done in place, and no extra space is used other than a few variables.
3. **Advantages:**
   - Simple and easy-to-understand logic.
   - Handles arrays with duplicate elements correctly (e.g., `{5, 8, 8, 3, 10}` → `8` is still the second largest).
4. **Limitations:**
   - Sorting the entire array is computationally more expensive than necessary. A linear $O(n)$ approach could be used to find the second largest element without sorting.
   - Throws an exception for arrays with fewer than two elements instead of providing a user-friendly message.
5. **Edge Cases:**
   - **Array with fewer than two elements:** Throws an exception.
   - **Array with duplicates:** Correctly identifies the second largest element. For example:
     - Input: `{5, 5, 8, 8, 10}`
     - Output: `8`.

5. Write Java code to print all the array elements that appear at least 2 times.

```java
import java.util.HashMap;

public class DuplicateElements {
    public static void findDuplicates(int[] arr) {
        HashMap<Integer, Integer> countMap = new HashMap<>();

        for (int num : arr) {
            countMap.put(num, countMap.getOrDefault(num, 0) + 1);
        }

        System.out.print("Duplicates: ");
        for (int key : countMap.keySet()) {
            if (countMap.get(key) >= 2) {
                System.out.print(key + " ");
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 5, 1, 6, 2};
        findDuplicates(arr);
    }
}
```

Code Structure:

1. **Import Statement:**
   - `import java.util.HashMap;`:
     - Imports the `HashMap` class from the Java library, which is used to store key-value pairs for counting occurrences.
2. **Class Declaration:**
   - `public class DuplicateElements`:
     - Defines the main class named `DuplicateElements`.
3. **Method:** `findDuplicates`:
   - **Parameters:**
     - Accepts an integer array (`int[] arr`) as input.
   - **Steps:**
     1. **Initialize a** `HashMap`:
        - `HashMap<Integer, Integer> countMap = new HashMap<>();`:
          - Creates a `HashMap` to store each unique number as the key and its occurrence count as the value.
     2. **Count Occurrences:**
        - `for (int num : arr)`:
          - Loops through each element in the array.
        - `countMap.put(num, countMap.getOrDefault(num, 0) + 1);`:
          - Updates the count of `num` in the `HashMap`.
          - If `num` is not already in the map, `getOrDefault(num, 0)` initializes its count to `0`.

3. Find Duplicates:
    - `for (int key : countMap.keySet())`:
        - Iterates through all the keys in the `HashMap`.
    - `if (countMap.get(key) >= 2)`:
        - Checks if the count of the current key is greater than or equal to `2` (i.e., it's a duplicate).
    - `System.out.print(key + " ");`:
        - Prints the duplicate number.
4. Main Method:
    - `public static void main(String[] args)`:
        - The entry point of the program.
    - Input Array:
        - `int[] arr = {1, 2, 3, 2, 4, 5, 1, 6, 2};`:
            - Defines the input array.
    - Call `findDuplicates`:
        - Passes the array `arr` to the `findDuplicates` method.

Execution:

1. Input Array: `{1, 2, 3, 2, 4, 5, 1, 6, 2}`
2. Build `countMap`:
    - After the loop, `countMap` contains:
        - `{1=2, 2=3, 3=1, 4=1, 5=1, 6=1}`
3. Identify Duplicates:
    - Iterates over keys:
        - Key `1`: Count = 2 → Duplicate → Printed: `1`
        - Key `2`: Count = 3 → Duplicate → Printed: `2`
        - Keys `3, 4, 5, 6`: Counts < 2 → Not printed.
4. Output:
    - Prints: `Duplicates: 1 2`

Output:

```makefile
Duplicates: 1 2
```

Key Points:

1. Logic:
    - A `HashMap` is used to store the frequency of each element.
    - Duplicates are identified when an element's frequency is `>= 2`.
2. Efficiency:
    - Time Complexity:
        - $O(n)$, where n is the size of the array.
        - Counting elements and iterating over keys are both $O(n)$ operations.
    - Space Complexity:
        - $O(n)$, as the `HashMap` may store up to n unique elements.
3. Advantages:
    - Handles arrays with multiple duplicates (e.g., `2` is counted only once as a duplicate even if it appears 3 times).
    - Works for arrays of any size.
4. Limitations:
    - Does not maintain the order of duplicates from the original array.
    - Prints duplicates immediately without additional formatting or processing.
5. Edge Cases:
    - Array with no duplicates: Outputs `Duplicates:` with no elements.
        - Example: `{1, 2, 3}` → No duplicates found.
    - Empty Array: `findDuplicates` does nothing, as there's no element to process.
    - Array with all duplicates:
        - Example: `{1, 1, 1}` → Correctly outputs: `1`.

6. Write Java code to remove duplicate elements from an array without using HashMap

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

public class Main {
    public static void main(String[] args) {
        // Initialize array with duplicate elements
        int[] arr = {1, 2, 2, 3, 4, 4, 5};

        // Create a List to store unique elements
        List<Integer> unique = new ArrayList<>();

        // Iterate through the array
        for (int num : arr) {
            // If the List does not contain the current number, add it
            if (!unique.contains(num)) {
                unique.add(num);
            }
        }

