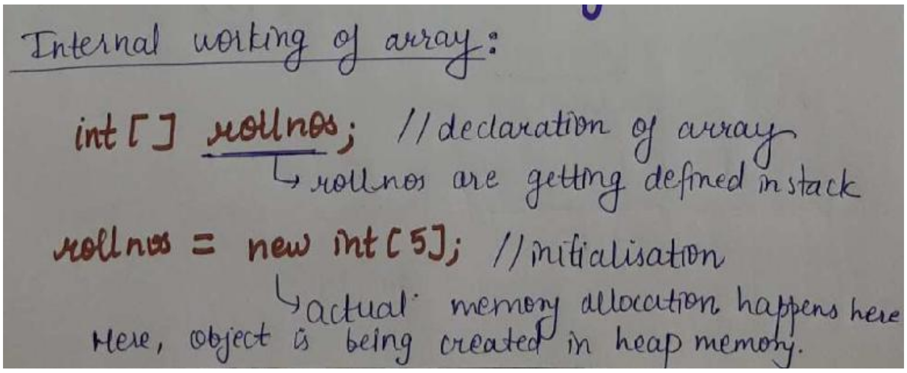
Arrays in Java

An **array** is a **fixed-size** collection of elements of the **same type**, stored in **contiguous memory locations**.

Key Properties of Arrays:

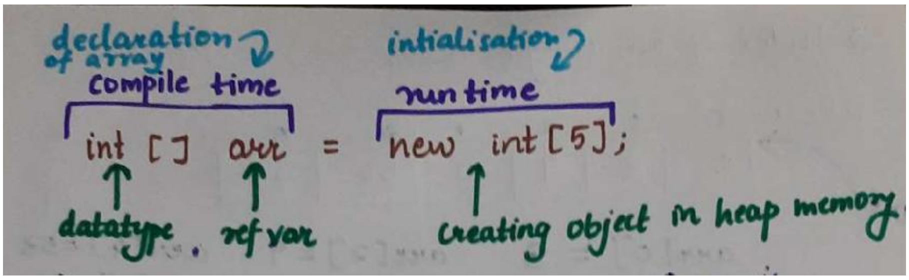
- **Fixed Size**: The size is defined at declaration and cannot be changed.
- **Zero-Based Indexing**: First element is at index 0.
- **Homogeneous Elements**: All elements must be of the same type.
- **Stored in Contiguous Memory**: Allows fast access but lacks dynamic resizing.



Internal working of array:

```
int[] rollnos; //declaration of array
      ↳ rollnos are getting defined in stack
rollnos = new int[5]; //initialisation
      ↳ actual memory allocation happens here
```

Here, object is being created in heap memory.



```
int [] arr = new int[5];
```

Accessing an **index out of bounds** (arr[10] when size is 5) throws **ArrayIndexOutOfBoundsException**.

For Traversing an Array use for loop or enhanced for loop (For-Each), for reach is best for read-only operations.

How Arrays are Stored in Memory

- When an array is created, **a continuous block of memory is allocated** in the heap.
- **Array elements are stored contiguously**, making random access (arr[i]) **fast (O(1))**.
- If the array is of **primitives (int, double, etc.)**, the values are **directly stored in the array**.

- If the array is of **objects (String[], Integer[])**, the array stores **references** to objects in heap memory.

Why is an array's size fixed in Java?
Arrays are allocated in contiguous memory, and resizing would require allocating a new array and copying elements.

Memory Layout of an Array

Every array in Java consists of:

1. **Header** (Metadata): Stores information like array length, type, and reference.
2. **Actual Data** (Elements): Stored in contiguous memory locations.

Understanding Array Object Representation in JVM

Array Header Information

- Every array in Java is an object that contains **header information** before the actual data.
- The **header** includes:
    1. **Type of array** (e.g., int[], double[], Object[], etc.)
    2. **Array length**
    3. **Garbage Collection metadata**

Internal Representation of Arrays in Java

1. Are Java Array Elements Stored in Contiguous Memory?

- **Yes, for primitive type arrays (int[], double[], char[], etc.)**.
- **No, for object type arrays (String[], CustomClass[])** because they store references, not actual objects.

