**Arrays**

An array is a data structure that holds a fixed number of primitive values or references to a single type of consecutive addresses in memory. Each item in an array is called an element, and each element is accessed by its numerical index. The length of an array is established when the array is created. After creation, its length is fixed.

**Size of Array**

int size = 42; // OK

int[] array = new int[size]; // OK

long size = 23L;

int[] array = new int[size]; // Compile-time error: incompatible types: possible lossy conversion from long to int

int[] array = new int[0]; // Compiles fine.

array[0] = 1; // Throws java.lang.ArrayIndexOutOfBoundsException.

int[] array = new int[-1]; // Throws java.lang.NegativeArraySizeException

**array.length** - public final field

array.length returns the actual size of the array and not the number of array elements which were assigned a value, unlike ArrayList#size() which returns the number of array elements which were assigned a value.

**Creating and Initializing**

- **Primitive Type Arrays**

int[] array1 = new int[] { 1, 2, 3 }; // Create an array with new operator and array // initializer.

int[] array2 = { 1, 2, 3 }; // Shortcut syntax with array initializer.

int[] array3 = new int[3]; // Equivalent to { 0, 0, 0 }

int[] array4 = null; // The array itself is an object, so it can be set as null.

int array5[]; /\* equivalent to \*/ int[] array5;

int a, b[], c[][]; /\* equivalent to \*/ int a; int[] b; int[][] c;

int[] a, b[]; /\* equivalent to \*/ int[] a; int[][] b;

int a, []b, c[][]; /\* Compilation Error, because [] is not part of the type at beginning

of the declaration, rather it is before 'b'. \*/

// The same rules apply when declaring a method that returns an array:

int foo()[] { ... } /\* equivalent to \*/ int[] foo() { ... }

float array[]; /\* and \*/ int foo()[] { ... } /\* are discouraged \*/

float[] array; /\* and \*/ int[] foo() { ... } /\* are encouraged \*/

- **Multi-Dimensional Arrays**

int[][] a = new int[2][3];

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

Jagged Array or Ragged Array :- int[][] a = { {1}, {2,3}, null }; not allowed in JAVA

- **Reference Type Arrays**

String[] array6 = new String[] { "Laurel", "Hardy" }; // Create an array with new operator and array initializer.

String[] array7 = { "Laurel", "Hardy" }; // Shortcut syntax with array

// initializer.

String[] array8 = new String[3]; // { null, null, null }

String[] array9 = null; // null

Object[] array10 = { new Object(), new Object() };

- **Generic Type Arrays**

public class MyGenericClass<T> {

private T[] a;

public MyGenericClass() {

a = new T[5]; // Compile time error: generic array creation

}

}

they can be created using one of the following methods:

1. By creating an Object array, and casting it to the generic type:

a = (T[])new Object[5];

This is the simplest method, but since the underlying array is still of type Object[], this method does not provide type safety, so the array is best used only within the generic class, not exposed publicly.

2. By using Array.newInstance with a class parameter:

public MyGenericClass(Class<T> clazz) {

a = (T[]) Array.newInstance(clazz, 5);

}

Here the class of T has to be explicitly passed to the constructor. The return type of Array.newInstance is always Object. However, the newly created array is in fact of type T[], and can be safely externalized.

- **Separate declaration and initialization of arrays**

int[] array9; // Array declaration - uninitialized

array9 = new int[3]; // Initialize array - { 0, 0, 0 }

array9[0] = 10; // Set index 0 value - { 10, 0, 0 }

array9[1] = 20; // Set index 1 value - { 10, 20, 0 }

array9[2] = 30; // Set index 2 value - { 10, 20, 30 }

- **Arrays may not be re-initialized with array initializer shortcut syntax**

int[] array = new int[] { 1, 2, 3 };

array = new int[] { 4, 5, 6 };

array = { 1, 2, 3, 4 }; // Compile-time error! Can't re-initialize an array via shortcut syntax with array initialize.

