**Collections**

Index:

1. Arrays (1 -27)

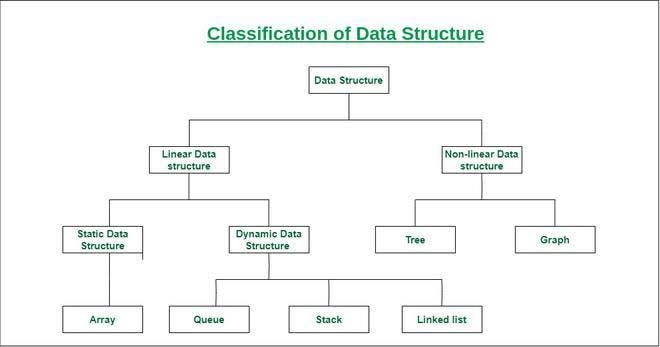
* Memory allocations
* Different methods for Arrays Operations - <https://ioflood.com/blog/java-array-methods/>
* Progarms - ? find the second highest value [1,2,3,4,5,6,7,8,9]?

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

**Choosing the Right Data Structure for Efficient Data Storage in the professional Sector**



Reference link: <https://medium.com/@tajbidtousif/choosing-the-right-data-structure-for-efficient-data-storage-in-the-professional-sector-3b30b14295c4>

Data structures are containers that hold and organize data to facilitate various operations like insertion, retrieval, and deletion. Choosing the right data structure depends on the type of data you're working with and the operations you need to perform. Here are some common data structures:

1. **Arrays**: Arrays are a simple and efficient way to store a collection of elements. They offer fast access times but have fixed sizes, making them less flexible when data size varies. Array is nothing but an index playing

2. **Linked Lists**: Linked lists consist of nodes, each containing data and a reference to the next node. They’re useful for dynamic data and efficient insertions or deletions.

3. **Stacks and Queues**: These are specialized data structures for specific use cases. Stacks follow the Last-In-First-Out (LIFO) principle, while queues follow the First-In-First-Out (FIFO) principle.

4. **Trees**: Trees like Binary Trees and Balanced Trees are ideal for hierarchical data storage and efficient searching. They are commonly used in databases and file systems.

5. **Hash Tables:** Hash tables provide fast data retrieval by using a hashing function to map keys to values. They are excellent for quick lookups but may have collision issues.

6. **Graphs**: Graphs are versatile data structures for modeling complex relationships between data points, commonly used in social networks and routing algorithms.

**How Indexing works:**

Reference links:

1. [Database Indexing In A Nutshell. Indexing is a performance-tuning method… | by Susith Hemathilaka | Nerd For Tech | Medium](https://medium.com/nerd-for-tech/database-indexing-in-a-nutshell-b525b73eadef)
2. [How Do Indexes Work Internally in a Database System? | by Hiteshvats | Medium](https://medium.com/@hiteshvats007/how-do-indexes-work-internally-in-a-database-system-2ebc0d5f9a6d)

**Database Indexing In A Nutshell:**

* Indexing is a performance-tuning method of allowing faster retrieval of records. An index creates an entry point for each value in indexed column(s).
* Indexes are lookup tables used in the background of a database to speed up querying. Indexes power queries by providing a method to quickly lookup the requested data.
* Simply , an index is a pointer to data in a table. An index in a database is very similar to an index page of a book.
* Index helps the database to find the row by providing a pointer to the associated value quickly.

**Why Indexing?**

When database tables get bigger, scanning through millions, billions or trillions of rows to return just two or three rows is a huge waste.

**Imagine when your teacher had asked you to open chapter number five and read some particular sub-topic from that particular chapter. You obviously didn’t scroll page by page to reach that particular sub-topic. You would go to the ‘index’ page, find page number for that particular sub-topic from chapter number five and directly open that page number.**

In the same manner databases use their index to find the requested data when you ask for a specific row, In case DBMS haven’t a index, when you ask for a specific row, it will go sequentially and check with every row; “Is this the row that I need?”, If yes return it, if no, keep searching till the end.

An index is simply a data structure which is stored inside the DBMS, most commonly as a B- tree or hash table.

**By default, Most of the DBMS automatically create an index on primary and unique columns.**

1. **Clustered Index:**

A clustered index defines the order in which data is stored in the table which can be sorted in only one way. So, there can be only a single clustered index per table.

In an RDBMS, usually, the primary key allows you to create a clustered index based on that specific column. Any column values which are sorted and unique can be used as clustered index and binary search algorithm is used most of the time to retrieve the data efficiently.

2. **Non-Clustered Index**

When you need to query the data using another column, we can create a secondary index called a non-clustered index. Let’s say we want to create a secondary index for the first name in the employee table, while the clustered index is the employee id.

For tables with higher number of insert/update operations, non-clustered indexing should be preferred, since it would not require extra time and CPU computations required for sorting the rows after every insert/update on the actual table.

|  |
| --- |
| But **Non Clustered index** is lower than Clustered index since we still need to read from the table created for secondary index then jump to the employee table to get to a specific employee. But, it’s much faster than a full table scan. |

|  |
| --- |
| CREATE INDEX employee\_idx  ON employee(employee\_id); |

**Note**: How Index’s works in database, same way it works in java data structures to find out the element of that object located in corresponding memory location.

**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:**

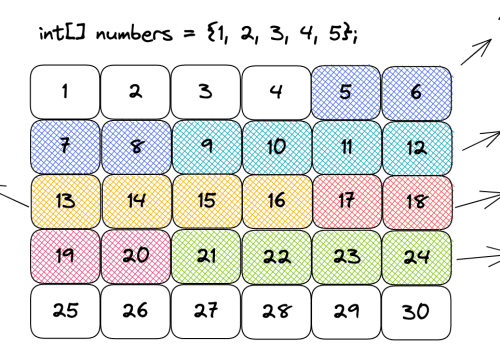
|  |
| --- |
| **int arr[6] = {3, 4, 7, 9, 7, 1}**  address of arr[0] (base address) = 0 x 61fe00  address of arr[3] (element address) = (base address) + (element index \* size of a single element)  0 x 61fe00 + ( 3 \* 4) = 0 x 61fe0c  Here, size of a single element is 4-bytes as it is int- type array.  **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...    } |

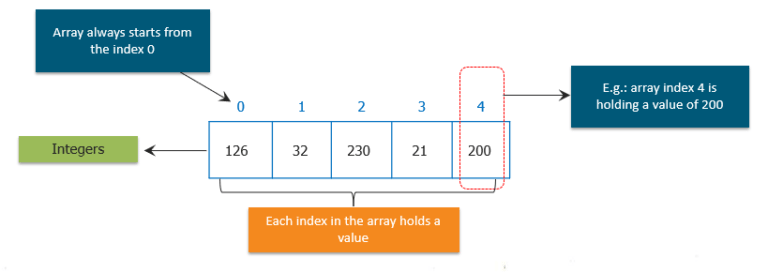
* The elementData variable is the one that will store the data behind the scenes for the ArrayList. Therefore, notice that the dynamic array actually manipulates a static array to behave as dynamic.
* When adding an element to an ArrayList, it will check if the size is greater than 10 and if that is 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.

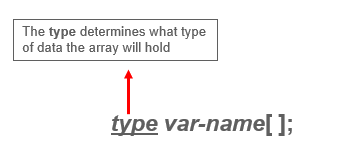
**What are Java Arrays?**

* 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.

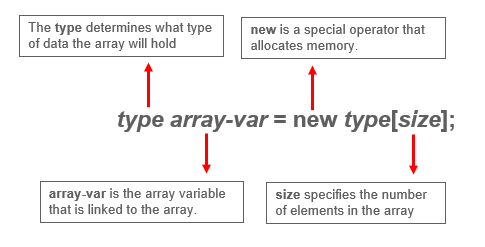
Obtaining an array is a two-step process.

* 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

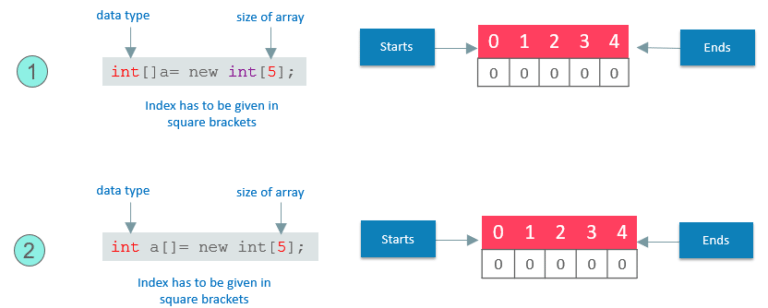
So, let us see how can we declare arrays in different ways.**General Form of Java Array Initialization**

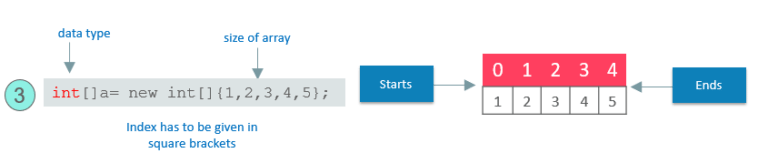


Example:- int month\_days[];

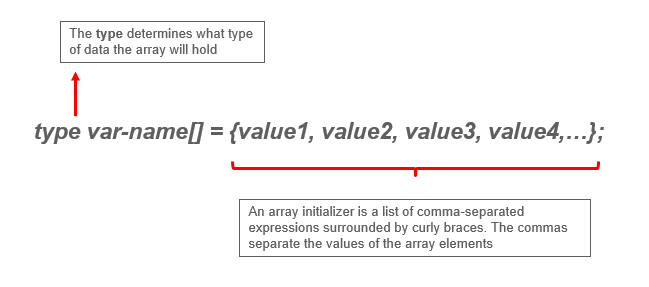
**General Form of Java Array Initialization**

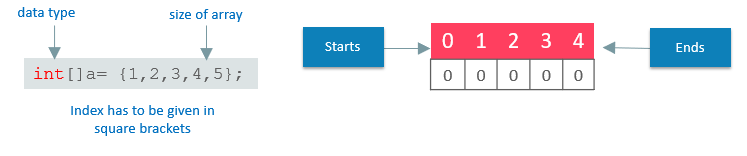
**Example:**





* **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:

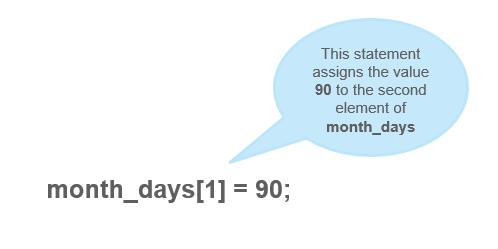


The following code creates an initialized array of integers:

|  |
| --- |
| 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:**



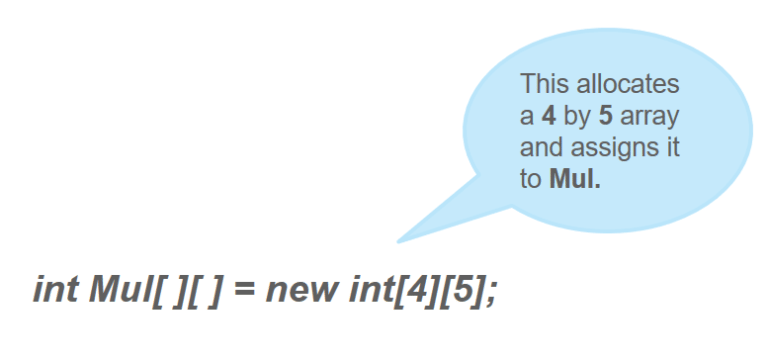
Putting together all the pieces:

|  |
| --- |
| 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.