2. Primitive Type Arrays (int[], double[], char[], etc.)

- **Stored in contiguous memory locations** in the heap.
- Each element is placed **one after another** in memory.

- This ensures **fast indexing (O(1))**.

Example:

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

Memory Representation (Contiguous Allocation in Heap):

Address	**Value**
0x1000	10
0x1004	20
0x1008	30
0x100C	40

Why Contiguous?

- Since **primitive types** have a **fixed size**, Java can allocate memory **sequentially**.
- This allows efficient **cache locality** and **faster memory access**.

3. Object Type Arrays (String[], CustomClass[])

- **Not stored in contiguous memory**.
- The array **stores references (addresses) to actual objects**.
- The objects themselves **are stored in different locations in the heap**.

Example:

String[] arr = {"Java", "Python", "C++"};

Memory Representation:

Array Address	**Value (Reference to Object)**
0x2000	0x3000 (points to "Java")
0x2004	0x4000 (points to "Python")
0x2008	0x5000 (points to "C++")

- **"Java"**, **"Python"**, and **"C++"** are stored separately in **heap memory**, and the array holds **only references** to them.

Why Not Contiguous?

- **Object sizes vary**, so Java cannot allocate them contiguously.

- Each element is just a **reference (4 or 8 bytes depending on JVM) pointing to different heap locations**.

4. Multi-Dimensional Arrays (2D Arrays)

Are 2D Arrays Contiguous?

- **No, because Java implements 2D arrays as an array of arrays.**
- Each **row is a separate array** stored in different locations.

Example:

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

Memory Layout (Non-Contiguous Storage):

```
matrix ---> [0x5000]  // Row 0 reference
         [0x6000]  // Row 1 reference
0x5000 ---> [1, 2, 3] (Heap)
0x6000 ---> [4, 5, 6] (Heap)
```

- Each row is an **independent object stored in different heap locations**.
- This allows **jagged arrays**, where different rows can have different sizes.

5. Conclusion

Array Type	**Contiguous Memory?**	**Why?**
Primitive Arrays (int[], char[], double[])	**Yes**	Fixed-size elements allow sequential memory allocation.
Object Arrays (String[], UserDefinedClass[])	**No**	Stores references, and objects are allocated separately in heap.
Multi-dimensional Arrays (int[][], Object[][])	**No**	Implemented as an array of arrays, each row is separate.

JVM Memory Management for Arrays in Java

1. How JVM Allocates Memory for Arrays?

When an array is created in Java, the **JVM (Java Virtual Machine)** allocates memory for it in the **heap memory**. The allocation process depends on whether the array contains **primitives** or **objects**.

Memory Areas in JVM:

Memory Area	Description
Heap	Stores array objects and dynamically allocated memory.
Stack	Stores references to arrays and local variables.
Method Area (MetaSpace in Java 8+)	Stores class definitions and static fields.
PC Register & Native Stack	Internal JVM processing.

2. Memory Layout for Primitive Type Arrays

Example:

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

- The **array object is stored in the heap**.
- The **reference to the array is stored in the stack**.
- **Efficient cache utilization** due to contiguous storage.

3. Memory Layout for Object Type Arrays

```
String[] arr = {"Java", "Python", "C++"};
```

- The **array itself is stored in the heap**.
- Each **element is a reference to an object in the heap**.
- **References are stored contiguously, but actual objects are not.**
- **Accessing an element requires two memory lookups**:
    1. First to get the reference.
    2. Second to get the object.

4. Garbage Collection & Arrays

- **Arrays in Java are eligible for garbage collection** when they are **no longer referenced**.
- The **Garbage Collector (GC) automatically frees memory** when an array goes out of scope.

Example (Eligible for GC):

```
int[] arr = new int[5];  // Array allocated in heap

arr = null;  // The original array is now unreachable → Eligible for GC
```

- When arr = null;, the old array becomes **unreachable**, so JVM **marks it for garbage collection**.
- This **frees memory** for new allocations.

5. Array Copying and Memory Impact

When arrays are copied, **JVM either creates a new array or copies references**.

