Arrays in Java

In Java, an array is a data structure that allows you to store a fixed-size sequential collection of elements of the same type.

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. Arrays in Java are indexed, starting from 0 to the length of the array minus one. They can be easily created, initialized, and manipulated using loops and methods.

- Arrays are groups of similar typed variables referred to by a common name.
- Arrays in Java are dynamically allocated.
- Arrays may be stored in contiguous memory locations, allowing random access.
- Arrays can contain both primitive types (int, char, etc.) and object references.
- Java arrays are objects and their length can be accessed using the `length` property.
- Arrays can be used as static fields, local variables, or method parameters.

Advantages:

- Random Access
- Cache Friendly

Disadvantages:

- Insertion and Deletion are slow
- Search is also slow for unsorted

Declaring and Initializing Arrays:

- Arrays can be declared with `type variable_name[]` or `type[] variable_name`.

  Example:

```
int intArray[];
int[] intArray;
```

- Use `new` to allocate memory for an array.

  Example:

```
intArray = new int[20];
int[] intArray = new int[20];
```

Array Literals:

- Arrays can be initialized directly using array literals.

Example:

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

Accessing Array Elements:

- Array elements are accessed via their index, starting from 0.

  Example with for loop:

```
for (int i = 0; i < arr.length; i++)
    System.out.println(arr[i]);
```

Arrays of Objects:

- Arrays can store references to objects.

  Example:

```
Student[] arr = new Student[5];
arr[0] = new Student(1, "Aman");
```

Multidimensional Arrays:

- Java supports multidimensional arrays (arrays of arrays).

  Example of 2D array:

```
int[][] arr = new int[3][3];
```

Passing Arrays to Methods:

- Arrays can be passed to methods.

  Example:

```
public static void sum(int[] arr) {
    int sum = 0;
    for (int i : arr) sum += i;
    System.out.println("Sum: " + sum);
}
```

Returning Arrays from Methods:

- Methods can return arrays.

  Example:

```
public static int[] getArray() {
    return new int[] {1, 2, 3};
}
```

Cloning Arrays:

- Cloning a single-dimensional array creates a deep copy.
- Cloning a multidimensional array creates a shallow copy.

  Example of single-dimensional array cloning:

  ```
  int[] cloneArray = intArray.clone();
  ```

ArrayIndexOutOfBoundsException:

- Accessing an array with an illegal index (negative or beyond array size) throws this exception.

  Example:

  ```
  int[] arr = new int[4];
  System.out.println(arr[5]); // Throws ArrayIndexOutOfBoundsException
  ```

Class Objects for Arrays:

- Arrays have associated class objects.

  Example:

  ```
  System.out.println(intArray.getClass()); // Output: class [I
  ```

Arrays class in Java

The `Arrays` class in the `java.util` package is an integral part of the Java Collection Framework. This utility class provides a variety of static methods for dynamically creating and manipulating arrays, which simplifies many common array operations. Although developers can declare, initialize, and perform operations on arrays using loops, the Arrays class offers optimized, built-in methods that make code more concise and efficient.

The class inherits from `java.lang.Object` and includes methods to fill arrays with specific values, sort arrays, search for elements using binary search, and more. These methods can be called directly using the class name, enhancing code readability and maintainability.

Class Hierarchy:

java.lang.Object
↳ java.util.Arrays

Method	Action
`asList()`	Returns a fixed-size list backed by the specified array.
`binarySearch(array, key)`	Searches for the specified element using the Binary Search Algorithm.
`binarySearch(array, fromIndex, toIndex, key, Comparator)`	Searches a range of the specified array for the specified object using the Binary Search Algorithm.
`compare(array1, array2)`	Compares two arrays lexicographically.
`copyOf(originalArray, newLength)`	Copies the specified array, truncating or padding with the default value if necessary.
`copyOfRange(originalArray, fromIndex, endIndex)`	Copies a specified range of the specified array into a new array.
`deepEquals(Object[] a1, Object[] a2)`	Returns `true` if the two specified arrays are deeply equal.
`deepHashCode(Object[] a)`	Returns a hash code based on the "deep contents" of the specified array.
`deepToString(Object[] a)`	Returns a string representation of the "deep contents" of the specified array.
`equals(array1, array2)`	Checks if both arrays are equal.
`fill(originalArray, fillValue)`	Assigns the specified value to each element of the array.
`hashCode(originalArray)`	Returns an integer hash code for the array.
`mismatch(array1, array2)`	Finds and returns the index of the first unmatched element between two arrays.
`parallelPrefix(originalArray, fromIndex, endIndex, functionalOperator)`	Performs a parallel prefix operation on a range of the array using the specified functional operator.
`parallelPrefix(originalArray, operator)`	Performs a parallel prefix operation on the entire array using the specified functional operator.
`parallelSetAll(originalArray, functionalGenerator)`	Sets all elements of the array in parallel using the provided generator function.
`parallelSort(originalArray)`	Sorts the specified array using parallel sort.
`setAll(originalArray, functionalGenerator)`	Sets all elements of the array using the provided generator function.
`sort(originalArray)`	Sorts the entire array in ascending order.
`sort(originalArray, fromIndex, endIndex)`	Sorts a specified range of the array in ascending order.

`sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)`	Sorts a specified range of the array using the specified comparator.
`sort(T[] a, Comparator<? super T> c)`	Sorts the array using the specified comparator.
`spliterator(originalArray)`	Returns a `Spliterator` covering all elements of the array.
`spliterator(originalArray, fromIndex, endIndex)`	Returns a `Spliterator` covering the specified range of the array.
`stream(originalArray)`	Returns a sequential stream with the array as its source.
`toString(originalArray)`	Returns a string representation of the array's contents.

Multidimensional Arrays in Java

Multidimensional arrays in Java can be understood as arrays of arrays. They allow the storage of data in a tabular form, which is particularly useful for representing structures like matrices or grids.

Syntax
data_type[1st dimension][2nd dimension]...[Nth dimension] array_name = new data_type[size1][size2]...[sizeN];

Examples:

Two-Dimensional Array

Three-Dimensional Array

Size Calculation

The total number of elements in a multidimensional array is the product of the sizes of all dimensions. For example:

- int[][] x = new int[10][20]; can store 10×20=20010 \times 20 = 20010×20=200 elements.
- int[][][] x = new int[5][10][20]; can store 5×10×20=10005 \times 10 \times 20 = 10005×10×20=1000 elements.

Applications of Multidimensional Arrays

- **Tabular Data**: Storing data in a tabular form, such as student roll numbers and marks.
- **Dynamic Programming**: Representing the states of a problem.
- **Algorithms**: Implementing matrix multiplication, adjacency matrix representation in graphs, and grid search problems.

Two-Dimensional Arrays (2D-Arrays)

Declaration:

```
int[][] arr = new int[10][20];
```

Initialization:

```
arr[0][0] = 1;
```

Example:

```
public class TwoDArray {
   public static void main(String[] args) {
      int[][] arr = new int[10][20];
      arr[0][0] = 1;
      System.out.println("arr[0][0] = " + arr[0][0]);
   }
}
```

Direct Method of Declaration:

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

Example:

```
public class DirectDeclaration {
   public static void main(String[] args) {
      int[][] arr = { { 1, 2 }, { 3, 4 } };
      for (int i = 0; i < 2; i++)
         for (int j = 0; j < 2; j++)
            System.out.println("arr[" + i + "][" + j + "] = " + arr[i][j]);
   }
}
```

Three-Dimensional Arrays (3D-Arrays)

Declaration:

```
int[][][] arr = new int[10][20][30];
```

Initialization:

```
arr[0][0][0] = 1;
```

Example:

```
public class ThreeDArray {
   public static void main(String[] args) {
      int[][][] arr = new int[10][20][30];
      arr[0][0][0] = 1;
      System.out.println("arr[0][0][0] = " + arr[0][0][0]);
   }
}
```

Direct Method of Declaration:

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

Example:

```
public class DirectDeclaration {
   public static void main(String[] args) {
      int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };
      for (int i = 0; i < 2; i++)
         for (int j = 0; j < 2; j++)
            for (int k = 0; k < 2; k++)
               System.out.println("arr[" + i + "][" + j + "][" + k + "] = " + arr[i][j][k]);
   }
}
```

Jagged Arrays in Java

A jagged array is an array of arrays where the member arrays can have different sizes. This allows the creation of a 2-D array with a variable number of columns in each row.

Advantages:

- **Dynamic Allocation**: Memory for each sub-array can be allocated at runtime.
- **Space Utilization**: Saves memory when sub-arrays have different sizes.
- **Flexibility**: Useful for storing arrays of different lengths or when the number of elements is not known in advance.
- **Improved Performance**: Can be faster for certain operations due to a more compact memory layout.

Disadvantages:

- Increased complexity in code.
- Potentially less readable codebase.

Syntax:

```
// Declaration
data_type array_name[][] = new data_type[n][];  // n: number of rows

// Initialization
array_name[0] = new data_type[n1]; // n1: number of columns in row-1
array_name[1] = new data_type[n2]; // n2: number of columns in row-2
//...
array_name[k] = new data_type[nk]; // nk: number of columns in row-n

// Alternative Initialization
int arr_name[][] = new int[][] {
   new int[] {10, 20, 30, 40},
   new int[] {50, 60, 70, 80, 90, 100},
   new int[] {110, 120}
};

int[][] arr_name = {
   new int[] {10, 20, 30, 40},
   new int[] {50, 60, 70, 80, 90, 100},
   new int[] {110, 120}
};

int[][] arr_name = {
   {10, 20, 30, 40},
   {50, 60, 70, 80, 90, 100},
   {110, 120}
};
```

Final Arrays in Java

Final Variables: A variable declared as final can only be assigned once. After the initial assignment, its value cannot be changed.

Final Object References: When an object reference is declared as final, it means that the reference cannot be changed to point to another object. However, the state of the object it refers to can be changed.

Arrays as Objects: Arrays in Java are objects, and the rules for final object references apply to arrays as well. This means:

- You cannot reassign the reference of a final array to a new array.
- You can modify the elements within the final array.

Declaring a final array:

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

Reflection Array Class in Java

The Array class in the java.lang.reflect package is a part of Java Reflection. It provides static methods to create and access Java arrays dynamically. It is a final class, meaning it cannot be instantiated or extended. The methods of this class must be accessed using the class name itself.

- java.lang.reflect.Array: Provides static methods to create and access arrays dynamically, maintaining type safety.
- java.util.Arrays: Contains various methods for manipulating arrays (such as sorting and searching).