Chapter 40: Arrays

Parameter	Details
`ArrayType`	Type of the array. This can be primitive (**`int`**, **`long`**, **`byte`**) or Objects (`String`, `MyObject`, etc).
index	Index refers to the position of a certain Object in an array.
length	Every array, when being created, needs a set length specified. This is either done when creating an empty array (**`new int`**`[3]`) or implied when specifying values (`{1, 2, 3}`).

Arrays allow for the storage and retrieval of an arbitrary quantity of values. They are analogous to vectors in mathematics. Arrays of arrays are analogous to matrices, and act as multidimensional arrays. Arrays can store any data of any type: primitives such as **`int`** or reference types such as `Object`.

Section 40.1: Creating and Initializing Arrays

Basic cases

```
int[]   numbers1 = new int[3];                // Array for 3 int values, default value is 0
int[]   numbers2 = { 1, 2, 3 };               // Array literal of 3 int values
int[]   numbers3 = new int[] { 1, 2, 3 };     // Array of 3 int values initialized
int[][] numbers4 = { { 1, 2 }, { 3, 4, 5 } }; // Jagged array literal
int[][] numbers5 = new int[5][];              // Jagged array, one dimension 5 long
int[][] numbers6 = new int[5][4];             // Multidimensional array: 5x4
```

Arrays may be created using any primitive or reference type.

```
float[]  boats = new float[5];          // Array of five 32-bit floating point numbers.
double[] header = new double[] { 4.56, 332.267, 7.0, 0.3367, 10.0 };
                                        // Array of five 64-bit floating point numbers.
String[] theory = new String[] { "a", "b", "c" };
                                        // Array of three strings (reference type).
Object[] dArt = new Object[] { new Object(), "We love Stack Overflow.", new Integer(3) };
                                        // Array of three Objects (reference type).
```

For the last example, note that subtypes of the declared array type are allowed in the array.

Arrays for user defined types can also be built similar to primitive types

```
UserDefinedClass[] udType = new UserDefinedClass[5];
```

Arrays, Collections, and Streams

Version ≥ Java SE 1.2

```
// Parameters require objects, not primitives

// Auto-boxing happening for int 127 here
Integer[]       initial        = { 127, Integer.valueOf( 42 ) };
List<Integer>   toList         = Arrays.asList( initial );  // Fixed size!

// Note: Works with all collections
Integer[]       fromCollection = toList.toArray( new Integer[toList.size()] );

//Java doesn't allow you to create an array of a parameterized type
List<String>[]  list = new ArrayList<String>[2];  // Compilation error!
```

Version ≥ Java SE 8

```
// Streams - JDK 8+
Stream<Integer> toStream       = Arrays.stream( initial );
Integer[]       fromStream     = toStream.toArray( Integer[]::new );
```

Intro

An *array* is a data structure that holds a fixed number of primitive values **or** references to object instances.

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:

```
int size = 42;
int[] array = new int[size];
```

The size of an array is fixed at runtime when initialized. It cannot be changed after initialization. If the size must be mutable at runtime, a `Collection` class such as `ArrayList` should be used instead. `ArrayList` stores elements in an array and supports resizing by allocating a new array and copying elements from the old array.

If the array is of a primitive type, i.e.

```
int[] array1 = { 1,2,3 };
int[] array2 = new int[10];
```

the values are stored in the array itself. In the absence of an initializer (as in `array2` above), the default value assigned to each element is `0` (zero).

If the array type is an object reference, as in

```
SomeClassOrInterface[] array = new SomeClassOrInterface[10];
```

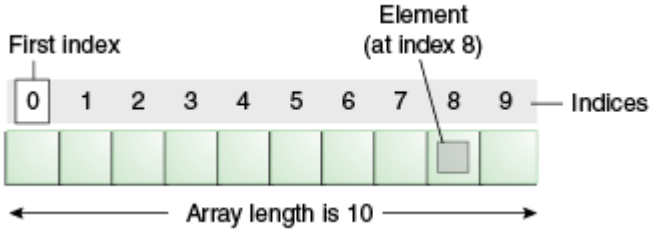
then the array contains *references* to objects of type `SomeClassOrInterface`. Those references can refer to an instance of `SomeClassOrInterface` *or any subclass (for classes) or implementing class (for interfaces) of* `SomeClassOrInterface`. If the array declaration has no initializer then the default value of **`null`** is assigned to each element.

Because all arrays are **`int`**-indexed, the size of an array must be specified by an **`int`**. The size of the array cannot be specified as a **`long`**:

```
long size = 23L;
int[] array = new int[size]; // Compile-time error:
                             // incompatible types: possible lossy conversion from
                             // long to int
```

Arrays use a **zero-based index** system, which means indexing starts at `0` and ends at `length - 1`.