Shallow Copy (Reference Copy)

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

int[] copy = original;  // Both point to the same memory
```

- **No new memory is allocated.**
- Both original and copy point to the same array in **heap memory**.

Deep Copy (New Memory Allocation)

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

int[] copy = Arrays.copyOf(original, original.length);  // New array created
```

- **A new array is created** in heap memory.
- Changes in copy **do not affect** original.

6. JVM Optimization for Arrays

- **Escape Analysis:** Determines if an array is used within a method. If yes, it may be allocated on the **stack** instead of the heap.
- **Compressed Oops:** Optimizes memory usage for object references in 64-bit JVMs.
- **Array Bounded Checking:** JVM checks for **out-of-bounds access** to prevent segmentation faults.

Interview Questions on JVM Array Memory Management

Q1: Where are arrays stored in Java memory?

A: Arrays are stored in the **heap memory**, and references to them are stored in the **stack memory**.

Q2: Are primitive type arrays stored contiguously in memory?

A: Yes, primitive type arrays (int[], char[], etc.) are stored in **contiguous memory locations**.

Q3: Why are object type arrays not stored contiguously?

A: Because object arrays **store references**, not actual objects. The referenced objects are allocated **separately in heap memory**.

Q4: How are 2D arrays stored in memory?

A: 2D arrays are stored as **arrays of arrays**, where each row is a separate array stored at different memory locations.

Q5: What happens when an array is set to null?

A: The array becomes **unreachable** and is **eligible for garbage collection**.

Q6: What is the difference between a shallow copy and a deep copy in arrays?

A:

- **Shallow Copy:** Copies only references, so changes in one affect the other.
- **Deep Copy:** Creates a new array with duplicated values.

9. Summary

Concept	Details
Where are arrays stored?	Heap memory
Are primitive type arrays contiguous?	Yes
Are object type arrays contiguous?	No (only references are stored contiguously)
Are 2D arrays stored in a single block?	No, each row is stored separately
How does JVM optimize array storage?	Escape analysis, compressed Oops, array bound checking
What happens when an array is nullified?	Eligible for garbage collection

Array vs. ArrayList

Feature	Array	ArrayList
Size	Fixed at declaration	Dynamically resizable
Type	Can store **primitives & objects**	Stores only **objects** (autoboxing for primitives)
Performance	Faster (direct memory access)	Slower (extra memory for dynamic resizing)
Memory Usage	Less (no overhead)	More (additional wrapper class and growth strategy)
Adding Elements	Manual index assignment	add() method
Removing Elements	Manual shifting	remove() method shifts elements automatically
Flexibility	Static, cannot grow/shrink	Can dynamically grow/shrink
Syntax	int[] arr = new int[5];	ArrayList<Integer> list = new ArrayList<>();

2. Performance & Memory Considerations

When to use Array?

Use arrays when:

- The size is **fixed**.
- Performance is **critical** (no resizing overhead).
- Working with **primitive types** (avoids autoboxing).

Example (Primitive Type Array - Less Memory Overhead)

```
int[] arr = new int[1000];  // Directly allocated in heap memory
```

Efficient because primitive types don’t require extra object wrapping.

When to use ArrayList?

Use ArrayList when:

- The size is **unknown or dynamic**.
- You need **convenience methods** (add(), remove(), contains()).
- Objects are used (e.g., String, Employee).

Example (ArrayList - Higher Memory Usage due to Wrapper Classes)

```
ArrayList<Integer> list = new ArrayList<>();
for (int i = 0; i < 1000; i++) {
   list.add(i);  // Each int is autoboxed into Integer object
}
```

Integer takes extra memory because it’s an object, not a primitive.

Internal Working of ArrayList

- **ArrayList uses an internal array (Object[] elementData).**
- **When you add an element, it stores the object reference in the array.**
- **If capacity is exceeded, it creates a new larger array (1.5x size) and copies elements.**

How ArrayList Grows Internally?

1. **Initial Capacity:** Default is 10.
2. **Expansion:** When full, size increases by **1.5x (newCapacity = oldCapacity + (oldCapacity / 2))**.
3. **Reallocation:** A **new array** is created and elements are copied.