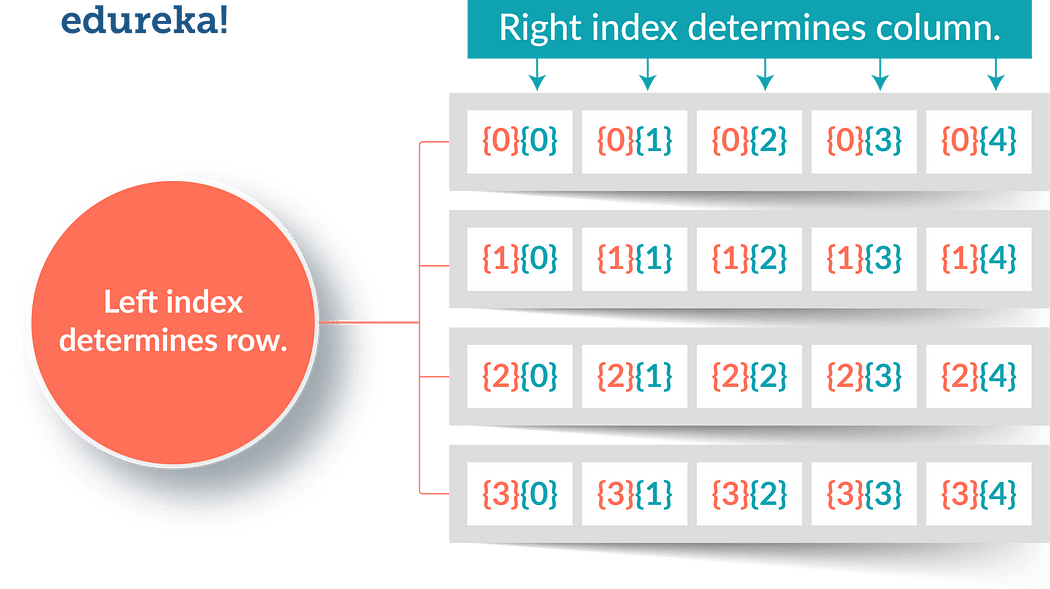
**Java Multidimensional Array:**

* 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.

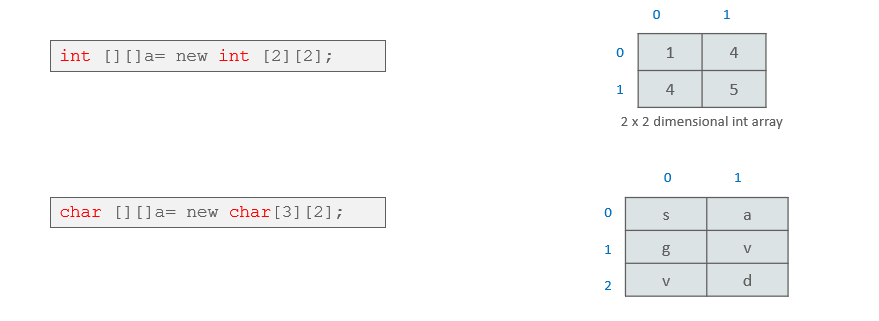
The following program, numbers each element in the array from left to right, top to bottom, and then displays these values:  


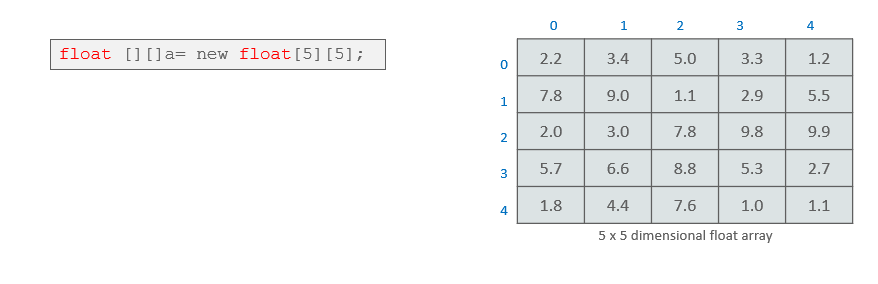
|  |
| --- |
| class Mul2D{  public static void main(String args[]) {  int mul2d[][]= new int[4][5];  int i, j, k = 0;  for(i=0; i&lt;4; i++)  for(j=0; j&lt;5; j++) {  Mul2D[i][j] = k;  k++;  }  for(i=0; i&lt;4; i++) {  for(j=0; j&lt;5; j++);  System.out.print(mul2d[i][j] + " ");  System.out.println();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

**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:**

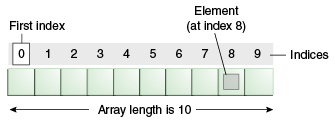
|  |
| --- |
| public class PMethods{  public static void display(int y[])  {  System.out.println(y[0]);  System.out.println(y[1]);  System.out.println(y[2]);    }  public static void main(String args[])  {  int x[] = { 1, 2, 3 };  display(x);  }  } |

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

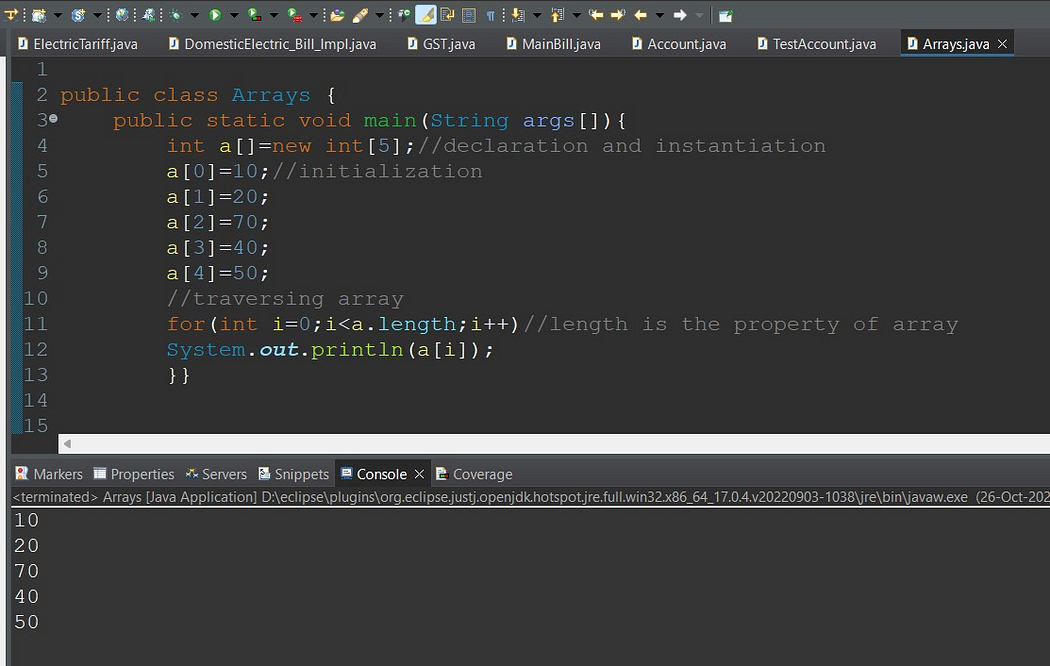
|  |
| --- |
| dataType[] arr; (or)  dataType []arr; (or)  dataType arr[]; |

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.

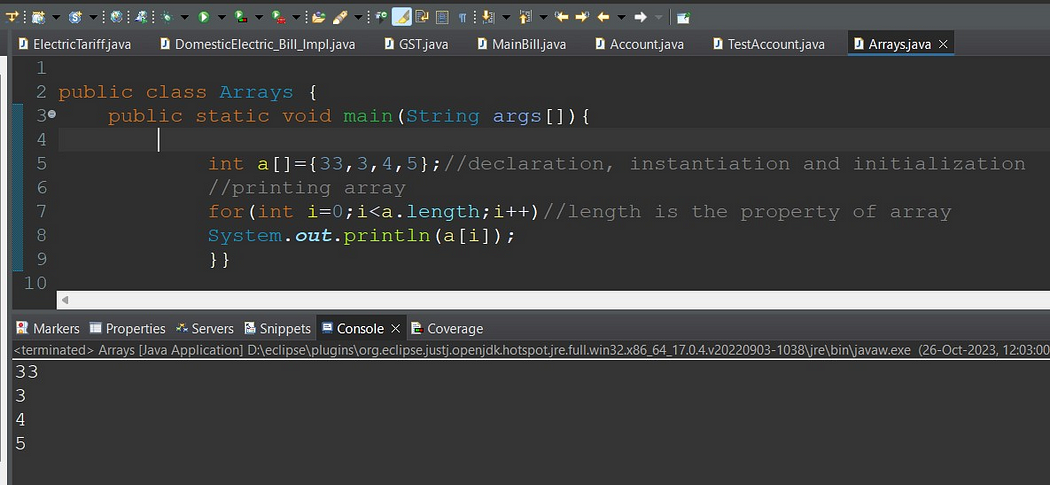


Declaration, Instantiation and Initialization of Java Array:

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:



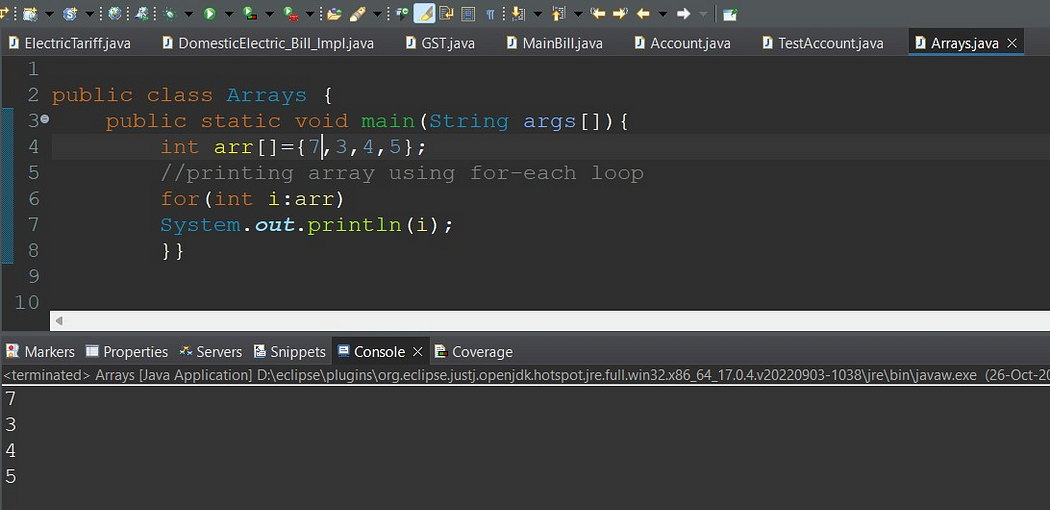
For-each Loop for Java Array:

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:

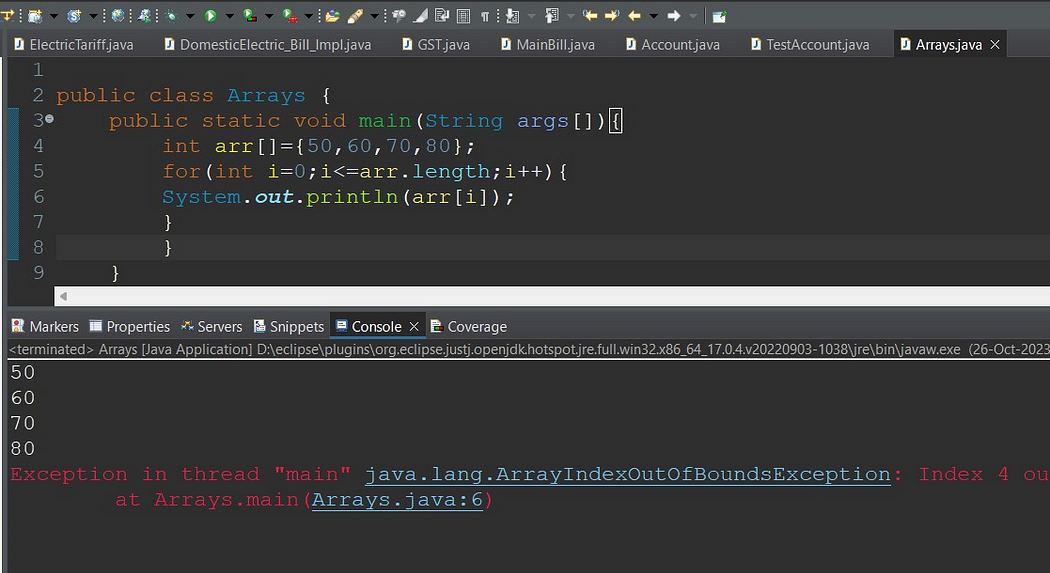
|  |
| --- |
| for(data\_type variable:array){  //body of the loop  } |

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:

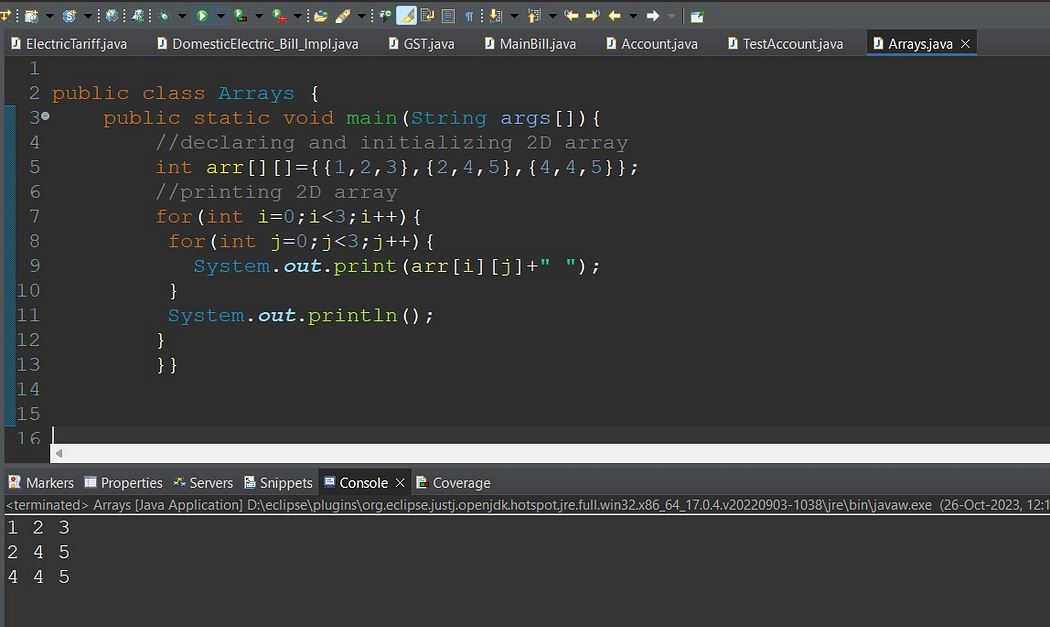
1. **int[][] arr=new int[3][3];//3 row and 3 column**

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.



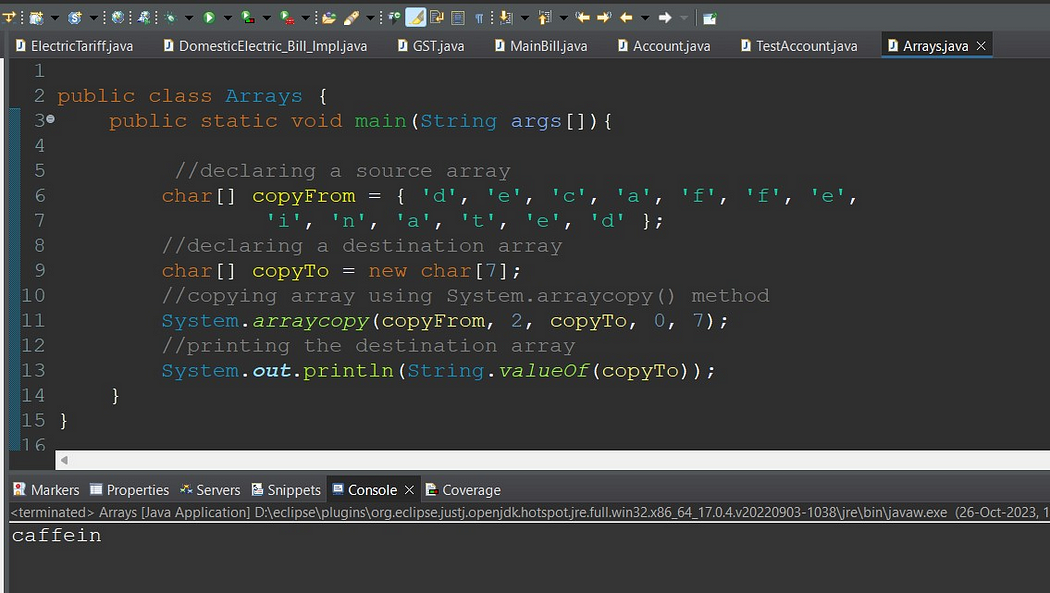
**Copying a Java Array:**

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

**Syntax of arraycopy method**

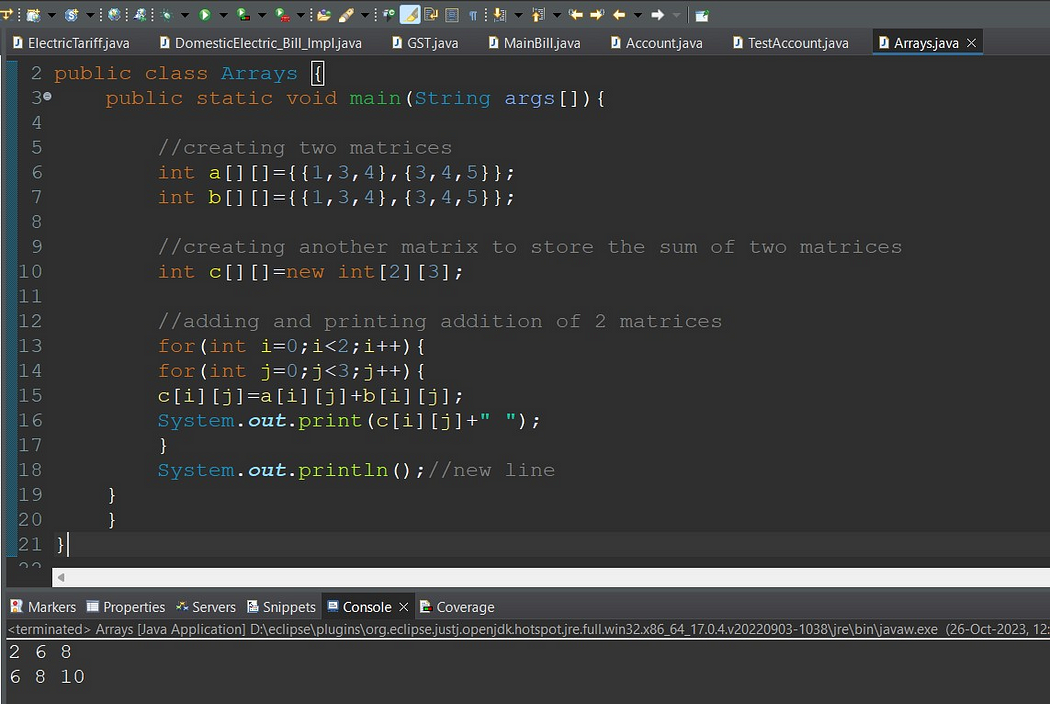
|  |
| --- |
| public static void arraycopy(Object src, int srcPos,Object dest, int destPos, int length) |