For example, the following image represents an array with size `10`. Here, the first element is at index `0` and the last element is at index `9`, instead of the first element being at index `1` and the last element at index `10` (see figure below).



Accesses to elements of arrays are done in **constant time**. That means accessing to the first element of the array has the same cost (in time) of accessing the second element, the third element and so on.

Java offers several ways of defining and initializing arrays, including **literal** and **constructor** notations. When

declaring arrays using the `new Type[length]` constructor, each element will be initialized with the following default values:

- `0` for primitive numerical types: **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, and **`double`**.
- `'\u0000'` (null character) for the **`char`** type.
- **`false`** for the **`boolean`** type.
- **`null`** for reference types.

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.
```

When declaring an array, [ ] will appear as part of the type at the beginning of the declaration (after the type name), or as part of the declarator for a particular variable (after variable name), or both:

```
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() { ... }
```

In the following example, both declarations are correct and can compile and run without any problems. However, both the Java Coding Convention and the Google Java Style Guide discourage the form with brackets after the variable name—the brackets identify the array type and should appear with the type designation. The same should be used for method return signatures.

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

The discouraged type is meant to accommodate transitioning C users, who are familiar with the syntax for C which has the brackets after the variable name.

In Java, it is possible to have arrays of size `0`:

```
int[] array = new int[0]; // Compiles and runs fine.
int[] array2 = {};        // Equivalent syntax.
```

However, since it's an empty array, no elements can be read from it or assigned to it:

```
array[0] = 1;     // Throws java.lang.ArrayIndexOutOfBoundsException.
int i = array2[0]; // Also throws ArrayIndexOutOfBoundsException.
```

Such empty arrays are typically useful as return values, so that the calling code only has to worry about dealing with an array, rather than a potential **`null`** value that may lead to a `NullPointerException`.

The length of an array must be a non-negative integer:

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

The array size can be determined using a public final field called `length`:

```
System.out.println(array.length); // Prints 0 in this case.
```

Note: `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 multi-dimensional arrays

The simplest way to create a multi-dimensional array is as follows:

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

It will create two three-length `int` arrays—`a[0]` and `a[1]`. This is very similar to the classical, C-style initialization of rectangular multi-dimensional arrays.

You can create and initialize at the same time:

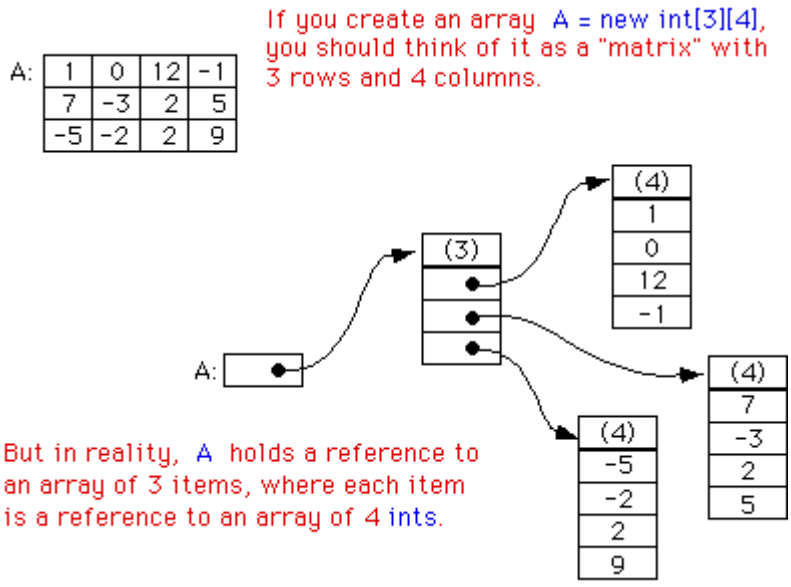
```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

Unlike C, where only rectangular multi-dimensional arrays are supported, inner arrays do not need to be of the same length, or even defined:

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

Here, `a[0]` is a one-length `int` array, whereas `a[1]` is a two-length `int` array and `a[2]` is `null`. Arrays like this are called jagged arrays or ragged arrays, that is, they are arrays of arrays. Multi-dimensional arrays in Java are implemented as arrays of arrays, i.e. `array[i][j][k]` is equivalent to `((array[i])[j])[k]`. Unlike C#, the syntax `array[i,j]` is not supported in Java.