        // Print the unique elements
        System.out.println("Unique elements: " + unique);
    }
}
```

Code Structure:

1. Import Statements:
   - `import java.util.ArrayList;`:
     - Imports the `ArrayList` class, which is a dynamic array implementation in Java.
   - `import java.util.List;`:
     - Imports the `List` interface, which `ArrayList` implements.
2. Class Declaration:
   - `public class Main`:
     - Defines the main class named `Main`.
3. Main Method:
   - `public static void main(String[] args)`:
     - Entry point of the program.
4. Steps:
   1. Initialize the Array:
      - `int[] arr = {1, 2, 2, 3, 4, 4, 5};`:
        - Defines an integer array with duplicate elements.
   2. Create an Empty List:
      - `List<Integer> unique = new ArrayList<>();`:
        - Initializes an empty `ArrayList` to store unique elements.
   3. Iterate Through the Array:
      - `for (int num : arr)`:
        - Loops through each element (`num`) in the array.
   4. Check for Duplicates:
      - `if (!unique.contains(num))`:
        - Checks if the `unique` list already contains the current element (`num`).
      - `unique.add(num)`:
        - If the element is not in the list, adds it to `unique`.
   5. Print Unique Elements:
      - `System.out.println("Unique elements: " + unique);`:
        - Prints the elements stored in the `unique` list.

Execution:

1. Input Array:
   - `{1, 2, 2, 3, 4, 4, 5}`
2. Iteration:
   - Step-by-Step:
     1. `num = 1`: Not in `unique` → Add to `unique`: `[1]`.
     2. `num = 2`: Not in `unique` → Add to `unique`: `[1, 2]`.
     3. `num = 2`: Already in `unique` → Skip.
     4. `num = 3`: Not in `unique` → Add to `unique`: `[1, 2, 3]`.
     5. `num = 4`: Not in `unique` → Add to `unique`: `[1, 2, 3, 4]`.
     6. `num = 4`: Already in `unique` → Skip.
     7. `num = 5`: Not in `unique` → Add to `unique`: `[1, 2, 3, 4, 5]`.
3. Final Output:
   - Prints: `Unique elements: [1, 2, 3, 4, 5]`

Output:

```less
Unique elements: [1, 2, 3, 4, 5]
```

Key Points:

1. Logic:
   - Uses a `List` (`ArrayList`) to store elements from the array.
   - Checks for duplicates using the `contains()` method before adding an element to the list.
2. Efficiency:
   - Time Complexity:
     - $O(n^2)$ in the worst case:
       - Outer loop iterates over the array ($O(n)$).
       - `contains()` method checks for duplicates ($O(n)$ for an unsorted `ArrayList`).

2. Efficiency:
   - Time Complexity:
     - $O(n^2)$ in the worst case:
       - Outer loop iterates over the array ($O(n)$).
       - `contains()` method checks for duplicates ($O(n)$ for an unsorted `ArrayList`).
   - Space Complexity:
     - $O(n)$, as the `ArrayList` stores unique elements from the array.
3. Advantages:
   - Simple implementation.
   - Easy to understand and use for small arrays.
4. Limitations:
   - Not efficient for large arrays due to $O(n^2)$ time complexity.
   - Relies on linear search in the `ArrayList` for checking duplicates.
5. Edge Cases:
   - Empty Array: Outputs an empty list:
     - Input: `int[] arr = {};` → Output: `Unique elements: []`.
   - Array with All Duplicates:
     - Input: `int[] arr = {2, 2, 2, 2};` → Output: `Unique elements: [2]`.
   - Array with All Unique Elements:
     - Input: `int[] arr = {1, 2, 3, 4, 5};` → Output: `Unique elements: [1, 2, 3, 4, 5]`.

How It Works:

1. Iterates through the array.
2. Checks each element for duplication in the `ArrayList`.
3. Adds unique elements to the list.
4. Prints the final list of unique elements.

7. Initialize the array and find the missing letters (10, 9, 2, 1) and print:

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] arr = {10, 9, 2, 1};
        Arrays.sort(arr); // Sort the array for easier checking

        System.out.println("Missing elements:");
        for (int i = 1; i < arr[arr.length - 1]; i++) {
            boolean found = false;
            for (int j = 0; j < arr.length; j++) {
                if (arr[j] == i) {
                    found = true;
                    break;
                }
            }
            if (!found) {
                System.out.println(i);
            }
        }
    }
}
```

1. **Import Statement:**
   - `import java.util.Arrays;`:
     - Imports the `Arrays` class from the Java standard library to use its sorting method.
2. **Class Declaration:**
   - `public class Main`:
     - Defines the main class `Main`.
3. **Main Method:**
   - `public static void main(String[] args)`:
     - The entry point of the program.
4. **Steps:**
   1. **Initialize the Array:**
      - `int[] arr = {10, 9, 2, 1};`:
        - Defines an array of integers with some missing elements from a sequential range.
   2. **Sort the Array:**
      - `Arrays.sort(arr);`:
        - Sorts the array in ascending order to make checking for missing elements easier.
        - After sorting: `arr = [1, 2, 9, 10]`
   3. **Print Missing Elements:**
      - `System.out.println("Missing elements:");`:
        - Prints the message indicating the start of missing elements.
   4. **Outer Loop** (Iterate over the range of numbers from 1 to the largest element in the array):
      - `for (int i = 1; i < arr[arr.length - 1]; i++)`:

3. **Print Missing Elements**:
   - `System.out.println("Missing elements:");`:
     - Prints the message indicating the start of missing elements.
4. **Outer Loop** (Iterate over the range of numbers from 1 to the largest element in the array):
   - `for (int i = 1; i < arr[arr.length - 1]; i++)`:
     - Iterates over numbers starting from 1 up to the last number in the array (the largest number).
     - `arr[arr.length - 1]` gives the largest element in the sorted array (`10`).
5. **Inner Loop** (Check if the current number exists in the array):
   - `for (int j = 0; j < arr.length; j++)`:
     - Loops through each element in the sorted array to check if the current number `i` is present.
   - If the number `i` is found, it sets `found = true` and breaks out of the inner loop.
6. **Identify and Print Missing Elements**:
   - `if (!found)`:
     - If the number `i` was not found in the array (i.e., it is missing), the program prints it.
     - `System.out.println(i);` prints the missing number.

