**Collections**

Index:

1. Arrays (1 -27)

* Memory allocations
* Different methods for Arrays Operations - <https://ioflood.com/blog/java-array-methods/>
* Progarms - ? find the second highest value [1,2,3,4,5,6,7,8,9]?

1. Why do we need collections framework?(28-

* Overview about Collections Framework

1. List(Arraylist,Vector), Set, Queue (internal working)
2. Map internal working
3. Java8 HashMap improvements
4. Why Generic and how to use?
5. Collections, CollectionsUtils,
6. ConcurrentModificationsException – Iterator vs ListIterator
7. Comparator vs Comparable

**1. Arrays:**

Reference Link : <https://medium.com/edureka/java-array-tutorial-50299ef85e5>

**Definition:**

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java.

Arrays in Java are a fundamental data structure used to store collections of elements, typically of the same type. They are objects in Java, which means they are created on the heap, and can hold primitives (like int , char ) or objects (like String , Integer ).

Java is a general-purpose, concurrent, object-oriented, class-based, and the runtime environment(JRE). Through this blog on Java Array, I will explain you the concepts of Arrays in Java and how single & multi-dimensional arrays work.

**How exactly does indexing works in Arrays?**

First, let’s understand arrays, It is a collection of items stored at contiguous memory locations. The basic idea is to store multiple items of the same type together which can be accessed by index/key (a number).

The contiguous memory of declared size is allocated on heap/stack and then the address of the element is calculated mathematically during run-time as:-

|  |
| --- |
| element address = (base address) + (element index \* size of a single element) |

**Base address**: It is the address of the element at the index 0 or the location of the first element of the array in the memory. The compiler knows this address as the memory location of the array.

**Element index:** It is the sequential number (index/key) assigned to the element where the first element of the array is assigned 0. It can also be defined as the number of elements prior to that particular element in the array.

**Size of a single element:** Elements in the array need to be of the same data type or object. The size of the single element is the number of bytes required in memory to store a single element of that kind.

For example:

* **Int** type requires 4-bytes (32-bit)
* **char** type requires a 1-byte (8-bit)
* **long** type requires 8-byte (64-bit) etc.

**Example of above implementation:**

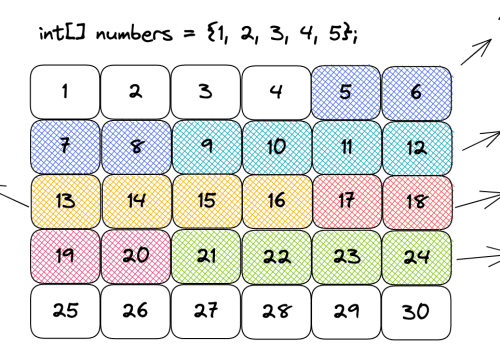
|  |
| --- |
| **int arr[6] = {3, 4, 7, 9, 7, 1}**  address of arr[0] (base address) = 0 x 61fe00  address of arr[3] (element address) = (base address) + (element index \* size of a single element)  0 x 61fe00 + ( 3 \* 4) = 0 x 61fe0c  Here, size of a single element is 4-bytes as it is int- type array.  **long long arr[6]={100, 12, 123, 899,124, 849}**  address of arr[0] (base address) = 0x61fdf0  address of arr[3] (element address) = (base address) + (element index \* size of a single element)  0x61fdf0 + ( 3 \* 8) = 0x61fe08  Here, size of a single element is 8-bytes as it is long – type array.  **Note**:  Here addresses are of Hexadecimal form.  x- means nth term values – 0th address of the value |
| **Note**: arr[9] – Then it will calculate the actual memory location of 9th index value using above formula and it will jump directly into that address and fetch the value. That is the reason Arrays are much preferable during searching process.  Reference Link: <https://javachallengers.com/array-data-structure/#:~:text=Simply%20put%2C%20an%20array%20is,a%20reference%20for%20this%20object>. |

**Array Data Structure with Java**

* The array data structure is probably the most used in every application. If not directly used it’s indirectly used with ArrayList, ArrayDeque, Vector, and other classes.
* Simply put, an array is a data structure that stores multiple variables into it so that there is no need to create many variables with different names.
* Arrays in Java are always an object, therefore, they will occupy space in the memory heap and it will create a reference for this object.

**Array Memory Allocation**

* The array is stored in the memory RAM and takes up space back to back. Most memory RAM stores 1 byte (8 bits) in each space. In Java, each primitive int number occupies 4 bytes in memory. Therefore, let’s how this would work in the following memory model if we create an array with 5 int elements:



* As you can see in the above diagram, when we are creating an array we must pass the type of the elements and size from it. That’s because when creating the array the memory allocation has to be back to back.
* That’s the reason why we can very quickly access an element from an array. Since we know where the array starts, in the case of the diagram above it’s in the memory address 5, the compiler makes a simple calculation.
* Considering the index we pass to the array, the size, and the type of the element, we can easily calculate where the element is present in memory. To get the second element from the array, for example, we would add 4 bytes to the first element index and we would know where the second element is allocated. For this reason, to access an element in an array the time complexity will be **always O(1).**
* To change an element in the array is also constant **time O(1).** That’s because we can quickly access the variable index and assign a new value to it.
* To create an array the time complexity **is O(n) because when creating it**, we will define the type and size of the array and the **required space in memory will be allocated**.

**Static Arrays**

* As the name suggests, a static array is an array that can’t be changed. We need to pass the type and size of the array and after that, we can’t change the type or size of the array.
* Let’s see in the following code how to create a static array with the int type and show all the elements:

|  |
| --- |
| int[] array = **{1, 2, 3, 4, 5}; // Create an array O(n)**  for (int i = 0; i < array.length; i++) {  System.out.print(array[i] + " ");  }  Output:  1 2 3 4 5 |

* Notice in the code above that we create an array of int values. Once it’s created we can’t change either the type or the size of the array, that’s what makes it static.
* Then we access each element of the array by index and show the values that will be 0 because those are the default values for primitive int in Java.

**Insert an Element in the Middle of the Array**

* To insert an element in the middle of the array it will be necessary to shift the elements. Therefore, the time complexity will be O(n).
* Let’s imagine we want to put 3 in the middle of 2 and 4 in the following array: { 1, 2, 4, 5 }. To do that, we will have to first create a new array with the size of 5.
* Then, it will be necessary to move elements 4 and 5 to one position ahead. Once this is done we can access index 2 and insert element 3.

**Dynamic Arrays**

* There are classes in Java that make use of a dynamic array such as ArrayList, Vector, and others. When we create an ArrayList, we have a static array under the hood that starts as empty but after adding the first element the size goes to 10. Then it doubles every time it’s needed as you can see in the following code of the JVM:

|  |
| --- |
| public class ArrayList<E> extends AbstractList<E>  implements List<E>, RandomAccess, Cloneable, java.io.Serializable  {  private static final int DEFAULT\_CAPACITY = 10;  transient Object[] elementData;  private int size;    private static final Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {};    public ArrayList() {  this.elementData = DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA;  }    // This is the method that will double the array whenever it's needed  private Object[] grow(int minCapacity) {  int oldCapacity = elementData.length;  if (oldCapacity > 0 || elementData != DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {  int newCapacity = ArraysSupport.newLength(oldCapacity,  minCapacity - oldCapacity, /\* minimum growth \*/  oldCapacity &gt;&gt; 1 /\* preferred growth \*/);  return elementData = Arrays.copyOf(elementData, newCapacity);  } else {  return elementData = new Object[Math.max(DEFAULT\_CAPACITY, minCapacity)];  }  }    // Omitted other methods...    } |