**Arrays.fill()**

Arrays.fill(array8, "abc"); // { "abc", "abc", "abc" }

Arrays.fill(array8, 1, 2, "aaa"); // Placing "aaa" from index 1 to 2.

Java SE8

int[] array = new int[5];

Arrays.setAll(array, i -> i); // The array becomes { 0, 1, 2, 3, 4 }.

**Creating List from an Array -** Arrays.asList()

The [Arrays.asList()](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html" \l "asList-T...-) method can be used to return a fixed-size List containing the elements of the given array. The resulting List will be of the same type as the array.

String[] stringArray = new String[] {"foo", "bar", "baz"};

List<String> stringList = Arrays.asList(stringArray);

**Note**: this list is backed by (*a view* of) the array, meaning any changes to the list changes the initial array and vice versa.

**Java SE 5 – 6**

String[] stringArray = new String[] {"foo", "bar", "baz"};

List<String> stringList = new ArrayList<String>(Arrays.asList(stringArray));

**Java SE 7**

A pair of angle brackets <> (empty set of type arguments) can be used, which is called the **Diamond Operator**. The compiler can determine the type arguments from the context. This means the type information can be left out when calling the constructor of ArrayList and it will be inferred automatically during compilation. This is called **Type Inference** which is a part of Java.

String[] stringArray = new String[] {"foo", "bar", "baz"};

List<String> stringList = new ArrayList<>(Arrays.asList(stringArray));

// Using ArrayList.addAll()

String[] stringArray = new String[] {"foo", "bar", "baz"};

ArrayList<String> list = new ArrayList<>();

list.addAll(Arrays.asList(stringArray));

// Using Collections.addAll()

String[] stringArray = new String[] {"foo", "bar", "baz"};

ArrayList<String> list = new ArrayList<>();

Collections.addAll(list, stringArray);

**Java SE 8**

// Using Streams

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

List<Integer> list = Arrays.stream(ints).boxed().collect(Collectors.toList());

String[] stringArray = new String[] {"foo", "bar", "baz"};

List<Object> list = Arrays.stream(stringArray).collect(Collectors.toList());

Notes related to use of Arrays.asList() :-

1. Changes to the List affect the array, and vice-versa. arr[1] = “abc” or listArr.set(2,”def”) will cause change to both the array and the list
2. The resulting List is of fixed-size. That means, adding or removing elements is not supported and will throw an **UnsupportedOperationException**. listArr.add() will throw **UnsupportedOperationException.**
3. A new List can be created by passing the original array-backed List to the constructor of a new List. This creates a new copy of the data, which is not of fixed-size anymore, but also not backed by the original array.

List<String> modifiableList = new ArrayList<>(Arrays.asList("foo", "bar"));

1. Calling <T> List<T> asList(T... a) on a primitive array, such as an int[], will produce a List<int[]> whose only element is the source primitive array instead of the actual elements of the source array. The reason for that behaviour is that primitive types cannot be used in place of generic type parameters, so the entire primitive array replaces the generic type parameter in this case. In order to convert a primitive array to a List, first, convert the primitive array to an array of the corresponding wrapper type (i.e. call Arrays.asList on an Integer[] instead of an int[]).

[**Creating an Array from a Collection**](http://stackoverflow.com/documentation/java/99/arrays/433/creating-an-array-from-a-collection)

**Object[] toArray()**

Set<Integer> set = new HashSet<Integer>();

set.add(0);

set.add(1);

// although a set was of type Integer, a returned array is of type Object

Object[] objectArray = set.toArray();

Object[] toArray() uses vectorized arraycopy, which is much faster than the type checked arraycopy used in T[] toArray(T[] a).

**<T> T[] toArray(T[] a)**

Set<Integer> set = new HashSet<Integer>();

set.add(0);

set.add(1);

/\*\*

\* Note that the array does not need to be created up front with the correct size.

\* Only its type matters.