Execution:

1. **Input Array:**
   - `{10, 9, 2, 1}`
2. **Sort the Array:**
   - Sorted array: `{1, 2, 9, 10}`

Execution:

1. Input Array:
   - `{10, 9, 2, 1}`
2. Sort the Array:
   - Sorted array: `{1, 2, 9, 10}`

3. Iterate through Range:
   - Range is from 1 to 9 (largest number in the array is 10, so we iterate from 1 to 9).
4. Check for Missing Numbers:
   - Check `i = 1`: Found (present in array).
   - Check `i = 2`: Found (present in array).
   - Check `i = 3`: Not found → Print `3`.
   - Check `i = 4`: Not found → Print `4`.
   - Check `i = 5`: Not found → Print `5`.
   - Check `i = 6`: Not found → Print `6`.
   - Check `i = 7`: Not found → Print `7`.
   - Check `i = 8`: Not found → Print `8`.
   - Check `i = 9`: Found (present in array).
5. Output:
   - Prints the missing numbers.

Output:

mathematica Copy code

```mathematica
Missing elements:
3
4
5
6
7
8
```

8. Move all zeros in an array to the end

```java
public class MoveZerosToEnd {
    public static void moveZerosToEnd(int[] arr) {
        int nonZeroIndex = 0;
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] != 0) {
                arr[nonZeroIndex++] = arr[i];
            }
        }
        while (nonZeroIndex < arr.length) {
            arr[nonZeroIndex++] = 0;
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 0, 7, 0, 4, 0, 5, 0};
        moveZerosToEnd(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Code Structure:

1. Class Declaration:
   - `public class MoveZerosToEnd`:
     - Defines the main class `MoveZerosToEnd`.
2. Method Declaration:
   - `public static void moveZerosToEnd(int[] arr)`:
     - Defines the method `moveZerosToEnd`, which takes an integer array as an argument and modifies it in-place to move all zeros to the end.
3. Steps in `moveZerosToEnd` Method:
   1. Initialize the Index for Non-Zero Elements:
      - `int nonZeroIndex = 0;`:
        - Initializes `nonZeroIndex` to track the position where the next non-zero element should be placed in the array.
   2. Loop Through the Array:
      - `for (int i = 0; i < arr.length; i++)`:
        - Loops through each element of the array.
   3. Move Non-Zero Elements to the Front:
      - `if (arr[i] != 0)`:
        - Checks if the current element is non-zero.
        - If it is, it places the element at the current position of `nonZeroIndex` and increments `nonZeroIndex`.
      - `arr[nonZeroIndex++] = arr[i];`:
        - Assigns the non-zero value to the position at `nonZeroIndex` and increments `nonZeroIndex`.
   4. Fill the Remaining Array with Zeros:
      - `while (nonZeroIndex < arr.length)`:
        - Once all non-zero elements are moved to the front, this loop fills the remaining positions with zeros.
      - `arr[nonZeroIndex++] = 0;`:
        - Sets the remaining elements ero by incrementing `nonZeroIndex` after assigning each zero.

`nonZeroIndex`.

4. **Fill the Remaining Array with Zeros:**
   - `while (nonZeroIndex < arr.length)`:
     - Once all non-zero elements are moved to the front, this loop fills the remaining positions with zeros.
   - `arr[nonZeroIndex++] = 0;`:
     - Sets the remaining elements to zero by incrementing `nonZeroIndex` after assigning each zero.

4. **Main Method:**
   - `public static void main(String[] args)`:
     - The entry point of the program.

   1. **Initialize Array:**
      - `int[] arr = {1, 0, 7, 0, 4, 0, 5, 0};`:
        - Defines an integer array with both non-zero and zero elements.
   2. **Call `moveZerosToEnd` Method:**
      - `moveZerosToEnd(arr);`:
        - Calls the `moveZerosToEnd` method to modify the array by moving zeros to the end.
   3. **Print the Modified Array:**
      - `for (int num : arr)`:
        - Loops through the modified array and prints each element.
      - `System.out.print(num + " ");`:
        - Prints the elements of the array after modification.

Execution:

1. **Input Array:**
   - `{1, 0, 7, 0, 4, 0, 5, 0}`
2. **Move Non-Zero Elements:**
   - The first loop moves non-zero elements to the front:
     - First `1` goes to index `0`.
     - Then `7` goes to index `1`.
     - Then `4` goes to index `2`.
     - Finally `5` goes to index `3`.
   - After moving non-zero elements, the array becomes:

   - The first loop moves non-zero elements to the front:
     - First `1` goes to index `0`.
     - Then `7` goes to index `1`.
     - Then `4` goes to index `2`.
     - Finally `5` goes to index `3`.
   - After moving non-zero elements, the array becomes:

     - `{1, 7, 4, 5, ?, ?, ?, ?}` (where `?` represents empty spots).
3. **Fill Remaining Spots with Zeros:**
   - The second loop fills the remaining positions with zeros:
     - Index `4`, `5`, `6`, and `7` are filled with zeros.
4. **Modified Array:**
   - `{1, 7, 4, 5, 0, 0, 0, 0}`

Output:

```
1 7 4 5 0 0 0 0
```

Key Points:

1. **Logic:**
   - The program efficiently moves all non-zero elements to the beginning of the array while maintaining their relative order.
   - Then, it fills the rest of the array with zeros.
2. **Efficiency:**
   - **Time Complexity:**
     - $O(n)$, where n is the length of the array.
     - Both loops (moving non-zeros and filling zeros) each run once through the array.
   - **Space Complexity:**
     - $O(1)$, since no extra space is used (the array is modified in place).
3. **Advantages:**
   - Efficient solution with linear time complexity.
   - In-place modification of the array without using extra space.

9. Move all odd numbers to the front and even numbers to the end in an array.

```java
public class Main {
    public static void moveOddEven(int[] arr) {
        int oddIndex = 0, evenIndex = arr.length - 1;
        while (oddIndex < evenIndex) {
            if (arr[oddIndex] % 2 != 0) {
                oddIndex++;
            } else if (arr[evenIndex] % 2 == 0) {
                evenIndex--;
            } else {
                int temp = arr[oddIndex];
                arr[oddIndex] = arr[evenIndex];
                arr[evenIndex] = temp;
                oddIndex++;
                evenIndex--;
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8};
        moveOddEven(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

- `public static void moveOddEven(int[] arr)`:
  - This method takes an integer array `arr` as input and rearranges its elements in-place such that all odd numbers are moved to the left and all even numbers to the right.