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



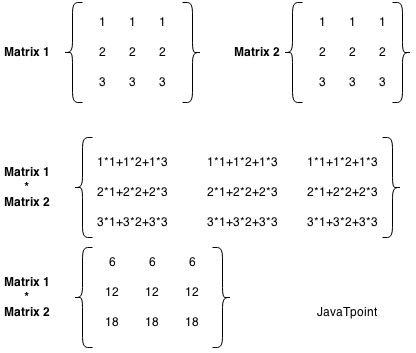
**Addition of 2 Matrices in Java:**

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

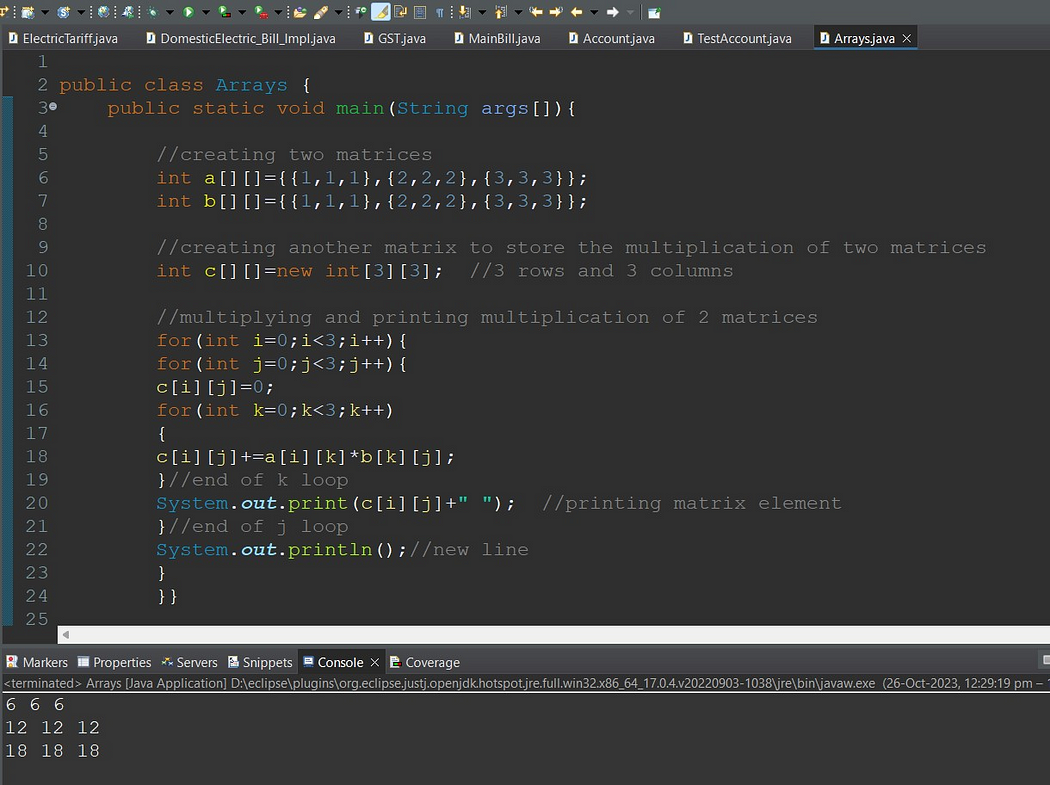


**Multiplication of 2 Matrices in Java:**

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.

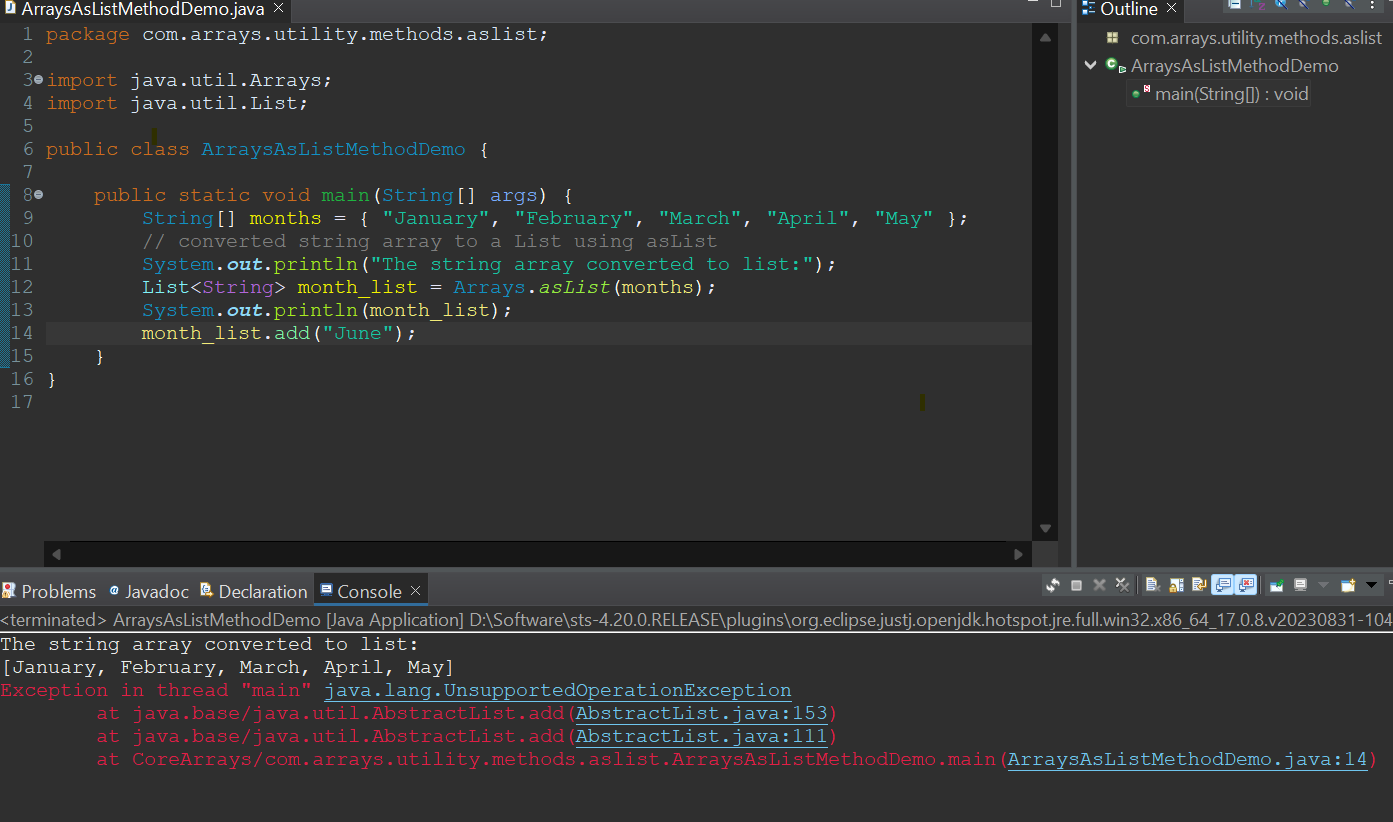
Arrays.compare(array1,arrays2)

* 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:

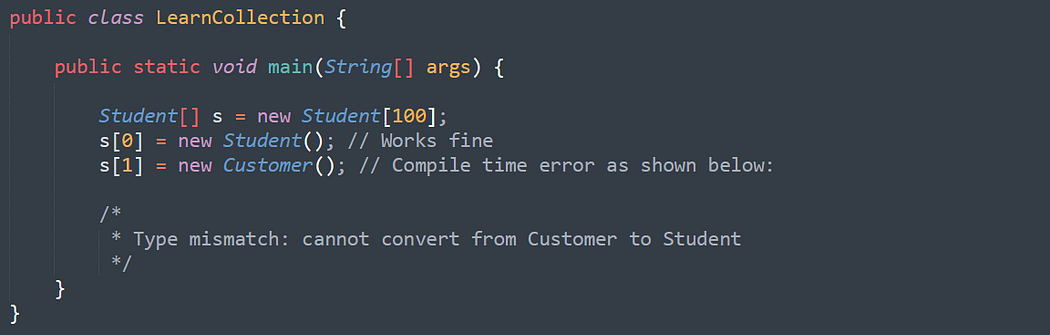
Let’s suppose, you want to represent huge number of values then using separate variable for each value is not a good programming practice. Hence, to overcome this problem, we can go for Array concept.

Array

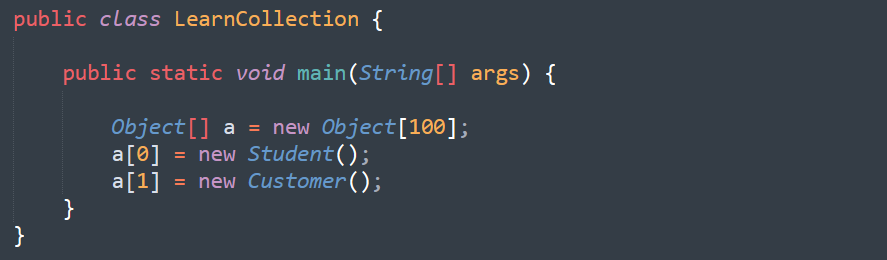
An Array is an indexed collection of fixed number of homogeneous data elements. The main advantage of arrays is, we can represent multiple values by using single variable. So that readability of code will be improved.

**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.



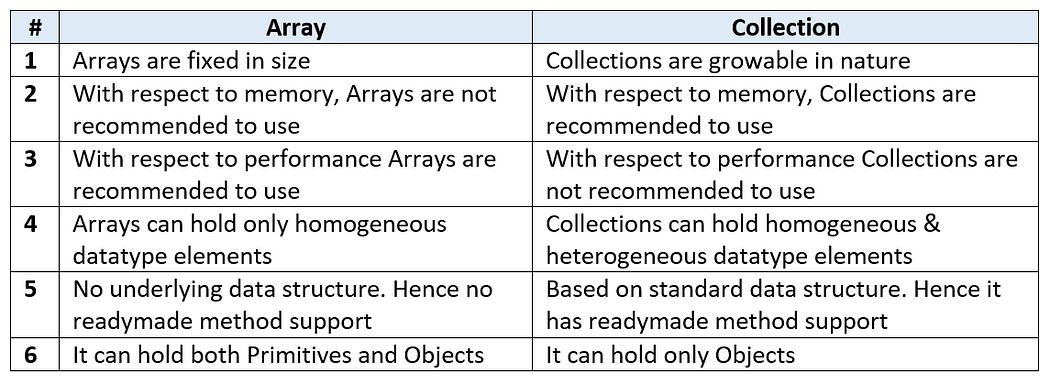
* Arrays concept is not implemented based on some standard data structure and hence readymade method support is not available. For every requirement, we have to write the code explicitly. Which increases complexity of programming.

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**

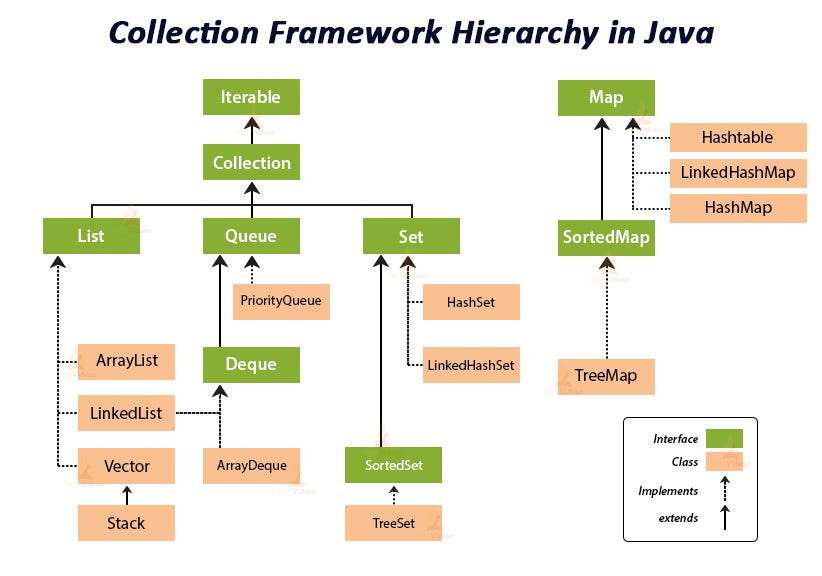
* A collections framework is a unified architecture for representing and manipulating collections.
* Java Collections framework provides many interfaces and classes in order to store data.