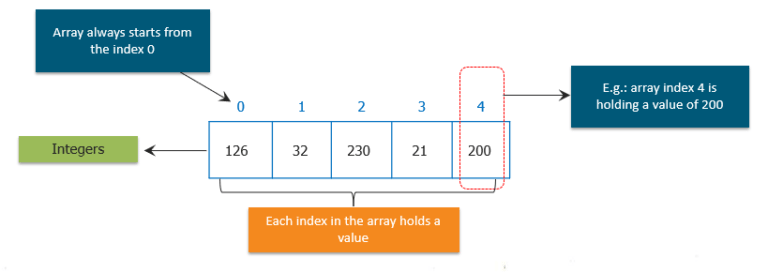
* The elementData variable is the one that will store the data behind the scenes for the ArrayList. Therefore, notice that the dynamic array actually manipulates a static array to behave as dynamic.
* When adding an element to an ArrayList, it will check if the size is greater than 10 and if that is true the time complexity will be O(n). That happens because since the array was created with the size of 10, it’s necessary to create a new array with the size of 20 copying all the elements into the new one. To do so, we need to traverse the whole array and copy element by element. Then we add the 11th element to the array.
* When the array behind the scenes is created with the size of 20 then whenever we add an element we will have the time complexity of O(1). Notice that the vast majority of the time when adding an element to a dynamic array will be pretty fast, it will be O(1). Only on the edge-case scenarios when the array size needs to be doubled the time complexity will be O(n). This is also called amortized complexity.

**Summary**

* An array is allocated in memory from back to back.
* Accessing an array by index has the time complexity of O(1), it’s pretty fast.
* Static array is the array that is created with a size and a type pre-defined.
* Dynamic array is an adaptation of the static array that automatically resizes it when necessary.
* A dynamic array will double its size when necessary.
* Adding an element to a dynamic array will be mostly O(1) because there will be space more often.
* When adding an element to a dynamic array exceeds the size of the static array under the hood, it will be necessary to create a new array, copy the elements from the existing array, then add the new element. Therefore, the time complexity will be O(n).
* Adding an element in the middle of the array will have the time complexity of O(n). That’s because it will be necessary to shift all the elements from the right side to one position on the right. Only then we will be able to insert the element by index.
* To remove the first element from the array, it will be necessary to shift all the elements from the right to the left. Therefore, the time complexity is O(n).
* To remove an element from the array in the last position takes O(1) complexity. That’s because we only need to remove the last value and we have direct access to it.

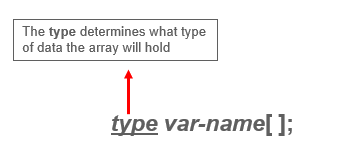
**What are Java Arrays?**

* Arrays in Java are homogeneous data structures implemented in Java as objects.
* Arrays store one or more values of a specific data type and provide indexed access to store the same.
* A specific element in an array is accessed by its index.
* Arrays offer a convenient means of grouping related information.

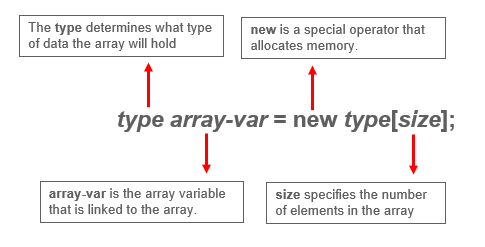
Obtaining an array is a two-step process.

* First, you must declare a variable of the desired array type
* Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable

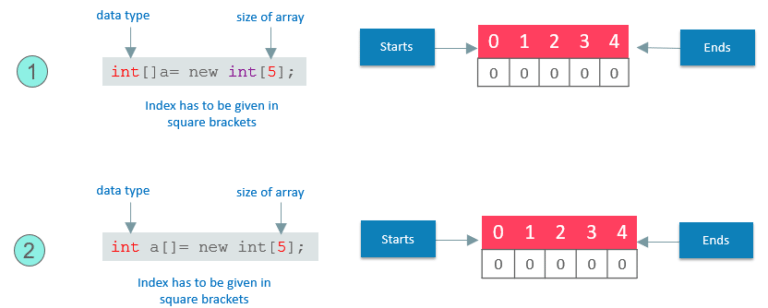
So, let us see how can we declare arrays in different ways.**General Form of Java Array Initialization**

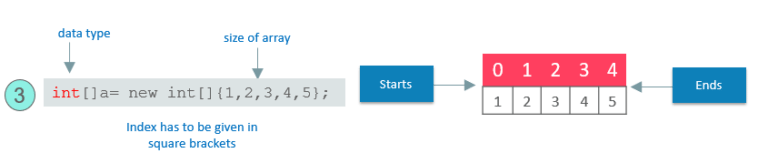


Example:- int month\_days[];

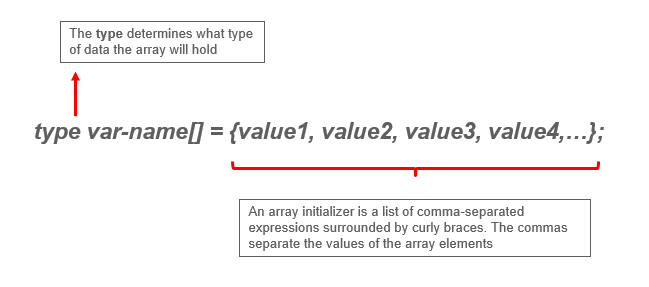
**General Form of Java Array Initialization**

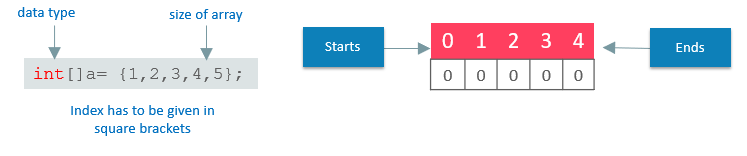
**Example:**





* **Static Arrays** will give arrays size with initialization values.
* Arrays can be initialized when they are declared.
* The array will automatically be created large enough to hold the number of elements you specify in the array initializer.
* There is no need to use new.
* Now, let us see how we can implement this.

General Form of Java Array Initialization:

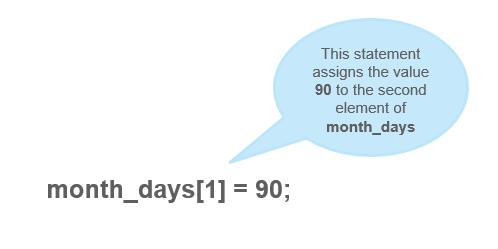


The following code creates an initialized array of integers:

|  |
| --- |
| class MyArray{    public static voide main(String args[]){    int month\_days[ ] = {31,28,31,30,31,30,31,30,31,30,31};    System.out.println("April has " + month+days[3] + "days.");    }    } |

* It will only be fair if I explain how you can access elements in a Java Array.
* Accessing a Specific Element in a Java Array
* In arrays, we can access the specific element by its index within square brackets.