2. Index Initialization:
   - `int oddIndex = 0, evenIndex = arr.length - 1;`:
     - `oddIndex` starts from the beginning (index `0`), representing the position where the next odd number should be placed.
     - `evenIndex` starts from the end (index `arr.length - 1`), representing the position where the next even number should be placed.
3. While Loop:
   - `while (oddIndex < evenIndex)`:
     - The loop continues until `oddIndex` is less than `evenIndex`. This ensures that we process the entire array and avoid swapping elements that are already in their correct positions.
4. Checking Odd and Even Elements:
   - Odd Element at `oddIndex`:
     - `if (arr[oddIndex] % 2 != 0) { oddIndex++; }`:
       - If the element at `oddIndex` is odd (`arr[oddIndex] % 2 != 0`), increment `oddIndex` to move it to the next position.
   - Even Element at `evenIndex`:
     - `else if (arr[evenIndex] % 2 == 0) { evenIndex--; }`:
       - If the element at `evenIndex` is even (`arr[evenIndex] % 2 == 0`), decrement `evenIndex` to move it to the previous position.
5. Swap Logic:
   - `else` block (when one element is odd and the other is even):
     - Swap the elements at `oddIndex` and `evenIndex`:
       - `int temp = arr[oddIndex];`
       - `arr[oddIndex] = arr[evenIndex];`
       - `arr[evenIndex] = temp;`
     - After swapping, increment `oddIndex` and decrement `evenIndex` to continue processing the remaining elements.
     - `oddIndex++` and `evenIndex--` ensure that we move towards the center of the array.

6. Main Method:
   - `public static void main(String[] args)`:
     - This is the entry point of the program.
   - Initialize Array:
     - `int[] arr = {1, 2, 3, 4, 5, 6, 7, 8};`
     - The input array contains a mix of odd and even numbers.
   - Call `moveOddEven` Method:
     - `moveOddEven(arr);`
     - This rearranges the array such that odd numbers are on the left and even numbers on the right.
   - Print the Result:
     - `for (int num : arr) { System.out.print(num + " "); }`
     - This prints the modified array with the odd numbers on the left and even numbers on the right.

Execution:

1. Input Array:
   - `{1, 2, 3, 4, 5, 6, 7, 8}`
2. Process:
   - `oddIndex = 0`, `evenIndex = 7` (starts from opposite ends of the array).
3. Iterate:
   - First check: `arr[oddIndex] = 1` (odd) → `oddIndex++` → `oddIndex = 1`.
   - Next check: `arr[evenIndex] = 8` (even) → `evenIndex--` → `evenIndex = 6`.
   - Next check: `arr[oddIndex] = 2` (even) and `arr[evenIndex] = 7` (odd).
     - Swap: `arr[oddIndex] = 7` and `arr[evenIndex] = 2`.
     - Array after swap: `{1, 7, 3, 4, 5, 6, 2, 8}`.
   - Continue checking and swapping elements until the entire array is processed.
4. Final Array:
   - `{1, 7, 3, 5, 4, 6, 2, 8}`

Output:

```
Copy code
1 7 3 5 4 6 2 8
```

Key Points:

1. Logic:
   - The program uses two indices (`oddIndex` and `evenIndex`) to traverse the array from both ends and rearrange the elements without using extra space.
   - Odd numbers are moved to the front and even numbers are moved to the back by swapping elements when necessary.
2. Efficiency:
   - Time Complexity:
     - $O(n)$, where n is the length of the array. We are iterating over the array once, performing constant-time operations for each element.
   - Space Complexity:
     - $O(1)$, since the array is modified in place without using extra space (apart from a few integer variables).
3. Advantages:
   - Efficient approach with a single pass through the array.
   - In-place swapping minimizes the use of additional memory.
4. Edge Cases:
   - Array with all odd numbers:
     - If the array contains only odd numbers, the array will remain unchanged.
     - Example: `{1, 3, 5}` → Output: `{1, 3, 5}`.
   - Array with all even numbers:
     - If the array contains only even numbers, the array will remain unchanged.
     - Example: `{2, 4, 6}` → Output: `{2, 4, 6}`.
   - Array with only one element:
     - If the array contains a single element, it will remain unchanged.
     - Example: `{1}` → Output: `{1}`.