**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.



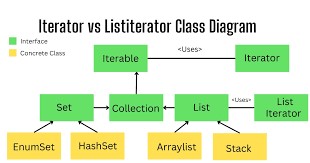
**Java Interfaces**

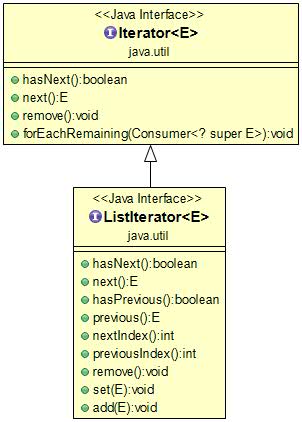
* In interfaces in Java, you write methods which must be implemented by Java objects as rules.
* Interfaces are the reference types which are similar to classes but contains only abstract methods as rules.

1. **Iterable**

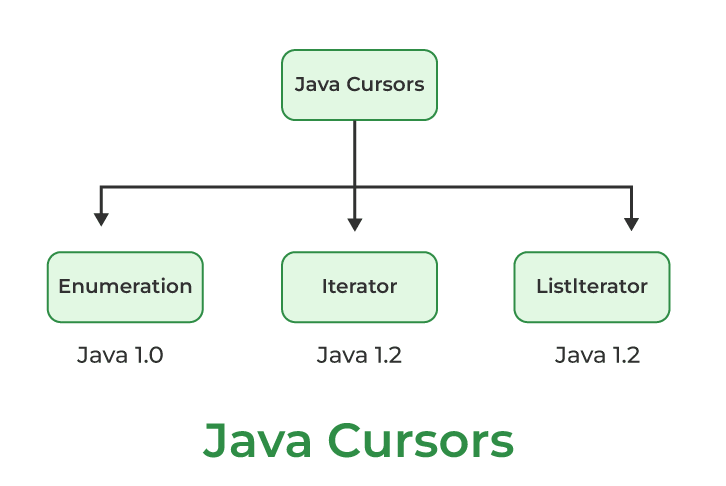
* The Iterable interface is the root interface for all the Collection classes.
* The Collection interface along with all its subclasses also implement the Iterable interface.
* The Iterable interface is a core part of Java and is used for enabling objects to be the target of the 'for-each loop' statement. 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.

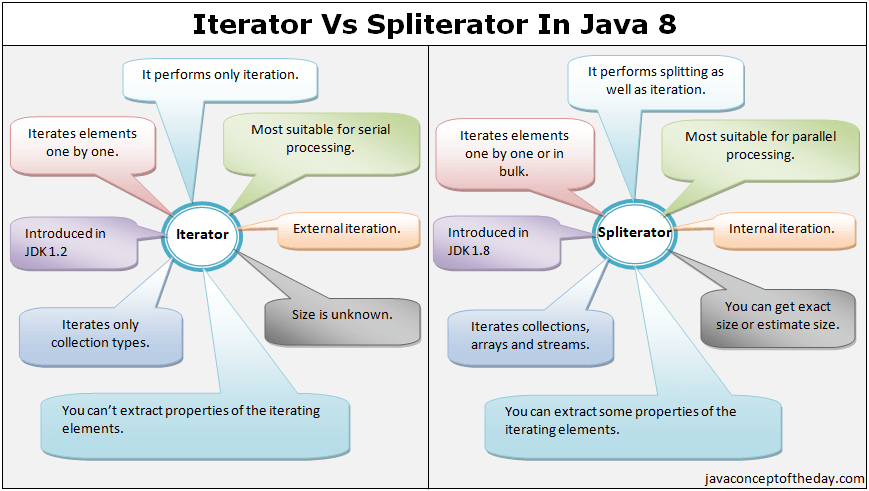
**Methods: Iterator <T> iterator()**

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



A Java Cursor is an Iterator, which is used to **iterate or traverse or retrieve a Collection or Stream object's elements one by one.**





**Iterator and Iterable difference:**

**Difference between iterator and iterable** - iterable will give capability of looping feature of any custom class and Iterator will take the collection instance and point single element and will give you the (hasNext(), next() and remove() ) feature on the **collection data structure.**

* 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.**

**Example:**

|  |
| --- |
| package com.collections.iterable;  import java.util.ArrayList;  import java.util.Arrays;  import java.util.Iterator;  import java.util.List;  public class CustomIterable<T> implements Iterable<T> {  List<T> values = new ArrayList<>();  public void add(T value) {  values.add(value);  }  @Override  public Iterator<T> iterator() {  return new CustomIterator<T>(values);  }  public class CustomIterator<E> implements Iterator<E> {  List<E> data;  int position = 0;  public CustomIterator(List<E> data) {  this.data = data;  }  @Override  public boolean hasNext() {  System.out.println("Hasnext called position: "+ position);  System.out.println("data.size() > position + 1" + (data.size() > position + 1));  return data.size() > position + 1;  }  @Override  public E next() {  E nextValue = this.data.get(position);  position++;  System.out.println("next method caleld and value is : " + nextValue.toString());  return nextValue;  }  }  public static void main(String[] args) {  CustomIterable<Integer> integers = new CustomIterable<>();  integers.add(1);  integers.add(2);  integers.add(3);  integers.add(4);  integers.add(5);  for (Integer integer : integers)  System.out.println("In main method" + integer);    List<String> names = Arrays.asList("A","B","C")    }  } |

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.

**Difference between traditional for loop and enhanced for loop:**

**traditional for loop:**

* Using this for loop we can iterate on any container object.
* **Using this for loop we can iterate on any container object.**
* This for-loop is present from JDK1
* In a normal for-loop, we can increase the counter as per our wish by using

i=i+x( where x is any constant x=1,2,3…)

* Using this for loop we can iterate on any container object.
* In this for-loop, we can iterate in both decrement or increment order.
* In this for-loop, we can replace elements at any specific index.
* By using normal for-loop we can print array elements either in the original order or in reverse order.

|  |
| --- |
| **Example: Printing element in a 1D array**  int[ ] x={1,2,3};  for(int i=0;i<x.length;i++)  {  System.out.println(x[i]);  }  **Example: Printing element in a 2D array using for loop**  int[ ][ ] x={{1,2,3},{4,5,6}};  for(int i=0;i<x.length;i++) {  for(int j=0; j<x[i].length;j++) {  System.out.println(x[i][j]);  }  } |

**Enhanced For Loop(Advance for loop):**

* This for loop is present from JDK5
* But enhanced for loop will execute in a sequential manner i.e counter will always increase by one.
* **We can only iterate on that container by using this loop to implement the iterable interface.**
* But in this for-loop, we can iterate only in increment order.
* But in this for-loop, we don’t have access to the index, so we cannot replace elements at any specific index.
* But in the for-each loop, we can print array element only in the original order, not in reverse order.

|  |
| --- |
| **Example: Printing element in a 1D array using for-each loop**  int[ ] x={1,2,3};  for(int a : x)  {  System.out.println(a);  }  **Example: Printing element in a 2D array using for-each loop**  int[ ][ ] x={{1,2,3},{4,5,6}} ;  for(int[ ] x1 :x){  for(int x2 : x1) {  System.out.println(x2);}  } |

**Exmaple 1:**

|  |
| --- |
| for(String s : list) {  System.out.println("s" + s);  list.remove(0);  }  **Note**: In advanced for loop, we can not use iterator directly. We can use data source collection and we have remove method by passing ides or object. Once you modify the structure during looping time we will get Concurrent modification exception. |

**Reference link :**

<https://anmolsehgal.medium.com/fail-fast-and-fail-safe-iterations-in-java-collections-11ce8ca4180e>

**Fail-fast and Fail-safe iterations in Java Collections:**

* Using iterations we can traverse over the collections objects. The iterators can be either fail-safe or fail-fast.
* Fail-safe iterators means they will not throw any exception even if the collection is modified while iterating over it.
* Whereas Fail-fast iterators throw an exception(**ConcurrentModificationException**) if the collection is modified while iterating over it.

Consider an example:

|  |
| --- |
| ArrayList<Integer> integers = new ArrayList<>();  integers.add(1);  integers.add(2);  integers.add(3);  Iterator<Integer> itr = integers.iterator();  while (itr.hasNext()) {  Integer a = itr.next();  integers.remove(a);  } |

**Explanation:**

* As arrayLists are fail-fast above code will throw an exception.
* First a will have value = 1, and then 1 will be removed in same iteration.
* Next when a will try to **get next(),** as the modification is made to the list, it will throw an exception here.

However if we use an fail-safe collection e.g. **CopyOnWriteArrayList** then no exception will occur:

|  |
| --- |
| List<Integer> integers = new CopyOnWriteArrayList<>();  integers.add(1);  integers.add(2);  integers.add(3);  Iterator<Integer> itr = integers.iterator();  while (itr.hasNext()) {  Integer a = itr.next();  integers.remove(a);  }  Here if we print the element a, then all the elements will be printed. |

**Fail-Fast Iterators internal working:**

* Every fail fast collection has a **modCount** field, to represent how many times the collection has **changed/modified.**
* So at every modification of this collection we increment the **modCount** value. For example the modCount is incremented in below cases:

1. When one or more elements are removed.

2. When one or more elements are added.

3. When the collection is replaced with other collection.

4. When the collection is sorted.

* So everytime there is some change in the collection structure, the mod count is incremented.

**Important Points:**

* Now the iterator stores the **modCount** value in the initialization as below:

|  |
| --- |
| int expectedModCount = modCount; |

* Now while the iteration is going on, expectedModCount will have old value of modCount.
* If there is any change made in the collection, the modCOunt will change and then an exception is thrown using:

|  |
| --- |
| if (modCount != expectedModCount)  throw new ConcurrentModificationException(); |

This code is used in most of the iterator methods e.g.

1. next()

2. remove()

So if we make any changes to the collection, the modCount will change, and expectedModCount will not be hence equal to the modCount. Then if we use any of the above methods of iterator, the ConcurrentModificationException will be thrown.

**Note**: If we remove/**add** the element using the remove() or add() of iterator instead of collection, then in that case no exception will opccur. It is because the remove/**add** methods of iterators call the remove/add method of collection internally, and also it reasigns the expectedModCount to new modCount value.

|  |
| --- |
| ArrayList.this.remove(lastRet);  cursor = lastRet;  lastRet = -1;  expectedModCount = modCount;  and  ArrayList.this.add(i, e);  expectedModCount = modCount; |

So the below code is safe as we are removing the element from the iterator here:

|  |
| --- |
| Iterator<Integer> itr = integers.iterator();  while (itr.hasNext()) {  if (itr.next() == 2) {  // will not throw Exception  itr.remove();  }  } |

Whereas the below code will throw an exception as we are removing the element from the collection here.

|  |
| --- |
| Iterator<Integer> itr = integers.iterator();  while (itr.hasNext()) {  if (itr.next() == 3) {  // will throw Exception on  // next call of next() method  integers.remove(3);  }  } |

**Fail-Safe Iterators internal working:**

* Unlike the fail-fast iterators, these iterators traverse over the clone of the collection. So even if the original collection gets structurally modified, no exception will be thrown.
* E.g. in case of CopyOnWriteArrayList the original collections is passed and is stored in the iterator:

|  |
| --- |
| public Iterator<E> iterator() {  return new COWIterator<E>(getArray(), 0);  } |

here iterator() method returns the iterator of the **CopyOnWriteArrayList**. As we can see, it passes the getArray() in the constructor of the iterator. This getArray() has all the collection elements.