Multidimensional array representation in Java



Source - Live on Ideone

Creating and initializing 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
```

Live on Ideone

In addition to the `String` literals and primitives shown above, the shortcut syntax for array initialization also works with canonical `Object` types:

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

Because arrays are covariant, a reference type array can be initialized as an array of a subclass, although an `ArrayStoreException` will be thrown if you try to set an element to something other than a `String`:

```
Object[] array11 = new String[] { "foo", "bar", "baz" };
array11[1] = "qux"; // fine
array11[1] = new StringBuilder(); // throws ArrayStoreException
```

The shortcut syntax cannot be used for this because the shortcut syntax would have an implicit type of `Object[]`.

An array can be initialized with zero elements by using `String[] emptyArray = new String[0]`. For example, an array with zero length like this is used for Creating an `Array` from a `Collection` when the method needs the runtime type of an object.

In both primitive and reference types, an empty array initialization (for example `String[] array8 = new String[3]`) will initialize the array with the default value for each data type.

Creating and initializing generic type arrays

In generic classes, arrays of generic types **cannot** be initialized like this due to type erasure:

```
public class MyGenericClass<T> {
    private T[] a;

    public MyGenericClass() {
        a = new T[5]; // Compile time error: generic array creation
    }
}
```

Instead, they can be created using one of the following methods: (note that these will generate unchecked warnings)

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. Therefore, this method of creating an 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, this method is safer because the newly created array is always of type `T[]`, and therefore can be safely externalized.

Filling an array after initialization

Version ≥ Java SE 1.2

`Arrays.fill()` can be used to fill an array with **the same value** after initialization:

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

Live on Ideone

`fill()` can also assign a value to each element of the specified range of the array:

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

Live on Ideone

Version ≥ Java SE 8

Since Java version 8, the method `setAll`, and its `Concurrent` equivalent `parallelSetAll`, can be used to set every element of an array to generated values. These methods are passed a generator function which accepts an index and returns the desired value for that position.

The following example creates an integer array and sets all of its elements to their respective index value:

```
int[] array = new int[5];
Arrays.setAll(array, i -> i); // The array becomes { 0, 1, 2, 3, 4 }.
```

Live on Ideone

Separate declaration and initialization of arrays

The value of an index for an array element must be a whole number (0, 1, 2, 3, 4, ...) and less than the length of the array (indexes are zero-based). Otherwise, an ArrayIndexOutOfBoundsException will be thrown:

```
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

It is not possible to re-initialize an array via a shortcut syntax with an array initializer since an array initializer can only be specified in a field declaration or local variable declaration, or as a part of an array creation expression.

However, it is possible to create a new array and assign it to the variable being used to reference the old array. While this results in the array referenced by that variable being re-initialized, the variable contents are a completely new array. To do this, the **new** operator can be used with an array initializer and assigned to the array variable:

```
// First initialization of array
int[] array = new int[] { 1, 2, 3 };
```

```
// Prints "1 2 3 ".
for (int i : array) {
    System.out.print(i + " ");
}

// Re-initializes array to a new int[] array.
array = new int[] { 4, 5, 6 };

// Prints "4 5 6 ".
for (int i : array) {
    System.out.print(i + " ");
}

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

Live on Ideone

Section 40.2: Creating a List from an Array

The `Arrays.asList()` 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 parameter type as the base type of the array.

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = Arrays.asList(stringArray);
```

Note: This list is backed by (*a view* of) the original array, meaning that any changes to the list will change the array and vice versa. However, changes to the list that would change its size (and hence the array length) will throw an exception.

To create a copy of the list, use the constructor of `java.util.ArrayList` taking a `Collection` as an argument:

Version ≥ Java SE 5

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<String>(Arrays.asList(stringArray));
```

Version ≥ Java SE 7

In Java SE 7 and later, a pair of angle brackets `<>` (empty set of type arguments) can be used, which is called the Diamond. 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 Generics.

```
// Using Arrays.asList()

String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<>(Arrays.asList(stringArray));

// Using ArrayList.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
list.addAll(Arrays.asList(stringArray));

// Using Collections.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, stringArray);
```

A point worth noting about the Diamond is that it cannot be used with Anonymous Classes.

Version ≥ Java SE 8

```
// Using Streams

int[] ints = {1, 2, 3};
List<Integer> list = Arrays.stream(ints).boxed().collect(Collectors.toList());

String[] stringArray = {"foo", "bar", "baz"};
List<Object> list = Arrays.stream(stringArray).collect(Collectors.toList());
```

Important notes related to using Arrays.asList() method

- This method returns `List`, which is an instance of `Arrays$ArrayList`(static inner class of `Arrays`) and not `java.util.ArrayList`. The resulting `List` is of fixed-size. That means, adding or removing elements is not supported and will throw an `UnsupportedOperationException`:

  ```
  stringList.add("something"); // throws java.lang.UnsupportedOperationException
  ```

- A new `List` can be created by passing an array-backed `List` to the constructor of a new `List`. This creates a new copy of the data, which has changeable size and that is not backed by the original array:

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

- 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 this behavior 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 of all, convert the primitive array to an array of the corresponding wrapper type (i.e. call `Arrays.asList` on an `Integer[]` instead of an `int[]`).

  Therefore, this will print `false`:

  ```
  int[] arr = {1, 2, 3};      // primitive array of int
  System.out.println(Arrays.asList(arr).contains(1));
  ```

  View Demo

  On the other hand, this will print `true`:

  ```
  Integer[] arr = {1, 2, 3};  // object array of Integer (wrapper for int)
  System.out.println(Arrays.asList(arr).contains(1));
  ```

  View Demo

  This will also print `true`, because the array will be interpreted as an `Integer[]`):

  ```
  System.out.println(Arrays.asList(1,2,3).contains(1));
  ```

  View Demo

Section 40.3: Creating an Array from a Collection

Two methods in `java.util.Collection` create an array from a collection:

- `Object[] toArray()`
- `<T> T[] toArray(T[] a)`

`Object[] toArray()` can be used as follows:

Version ≥ Java SE 5

```
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// although set is a Set<String>, toArray() returns an Object[] not a String[]
Object[] objectArray = set.toArray();
```

`<T> T[] toArray(T[] a)` can be used as follows:

Version ≥ Java SE 5

```
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// The array does not need to be created up front with the correct size.
// Only the array type matters. (If the size is wrong, a new array will
// be created with the same type.)
String[] stringArray = set.toArray(new String[0]);

// If you supply an array of the same size as collection or bigger, it
// will be populated with collection values and returned (new array
// won't be allocated)
String[] stringArray2 = set.toArray(new String[set.size()]);
```

The difference between them is more than just having untyped vs typed results. Their performance can differ as well (for details please read this performance analysis section):

- `Object[] toArray()` uses vectorized `arraycopy`, which is much faster than the type-checked `arraycopy` used in `T[] toArray(T[] a)`.
- `T[] toArray(new T[non-zero-size])` needs to zero-out the array at runtime, while `T[] toArray(new T[0])` does not. Such avoidance makes the latter call faster than the former. Detailed analysis here : Arrays of Wisdom of the Ancients.

Version ≥ Java SE 8

Starting from Java SE 8+, where the concept of `Stream` 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` method.

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

Examples taken from two answers (1, 2) to Converting 'ArrayList to 'String[]' in Java on Stack Overflow.

Section 40.4: Multidimensional and Jagged Arrays

It is possible to define an array with more than one dimension. Instead of being accessed by providing a single index, a multidimensional array is accessed by specifying an index for each dimension.

The declaration of multidimensional array can be done by adding [ ] for each dimension to a regular array declaration. For instance, to make a 2-dimensional `int` array, add another set of brackets to the declaration, such as `int[][]`. This continues for 3-dimensional arrays (`int[][][]`) and so forth.

To define a 2-dimensional array with three rows and three columns:

```
int rows = 3;
int columns = 3;
int[][] table = new int[rows][columns];
```

The array can be indexed and assign values to it with this construct. Note that the unassigned values are the default values for the type of an array, in this case `0` for `int`.

```
table[0][0] = 0;
table[0][1] = 1;
table[0][2] = 2;
```

It is also possible to instantiate a dimension at a time, and even make non-rectangular arrays. These are more commonly referred to as **jagged arrays**.

```
int[][] nonRect = new int[4][];
```

It is important to note that although it is possible to define any dimension of jagged array, it's preceding level ***must*** be defined.

```
// valid
String[][] employeeGraph = new String[30][];

// invalid
int[][] unshapenMatrix = new int[][10];

// also invalid
int[][][] misshapenGrid = new int[100][][10];
```

How Multidimensional Arrays are represented in Java

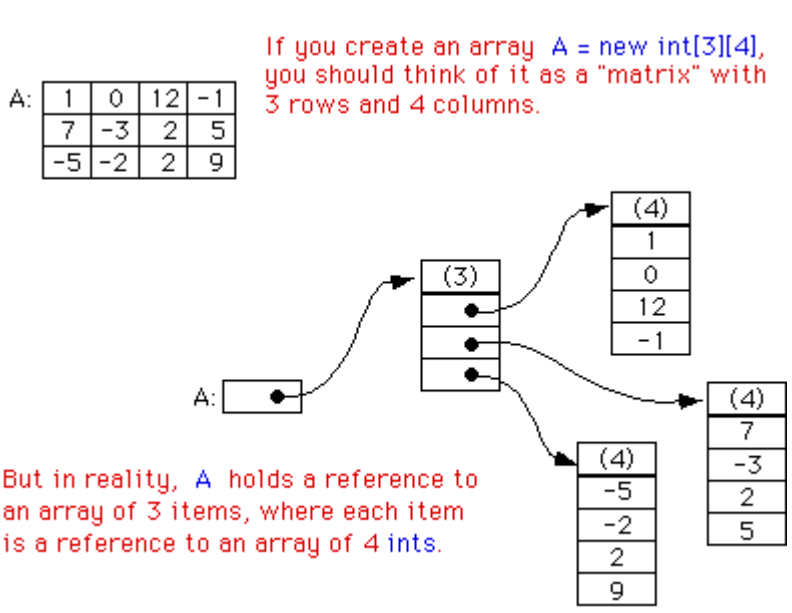


Image source: http://math.hws.edu/eck/cs124/javanotes3/c8/s5.html

Jagged array literal intialization

Multidimensional arrays and jagged arrays can also be initialized with a literal expression. The following declares and populates a 2x3 `int` array:

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

Note: Jagged subarrays may also be `null`. For instance, the following code declares and populates a two dimensional `int` array whose first subarray is `null`, second subarray is of zero length, third subarray is of one length and the last subarray is a two length array:

```
int[][] table = {
    null,
    {},
    {1},
    {1,2}
};
```

For multidimensional array it is possible to extract arrays of lower-level dimension by their indices:

```
int[][][] arr = new int[3][3][3];
int[][] arr1 = arr[0]; // get first 3x3-dimensional array from arr
int[] arr2 = arr1[0]; // get first 3-dimensional array from arr1
int[] arr3 = arr[0]; // error: cannot convert from int[][] to int[]
```

Section 40.5: ArrayIndexOutOfBoundsException

The `ArrayIndexOutOfBoundsException` is thrown when a non-existing index of an array is being accessed.

Arrays are zero-based indexed, so the index of the first element is `0` and the index of the last element is the array capacity minus 1 (i.e. `array.length - 1`).

Therefore, any request for an array element by the index `i` has to satisfy the condition `0 <= i < array.length`, otherwise the `ArrayIndexOutOfBoundsException` will be thrown.

The following code is a simple example where an `ArrayIndexOutOfBoundsException` is thrown.

```
String[] people = new String[] { "Carol", "Andy" };

// An array will be created:
// people[0]: "Carol"
// people[1]: "Andy"

// Notice: no item on index 2. Trying to access it triggers the exception:
System.out.println(people[2]);  // throws an ArrayIndexOutOfBoundsException.
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
at your.package.path.method(YourClass.java:15)
```

Note that the illegal index that is being accessed is also included in the exception (2 in the example); this information could

be useful to find the cause of the exception.

To avoid this, simply check that the index is within the limits of the array:

```
int index = 2;
if (index >= 0 && index < people.length) {
    System.out.println(people[index]);
}
```

Section 40.6: Array Covariance

Object arrays are covariant, which means that just as `Integer` is a subclass of `Number`, `Integer[]` is a subclass of `Number[]`. This may seem intuitive, but can result in surprising behavior:

```
Integer[] integerArray = {1, 2, 3};
Number[] numberArray = integerArray;  // valid
Number firstElement = numberArray[0]; // valid
numberArray[0] = 4L;                  // throws ArrayStoreException at runtime
```

Although `Integer[]` is a subclass of `Number[]`, it can only hold `Integer`s, and trying to assign a `Long` element throws a runtime exception.

Note that this behavior is unique to arrays, and can be avoided by using a generic `List` instead:

```
List<Integer> integerList = Arrays.asList(1, 2, 3);
//List<Number> numberList = integerList;  // compile error
List<? extends Number> numberList = integerList;
Number firstElement = numberList.get(0);
//numberList.set(0, 4L);                  // compile error
```

It's not necessary for all of the array elements to share the same type, as long as they are a subclass of the array's type:

```
interface I {}

class A implements I {}
class B implements I {}
class C implements I {}

I[] array10 = new I[] { new A(), new B(), new C() }; // Create an array with new
                                                     // operator and array initializer.

I[] array11 = { new A(), new B(), new C() };         // Shortcut syntax with array
                                                     // initializer.

I[] array12 = new I[3];                              // { null, null, null }

I[] array13 = new A[] { new A(), new A() };          // Works because A implements I.

Object[] array14 = new Object[] { "Hello, World!", 3.14159, 42 }; // Create an array with
                                                     // new operator and array initializer.