Example: Resizing in Action

```
ArrayList<Integer> list = new ArrayList<>(3);

list.add(1);

list.add(2);

list.add(3); // Capacity = 3

list.add(4); // Triggers resizing (new capacity = 3 + 3/2 = 4)
```

Avoid Frequent Resizing: If size is known, use:

```
ArrayList<Integer> list = new ArrayList<>(1000); // Preallocates memory
```

Pre-allocating prevents multiple resize operations, improving performance.

ArrayList uses extra memory for object wrappers (autoboxing overhead).

Memory Optimization Techniques

1. Prefer int[] Over ArrayList<Integer> for Large Data

Arrays are more memory-efficient for primitive data:

```
int[] arr = new int[1000000];  // More efficient than ArrayList<Integer>
```

2. Pre-Allocate ArrayList Size if Known

Avoid multiple resizes:

```
ArrayList<Integer> list = new ArrayList<>(1000);  // Reduces unnecessary reallocations
```

How Garbage Collection Works in ArrayList

- **ArrayList uses an internal array (Object[]).**
- When elements are **removed**, the reference is set to null, but **GC doesn't reclaim memory immediately**.
- If the **ArrayList is resized**, the **old array is garbage collected**.

Use ArrayDeque Instead of ArrayList for Insertions/Deletions

Better for frequent add/remove operations:

```
Deque<Integer> deque = new ArrayDeque<>();

deque.addFirst(10);  // Faster than ArrayList’s shifting mechanism
```

Convert ArrayList to Array When Processing Large Data

For better performance in computations:

```
Integer[] arr = list.toArray(new Integer[0]);  // Avoids unnecessary dynamic resizing
```

Optimizing Arrays

Use primitive arrays for large data (avoid autoboxing).
Use Arrays.fill() to quickly initialize arrays.
If arrays are large, manually set them to null before exiting a method to help GC.

```
int[] arr = new int[1000000];

arr = null;  // Helps GC reclaim memory
```

Optimizing ArrayLists

Preallocate size (new ArrayList<>(size)) to avoid frequent resizing.
Use trimToSize() after removing many elements.
Prefer LinkedList if frequent insertions/deletions are needed.
Convert to Array for large processing.

```
Integer[] arr = list.toArray(new Integer[0]);
```

Interview Questions on Array vs. ArrayList

Q1: When should you use Array instead of ArrayList?

A: Use arrays when **size is fixed, performance is critical, and you need to store primitives**.

Q2: How does ArrayList resize itself?

A: When full, it creates a **new array with 1.5x capacity** and **copies old elements**.

Q3: What is the default capacity of ArrayList?

A: 10. When exceeded, capacity grows by **1.5x**.

Q4: How is memory allocated for an ArrayList of Integer?

A: References are stored contiguously in an array, but **actual Integer objects are separate in heap**.

Q5: Why is ArrayList<Integer> less memory-efficient than int[]?

A: ArrayList<Integer> **stores objects**, leading to **autoboxing overhead**.

Q6: How to optimize memory usage in ArrayList?

A: Use **pre-allocation (new ArrayList<>(size))**, convert to **arrays (toArray())**, or **use primitive arrays (int[])**.

Why is ArrayList<Integer> less memory efficient than int[]?

A: ArrayList<Integer> stores **Integer objects**, leading to **autoboxing overhead**. int[] directly stores values.

Feature	**Array (int[])**	**ArrayList (ArrayList<Integer>)**
Memory Efficiency	**High (No object overhead)**	**Low (Autoboxing overhead)**
Garbage Collection	**Fast (single array object)**	**Slower (array + wrapper objects)**
Resizing	**Not possible**	**Grows dynamically (1.5x)**
Performance	**Faster (direct memory)**	**Slower (resizing overhead)**

Feature	**Array (int[])**	**ArrayList (ArrayList<Integer>)**
Memory Efficiency	High (No object overhead)	Low (Autoboxing overhead)
Size	Fixed	Dynamic
Performance	Faster (direct memory)	Slower (resizing overhead)
When to Use?	When size is **known** and **performance matters**	When **dynamic growth** is required
Resizing	Not possible	Grows dynamically (1.5x)
Insert/Delete Performance	Shifting required	Automatic shifting

Arrays Class in Java

1. Introduction to the Arrays Class

- Part of java.util package.
- Provides utility methods for **sorting, searching, comparing, copying, filling, parallel operations**, etc.
- Supports both **primitive and object** arrays.
- Methods are **static**, so they can be accessed directly without creating an instance of the Arrays class.