**Example:**



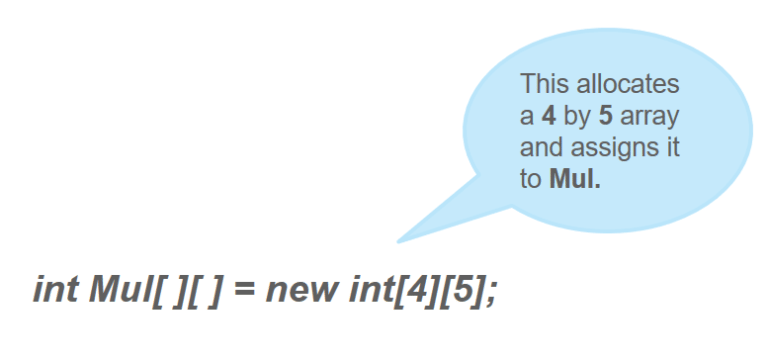
Putting together all the pieces:

|  |
| --- |
| public static void main(String args[]) {  int month\_days[];  month\_days = new int[12];  month\_days[0] = 31;  month\_days[1] = 28;  month\_days[2] = 31;  month\_days[3] = 30;  month\_days[4] = 31;  month\_days[5] = 30;  month\_days[6] = 31;  month\_days[8] = 30;  month\_days[9] = 31;  month\_days[10] = 30;  month\_days[11] = 31;  System.out.println("April has " + month\_days[3] + " days.");  }  } |

* So, this was all about the arrays and its declaration and how single dimension arrays can be used.
* What if I tell you, there can be an array inside an array. I know it sounds a bit complex, but don’t worry, I know how to make it easy for you.

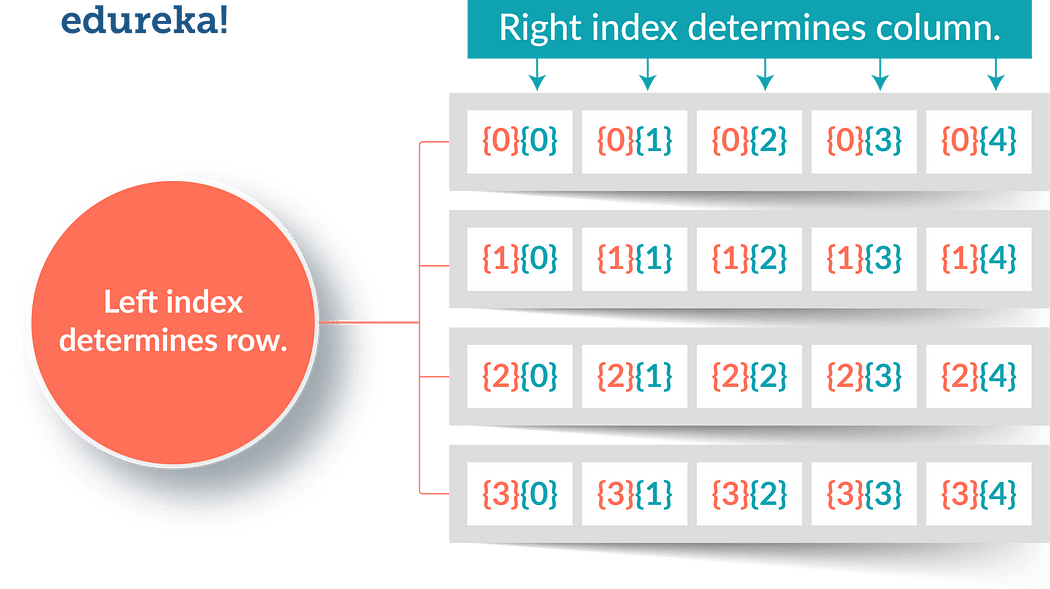
**Java Multidimensional Array:**

* Multidimensional arrays are arrays of arrays.
* Declaring Multidimensional Array
* To declare it, we have to specify each additional index using another set of square brackets.



Conceptually, the array declared above would be represented as shown in the figure:

Let us now Demonstrate Multidimensional Array.

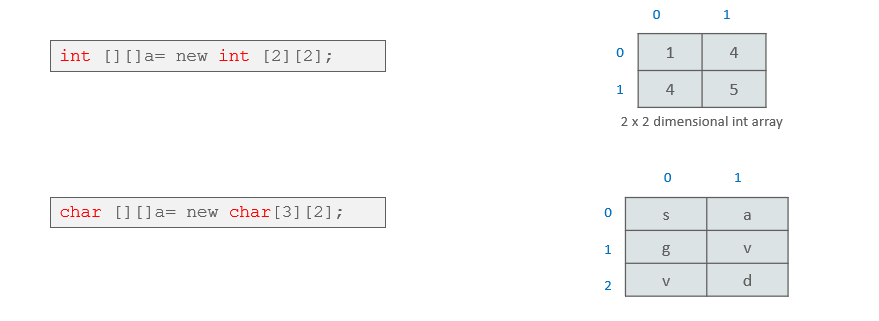
The following program, numbers each element in the array from left to right, top to bottom, and then displays these values:  


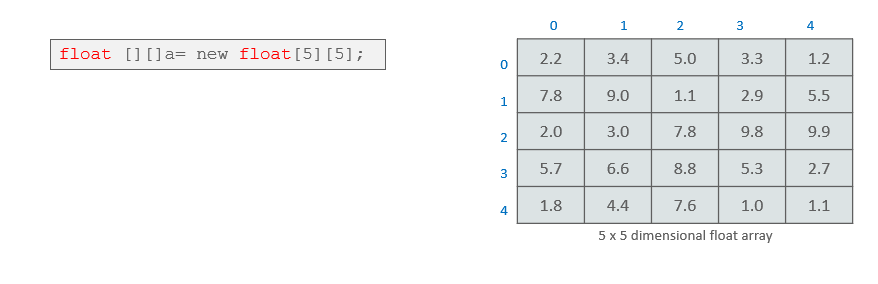
|  |
| --- |
| class Mul2D{  public static void main(String args[]) {  int mul2d[][]= new int[4][5];  int i, j, k = 0;  for(i=0; i&lt;4; i++)  for(j=0; j&lt;5; j++) {  Mul2D[i][j] = k;  k++;  }  for(i=0; i&lt;4; i++) {  for(j=0; j&lt;5; j++);  System.out.print(mul2d[i][j] + " ");  System.out.println();  }  }  } |

**This program generates the following output:**

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

These are other Multidimensional arrays representation of other data types.





* So, this was all about the Multidimensional Arrays. Now, Let us see, how to pass an array to a method as a parameter like the other data types.

**Passing Java Array to a Method**

* We can also pass arrays to methods just as we can pass primitive type values to methods.

**Example:**

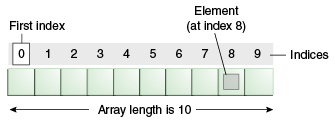
|  |
| --- |
| public class PMethods{  public static void display(int y[])  {  System.out.println(y[0]);  System.out.println(y[1]);  System.out.println(y[2]);    }  public static void main(String args[])  {  int x[] = { 1, 2, 3 };  display(x);  }  } |

This will be the output of the program:

|  |
| --- |
| 1  2  3 |

Java Arrays: (<https://medium.com/@nikhilsalvi011/java-arrays-d3bc832e8292>)

* Normally, an array is a collection of similar type of elements which has contiguous memory location.
* Java Array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
* Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.
* Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the **sizeof** operator.
* In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.
* Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



**Advantages:**

* Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.
* Random access: We can get any data located at an index position.

**Disadvantages:**

* Size Limit: We can store only the fixed size of elements in the array.
* It doesn’t grow its size at runtime.
* To solve this problem, collection framework is used in Java which grows automatically.

**Types of Array in java:**

There are two types of array.