Now the iterator(**COWIterator** here) will save this to traverse upon as:

|  |
| --- |
| COWIterator(Object[] elements, int initialCursor) {  cursor = initialCursor;  **snapshot** = elements;  } |

**So the original collection elements are saved in the snapshot field variable.**

So all the iterator methods will work on this snapshot method. So even if there is any change in the original collection, no exception will be thrown. But note the the iterator will not reflect the latest state of the collection.

Iterator<Integer> itr = integers.iterator();

while (itr.hasNext()) {

int a = itr.next();

if (a == 1) {

integers.remove(Integer.valueOf(a));

}

System.out.print(a);

}

So as we are removing the element from the collection. the collection now has only elements 2 and 3 in it. But the iterator will print all the elements 1,2,3 because it traverses over the snapshot of the collection elements.

We can print the collection elements after the above code. It will print only 2,3 as:

Iterator<Integer> itr = integers.iterator();

while (itr.hasNext()) {

int a = itr.next();

System.out.print(a);

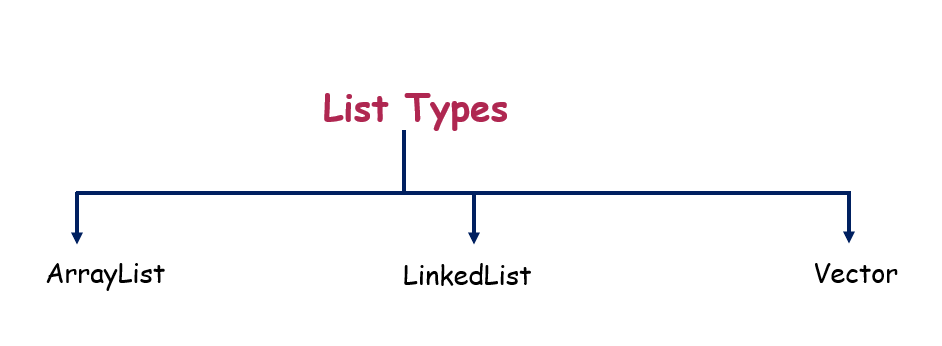
}

**Note**: although it does not throw any exception, but the downsides of this iterator are:

1. They will not reflect the latest state of the collection.

2. It requires extra memory as it clones the collection.

**List Interface Overview:**



Traditional loop and

**ArrayList**:

<https://medium.com/@reetesh043/java-collection-arraylist-a1a830864e81>

**Create ArrayList through Constructor:**

The core data structure of ArrayList is an array that holds the elements. When we create an ArrayList, it creates an internal array of a certain initial capacity. By default, the initial capacity is **10**, but we can specify a different capacity in the **constructor**.

|  |
| --- |
| // Default initial capacity.  private static final int DEFAULT\_CAPACITY = 10;  /\*\*  \* The array buffer into which the elements of the ArrayList are stored.  \*/  transient Object[] elementData; |

* ArrayList is a part of collection framework and is present in java.util package. It provides us dynamic arrays in Java.
* Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.
* ArrayList inherits AbstractList class and implements List interface.
* ArrayList is initialized by a size, however the size can increase if collection grows or shrink if objects are removed from the collection.
* Java ArrayList allows us to randomly access the list.
* ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases (see this for details).
* ArrayList in Java can be seen as similar to vector in C++.

Must know about array list and how it works internally?

create arrayList()? What is happening?

1. **How to Create arrayList and how it created internally?**

Reference Link - [**https://www.geeksforgeeks.org/internal-working-of-arraylist-in-java/**](https://www.geeksforgeeks.org/internal-working-of-arraylist-in-java/)

* ArrayList is a resizable array implementation in Java. ArrayList grows dynamically and ensures that there is always a space to add elements.
* The backing data structure of ArrayList is an array of Object classes. ArrayList class in Java has 3 constructors. It has its own version of readObject and writeObject methods.
* Object Array in ArrayList is transient. It implements RandomAccess, Cloneable, and java.io.Serializable (which are Marker Interface in Java)

|  |  |  |
| --- | --- | --- |
| public ArrayList() {  this.elementData = DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA;  } | public ArrayList(int initialCapacity) { | public ArrayList(Collection<? extends E> c) |

Syntax:

public class ArrayList<E>

extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, java.io.Serializable

Internally an ArrayList uses an Object[] Array which is an array of objects. All operation like deleting, adding, and updating the elements happens in this Object[] array.

/\*\*

\* The array buffer into which the elements of the ArrayList are stored.

\* The capacity of the ArrayList is the length of this array buffer. Any

\* empty ArrayList with elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA

\* will be expanded to DEFAULT\_CAPACITY when the first element is added.

\*/

|  |
| --- |
| **transient Object[] elementData; // non-private to simplify nested class access** |

* The above code is from Java 8 in Java 7 the array is declared as a Private transient Object but in Java 8 it’s not Private because non-private is to simplify access for a nested class like Itr, ListItr, SubList.

Initialization

List<String> arrayList = new ArrayList<String>();

while declaring ArrayList below code is executed as the default constructor of the ArrayList class is invoked.

**In Java 7**

public ArrayList() {

this(10);

}

Hereby default capacity of the Array size is 10.

**In Java 8**

|  |
| --- |
| private static final int DEFAULT\_CAPACITY = 10;\\ Default initial capacity.    // Shared empty array instance used for empty instances.  private static final Object[] EMPTY\_ELEMENTDATA = {};  /\*\*  \* Shared empty array instance used for default sized empty instances. We  \* distinguish this from EMPTY\_ELEMENTDATA to know how much to inflate when  \* first element is added.  \*/ |

Here, the List is initialized with a default capacity of 10. Array List with elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA will be expanded to DEFAULT\_CAPACITY when the first element is inserted into the Array list ( Adding 1st element to ArrayList).

private static final Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {};

**Example:**

|  |
| --- |
| import java.util.ArrayList;  import java.util.Collection;    public class Main {  public static void main(String args[])  {  Collection<Integer> arr = new ArrayList<Integer>();  arr.add(1);  arr.add(2);  arr.add(3);  arr.add(4);  arr.add(5);  System.out.println("This is arr " + arr);    ArrayList<Integer> newList  = new ArrayList<Integer>(arr);    System.out.println("This is newList " + newList);  newList.add(7);  newList.add(700);    System.out.println(  "This is newList after adding elements "  + newList);  }  }  **Output**  This is arr [1, 2, 3, 4, 5]  This is newList [1, 2, 3, 4, 5]  This is newList after adding elements [1, 2, 3, 4, 5, 7, 700] |

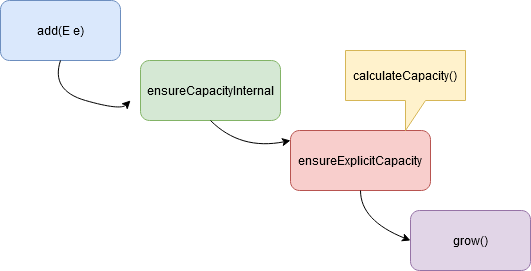
* Here the elements in the arr are passed to newList. So elements of arr were copied to newList this is shown in the above example.
* How the size of ArrayList grows dynamically? [add() method]

Let’s deep dive into how to add method that works in Array list with help of the internal Java 8 code of ArrayList.

If we try to add an element using the add() method in the array list Internally then it checks for the capacity to store the new element or not, If not then the new capacity is calculated as shown in the internal code of the add() method.

**add() method works:** we have two methods

* add(int index, E obj)
* add(E obj)



|  |
| --- |
| public boolean add(E e) {  ensureCapacityInternal(size + 1); // Size Increments  elementData[size++] = e;  return true;  } |

Explanation:

* Here in the add(Object ) method object is passed and the size is increased. Internal capacity of the array is ensured by the ensureCapacityInternal() method

|  |
| --- |
| private void ensureCapacityInternal(int minCapacity) {  if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {  minCapacity = Math.max(DEFAULT\_CAPACITY, minCapacity);  }  ensureExplicitCapacity(minCapacity);  } |

* This ensureExplicitCapacity method determines what is the current size of elements and what is the maximum size of the array. here Minimum capacity will be the maximum of default capacity and mincapacity then goes for ensureExplicitCapacity method mincapacity as an argument.

|  |
| --- |
| private void ensureExplicitCapacity(int minCapacity) {  modCount++;    // overflow-conscious code  if (minCapacity - elementData.length > 0)  grow(minCapacity);  } |

* Here if the (mincapacity – arraylength) is greater than 0(>0) then the Array size will grow by calling the grow() method and mincapacity as the argument passed.

|  |
| --- |
| /\*\*  \* Increases the capacity to ensure that it can hold at least the  \* number of elements specified by the minimum capacity argument.  \*  \* @param minCapacity the desired minimum capacity  \*/  private void grow(int minCapacity) {  // overflow-conscious code  int oldCapacity = elementData.length;  **int newCapacity = oldCapacity + (oldCapacity >> 1);**  if (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  if (newCapacity - MAX\_ARRAY\_SIZE > 0)  newCapacity = hugeCapacity(minCapacity);  // minCapacity is usually close to size, so this is a win:  elementData = Arrays.copyOf(elementData, newCapacity);  } |

* The grow method in the ArrayList class gives the new size array. In Java 8 and later The new capacity is calculated which is 50% more than the old capacity and the array is increased by that capacity. It uses **Arrays.copyOf** which gives the array increased to the new length by right shift operator also it will grow by 50% of old capacity.

|  |
| --- |
| int newCapacity = oldCapacity + (oldCapacity >> 1); |

* For example, if the Array size is 10 and already all the rooms were filled by the elements, while we are adding a new element now the array capacity will be increased as **10+ (10>>1) => 10+ 5 => 15.** Here the size is increased from **10 to 15**. To increase the size by 50% we use the right shift operator. While in Java 6 it’s totally different from the ab**ove calculation on increasing the size of the Array, in java 6 the capacity increases by the amount to 1.5X**

|  |
| --- |
| int newCapacity = (oldCapacity \* 3)/2 + 1; -- Java 6 |

**How does the remove method work in ArrayList? [ArrayList shrinks automatically]**

* To remove an element from ArrayList in Java , we can use either **remove(int i)** [0 index based] or remove(Object o) .
* while removing any element from an ArrayList, internally all the subsequent elements are to be shifted left to fill the gap created by the removed element in the array then subtracts one from their indices.
* size of the array will be decreased by 1 ( – – size).

|  |
| --- |
| // Removes the element at the specified position in this list.  // Shifts any subsequent elements to the left (subtracts one from their indices).  public E **remove**(int index) {  rangeCheck(index);  **modCount**++;  E **oldValue** = elementData(index);  int numMoved = size - index - 1;  if (numMoved > 0)  **System.arraycopy**(elementData, index+1, elementData, index, numMoved);  **elementData**[--size] = null; // clear to let GC do its work  return oldValue;  } |

* **System.arrayCopy** method is used for this purpose. Here **index+1** is the initial position and index is the final position.
* Since the element at the position index is removed so elements starting from index+1 are copied to the destination starting from the **index**.

|  |
| --- |
| System.arraycopy(elementData, index+1, elementData, index, numMoved);  **This is how ArrayList shrinks automatically.//** |

**Best practices in creating ArrayList**

* Whenever an ArrayList exceeds the capacity of the internal array it creates a new internal array with 50% more capacity and copies all old elements from the old array to the new one. Repeated resizes will add load for memory allocation and garbage collection and the time to do the copy.
* If you have a clear idea of the eventual size of your ArrayList it is best to pass that size as the initialCapacity to the ArrayList constructor.
* However if you are copying another collection into the list it is better to pass that collection into the ArrayList constructor.