Important Methods of the Arrays Class

(1) Sorting an Array – Arrays.sort()

This method sorts the elements of an array in **ascending order** using **Dual-Pivot Quicksort** for primitive types and **Timsort** for objects.

```
int[] numbers = {5, 1, 4, 2, 8};

Arrays.sort(numbers);

System.out.println(Arrays.toString(numbers)); // Output: [1, 2, 4, 5, 8]
```

Sorting a Subarray

```
int[] arr = {5, 3, 8, 6, 2, 7};
```

```
Arrays.sort(arr, 1, 4);  // Sorts only from index 1 to 3 (4 is excluded)

System.out.println(Arrays.toString(arr)); // Output: [5, 2, 3, 8, 6, 7]
```

Sorting in Descending Order

```
Integer[] arr = {10, 2, 8, 6};

Arrays.sort(arr, Collections.reverseOrder());

System.out.println(Arrays.toString(arr)); // Output: [10, 8, 6, 2]
```

(2) Binary Search – Arrays.binarySearch()

This method searches for an element in a **sorted array** using **Binary Search** (O(log n) complexity).

Usage:

```
int[] arr = {2, 4, 6, 8, 10};

int index = Arrays.binarySearch(arr, 6);

System.out.println("Element found at index: " + index); // Output: 2
```

Binary Search on an Unsorted Array (Incorrect Result)

```
int[] arr = {10, 5, 8, 2, 7};

int index = Arrays.binarySearch(arr, 8); // Wrong result since array is unsorted
```

Solution: Sort the array first before using binarySearch().

(3) Filling an Array – Arrays.fill()

Used to **initialize or replace all elements** of an array with a specific value.

Usage:

```
int[] arr = new int[5];

Arrays.fill(arr, 10);

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

Filling a Part of the Array

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

Arrays.fill(arr, 1, 4, 9); // Fill indexes 1 to 3 with 9
```

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

(4) Copying an Array – Arrays.copyOf() and Arrays.copyOfRange()

Used to **create a new array** by copying an existing array.

Copying the Entire Array

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

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

System.out.println(Arrays.toString(copy)); // Output: [1, 2, 3, 4, 5]
```

Copying a Part of the Array

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

int[] newArr = Arrays.copyOfRange(arr, 1, 4); // Copies index 1 to 3

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

(5) Comparing Arrays – Arrays.equals()

Checks if two arrays are **equal** (element-wise comparison).

Usage:

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

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

int[] arr3 = {1, 2, 4};

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

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

(6) Converting an Array to a String – Arrays.toString()

Converts an array to a readable string format.

```
int[] arr = {5, 10, 15};

System.out.println(Arrays.toString(arr)); // Output: [5, 10, 15]
```

(7) Converting a 2D Array to a String – Arrays.deepToString()

Used for **multidimensional arrays**.

```
int[][] matrix = {
   {1, 2, 3},
   {4, 5, 6}
};
System.out.println(Arrays.deepToString(matrix));
// Output: [[1, 2, 3], [4, 5, 6]]
```

(8) Parallel Sorting – Arrays.parallelSort()

This method sorts an array using **multithreading**, which is **faster for large arrays**.

Usage:

```
int[] arr = {9, 5, 1, 7, 3};
Arrays.parallelSort(arr);
System.out.println(Arrays.toString(arr)); // Output: [1, 3, 5, 7, 9]
```

Interview Questions on the Arrays Class

Theoretical Questions

Q1. What is the difference between Arrays.sort() and Arrays.parallelSort()?
A1. Arrays.sort() uses a **single-threaded** sorting algorithm, whereas Arrays.parallelSort() uses **multithreading** for faster execution on large arrays.

Q2. Can Arrays.equals() be used for comparing multidimensional arrays?
A2. No, for **multidimensional arrays**, use Arrays.deepEquals().

Q3. What is the difference between Arrays.copyOf() and System.arraycopy()?
A3.

- Arrays.copyOf() creates a **new array** and copies elements.
- System.arraycopy() is **faster** as it copies elements **in place** without creating a new array.

Summary Table of Arrays Methods

Method	Description
sort(arr)	Sorts an array in ascending order
binarySearch(arr, key)	Searches for a key using binary search
fill(arr, val)	Fills array with a value
copyOf(arr, size)	Copies an array to a new size
equals(arr1, arr2)	Compares two arrays
toString(arr)	Converts array to string
deepToString(arr2D)	Converts 2D array to string
parallelSort(arr)	Sorts using multiple threads

Advanced Concepts of Arrays in Java

1. Multi-Dimensional Arrays in Java

Q1: What are Multi-Dimensional Arrays?

Multi-dimensional arrays are arrays of arrays, where each element is another array.

Q2: How to Declare and Initialize a Multi-Dimensional Array?

```
int[][] arr = new int[3][4];  // 3 rows, 4 columns
```

OR

```
int[][] arr = {
   {1, 2, 3, 4},
   {5, 6, 7, 8},
   {9, 10, 11, 12}
};
```

Q3: How are Multi-Dimensional Arrays Stored in Memory?

- Java uses **row-major order**, meaning elements of each row are stored in contiguous memory.
- It is an **array of arrays**, meaning each row is a separate array stored at different memory locations.

Q4: How to Access Multi-Dimensional Arrays?

```
System.out.println(arr[1][2]);  // Output: 7
```

Q5: Traversing a Multi-Dimensional Array

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

2. Jagged Arrays in Java

Q6: What is a Jagged Array?

A **jagged array** is an array where rows have different column sizes.

Q7: How to Declare a Jagged Array?

```
int[][] jaggedArr = new int[3][];  // Only row size is fixed

jaggedArr[0] = new int[2];  // Row 0 has 2 columns

jaggedArr[1] = new int[4];  // Row 1 has 4 columns

jaggedArr[2] = new int[3];  // Row 2 has 3 columns
```

Q8: Traversing a Jagged Array

```
for (int i = 0; i < jaggedArr.length; i++) {
    for (int j = 0; j < jaggedArr[i].length; j++) {
        System.out.print(jaggedArr[i][j] + " ");
    }
    System.out.println();
}
```

3. Array Performance & Limitations

Q9: What are the Performance Characteristics of Arrays?

Operation	Time Complexity
Access (arr[i])	O(1)
Search (Linear)	O(n)
Search (Binary)	O(log n)
Insertion (End)	O(1)
Insertion (Middle)	O(n)
Deletion (End)	O(1)
Deletion (Middle)	O(n)

Q10: Why are Arrays Fixed in Size?

- Java arrays use **contiguous memory allocation**, making resizing impossible.
- To overcome this, Java provides **ArrayList**, which dynamically resizes.

4. Resizing Arrays

Q11: How does Java Handle Resizable Arrays?

Java provides **ArrayList**, which dynamically resizes using an internal array.

Q12: How does ArrayList Resize Internally?

- Starts with a default capacity (10).
- When full, it creates a new array with **1.5x or 2x capacity** and copies old elements.

```
ArrayList<Integer> list = new ArrayList<>();

list.add(10);

list.add(20);

System.out.println(list.size());  // Output: 2
```

5. Sparse Arrays

Q13: What is a Sparse Array?

- An array where **most elements are zeros**.
- Instead of storing all elements, **only non-zero values and their positions are stored**.

Q14: How to Represent a Sparse Array?

Example:

```
int[][] sparseArr = {
  {0, 0, 5, 0},
  {0, 10, 0, 0},
  {0, 0, 0, 15}
};
```

Converted to **compressed form**:

Row	Col	Value
0	2	5
1	1	10
2	3	15

6. Implementing Custom Dynamic Array (Like ArrayList)

Q15: How to Create a Custom Dynamic Array?