* Single Dimensional Array
* Multidimensional Array

Single Dimensional Array in Java:

Syntax to Declare an Array in Java

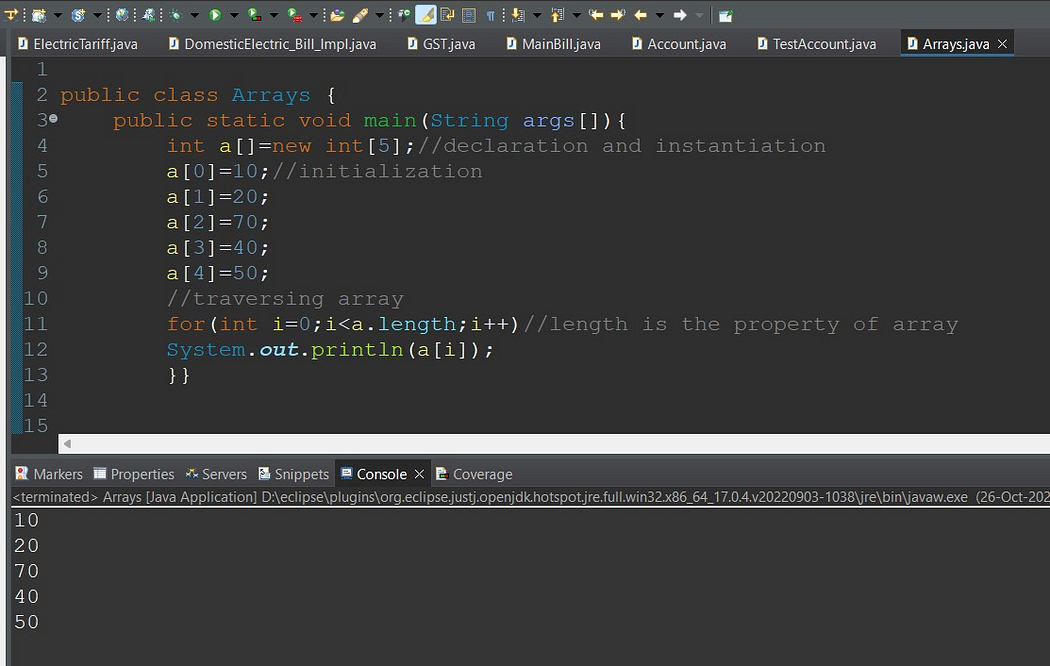
|  |
| --- |
| dataType[] arr; (or)  dataType []arr; (or)  dataType arr[]; |

Instantiation of an Array in Java

|  |
| --- |
| arrayRefVar=new datatype[size]; |

**Example of Java Array :**

Let’s see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

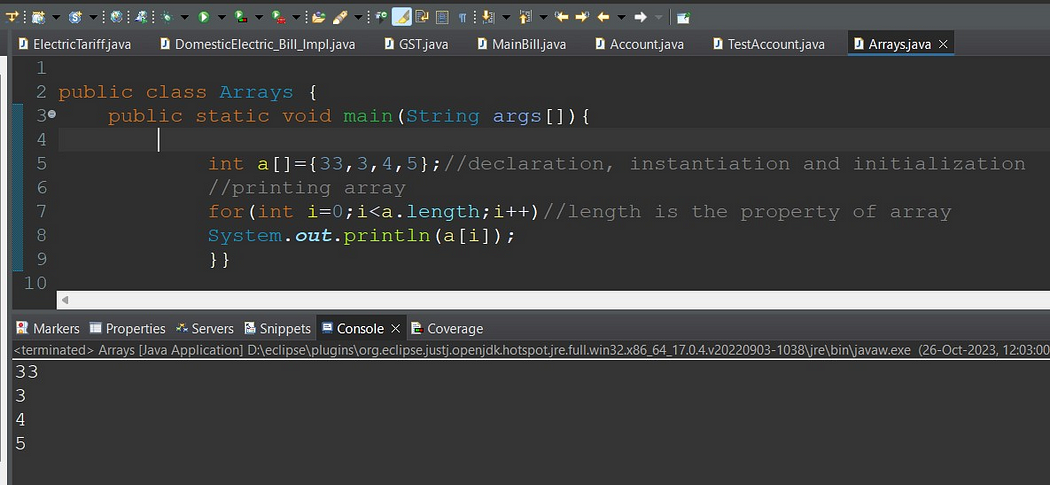


Declaration, Instantiation and Initialization of Java Array:

We can declare, instantiate and initialize the java array together by:

|  |
| --- |
| int a[]={33,3,4,5}; //declaration, instantiation and initialization |

Example to print this array:



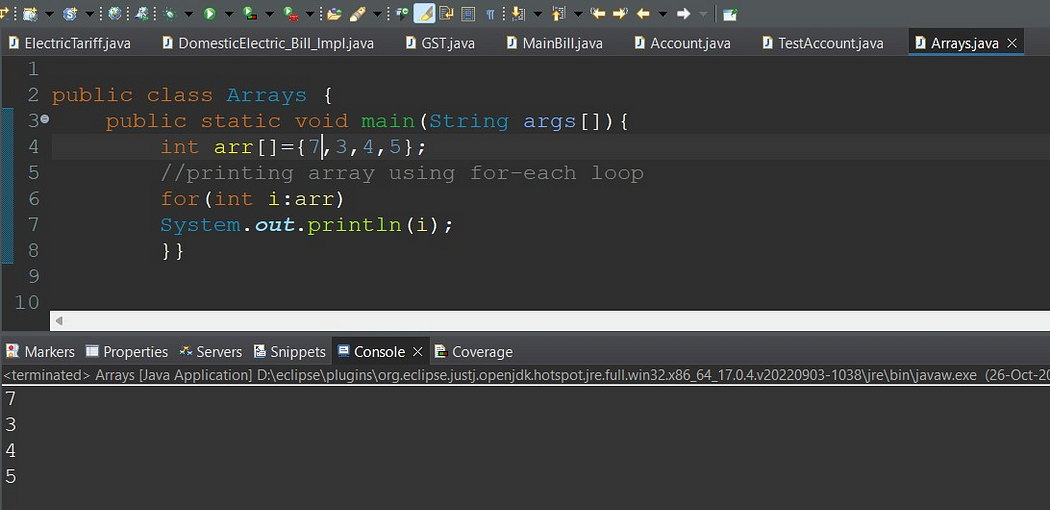
For-each Loop for Java Array:

We can also print the Java array using For-each loop. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

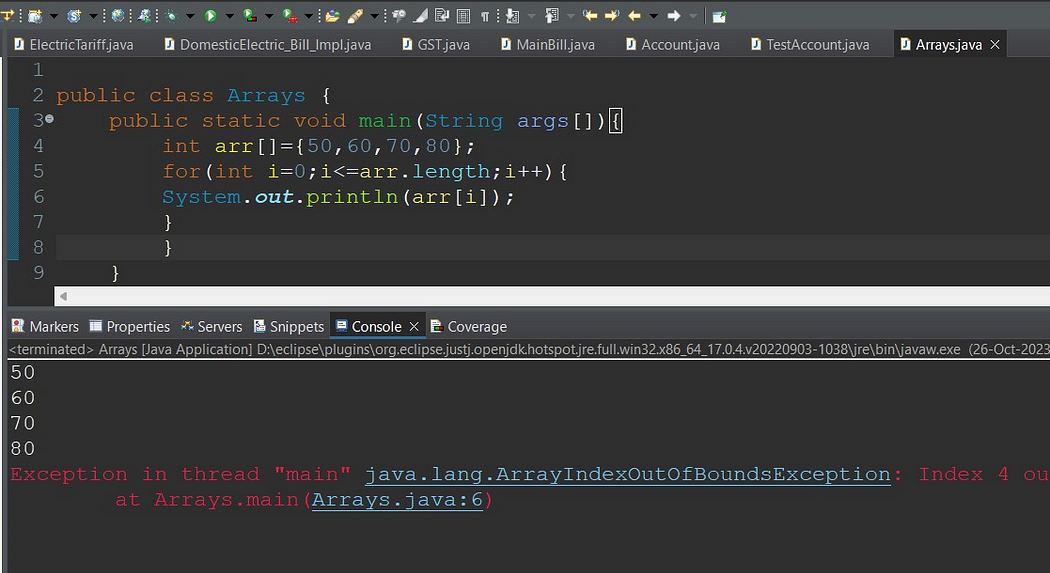
|  |
| --- |
| for(data\_type variable:array){  //body of the loop  } |

Example Java array using the for-each loop



**ArrayIndexOutOfBoundsException** :

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.