Object[] array15 = { new A(), 64, "My String" };     // Shortcut syntax
                                                     // with array initializer.
```

Section 40.7: Arrays to Stream

Version ≥ Java SE 8

Converting an array of objects to `Stream`:

```
String[] arr = new String[] {"str1", "str2", "str3"};
Stream<String> stream = Arrays.stream(arr);
```

Converting an array of primitives to `Stream` using `Arrays.stream()` will transform the array to a primitive specialization of Stream:

```
int[] intArr = {1, 2, 3};
IntStream intStream = Arrays.stream(intArr);
```

You can also limit the `Stream` to a range of elements in the array. The start index is inclusive and the end index is exclusive:

```
int[] values = {1, 2, 3, 4};
IntStream intStream = Arrays.stream(values, 2, 4);
```

A method similar to `Arrays.stream()` appears in the `Stream` class: `Stream.of()`. The difference is that `Stream.of()` uses a varargs parameter, so you can write something like:

```
Stream<Integer> intStream = Stream.of(1, 2, 3);
Stream<String> stringStream = Stream.of("1", "2", "3");
Stream<Double> doubleStream = Stream.of(new Double[]{1.0, 2.0});
```

Section 40.8: Iterating over arrays

You can iterate over arrays either by using enhanced for loop (aka foreach) or by using array indices:

```
int[] array = new int[10];

// using indices: read and write
for (int i = 0; i < array.length; i++) {
    array[i] = i;
}
```

Version ≥ Java SE 5

```
// extended for: read only
for (int e : array) {
    System.out.println(e);
}
```

It is worth noting here that there is no direct way to use an Iterator on an Array, but through the Arrays library it can be easily converted to a list to obtain an `Iterable` object.

For boxed arrays use Arrays.asList:

```
Integer[] boxed = {1, 2, 3};
Iterable<Integer> boxedIt = Arrays.asList(boxed); // list-backed iterable
Iterator<Integer> fromBoxed1 = boxedIt.iterator();
```

For primitive arrays (using java 8) use streams (specifically in this example - Arrays.stream -> IntStream):

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

```
IntStream primitiveStream = Arrays.stream(primitives); // list-backed iterable
PrimitiveIterator.OfInt fromPrimitive1 = primitiveStream.iterator();
```

If you can't use streams (no java 8), you can choose to use google's guava library:

```
Iterable<Integer> fromPrimitive2 = Ints.asList(primitives);
```

In two-dimensional arrays or more, both techniques can be used in a slightly more complex fashion.

Example:

```
int[][] array = new int[10][10];

for (int indexOuter = 0; indexOuter < array.length; indexOuter++) {
    for (int indexInner = 0; indexInner < array[indexOuter].length; indexInner++) {
        array[indexOuter][indexInner] = indexOuter + indexInner;
    }
}
```

Version ≥ Java SE 5

```
for (int[] numbers : array) {
    for (int value : numbers) {
        System.out.println(value);
    }
}
```

It is impossible to set an Array to any non-uniform value without using an index based loop.

Of course you can also use **`while`** or `do-`**`while`** loops when iterating using indices.

One note of caution: when using array indices, make sure the index is between `0` and `array.length - 1` (both inclusive). Don't make hard coded assumptions on the array length otherwise you might break your code if the array length changes but your hard coded values don't.

Example:

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

public void incrementNumbers() {
    // DO THIS :
    for (int i = 0; i < numbers.length; i++) {
        numbers[i] += 1; //or this: numbers[i] = numbers[i] + 1; or numbers[i]++;
    }

    // DON'T DO THIS :
    for (int i = 0; i < 4; i++) {
        numbers[i] += 1;
    }
}
```

It's also best if you don't use fancy calculations to get the index but use the index to iterate and if you need different values calculate those.

Example:

```
public void fillArrayWithDoubleIndex(int[] array) {
    // DO THIS :
    for (int i = 0; i < array.length; i++) {
        array[i] = i * 2;
```

```
    }

    // DON'T DO THIS :
    int doubleLength = array.length * 2;
    for (int i = 0; i < doubleLength; i += 2) {
        array[i / 2] = i;
    }
}
```

Accessing Arrays in reverse order

```
int[] array = {0, 1, 1, 2, 3, 5, 8, 13};
for (int i = array.length - 1; i >= 0; i--) {
   System.out.println(array[i]);
}
```

Using temporary Arrays to reduce code repetition

Iterating over a temporary array instead of repeating code can make your code cleaner. It can be used where the same operation is performed on multiple variables.

```
// we want to print out all of these
String name = "Margaret";
int eyeCount = 16;
double height = 50.2;
int legs = 9;
int arms = 5;


// copy-paste approach:
System.out.println(name);
System.out.println(eyeCount);
System.out.println(height);
System.out.println(legs);
System.out.println(arms);


// temporary array approach:
for(Object attribute : new Object[]{name, eyeCount, height, legs, arms})
    System.out.println(attribute);

// using only numbers
for(double number : new double[]{eyeCount, legs, arms, height})
    System.out.println(Math.sqrt(number));
```

Keep in mind that this code should not be used in performance-critical sections, as an array is created every time the loop is entered, and that primitive variables will be copied into the array and thus cannot be modified.