```
class DynamicArray {
  private int[] arr;
  private int size;
  private int capacity;

  public DynamicArray(int capacity) {
    this.capacity = capacity;
    arr = new int[capacity];
    size = 0;
    }

    public void add(int value) {
        if (size == capacity) {
            resize();
        }
        arr[size++] = value;
    }

    private void resize() {
        capacity *= 2;
        int[] newArr = new int[capacity];
        System.arraycopy(arr, 0, newArr, 0, size);
        arr = newArr;
    }

    public int get(int index) {
        if (index < 0 || index >= size) throw new IndexOutOfBoundsException();
        return arr[index];
    }
}
```

Usage:

```
DynamicArray da = new DynamicArray(2);
da.add(10);
da.add(20);
da.add(30);  // Resizes array
System.out.println(da.get(2));  // Output: 30
```

7. Circular Arrays

Q16: What is a Circular Array?

A circular array **wraps around** when the end is reached.

Q17: How to Implement a Circular Array?

```
class CircularArray {
    private int[] arr;
    private int front, rear, size, capacity;

    public CircularArray(int capacity) {
        this.capacity = capacity;
        arr = new int[capacity];
        front = rear = -1;
        size = 0;
    }

    public void enqueue(int value) {
        if (size == capacity) throw new RuntimeException("Queue Full");
        rear = (rear + 1) % capacity;
        arr[rear] = value;
        if (front == -1) front = 0;
        size++;
    }

    public int dequeue() {
        if (size == 0) throw new RuntimeException("Queue Empty");
        int value = arr[front];
        front = (front + 1) % capacity;
        size--;
        return value;
    }
}
```

Usage:

```
CircularArray queue = new CircularArray(3);
queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);
System.out.println(queue.dequeue()); // Output: 10
```

Advanced Concepts

- **Jagged Arrays** – Arrays with different column sizes.
- **Sparse Arrays** – Arrays that store mostly default values.
- **Array Cloning** – arr.clone() creates a **shallow copy**.
- **Parallel Sorting** – Arrays.parallelSort(arr) for faster sorting using multithreading.

ArrayList in Java – Complete Guide

1. Introduction to ArrayList

Q1: What is an ArrayList?

ArrayList is a **resizable array implementation** of the List interface in Java, part of the **Java Collections Framework**. Unlike an array, it **can grow and shrink dynamically**.

Q2: Why Use ArrayList Instead of an Array?

Feature	Array	ArrayList
Fixed Size?	Yes	No (Dynamic)
Type of Elements?	Primitives & Objects	Only Objects (Wrapper classes for primitives)
Performance	Fast for direct index access	Slightly slower (Dynamic resizing)

Built-in Methods?	Few	Many (Sorting, Searching, etc.)

2. How to Create an ArrayList?

Q3: How to Declare and Initialize an ArrayList?

```
import java.util.ArrayList;  // Import required


public class Main {

   public static void main(String[] args) {

      ArrayList<Integer> list = new ArrayList<>();  // Default size 10

      ArrayList<String> names = new ArrayList<>(20);  // Custom initial size 20

   }

}
```

Q4: Can We Use Primitive Data Types in ArrayList?

No, only objects are allowed. Use **Wrapper Classes** instead.

```
ArrayList<int> list = new ArrayList<>();  //  Error!

ArrayList<Integer> list = new ArrayList<>();  // Works!
```

3. Adding Elements to an ArrayList

Q5: How to Add Elements?

Method	Description	Example
add(E e)	Adds element at the end	list.add(10);
add(int index, E e)	Adds element at a specific index	list.add(1, 20);