Performance of ArrayList :

The time complexity of the common operations in ArrayList java.

1. add(): For adding a single element O(1) . Adding n element in the array list takes O(n).
2. add(index, element): adding element in particular index in average runs in O(n) time.
3. get(): is always a constant time O(1) operation.
4. remove(): runs in linear O(n) time. We have to iterate the entire array to find the element fit for removal.
5. indexOf(): It runs over the whole array and iterates through each and every element worst case will be the size of the array n .so, it requires O(n) time.
6. contains(): implementation is based on indexOf(). So it will also run in O(n) time.
7. The size, isEmpty, set, iterator, and listIterator operations run in constant time O(1)

**Note:**

1. ArrayList is a resizable array implementation in java.
2. The backing data structure of ArrayList is an array of Object class.
3. When creating an ArrayList you can provide initial capacity then the array is declared with the given capacity.
4. The default capacity value is 10. If the initial capacity is not specified by the user then the default capacity is used to create an array of objects.
5. When an element is added to an ArrayList it first checks whether the new element has room to fill or it needs to grow the size of the internal array, If capacity has to be increased then the new capacity is calculated which is 50% more than the old capacity and the array is increased by that capacity.

### Methods of ArrayList in java:

|  |  |
| --- | --- |
| **Method** | **Description** |
| void [add](https://www.javatpoint.com/java-arraylist-add-method)(int index, E element) | It is used to insert the specified element at the specified position in a list. |
| boolean [add](https://www.javatpoint.com/java-arraylist-add-method)(E e) | It is used to append the specified element at the end of a list. |
| boolean [addAll](https://www.javatpoint.com/java-arraylist-addall-method)(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| boolean [addAll](https://www.javatpoint.com/java-arraylist-addall-method)(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |
| void [clear](https://www.javatpoint.com/java-arraylist-clear-method)() | It is used to remove all of the elements from this list. |
| void ensureCapacity(int requiredCapacity) | It is used to enhance the capacity of an ArrayList instance. |
| E get(int index) | It is used to fetch the element from the particular position of the list. |
| boolean isEmpty() | It returns true if the list is empty, otherwise false. |
| [Iterator()](https://www.javatpoint.com/java-arraylist-iterator-method) |  |
| [listIterator()](https://www.javatpoint.com/java-arraylist-listiterator-method) |  |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. |
| <T> T[] toArray(T[] a) | It is used to return an array containing all of the elements in this list in the correct order. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| boolean contains(Object o) | It returns true if the list contains the specified element. |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| E remove(int index) | It is used to remove the element present at the specified position in the list. |
| boolean [remove](https://www.javatpoint.com/java-arraylist-remove-method)(Object o) | It is used to remove the first occurrence of the specified element. |
| boolean [removeAll](https://www.javatpoint.com/java-arraylist-removeall-method)(Collection<?> c) | It is used to remove all the elements from the list. |
| boolean removeIf(Predicate<? super E> filter) | It is used to remove all the elements from the list that satisfies the given predicate. |
| protected void [removeRange](https://www.javatpoint.com/java-arraylist-removerange-method)(int fromIndex, int toIndex) | It is used to remove all the elements lies within the given range. |
| void replaceAll(UnaryOperator<E> operator) | It is used to replace all the elements from the list with the specified element. |
| void [retainAll](https://www.javatpoint.com/java-arraylist-retainall-method)(Collection<?> c) | It is used to retain all the elements in the list that are present in the specified collection. |
| E set(int index, E element) | It is used to replace the specified element in the list, present at the specified position. |
| void sort(Comparator<? super E> c) | It is used to sort the elements of the list on the basis of the specified comparator. |
| Spliterator<E> spliterator() | It is used to create a spliterator over the elements in a list. |
| List<E> subList(int fromIndex, int toIndex) | It is used to fetch all the elements that lies within the given range. |
| int size() | It is used to return the number of elements present in the list. |
| void trimToSize() | It is used to trim the capacity of this ArrayList instance to be the list's current size. |

**How index based mechanism will work in java:**

<https://www.codecademy.com/article/sql-indexes>

**Database Indexing In A Nutshell**

Indexing is a performance-tuning method of allowing faster retrieval of records. An index creates an entry point for each value in indexed column(s).

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

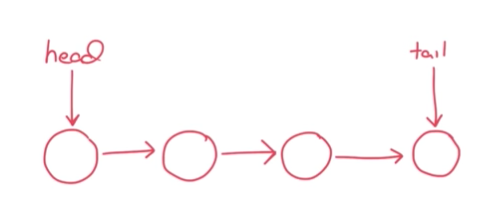
**Base address :** It is the address of the element at the index 0 or the location of the first element of the array in the memory.. The compiler knows this address as the memory location of the array.

**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.

**Linked List Data Structure:**

1. What is the use of linked list?
2. Where and all it is preferable to use?
3. Advantages and dis-advantages of linked list?
4. How linked list works internally?
5. How linked list operations works internally?
6. How many types of linked list are there and use of them?
7. Single linked list
8. Double linked list
9. Circular linked list



**What is the use of linked list:**

* A linked list is a **linear data structure**, in which the elements are not stored at contiguous memory locations.
* Every element in the linked list is a separate object with a **data part** and **address part**. Each of these elements is known as a **NODE**.
* Each node contains a data field and a reference(link) to the next node in the list.
* It is a part of the Collection framework present in java. util package.

**Reference links :**

Linked Lists in Java by Venkat - <https://projectjava.medium.com/linked-lists-in-java-4b7a33f5b493>

**How add(element), get(index), remove(index) works in linked list: (Custom Linked list):**

|  |
| --- |
| package com.customlinkedlist;  import java.util.Collection;  import java.util.Iterator;  import java.util.List;  import java.util.ListIterator;  class Node<T> {    **@Override**  **public String toString() {**  **return "Node [val=" + val + ", next=" + next + ", previus=" + previus + "]";**  **}**  private T val;  private Node next, previus;  public Node(T val) {  this.val = val;  this.next = this.previus = null;  }  }  public class CustomLinkedList<T> implements List<T> {    @Override  public String toString() {  return "CustomLinkedList [head=" + head + ", size=" + size + "]";  }  private Node<T> head;  private int size;  public CustomLinkedList() {  this.size = 0;  }  **@Override**  **public boolean add(T element) {**  **Node<T> newNode = new Node<>(element);**  **if (this.head == null) {**  **this.head = newNode;**  **} else {**  **Node<T> currHead = this.head;**  **while (currHead.getNext() != null) {**  **currHead = currHead.getNext();**  **}**  **currHead.setNext(newNode);**  **this.size++;**  **}**  **return true;**  **}**  @Override  public boolean addAll(Collection<? extends T> elements) {  for (T element : elements) {  this.add(element);  }  return true;  }  **@Override**  **public T get(int index) {// input as 10-- linkedlist.get(10)**  **if (index < 0 || index > this.size + 1) {**  **return null;**  **} else {**  **if (index == 0)**  **return this.head.getVal();// train engine**  **Node<T> currHead = this.head;**  **for (int i = 0; i <= index; i++) {**  **currHead = currHead.getNext();//**  **}**  **return currHead.getVal();**  }  }  //total- 50 elemebnst  //25th remove -- remove(25)--- get(25) -    **@Override**  **public T remove(int index) {**  **if (index < 0 || index > this.size + 1) {**  **return null;**  **} else {**  **T removedValue = this.get(index);**  **if (index == 0) {**  **this.head = this.head.getNext();**  **return removedValue;**  **}**  **Node<T> currHead = this.head;**  **for (int i = 0; i < index - 1; i++) {**  **currHead = currHead.getNext();**  **}**  **removedValue = (T) currHead.getNext().getVal();**  **currHead.setNext(currHead.getNext().getNext());**  **return removedValue;**  **}**  **}**  } |

**Linked is implementing Deque and Deque is extending Queue interface:**

What is the difference between Deque and Queue?

A queue follows the FIFO (First In First Out) rule, where the element that is inserted first is the one that gets removed first. However, a deque does not follow this rule. It allows you to add or remove elements from both the front and the rear ends, making it a versatile data structure. A queue is designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue. Where as Dequeue represents a queue where you can insert and remove elements from both ends of the queue.

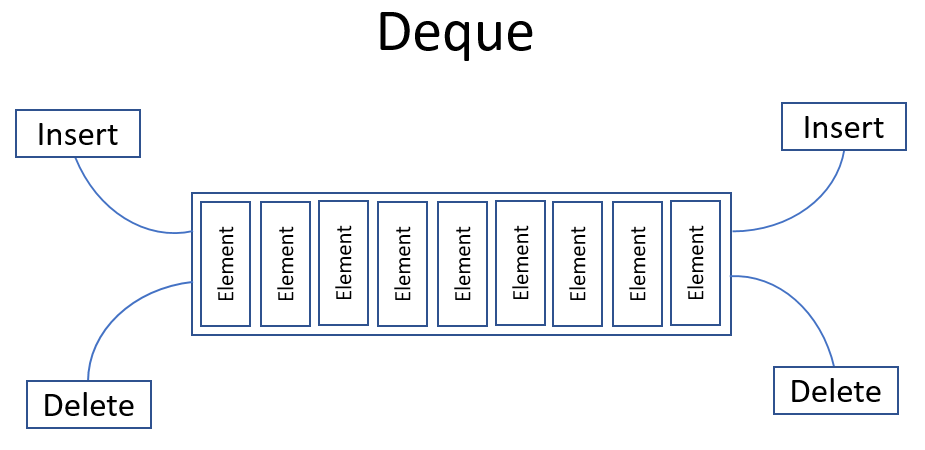
Deque is short for "double ended queue".