Section 40.9: Arrays to a String

Version ≥ Java SE 5

Since Java 1.5 you can get a `String` representation of the contents of the specified array without iterating over its every element. Just use `Arrays.toString(Object[])` or `Arrays.deepToString(Object[])` for multidimentional arrays:

```
int[] arr = {1, 2, 3, 4, 5};
System.out.println(Arrays.toString(arr));      // [1, 2, 3, 4, 5]
```

```
int[][] arr = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println(Arrays.deepToString(arr));  // [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`Arrays.toString()` method uses `Object.toString()` method to produce `String` values of every item in the array, beside primitive type array, it can be used for all type of arrays. For instance:

```
public class Cat { /* implicitly extends Object */
    @Override
    public String toString() {
      return "CAT!";
    }
}

Cat[] arr = { new Cat(), new Cat() };
System.out.println(Arrays.toString(arr));        // [CAT!, CAT!]
```

If no overridden `toString()` exists for the class, then the inherited `toString()` from `Object` will be used. Usually the output is then not very useful, for example:

```
public class Dog {
    /* implicitly extends Object */
}

Dog[] arr = { new Dog() };
System.out.println(Arrays.toString(arr));        // [Dog@17ed40e0]
```

Section 40.10: Sorting arrays

Sorting arrays can be easily done with the Arrays api.

```
import java.util.Arrays;

// creating an array with integers
int[] array = {7, 4, 2, 1, 19};
// this is the sorting part just one function ready to be used
Arrays.sort(array);
// prints [1, 2, 4, 7, 19]
System.out.println(Arrays.toString(array));
```

Sorting String arrays:

`String` is not a numeric data, it defines it's own order which is called lexicographic order, also known as alphabetic order. When you sort an array of String using `sort()` method, it sorts array into natural order defined by Comparable interface, as shown below :

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));
```

Output:

```
String array before sorting : [John, Steve, Shane, Adam, Ben]
String array after sorting in ascending order : [Adam, Ben, John, Shane, Steve]
```

Decreasing Order

```
Arrays.sort(names, 0, names.length, Collections.reverseOrder());
System.out.println("String array after sorting in descending order : " + Arrays.toString(names));
```

Output:

```
String array after sorting in descending order : [Steve, Shane, John, Ben, Adam]
```

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.

We can use either `sort(Object[])` method to sort an object array on its natural order, but you must ensure that all elements in the array must implement `Comparable`.

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.

```
// How to Sort Object Array in Java using Comparator and Comparable
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));
```

Output:

```
Object array before sorting : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 , #401 Scala@500 ]
Object array after sorting in natural order : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 ,
#401 Scala@500 ]
Object array after sorting by price : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 , #401
Scala@500 ]
Object array after sorting by name : [#101 Java@200 , #301 Python@400 , #201 Ruby@300 , #401
Scala@500 ]
```

Section 40.11: Getting the Length of an Array

Arrays are objects which provide space to store up to its size of elements of specified type. An array's size can not be modified after the array is created.

```
int[] arr1 = new int[0];
int[] arr2 = new int[2];
int[] arr3 = new int[]{1, 2, 3, 4};
int[] arr4 = {1, 2, 3, 4, 5, 6, 7};

int len1 = arr1.length; // 0
int len2 = arr2.length; // 2
int len3 = arr3.length; // 4
int len4 = arr4.length; // 7
```

The `length` field in an array stores the size of an array. It is a **`final`** field and cannot be modified.

This code shows the difference between the `length` of an array and amount of objects an array stores.

```
public static void main(String[] args) {
    Integer arr[] = new Integer[] {1,2,3,null,5,null,7,null,null,null,11,null,13};

    int arrayLength = arr.length;
    int nonEmptyElementsCount = 0;

    for (int i=0; i<arrayLength; i++) {
        Integer arrElt = arr[i];
        if (arrElt != null) {
            nonEmptyElementsCount++;
        }
    }

    System.out.println("Array 'arr' has a length of "+arrayLength+"\n"
                           + "and it contains "+nonEmptyElementsCount+" non-empty values");
}
```

Result:

```
Array 'arr' has a length of 13
and it contains 7 non-empty values
```

Section 40.12: Finding an element in an array

There are many ways find the location of a value in an array. The following example snippets all assume that the array is one of the following:

```
String[] strings = new String[] { "A", "B", "C" };
int[] ints = new int[] { 1, 2, 3, 4 };
```

In addition, each one sets `index` or `index2` to either the index of required element, or -1 if the element is not present.