```
ArrayList<Integer> list = new ArrayList<>();

list.add(10);

list.add(20);

list.add(1, 15);  // Inserts 15 at index 1

System.out.println(list);  // Output: [10, 15, 20]
```

4. Accessing Elements

Q6: How to Get an Element?

Method	**Description**	**Example**
get(int index)	Retrieves an element at the index	list.get(1);

System.out.println(list.get(1)); // Output: 15

5. Updating and Removing Elements

Q7: How to Update an Element?

Method	**Description**	**Example**
set(int index, E e)	Updates value at index	list.set(1, 25);

list.set(1, 25);

System.out.println(list); // Output: [10, 25, 20]

Q8: How to Remove an Element?

Method	**Description**	**Example**
remove(int index)	Removes by index	list.remove(1);
remove(Object obj)	Removes by value	list.remove(Integer.valueOf(25));

list.remove(1);

System.out.println(list); // Output: [10, 20]

6. Checking ArrayList Properties

Q9: How to Check the Size?

Method	**Description**	**Example**
size()	Returns the number of elements	list.size();

System.out.println(list.size()); // Output: 2

Q10: How to Check If ArrayList is Empty?

Method	**Description**	**Example**
isEmpty()	Returns true if empty	list.isEmpty();

System.out.println(list.isEmpty()); // Output: false

Q11: How to Check If an Element Exists?

Method	Description	Example
contains(Object obj)	Returns true if present	list.contains(20);

```
System.out.println(list.contains(20));  // Output: true
```

7. Iterating Over an ArrayList

Q12: What are Different Ways to Iterate Over an ArrayList?

(i) Using For Loop

```
for (int i = 0; i < list.size(); i++) {
   System.out.print(list.get(i) + " ");
}
// Output: 10 20
```

(ii) Using Enhanced For Loop

```
for (Integer num : list) {
   System.out.print(num + " ");
}
// Output: 10 20
```

(iii) Using Iterator

```
import java.util.Iterator;


Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
}
// Output: 10 20
```

(iv) Using Streams (Java 8)

```
list.forEach(System.out::println);
// Output: 10 20
```

8. Sorting and Searching

Q13: How to Sort an ArrayList?

Method	Description	Example
Collections.sort(List<E>)	Sorts in ascending order	Collections.sort(list);
Collections.reverseOrder()	Sorts in descending order	Collections.sort(list, Collections.reverseOrder());

```
import java.util.Collections;


Collections.sort(list);

System.out.println(list);  // Output: [10, 20]
```

Q14: How to Search for an Element?

Method	Description	Example
indexOf(E e)	Returns index, -1 if not found	list.indexOf(20);

```
System.out.println(list.indexOf(20));  // Output: 1
```

---

9. Converting ArrayList to Other Data Structures

Q15: How to Convert ArrayList to Array?

```
Integer[] arr = list.toArray(new Integer[0]);
```

Q16: How to Convert Array to ArrayList?

```
import java.util.Arrays;


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

ArrayList<Integer> list = new ArrayList<>(Arrays.asList(arr));
```

10. Advanced Features

Q17: How to Remove All Elements?

```
list.clear();

System.out.println(list);  // Output: []
```

Q18: How to Ensure Capacity?

```
list.ensureCapacity(100);
```

11. Performance Considerations

Q19: How Does ArrayList Resize?

1. Default capacity = **10**
2. When full, **new capacity = (old capacity * 1.5) + 1**
3. Resizing is costly, so use ensureCapacity() if the size is known.

Q20: When to Use LinkedList Instead of ArrayList?

Feature	ArrayList	LinkedList
Fast Random Access	Yes (O(1))	No (O(n))
Frequent Insert/Delete	Slow (O(n))	Fast (O(1) at ends)

12. Summary

Feature	Method
Add Element	add(E e), add(int index, E e)
Get Element	get(int index)
Update Element	set(int index, E e)
Remove Element	remove(int index), remove(Object obj)
Size Check	size()
Check If Empty	isEmpty()
Search Element	contains(Object obj), indexOf(E e)
Sort	Collections.sort(List<E>)
Clear All Elements	clear()

How ArrayList Works Internally

- **Underlying Data Structure:** ArrayList is backed by an **array** of type Object[] (not primitives).
- **Initial Capacity:** Default size is **10** when an ArrayList is created using new ArrayList<>().
- **Dynamic Resizing:** When the array gets **full**, a **new array** of **1.5x (or 1.5 times the current size)** is created, and all elements are copied into the new array.