10. Reverse an array in subsets of size N.

```java
public class ReverseInSubsets {
    public static void reverseInSubsets(int[] arr, int N) {
        for (int i = 0; i < arr.length; i += N) {
            int left = i;
            int right = Math.min(i + N - 1, arr.length - 1);
            while (left < right) {
                int temp = arr[left];
                arr[left] = arr[right];
                arr[right] = temp;
                left++;
                right--;
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        reverseInSubsets(arr, 3);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

1. Class Declaration:
   - `public class ReverseInSubsets`:
     - This defines the `ReverseInSubsets` class.
2. Method Declaration:
   - `public static void reverseInSubsets(int[] arr, int N)`:
     - This method takes two parameters:
       - `arr`: the input array that needs to be processed.
       - `N`: the size of the subsets that need to be reversed.
3. Loop Through the Array:
   - `for (int i = 0; i < arr.length; i += N)`:
     - This loop runs through the array in increments of `N`, meaning that each iteration processes a subset of size `N` in the array.
4. Calculate Subset Boundaries:
   - `int left = i;`:
     - `left` starts at the current index `i`, which represents the start of the subset.
   - `int right = Math.min(i + N - 1, arr.length - 1);`:
     - `right` is the end of the subset. It is calculated as `i + N - 1` (which would be the last index of the subset), but it ensures that it does not go beyond the last index of the array by using `Math.min`.
5. Reverse the Subset:
   - `while (left < right)`:
     - The `while` loop runs until the `left` and `right` indices meet or cross.
     - Swap Elements:
       - `int temp = arr[left];`:
         - A temporary variable `temp` stores the value of the element at the `left` index.
       - `arr[left] = arr[right];`:
         - The element at the `right` index is assigned to the `left` index.
       - `arr[right] = temp;`:
         - The value stored in `temp` is assigned to the `right` index.
     - Move Indices:
       - `left++` and `right--` increment and decrement the indices, respectively, moving toward the center of the subset, effectively reversing the elements in the subset.

6. Main Method:
   - `public static void main(String[] args)`:
     - The entry point of the program.
   1. Initialize Array:
      - `int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};`:
        - The input array for the program, which contains numbers from `1` to `9`.
   2. Call `reverseInSubsets` Method:
      - `reverseInSubsets(arr, 3);`:
        - This call reverses the array in subsets of size 3.
   3. Print the Result:
      - `for (int num : arr) { System.out.print(num + " "); }`:
        - This prints the modified array after reversing it in subsets.

Execution:

1. Input Array:
   - `{1, 2, 3, 4, 5, 6, 7, 8, 9}`
2. First Subset (`N = 3`):
   - Subset: `{1, 2, 3}`.
   - Reverse this subset to get `{3, 2, 1}`.
3. Second Subset (`N = 3`):
   - Subset: `{4, 5, 6}`.
   - Reverse this subset to get `{6, 5, 4}`.
4. Third Subset (`N = 3`):
   - Subset: `{7, 8, 9}`.
   - Reverse this subset to get `{9, 8, 7}`.
5. Final Array:
   - After reversing all subsets, the array becomes: `{3, 2, 1, 6, 5, 4, 9, 8, 7}`.

Output:

Copy code

```
3 2 1 6 5 4 9 8 7
```

Key Points:

1. Logic:
   - The program divides the array into subsets of size `N` and reverses each subset in place without using extra memory.
   - For each subset, the program uses two indices (`left` and `right`) to swap the elements until they meet in the middle.
2. Efficiency:
   - Time Complexity:
     - $O(n)$, where n is the length of the array.
     - Each element in the array is processed once in the reversing process.
   - Space Complexity:
     - $O(1)$, as the array is modified in place and no extra space is used.
3. Edge Cases:
   - Array with fewer than `N` elements:
     - If the array has fewer than `N` elements, the program will simply reverse the entire array.
     - Example: For an array `{1, 2}` and `N = 3`, the entire array is reversed to `{2, 1}`.
   - Array with exactly `N` elements:
     - If the array size is equal to `N`, the program will reverse the entire array.
     - Example: For an array `{1, 2, 3}` and `N = 3`, the array becomes `{3, 2, 1}`.
   - Array with less than or equal to 1 element:
     - If the array has 0 or 1 element, no reversing is required.

How It Works:

1. The program divides the array into subsets of size `N`.
2. For each subset, it reverses the order of elements using a two-pointer approach, where `left` and `right` indices swap elements until they meet.
3. The process continues for all subsets, and the final array with reversed subsets is returned.

11. Count Odd & Even Numbers in an Array

```java
public class CountOddEven {
    public static void countOddEven(int[] arr) {
        int oddCount = 0, evenCount = 0;
        for (int num : arr) {
            if (num % 2 == 0) {
                evenCount++;
            } else {
                oddCount++;
            }
        }
        System.out.println("Odd Count: " + oddCount);
        System.out.println("Even Count: " + evenCount);
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8};
        countOddEven(arr);
    }
}
```

1. **Class Declaration:**
   - `public class CountOddEven`:
     - The class `CountOddEven` is declared.
2. **Method Declaration:**
   - `public static void countOddEven(int[] arr)`:
     - This method takes an integer array `arr` as an input.
     - It counts how many odd and even numbers are in the array and prints the results.
3. **Initialize Counters:**
   - `int oddCount = 0, evenCount = 0;`:
     - Two counters are initialized:
       - `oddCount` to track the number of odd elements.
       - `evenCount` to track the number of even elements.
4. **Loop Through the Array:**
   - `for (int num : arr)`:
     - This loop iterates through each element `num` in the array `arr`.
5. **Check Odd or Even:**
   - `if (num % 2 == 0)`:
     - The condition checks whether the number is divisible by 2 (i.e., an even number).
     - If the condition is `true`, it means the number is even, so `evenCount` is incremented.
     - If the condition is `false`, it means the number is odd, so `oddCount` is incremented.
6. **Print Results:**
   - After iterating through the array, the program prints:
     - `System.out.println("Odd Count: " + oddCount);`
     - `System.out.println("Even Count: " + evenCount);`

6. **Print Results:**
   - After iterating through the array, the program prints:
     - `System.out.println("Odd Count: " + oddCount);`
     - `System.out.println("Even Count: " + evenCount);`
     - These lines display the counts of odd and even numbers.
7. **Main Method:**
   - `public static void main(String[] args)`:
     - The entry point of the program.

   1. **Initialize Array:**
      - `int[] arr = {1, 2, 3, 4, 5, 6, 7, 8};`:
        - The input array, which contains numbers from 1 to 8.
   2. **Call `countOddEven` Method:**
      - `countOddEven(arr);`:
        - The method `countOddEven` is called to count the odd and even numbers in the array.

Execution:

1. **Input Array:**
   - `{1, 2, 3, 4, 5, 6, 7, 8}`
2. **Counting Odd and Even Numbers:**
   - Odd numbers in the array: `{1, 3, 5, 7}` → 4 odd numbers.
   - Even numbers in the array: `{2, 4, 6, 8}` → 4 even numbers.

Execution:

1. **Input Array:**
   - `{1, 2, 3, 4, 5, 6, 7, 8}`
2. **Counting Odd and Even Numbers:**
   - Odd numbers in the array: `{1, 3, 5, 7}` → 4 odd numbers.
   - Even numbers in the array: `{2, 4, 6, 8}` → 4 even numbers.
3. **Output:**
   - The program will output:

```mathematica
Odd Count: 4
Even Count: 4
```

Key Points:

1. **Logic:**
   - The program loops through the array and checks whether each number is even or odd by using the modulo operator (`%`).
   - The number is classified as even if `num % 2 == 0`; otherwise, it is classified as odd.
2. **Efficiency:**
   - **Time Complexity:**
     - $O(n)$, where n is the length of the array.
     - The program iterates through each element in the array once.
   - **Space Complexity:**
     - $O(1)$, as only a constant amount of extra space is used (just the two counters).
3. **Edge Cases:**
   - **Empty Array:** If the array is empty, the program will output:

```mathematica
Odd Count: 0
Even Count: 0
```

   - **Array with All Odd or All Even:** The program will accurately count the odd and even numbers even if the array contains only odd or only even numbers.

12. Remove Duplicates in an Arrayusing Hashset

```java
import java.util.HashSet;

public class RemoveDuplicatesArray {
    public static void removeDuplicates(int[] arr) {
        HashSet<Integer> set = new HashSet<>();
        for (int num : arr) {
            set.add(num);
        }
        System.out.println("Array without duplicates: " + set);
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 3, 5, 2};
        removeDuplicates(arr);
    }
}
```

Code Breakdown:

1. Class Declaration:
   - `public class RemoveDuplicatesArray`:
     - The class `RemoveDuplicatesArray` is defined to handle removing duplicates from the array.
2. Method Declaration:
   - `public static void removeDuplicates(int[] arr)`:
     - This method takes an integer array `arr` as input and removes the duplicates.
3. Initialize a HashSet:
   - `HashSet<Integer> set = new HashSet<>();`:
     - A `HashSet` named `set` is created to store the unique elements from the array.
     - Since `HashSet` automatically handles uniqueness, it will only store each element once, removing any duplicates.
4. Iterate Over the Array:
   - `for (int num : arr)`:
     - A `for-each` loop is used to iterate through each element `num` in the input array `arr`.
5. Add Each Element to the HashSet:
   - `set.add(num);`:
     - The current element `num` is added to the `set`. If the element is already present, it will not be added again because a `HashSet` only stores unique values.
6. Print the Result:
   - `System.out.println("Array without duplicates: " + set);`:
     - After iterating through the entire array, the program prints the contents of the `set`, which now contains only the unique elements from the array.
7. Main Method:
   - `public static void main(String[] args)`:
     - The entry point of the program.
   1. Initialize Array:
      - `int[] arr = {1, 2, 3, 4, 3, 5, 2};`:
        - The input array, which contains some duplicate values: `{1, 2, 3, 4, 3, 5, 2}`.

1. Initialize Array:

   - `int[] arr = {1, 2, 3, 4, 3, 5, 2};` :
     - The input array, which contains some duplicate values: `{1, 2, 3, 4, 3, 5, 2}`.

2. Call `removeDuplicates` Method:

   - `removeDuplicates(arr);` :
     - This call removes the duplicate elements from the array and prints the result.

Execution:

1. Input Array:
   - `{1, 2, 3, 4, 3, 5, 2}`
2. Process:
   - The `HashSet` stores the elements as follows:
     - `set.add(1)` → `{1}`
     - `set.add(2)` → `{1, 2}`
     - `set.add(3)` → `{1, 2, 3}`
     - `set.add(4)` → `{1, 2, 3, 4}`
     - `set.add(3)` → `{1, 2, 3, 4}` (Duplicate 3 is ignored)
     - `set.add(5)` → `{1, 2, 3, 4, 5}`
     - `set.add(2)` → `{1, 2, 3, 4, 5}` (Duplicate 2 is ignored)
3. Output:
   - The program prints the `HashSet`, which contains only the unique elements from the array:

```sql
Array without duplicates: [1, 2, 3, 4, 5]
```

Key Points:

1. Logic:
   - The program uses a `HashSet` to automatically filter out duplicate values from the input array, as a `HashSet` does not allow duplicates.
2. Efficiency:
   - Time Complexity:
     - $O(n)$, where n is the number of elements in the array. Each element is added to the `HashSet` in constant time on aver-ge.

Key Points:

1. Logic:
   - The program uses a `HashSet` to automatically filter out duplicate values from the input array, as a `HashSet` does not allow duplicates.
2. Efficiency:
   - Time Complexity:
     - $O(n)$, where n is the number of elements in the array. Each element is added to the `HashSet` in constant time on average.

   - Space Complexity:
     - $O(n)$, as the `HashSet` stores all unique elements of the array.
3. Edge Cases:
   - Empty Array: If the array is empty, the program will print:

```sql
Array without duplicates: []
```

   - Array with All Unique Elements: If the array has no duplicates, the output will be the same as the input array.
   - Array with All Same Elements: If the array has all identical elements, the program will print just one element.
4. Additional Information:
   - The `HashSet` guarantees that there will be no duplicates in the resulting set.
   - The program does not preserve the order of elements as `HashSet` does not maintain the insertion order (though in this case, the elements are printed in arbitrary order due to the nature of `HashSet`).

How It Works:

1. The program initializes a `HashSet` to store unique elements from the array.
2. It iterates over the array and adds each element to the `HashSet`.
3. Since `HashSet` only stores unique elements, duplicates are automatically ignored.
4. Finally, it prints the elements of the `HashSet`, which contains only the unique elements.

13. Remove Duplicates from ArrayList

```java
import java.util.ArrayList;
import java.util.HashSet;

public class RemoveDuplicatesArrayList {
    public static void removeDuplicates(ArrayList<Integer> list) {
        HashSet<Integer> set = new HashSet<>(list);
        list.clear();
        list.addAll(set);
        System.out.println("ArrayList without duplicates: " + list);
    }