**Multidimensional Array in Java :**

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java :

|  |
| --- |
| dataType[][] arrayRefVar; (or)  dataType [][]arrayRefVar; (or)  dataType arrayRefVar[][]; (or)  dataType []arrayRefVar[]; |

Example to instantiate Multidimensional Array in Java:

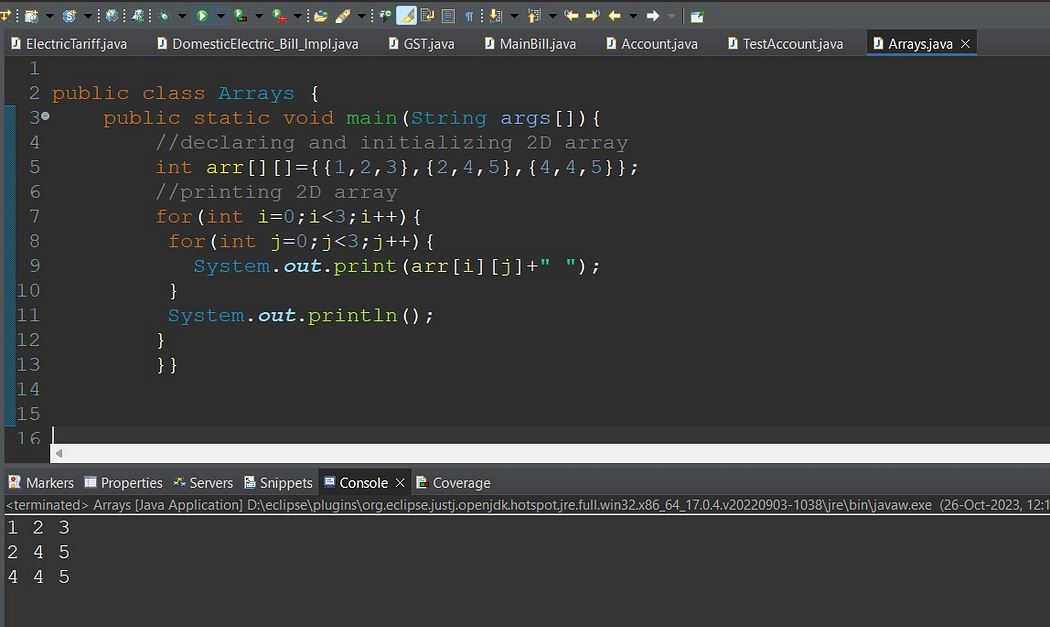
1. **int[][] arr=new int[3][3];//3 row and 3 column**

Example to initialize Multidimensional Array in Java

|  |
| --- |
| arr[0][0]=1;  arr[0][1]=2;  arr[0][2]=3;  arr[1][0]=4;  arr[1][1]=5;  arr[1][2]=6;  arr[2][0]=7;  arr[2][1]=8;  arr[2][2]=9; |

Example of Multidimensional Java Array :

Let’s see the simple example to declare, instantiate, initialize and print the 2Dimensional array.



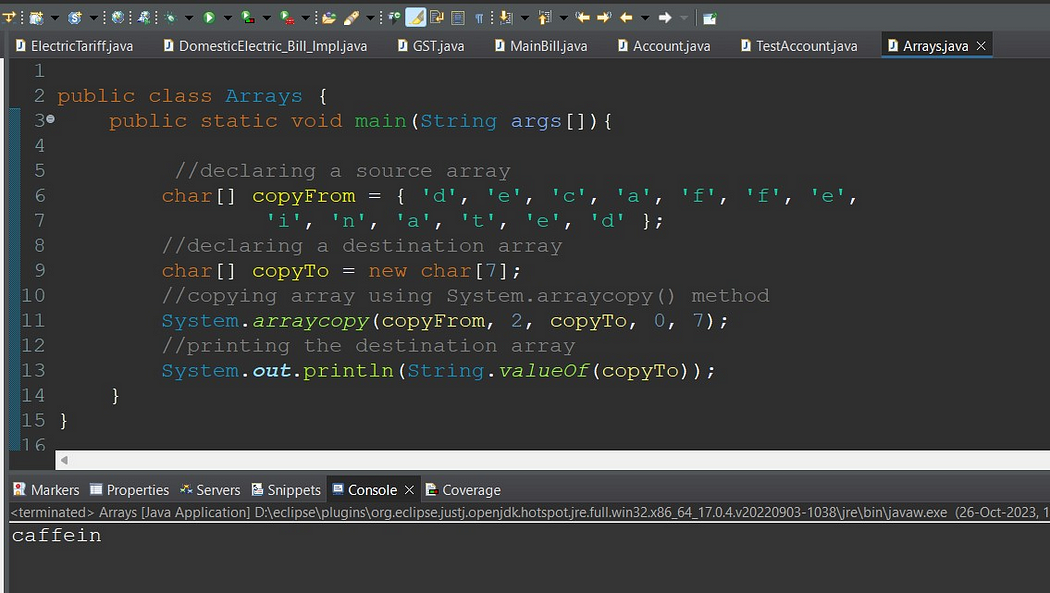
**Copying a Java Array:**

* We can copy an array to another by the arraycopy() method of System class.

**Syntax of arraycopy method**

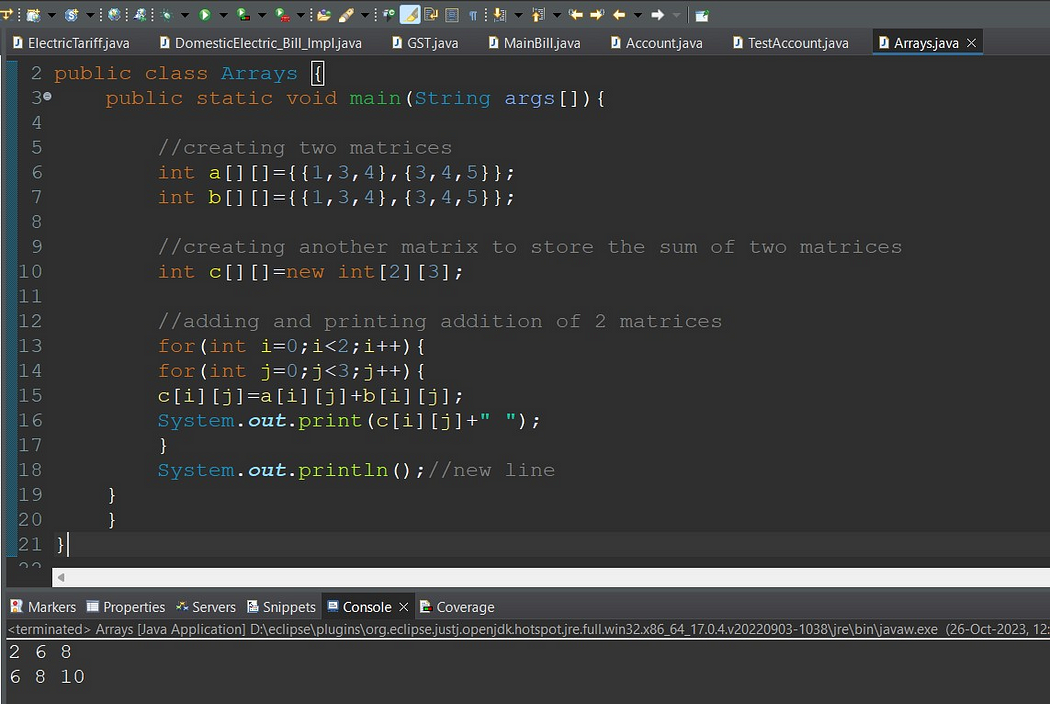
|  |
| --- |
| public static void arraycopy(Object src, int srcPos,Object dest, int destPos, int length) |