**What is the advantages of linked list by implementing deque?**

Reference Link : <https://www.linkedin.com/pulse/understanding-deque-java-arraydeque-linkedlist-diego-cardoso-de-melo-l2w5e/>

In the realm of Java collections, the Deque interface stands out for its unique capability to efficiently add or remove elements from both ends of a double-ended queue. This flexibility makes Deques extremely useful in a variety of programming scenarios, especially those requiring a queue or stack data structure with high efficiency and flexibility. Java provides two main classes that implement the Deque interface: ArrayDeque and LinkedList. Each of these classes has its own set of characteristics and use cases, making them suitable for different programming needs.

**What is a Deque?**



Before diving into the specifics of ArrayDeque and LinkedList, it's essential to understand what a Deque is.

A Deque (double-ended queue) is a linear collection of elements that supports insertion and removal of elements from both ends. This capability distinguishes Deques from standard queues, which typically allow element manipulation at the front and rear only.

**ArrayDeque**

ArrayDeque, as its name suggests, is implemented as a resizable array. This implementation provides a more efficient way to use Deques, especially when frequent insertions and deletions are expected.

**Characteristics of ArrayDeque:**

* **No Capacity Restrictions:** ArrayDeque can grow as needed to support usage.
* **Faster than LinkedList:** Due to its array-based implementation, ArrayDeque is generally faster than LinkedList, especially for queue operations.
* **Not Thread-Safe:** Like most collection classes, ArrayDeque is not synchronized (thread-safe). External synchronization is necessary for concurrent access by multiple threads.
* **Null Elements:** ArrayDeque does not allow the insertion of null elements. Attempting to add null results in a NullPointerException.

**Use Cases for ArrayDeque:**

* **Stack Implementation**: ArrayDeque can be used as a stack, with fast push and pop operations.
* **Queue Implementation:** It serves as an efficient queue for scenarios where elements are frequently added or removed from both ends.

**LinkedList**

* LinkedList implements the Deque interface using a doubly-linked list.
* This implementation provides flexibility and efficient operations at both ends of the list, at the cost of slower random access compared to **ArrayDeque**.

**Characteristics of LinkedList:**

* **Element Insertion and Removal:** Offers constant-time performance for operations at both ends, which can be advantageous for certain algorithmic problems.
* **Implements List and Deque**: Unlike ArrayDeque, LinkedList also implements the List interface, providing additional methods and flexibility.
* **Thread-Safety**: Like ArrayDeque, LinkedList is not thread-safe. Care must be taken when accessing a LinkedList concurrently from multiple threads.
* **Support for Null Elements:** LinkedList allows the insertion of null elements, providing greater flexibility in certain use cases.

**Use Cases for LinkedList:**

* **When List Interface is Needed**: If your scenario benefits from the methods provided by the List interface, LinkedList is the way to go.
* **Insertions/Deletions in the Middle**: For use cases where insertions or deletions occur not only at the ends but also in the middle of the collection, LinkedList's performance can be advantageous.

**Choosing Between ArrayDeque and LinkedList:**

When deciding between ArrayDeque and LinkedList, consider the following:

1. **Performance Needs:** If you need high performance for enqueue and dequeue operations and do not require null elements, ArrayDeque is typically the better choice.
2. **Functionality Requirements:** If you need the additional methods provided by the List interface or need to store null elements, LinkedList offers these capabilities.

**Conclusion**

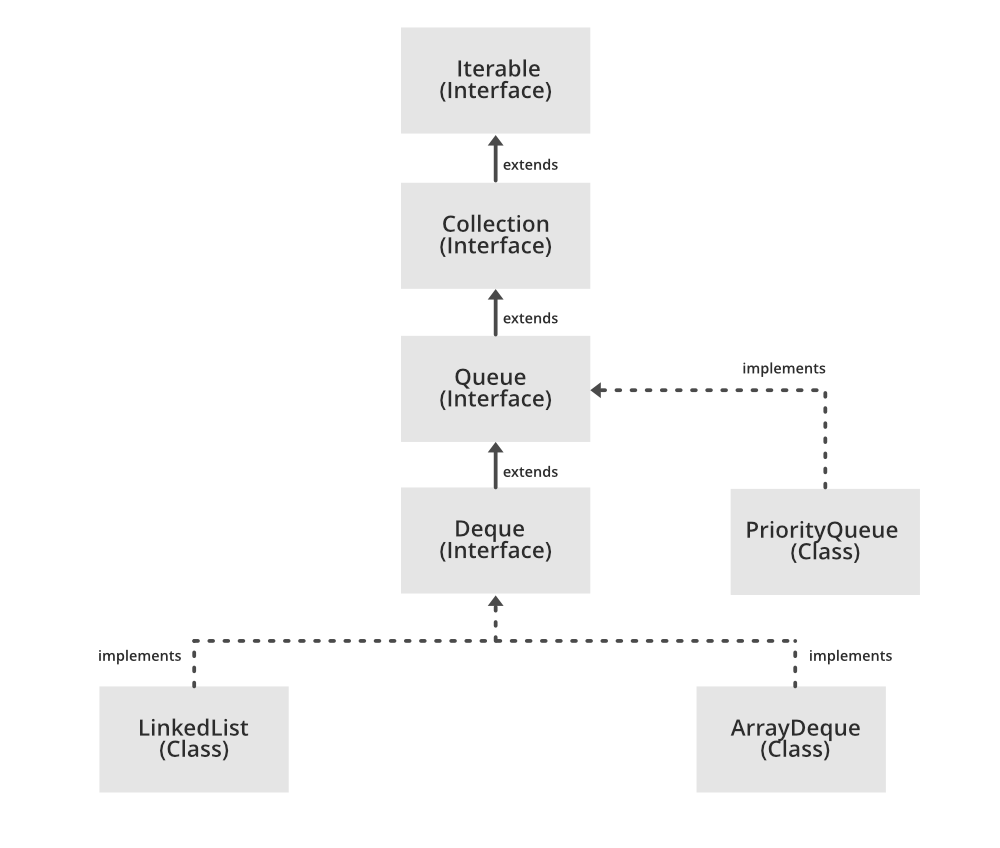
* Both **ArrayDeque and LinkedList** offer powerful features for implementing double-ended queues in Java, each with its strengths and ideal use cases.
* Understanding the differences between these two implementations allows developers to make informed decisions based on their specific requirements, optimizing the performance and efficiency of their Java applications.
* In terms of efficiency, ArrayDeque is more efficient than the LinkedList for add and remove operation at both ends. **The best operation in a LinkedList implementation is removing the current element during the iteration**. LinkedList implementations are not ideal structures to iterate.

**What are the advantages of using a linked list to implement a queue?**

* Using a linked list to implement a queue allows for efficient **enqueue** and **dequeue** operations, and **dynamic memory allocation**.
* A queue is a data structure that follows the First-In-First-Out (FIFO) principle, meaning the first element that was added will be the first one to be removed.
* When implementing a queue using a linked list, the **enqueue** operation (adding an element to the queue) can be performed in constant time**, O(1**), by simply adding a new node at the end of the list.
* Similarly, the dequeue operation (removing an element from the queue) can also be performed in constant time, O(1), by removing the node at the front of the list. This is a significant advantage over array-based implementations, where **enqueue and dequeue** operations can take linear time, O(n), due to the need to shift elements.
* Another advantage of using a linked list is that it allows for dynamic memory allocation. This means that the size of the **queue** can change during the execution of the program, with nodes being created and destroyed as needed.
* This is in contrast to an array-based implementation, where the size of the array must be determined at the time of creation and cannot be changed. **Dynamic memory** allocation can lead to more efficient use of memory, as it avoids the need to reserve space that may not be used.
* Furthermore, linked lists do not require contiguous memory space. Each node in the list is an independent object and can be located anywhere in memory.
* This makes linked lists more flexible and easier to manage in terms of memory allocation compared to arrays, which require a block of contiguous memory.
* In conclusion, implementing a queue using a linked list offers several advantages, including effici**ent enqueue and dequeue operations, dynamic memory allocation,** and **flexibility in memory management**. These features make linked lists a popular choice for implementing **queues** in many applications.

Reference Link: [**https://www.tutorchase.com/answers/ib/computer-science/what-are-the-advantages-of-using-a-linked-list-to-implement-a-queue**](https://www.tutorchase.com/answers/ib/computer-science/what-are-the-advantages-of-using-a-linked-list-to-implement-a-queue)

**Queue and Deque:**



**Queue has multiple implementations: from the API: All Known Implementing Classes:**

|  |
| --- |
| AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue,  DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList,  PriorityBlockingQueue, PriorityQueue, SynchronousQueue |

1. add() — This method in java is used to insert the specified element into the queue and if the task is successful, then the add() function returns true, else if it is not true then it throws an exception.
2. offer() — In the second method it is also used to insert the specified element into the queue and if the task is successful then the offer() method returns true, if not it returns false.
3. element() — In this method, it returns the head of the queue and throws an exception only and only if the queue is empty.
4. peek() — By the word, we can get an idea that this method returns the head of the queue and it returns null if the queue is empty.
5. remove() — This method returns and removes the head of the queue and it throws an exception only if the queue is empty.
6. poll() — This method returns and removes the head of the queue and returns null if the queue is empty.
7. size()- This method is used to Return the size or number of elements which is present in the queue.

Now that we know that a queue is an interface in java so let us see the syntax of how to implement a queue in java:

Since the queue is an inbuilt method in java so firstly we have to write the following line to import a queue in java:

Now as we have imported the queueing package we can create a queue in java using the following line:

|  |
| --- |
| Queue<String> str\_queue = new LinkedList<> ();  Queue<String> priority\_queue = new PriorityQueue<> ();  Queue<Integer> queue1 = new ArrayDeque<> (); |

Let’s us now read about the operations which we can perform in a queue:

|  |
| --- |
| Enqueue, Dequeue,Empty,Full,Peek,Front,Rear |

1. **Enqueue**: This operation is used to add elements to the queue. If in any condition the rear index exceeds the size of the queue then we can say that the queue is full and then it is said to be in an overflow condition. The time complexity to perform this operation is O(1).
2. **Dequeue**: Since the enqueue operation is used to insert elements in the queue so the dequeue operation is used to delete the elements from the queue. If the rear points to -1 that is there are no more elements to delete then we can say that the queue is in the underflow condition. The time complexity to perform this operation is O(1).
3. **Empty**: This operation in the queue is used to check whether the queue is empty or not.
4. **Full**: By the word is full we can understand that this operation is used to check whether the queue is full or not.
5. **Peek**: This operation is used to find the peek element which is the first element of the queue without deleting. It returns the value of the first element without deleting it.
6. **Front**: This operation is used to get the first element from the queue. The time complexity to perform this operation is O(1).
7. **Rear**: This operatio n is used to get the last element from the queue. The time complexity to perform the rear operation in the queue is O(1).

Reference Link: [Java Collections: Queue and Deque Interfaces (stackabuse.com)](https://stackabuse.com/java-collections-queue-and-deque-interfaces/)