    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(3);
        list.add(2);
        removeDuplicates(list);
    }
}
```

Code Breakdown:

1. Class Declaration:
   - `public class RemoveDuplicatesArrayList`:
     - The class `RemoveDuplicatesArrayList` is defined to handle removing duplicates from an `ArrayList`.
2. Method Declaration:
   - `public static void removeDuplicates(ArrayList<Integer> list)`:
     - This method takes an `ArrayList<Integer>` as input and removes duplicates from the list.
3. Using HashSet to Remove Duplicates:
   - `HashSet<Integer> set = new HashSet<>(list);`:
     - A `HashSet` is created from the `ArrayList`. The constructor of `HashSet` automatically removes duplicates because a `HashSet` does not allow duplicate entries.
4. Clearing the Original List:
   - `list.clear();`:
     - The original `ArrayList` is cleared, removing all of its elements.
5. Adding Unique Elements Back:
   - `list.addAll(set);`:
     - All the unique elements (from the `HashSet`) are added back into the `ArrayList`. This ensures the list only contains unique values.
6. Print the Result:
   - `System.out.println("ArrayList without duplicates: " + list);`:
     - The updated `ArrayList`, which now contains only unique elements, is printed.
7. Main Method:
   - `public static void main(String[] args)`:
     - The entry point of the program.
   1. Initialize ArrayList:
      - `ArrayList<Integer> list = new ArrayList<>();`:
        - An empty `ArrayList` of integers is created.

- An empty `ArrayList` of integers is created.

2. Add Elements to ArrayList:
   - `list.add(1); list.add(2); list.add(3); list.add(4); list.add(3); list.add(2);` :
     - Several integers, including duplicates, are added to the `ArrayList`.
3. Call `removeDuplicates` Method:
   - `removeDuplicates(list);` :
     - The method `removeDuplicates` is called to remove duplicate elements from the `ArrayList`.

Execution:

1. Input ArrayList:
   - `{1, 2, 3, 4, 3, 2}`
2. Process:
   - A `HashSet` is created from the `ArrayList`, removing duplicates automatically:
     - `set = {1, 2, 3, 4}` (duplicates removed)
   - The original `ArrayList` is cleared and the unique elements are added back into it.
3. Output:
   - The program prints the updated `ArrayList`:

```less
ArrayList without duplicates: [1, 2, 3, 4]
```

Key Points:

1. Logic:
   - The program uses a `HashSet` to eliminate duplicates because `HashSet` automatically ensures that all elements are unique.
   - The original list is cleared, and the unique elements from the `HashSet` are added back to the `ArrayList`.
2. Efficiency:
   - Time Complexity:
     - $O(n)$, where n is the number of elements in the `ArrayList`. The insertion of elements into the `HashSet` and the addition of unique elements back to the `ArrayList` takes linear time.

2. Efficiency:
   - Time Complexity:
     - $O(n)$, where n is the number of elements in the `ArrayList`. The insertion of elements into the `HashSet` and the addition of unique elements back to the `ArrayList` takes linear time.
   - Space Complexity:
     - $O(n)$, as a `HashSet` is used to store the unique elements, which requires additional space.
3. Edge Cases:
   - Empty List: If the `ArrayList` is empty, the output will be:

```less
ArrayList without duplicates: []
```

   - List with All Unique Elements: If the list has no duplicates, the output will be the same as the input list.
   - List with All Same Elements: If all elements are the same, the output will contain just one element.
4. Additional Information:
   - The order of the elements may not be preserved, as `HashSet` does not maintain the order of insertion (unless using `LinkedHashSet`, which preserves insertion order).
   - This approach does not handle primitive arrays; it is specifically for `ArrayList` objects.

How It Works:

1. The program first creates a `HashSet` from the `ArrayList`, which removes all duplicate elements automatically.
2. The original `ArrayList` is cleared, and the unique elements from the `HashSet` are added back into the list.
3. Finally, the updated `ArrayList` is printed, showing only the unique elements.

14. Search an Element in an Array

```java
public class SearchElement {
    public static boolean searchElement(int[] arr, int target) {
        for (int num : arr) {
            if (num == target) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int target = 3;
        System.out.println("Element found: " + searchElement(arr, target));
    }
}
```

Code Breakdown:

1. Class Declaration:
   - `public class SearchElement`:
     - The class `SearchElement` is defined to handle the search operation in an array.
2. Method Declaration:
   - `public static boolean searchElement(int[] arr, int target)`:
     - This method takes an integer array `arr` and an integer `target` as input parameters.
     - It returns `true` if the target element is found in the array and `false` if it is not found.
3. Iterate Over the Array:
   - `for (int num : arr)`:
     - A `for-each` loop is used to iterate through each element `num` in the array `arr`.
4. Check for Target Element:
   - `if (num == target)`:
     - Inside the loop, the code checks if the current element `num` is equal to the `target` element.
     - If a match is found, the method returns `true`.
5. Return `false` if Target is Not Found:
   - `return false;`:
     - If the loop completes without finding the `target`, the method returns `false`.
6. Main Method:
   - `public static void main(String[] args)`:
     - The entry point of the program.
   1. Initialize Array:
      - `int[] arr = {1, 2, 3, 4, 5};`:
        - The input array, containing integer elements: `{1, 2, 3, 4, 5}`.
   2. Set Target:
      - `int target = 3;`:
        - The `target` element to search for in the array, set to `3`.

3. Call `searchElement` Method:
   - `System.out.println("Element found: " + searchElement(arr, target));` :
     - The `searchElement` method is called to check if the `target` element exists in the array.
     - The result (`true` or `false`) is printed.

Execution:

1. Input Array:
   - `{1, 2, 3, 4, 5}`
2. Target Element:
   - `3`
3. Process:
   - The program iterates through the array:
     - For `num = 1` : It is not equal to `3`.
     - For `num = 2` : It is not equal to `3`.
     - For `num = 3` : It is equal to `3`, so the method returns `true`.
4. Output:
   - The program prints:

```yaml
Element found: true
```

Key Points:

1. Logic:
   - The program checks each element of the array against the target.
   - If a match is found, it returns `true`. Otherwise, it returns `false` after checking all elements.
2. Efficiency:
   - Time Complexity:
     - $O(n)$, where n is the number of elements in the array. In the worst case, the program needs to check each element once.
   - Space Complexity:
     - $O(1)$, as the program uses a constant amount of extra space (only a few variables).

Key Points:

1. Logic:
   - The program checks each element of the array against the target.
   - If a match is found, it returns `true`. Otherwise, it returns `false` after checking all elements.
2. Efficiency:
   - Time Complexity:
     - $O(n)$, where n is the number of elements in the array. In the worst case, the program needs to check each element once.
   - Space Complexity:
     - $O(1)$, as the program uses a constant amount of extra space (only a few variables).
3. Edge Cases:
   - **Element Found:** The program correctly returns `true` if the element exists in the array.
   - **Element Not Found:** If the element doesn't exist in the array, it returns `false`.
   - **Empty Array:** If the array is empty, the method will immediately return `false`.
4. Additional Information:
   - This is a linear search algorithm, which is simple and effective for unsorted arrays or small arrays.
   - If the array were sorted, more efficient search methods like binary search could be used, but this method works on any array without the need for sorting.

How It Works:

1. The program loops through the array to check each element against the target.
2. If the target is found, it returns `true`.
3. If the target is not found after checking all elements, it returns `false`.

15. Sort an Array

```java
public class SortArrayWithoutSort {
    public static void sortArrayWithoutSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] > arr[j]) {
                    int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
        }
        System.out.println("Sorted Array: " + Arrays.toString(arr));
    }

    public static void main(String[] args) {
        int[] arr = {5, 3, 8, 1, 2};
        sortArrayWithoutSort(arr);
    }
}
```

Code Breakdown:

1. Class Declaration:
   - `public class SortArrayWithoutSort`:
     - The class `SortArrayWithoutSort` is defined to implement the sorting logic.
2. Method Declaration:
   - `public static void sortArrayWithoutSort(int[] arr)`:
     - This method takes an integer array `arr` as input and sorts it in ascending order using the bubble sort algorithm.
3. Outer Loop (Iterating Through Array):
   - `for (int i = 0; i < arr.length - 1; i++)`:
     - This loop runs through the entire array from the first element to the second-to-last element.
     - `i` represents the current index being processed.
4. Inner Loop (Comparing Elements):
   - `for (int j = i + 1; j < arr.length; j++)`:
     - The inner loop starts at the element immediately following `arr[i]` (i.e., `i + 1`) and goes to the last element of the array.
     - This loop compares each element with the element at index `i`.
5. Swapping Elements if Not in Order:
   - `if (arr[i] > arr[j])`:
     - If the current element `arr[i]` is greater than `arr[j]`, the elements are swapped.
   - `int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;`:
     - The elements at indices `i` and `j` are swapped using a temporary variable `temp`.
     - This ensures that the smaller element is placed earlier in the array.
6. Printing the Sorted Array:
   - `System.out.println("Sorted Array: " + Arrays.toString(arr));`:
     - After sorting is complete, the array is printed using `Arrays.toString(arr)` to format it as a string.

7. Main Method:
  - `public static void main(String[] args)`:
    - The entry point of the program.
  1. Initialize Array:
    - `int[] arr = {5, 3, 8, 1, 2};`:
      - The input array `{5, 3, 8, 1, 2}` is provided to the method for sorting.
  2. Call `sortArrayWithoutSort` Method:
    - `sortArrayWithoutSort(arr);`:
      - The `sortArrayWithoutSort` method is called to sort the array.

Execution:

1. Input Array:
   - `{5, 3, 8, 1, 2}`
2. Process:
   - The outer loop (`i`) iterates over the elements, and for each element at index `i`, the inner loop (`j`) compares it with the remaining elements to the right (`arr[j]`).
   - Whenever an element is greater than the element being compared, they are swapped.
   - This process repeats until the array is fully sorted.
3. Sorted Array:
   - The array is sorted in ascending order: `{1, 2, 3, 5, 8}`.
4. Output:
   - The program prints the sorted array:

```javascript
Sorted Array: [1, 2, 3, 5, 8]
```

Key Points:

1. Logic:
   - The program implements **Bubble Sort**, where each element is compared with the next one, and if they are in the wrong order, they are swapped.
   - This continues until the array is sorted.

Key Points:

1. Logic:
   - The program implements **Bubble Sort**, where each element is compared with the next one, and if they are in the wrong order, they are swapped.
   - This continues until the array is sorted.
2. Efficiency:
   - Time Complexity:
     - $O(n^2)$, where n is the number of elements in the array. In the worst case, the algorithm performs n^2 comparisons and swaps.
   - Space Complexity:
     - $O(1)$, since the algorithm sorts the array in-place and does not require any extra space except for the temporary variable used in swapping.
3. Edge Cases:
   - **Empty Array:** If the array is empty, no changes will occur, and the output will be `Sorted Array: []`.
   - **Array with One Element:** If the array contains only one element, it is already sorted, and the output will be that element.
   - **Array with All Elements Equal:** If all elements are the same, the output will be the same array with no changes.
4. Comparison with Built-In Sort:
   - The program uses a manual sorting technique (Bubble Sort) instead of `Arrays.sort()`. While this demonstrates how sorting can be done from scratch, the built-in sorting methods in Java are much more efficient (with a time complexity of $O(n \log n)$).

How It Works:

1. The outer loop iterates through each element of the array.
2. The inner loop compares the current element with every element that follows it.
3. If the current element is greater than the next element, the two elements are swapped.
4. This continues until the array is sorted, and the sorted array is printed.