**Example of Copying an Array in Java:**



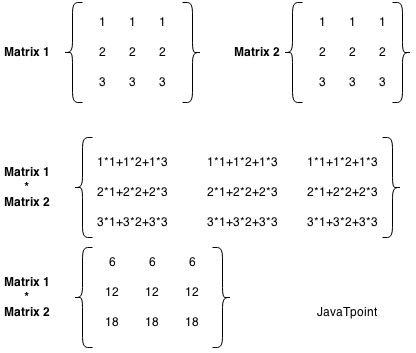
**Addition of 2 Matrices in Java:**

* Let’s see a simple example that adds two matrices.

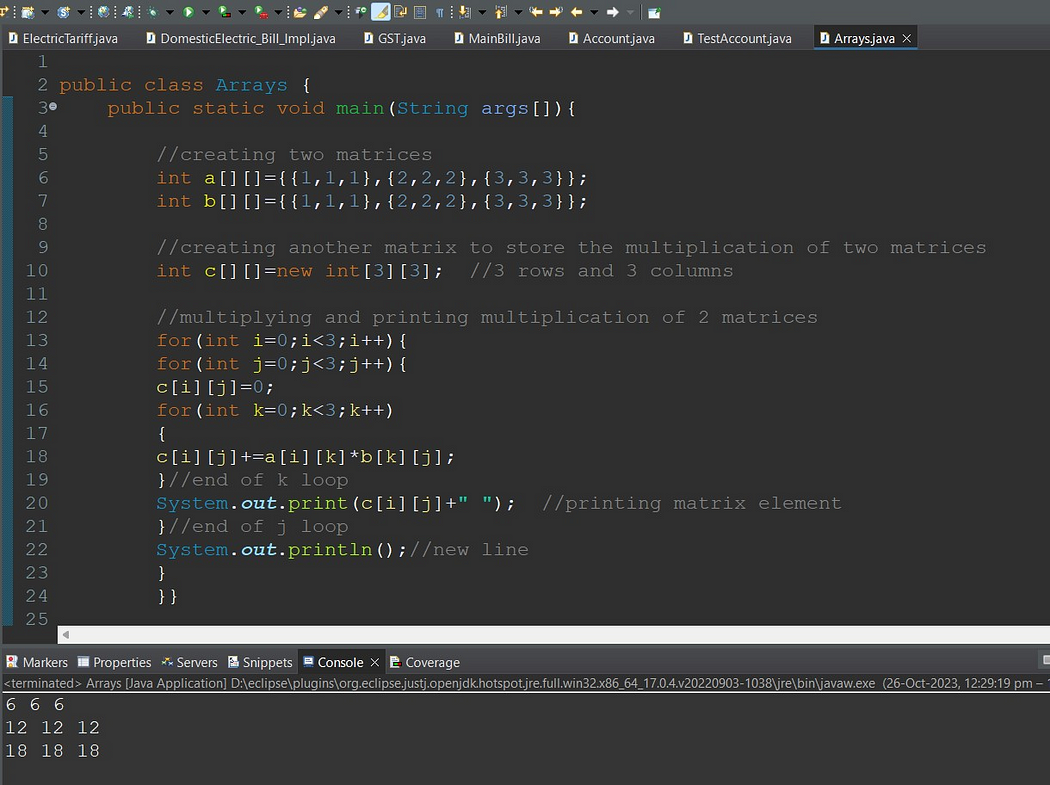


**Multiplication of 2 Matrices in Java:**

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.



Example to multiply two matrices of 3 rows and 3 columns:



**Different methods for Arrays Operations:**

Basic Array Manipulations in Java

* Sorting Arrays with Arrays.sort():

|  |
| --- |
| String[] fruits = {"Apple", "Cherry", "Banana", "Dragonfruit"};  Arrays.sort(fruits);  for (String fruit : fruits) {  System.out.print(fruit + " ");  }  // Output:  // `Apple Banana Cherry Dragonfruit ` |

**We have many overload methods for Arrays.sort() method and it accepts Comparator instance as well.**

* public static <T> void sort(T[] a, Comparator<? super T> c)
* public static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)
* Filling Arrays with Arrays.fill()

The Arrays.fill() method is used to fill an entire array with a single value. It can be useful when you want to reset all values in an array or initialize them to a specific value.

|  |
| --- |
| int[] array = new int[5];  Arrays.fill(array, 1);  for (int i : array) {  System.out.print(i + " ");  }  // Output:  // 1 1 1 1 1 |

* Arrays.toString()

The Arrays.toString() method is a simple and effective way to convert an entire array into a string format, which is useful for printing or logging purposes.

|  |
| --- |
| int[] array = {1, 2, 3};  System.out.println(Arrays.toString(array)); |

* Copying Arrays with Arrays.copyOf()

The Arrays.copyOf() method allows you to create a new array that is a copy of an existing array. This method is useful when you want to manipulate an array **without affecting the original data.**

**public static <T,U> T[] copyOfRange(U[] original, int from, int to, Class<? extends T[]> newType)**

|  |
| --- |
| Here’s how to use Arrays.copyOf():  int[] original = {1, 2, 3};  int[] copy = Arrays.copyOf(original, original.length);  System.out.println(Arrays.toString(copy));  // Output:  // [1, 2, 3] |

* Comparing Arrays with Arrays.equals()

The Arrays.equals() method is used to check if two arrays are equal, meaning **their length, order, and elements are the same.**

* We should use the **deepEquals** method when we want to check the equality between two nested or multidimensional arrays. Also, when we want to compare two arrays composed of user-defined objects, as we’ll see later, we must override the equals method.
* By analyzing the method’s internal implementation, we can see that the method not only checks the top-level elements of the arrays but also checks recursively every subelement of it.
* Therefore, we should avoid using the deepEquals method with arrays that have a self-reference because this will result in a java.lang.**StackOverflowError**.

|  |
| --- |
| Plane[][] planes1  = new Plane[][] { new Plane[]{new Plane("Plane 1", "A320")}, new Plane[]{new Plane("Plane 2", "B738") }};  Plane[][] planes2  = new Plane[][] { new Plane[]{new Plane("Plane 1", "A320")}, new Plane[]{new Plane("Plane 2", "B738") }};    System.out.println(Arrays.equals(planes1, planes2));  System.out.println(Arrays.deepEquals(planes1, planes2)); |

|  |
| --- |
| Let’s see it in action:  int[] array1 = {1, 2, 3};  int[] array2 = {1, 2, 3};  boolean isEqual = Arrays.equals(array1, array2);  System.out.println(isEqual);  // Output:  // true |

* Searching in Arrays with Arrays.binarySearch()

The Arrays.binarySearch() method is used to search for a specific element in an array. It uses the binary search algorithm, which is more efficient than a linear search, but requires the array to be sorted first.

|  |
| --- |
| int[] array = {1, 2, 3, 4, 5};  int **index** = Arrays.binarySea**rch(array, 3);**  **System.out.println(index);**  // Output:  // 2 |

In this example, Arrays.binarySearch() returns 2, which is the index of the number 3 in the array.