\*\*/

Integer[] integerArray = set.toArray(new Integer[0]);

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

list.add("android");

list.add("apple");

// toArray() method here uses generics, so the returned array is of type String

String[] stringArray = list.toArray(new String[list.size()]);

T[] toArray(new T[non-zero-size]) needs to zero the array, while T[] toArray(new T[0])does not. Such avoidance makes the latter call faster than the former.

**Java SE 8**

Starting from Java SE 8+, where the concept of [Stream](http://stackoverflow.com/documentation/java/88/streams/383/using-streams) has been introduced, it is possible to use the Stream produced by the collection in order to create a new Array using the [Stream.toArray](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html" \l "toArray-java.util.function.IntFunction-)method.

String[] strings = list.stream().toArray(String[]::new);

**Copying Arrays**

1. **for loop**

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

int[] b = new int[a.length];

for (int i = 0; i < a.length; i++) {

b[i] = a[i];

}

Note that using this option with an Object array instead of primitive array will fill the copy with reference to the original content instead of copy of it.

1. **Object.clone()**

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

int[] b = a.clone(); // [4, 1, 3, 2]

Note that the Object.clone method for an array performs a **shallow copy**, i.e. it returns a reference to a new array which references the **same** elements as the source array.

1. **Arrays.copyOf()**

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

int[] b = Arrays.copyOf(a, a.length); // [4, 1, 3, 2]

Note that **Arrays.copyOf** also provides an overload which allows you to change the type of the array:

Double[] doubles = { 1.0, 2.0, 3.0 };

Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);

1. **System.arraycopy()**

public static void **arraycopy(Object src, int srcPos, Object dest, int destPos, int length)** Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

Below an example of use

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

int[] b = new int[a.length];

System.arraycopy(a, 0, b, 0, a.length); // [4, 1, 3, 2]

1. **Arrays.copyOfRange()**

Mainly used to copy a part of an Array, you can also use it to copy whole array to another as below:

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

int[] b = Arrays.copyOfRange(a, 0, a.length); // [4, 1, 3, 2]

**Sorting Arrays**

1. **Increasing Order**

String[] names = {"John", "Steve", "Shane", "Adam", "Ben"};

System.out.println("String array before sorting : " + Arrays.toString(names));

Arrays.sort(names);

System.out.println("String array after sorting in ascending order : " + Arrays.toString(names));

1. **Decreasing Order**

Arrays.sort(names, 0, names.length, Collections.reverseOrder());

System.out.println("String array after sorting in descending order : " + Arrays.toString(names));

1. **Sorting an Object Array**

In order to sort an object array, all elements must implement either Comparable or Comparator interface to define the order of the sorting.

Furthermore, they must be mutually comparable as well, for example e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array. Alternatively you can sort an Object array on custom order using sort(T[], Comparator) method as shown in following example.

Course[] courses = new Course[4];

courses[0] = new Course(101, "Java", 200);

courses[1] = new Course(201, "Ruby", 300);

courses[2] = new Course(301, "Python", 400);

courses[3] = new Course(401, "Scala", 500);

System.out.println("Object array before sorting : " + Arrays.toString(courses));

Arrays.sort(courses);

System.out.println("Object array after sorting in natural order : " + Arrays.toString(courses));

Arrays.sort(courses, new Course.PriceComparator());

System.out.println("Object array after sorting by price : " + Arrays.toString(courses));

Arrays.sort(courses, new Course.NameComparator());

System.out.println("Object array after sorting by name : " + Arrays.toString(courses));

**Array vs List**

|  |  |  |
| --- | --- | --- |
|  | **Array** | **List** |
| **Size** | Fixed length. Cannot change the size after creation | Dynamic in size. Capacity grows as the elements are added |
| **Content** | Can contain primitive data types and objects | Objects only. No primitive data types |
| **Dimension** | Can be multi-dimensional | Always single-dimensional |
| **Type-Safety** | Typesafe, meaning that array will contain objects of specific class or primitives of specific data type | Not type safe by default. Genericscan be used to make a List type safe |
| **Insertion/Deletion** | Shifting existing elements may be needed if action is performed not at the end of the array | Easy insertion/deletion methods provided |