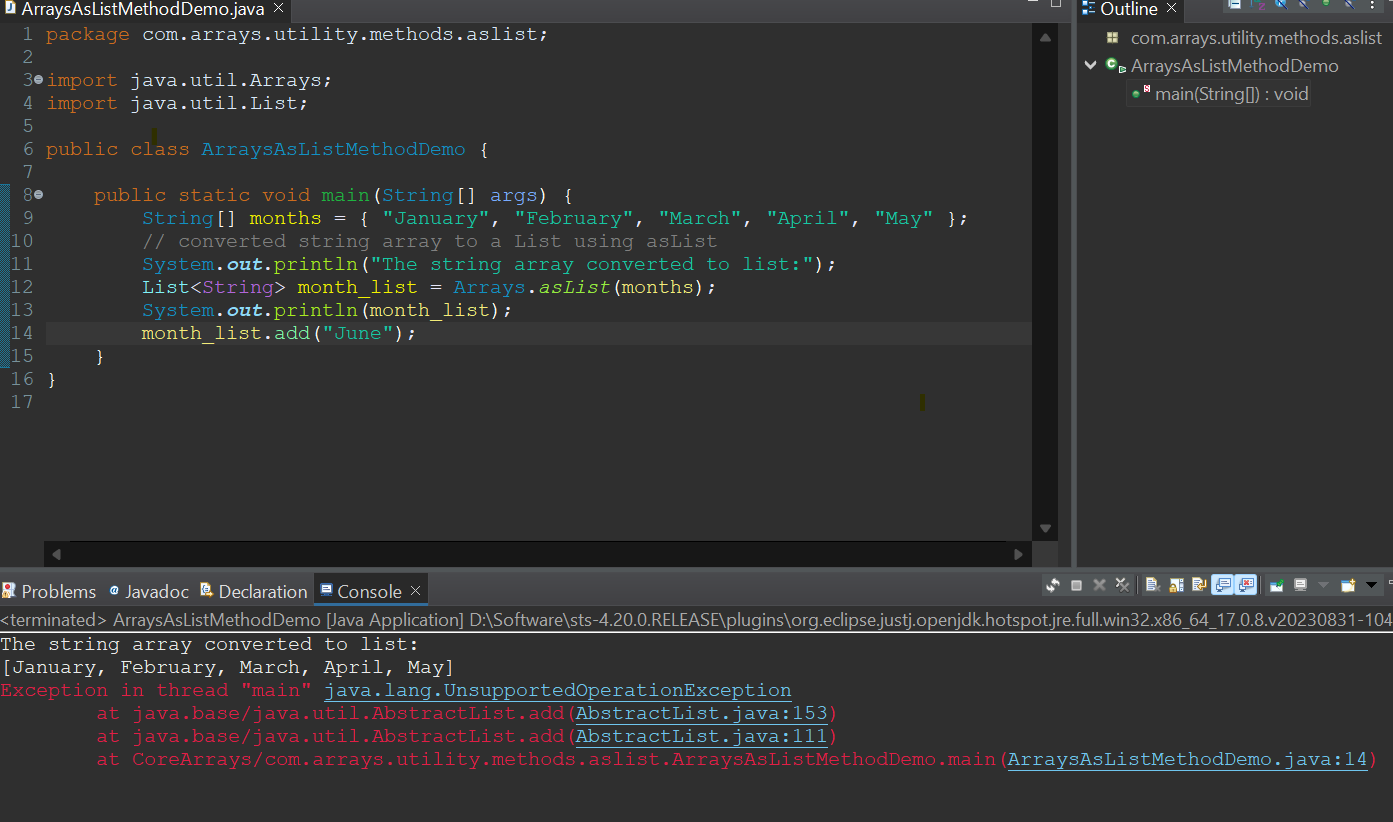
Arrays.compare(array1,arrays2)

* lexicographical order is alphabetical order. The other type is numerical ordering. Consider the following values,
* 1, 10, 2
* Those values are in lexicographical order. 10 comes after 2 in numerical order, but 10 comes before 2 in "alphabetical" order.
* abc, abb, azc
* abb,abc, azc

|  |
| --- |
| public class ArraysLexicographicallyEqualDemo {  public static void main(String[] args)  {  int array1[]={19,27,55,80};  int array2[]={19,27,55,80};  System.out.println(Arrays.compare(array1,array2));  }    } |

* **Arrays. asList()**

<https://medium.com/@arungupta651/how-to-use-arrays-aslist-in-java-with-examples-732fc6a0e92b>



**Exploring Alternative Approaches to Array Manipulation**

* Using ArrayLists for Dynamic Arrays

While Java arrays are powerful, they have a limitation: their size is fixed at the time of creation. To overcome this, we can use ArrayList, a resizable array implementation in the Java Collections Framework.

Here’s how you can create an ArrayList and add elements to it:

|  |
| --- |
| List<Integer> list = new ArrayList<>();  list.add(1);  list.add(2);  list.add(3);  System.out.println(list);  // Output:  // [1, 2, 3] |

Streamlining Array Operations with Java Streams

* Java Streams, introduced in Java 8, provide a powerful and flexible way to process data structures, including arrays.
* With Streams, you can easily perform complex data transformations using functional programming style.
* Here’s an example of using a Stream to filter and transform an array:

|  |
| --- |
| int[] array = {1, 2, 3, 4, 5};  int[] evenSquares = **Arrays.stream(array)**  .filter(n -> n % 2 == 0)  .map(n -> n \* n)  .toArray();  System.out.println(Arrays.**toString**(evenSquares));  // Output:  // [4, 16] |

* In this example, we start with an array of numbers. We create a Stream from the array, filter out the odd numbers, square the remaining even numbers, and collect the results back into a new array.
* Streams provide a powerful and expressive way to manipulate arrays in Java.

Troubleshooting Common Issues with Java Array Methods:

* Handling ArrayIndexOutOfBoundsException

One common issue when working with arrays is the ArrayIndexOutOfBoundsException. This exception is thrown to indicate that you’ve attempted to access an array with an illegal index, either negative or greater than the array’s size.

Here’s a simple example of an **ArrayIndexOutOfBoundsException**:

|  |
| --- |
| int[] array = {1, 2, 3};  try {  System.out.println(array[3]);  } catch (ArrayIndexOutOfBoundsException e) {  e.printStackTrace();  }  // Output:  // java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3  Java  In this example, we try to access array[3], which doesn’t exist because our array’s length is 3 and array indices start at 0. To avoid this exception, always ensure your index is within the array’s bounds. |

Dealing with Null Values in Arrays:

Another common issue is d**ealing with null values in arrays. If you attempt to call a method on a null value, a NullPointerException** will be thrown.

Here’s an example:

|  |
| --- |
| Integer[] **array = {1, null, 3};**  try {  System.out.println(array[1].toString());  } catch (NullPointerException e) {  e.printStackTrace();  }  // Output:  // java.lang.**NullPointerException**  Java  In this example, we attempt to call toString() on a null value, which results in a NullPointerException. To avoid this, always check if an array element is null before attempting to call methods on it. |

**Collections Framework Overview:**

* <https://hansinirup.medium.com/collections-framework-in-java-7f2d13dca075>
* <https://medium.com/javarevisited/getting-started-with-collection-framework-part-1-f2b546adcf29>

Need for Collection Framework:

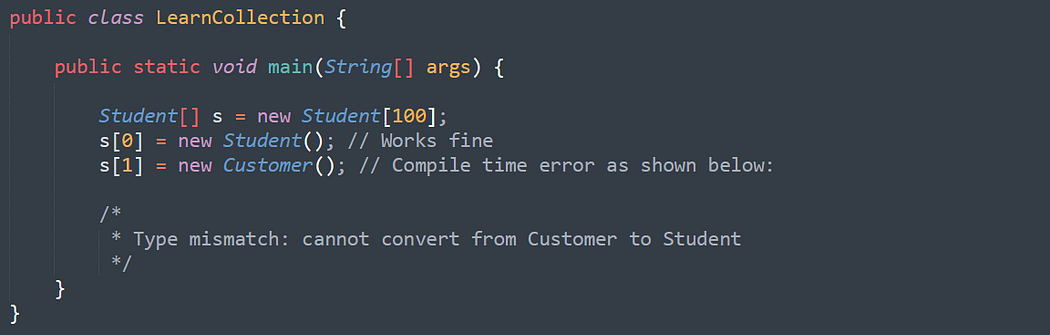
Let’s suppose, you want to represent huge number of values then using separate variable for each value is not a good programming practice. Hence, to overcome this problem, we can go for Array concept.

Array

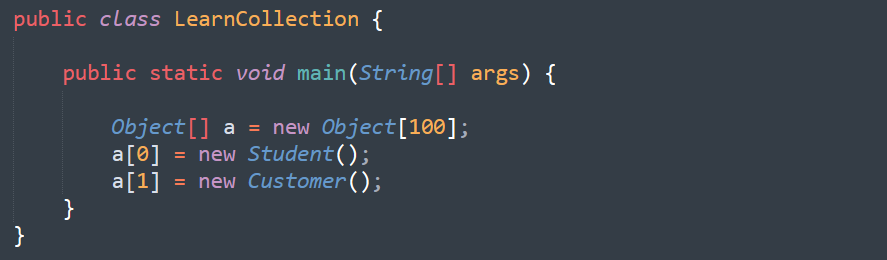
An Array is an indexed collection of fixed number of homogeneous data elements. The main advantage of arrays is, we can represent multiple values by using single variable. So that readability of code will be improved.

**Limitations of Arrays:**

* Arrays are fixed in size i.e. Once we create an array, there is no chance of increasing or decreasing the size based on our requirement.
* Due to this, to use the arrays concept, compulsory we should know the size in advance, which may not be possible always.
* Array can hold only homogeneous data type elements.



However, we can solve this problem by using Object type arrays.



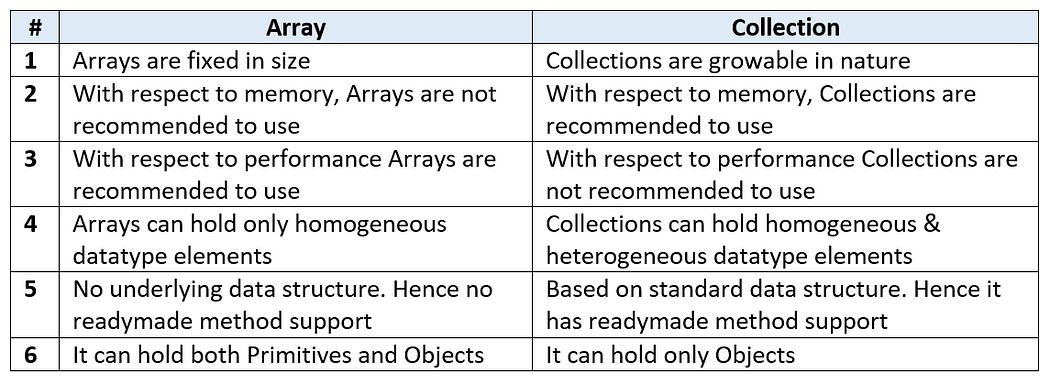
* Arrays concept is not implemented based on some standard data structure and hence readymade method support is not available. For every requirement, we have to write the code explicitly. Which increases complexity of programming.

To overcome above problems of Array, we should go for “Collection” concept.

* Collections are growable in nature i.e. based on our requirement we can increase/decrease the size.
* Collections can hold both homogeneous and heterogeneous elements.
* Every collection class is implemented based on some standard data structure. Hence for every requirement readymade method support is available.

|  |
| --- |
| Being a programmer, we are responsible to use those methods and we are not responsible to implement those methods. |

**What is the difference between Array and Collection:**



**Collections**

* Collections are the containers that group multiple items into a single unit.
* Collections are used to store, retrieve and manipulate data and to transmit data from one method to another.

**Java Collections Framework**

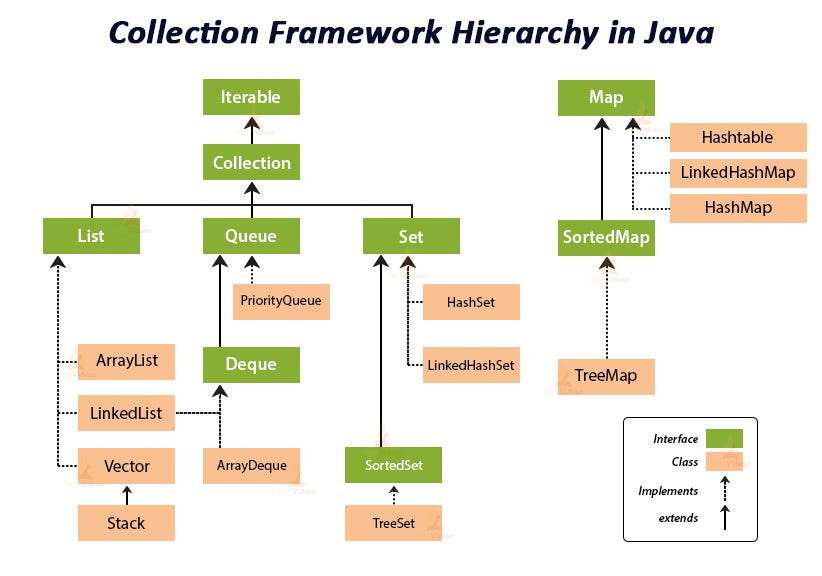
* A collections framework is a unified architecture for representing and manipulating collections.
* Java Collections framework provides many interfaces and classes in order to store data.

**Benefits of a Collections Framework**

* It reduces programming effort.
* It increases program speed and quality.
* It allows interoperability among unrelated APIs.
* It fosters software reuse.
* It reduces effort to design new APIs.

**Java Collections Framework Hierarchy**

* A set of APIs linked together as a parent-child relationship.



**Java Interfaces**

* In interfaces in Java, you write methods which must be implemented by Java objects as rules.
* Interfaces are the reference types which are similar to classes but contains only abstract methods as rules.

1. **Iterable**

* The Iterable interface is the root interface for all the Collection classes.
* The Collection interface along with all its subclasses also implement the Iterable interface.
* The Iterable interface is a core part of Java and is used for enabling objects to be the target of the 'for-each loop' statement. It is present in the java. lang package, and any class implementing this interface allows its objects to be iterated using the enhanced for loop.

**Methods: Iterator <T> iterator()**

* From the above method, we get reference of Iterator and we can iterate in the forward direction in any data structure.

Reference Links about Iterable:

1. **Collection**

* Collection interface is implemented by all the classes in the collection framework and declares the methods that every collection contain.
* This achieves Runtime Polymorphism.

**Methods: Boolean add (Object obj)**

* Boolean addAll (Object obj)
* void clear() , etc.

**List Interface Overview:**