Using `Arrays.binarySearch` (for sorted arrays only)

```
int index = Arrays.binarySearch(strings, "A");
int index2 = Arrays.binarySearch(ints, 1);
```

Using a `Arrays.asList` (for non-primitive arrays only)

```
int index = Arrays.asList(strings).indexOf("A");
int index2 = Arrays.asList(ints).indexOf(1);  // compilation error
```

Using a `Stream`

Version ≥ Java SE 8

```
int index = IntStream.range(0, strings.length)
                     .filter(i -> "A".equals(strings[i]))
                     .findFirst()
                     .orElse(-1); // If not present, gives us -1.
// Similar for an array of primitives
```

Linear search using a loop

```
int index = -1;
for (int i = 0; i < array.length; i++) {
    if ("A".equals(array[i])) {
        index = i;
        break;
    }
}
// Similar for an array of primitives
```

Linear search using 3rd-party libraries such as org.apache.commons

```
int index = org.apache.commons.lang3.ArrayUtils.contains(strings, "A");
int index2 = org.apache.commons.lang3.ArrayUtils.contains(ints, 1);
```

Note: Using a direct linear search is more efficient than wrapping in a list.

Testing if an array contains an element

The examples above can be adapted to test if the array contains an element by simply testing to see if the index computed is greater or equal to zero.

Alternatively, there are also some more concise variations:

```
boolean isPresent = Arrays.asList(strings).contains("A");
```

Version ≥ Java SE 8

```
boolean isPresent = Stream<String>.of(strings).anyMatch(x -> "A".equals(x));
```

```
boolean isPresent = false;
for (String s : strings) {
    if ("A".equals(s)) {
        isPresent = true;
        break;
    }
}

boolean isPresent = org.apache.commons.lang3.ArrayUtils.contains(ints, 4);
```

Section 40.13: How do you change the size of an array?

The simple answer is that you cannot do this. Once an array has been created, its size cannot be changed. Instead, an array can only be "resized" by creating a new array with the appropriate size and copying the elements from the existing array to the new one.

```
String[] listOfCities = new String[3];   // array created with size 3.
listOfCities[0] = "New York";
```

```
listOfCities[1] = "London";
listOfCities[2] = "Berlin";
```

Suppose (for example) that a new element needs to be added to the `listOfCities` array defined as above. To do this, you will need to:

1. create a new array with size 4,
2. copy the existing 3 elements of the old array to the new array at offsets 0, 1 and 2, and
3. add the new element to the new array at offset 3.

There are various ways to do the above. Prior to Java 6, the most concise way was:

```
String[] newArray = new String[listOfCities.length + 1];
System.arraycopy(listOfCities, 0, newArray, 0, listOfCities.length);
newArray[listOfCities.length] = "Sydney";
```

From Java 6 onwards, the `Arrays.copyOf` and `Arrays.copyOfRange` methods can do this more simply:

```
String[] newArray = Arrays.copyOf(listOfCities, listOfCities.length + 1);
newArray[listOfCities.length] = "Sydney";
```

For other ways to copy an array, refer to the following example. Bear in mind that you need an array copy with a different length to the original when resizing.

- Copying arrays

A better alternatives to array resizing

There two major drawbacks with resizing an array as described above:

- It is inefficient. Making an array bigger (or smaller) involves copying many or all of the existing array elements, and allocating a new array object. The larger the array, the more expensive it gets.
- You need to be able to update any "live" variables that contain references to the old array.

One alternative is to create the array with a large enough size to start with. This is only viable if you can determine that size accurately *before allocating the array*. If you cannot do that, then the problem of resizing the array arises again.

The other alternative is to use a data structure class provided by the Java SE class library or a third-party library. For example, the Java SE "collections" framework provides a number of implementations of the `List`, `Set` and `Map` APIs with different runtime properties. The `ArrayList` class is closest to performance characteristics of a plain array (e.g. O(N) lookup, O(1) get and set, O(N) random insertion and deletion) while providing more efficient resizing without the reference update problem.

(The resize efficiency for `ArrayList` comes from its strategy of doubling the size of the backing array on each resize. For a typical use-case, this means that you only resize occasionally. When you amortize over the lifetime of the list, the resize cost per insert is `O(1)`. It may be possible to use the same strategy when resizing a plain array.)

Section 40.14: Converting arrays between primitives and boxed types

Sometimes conversion of primitive types to boxed types is necessary.

To convert the array, it's possible to use streams (in Java 8 and above):

Version ≥ Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray =
    Arrays.stream(primitiveArray).boxed().toArray(Integer[]::new);
```

With lower versions it can be by iterating the primitive array and explicitly copying it to the boxed array:

Version < Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray = new Integer[primitiveArray.length];
for (int i = 0; i < primitiveArray.length; ++i) {
    boxedArray[i] = primitiveArray[i]; // Each element is autoboxed here
}
```

Similarly, a boxed array can be converted to an array of its primitive counterpart:

Version ≥ Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};
int[] primitiveArray =
    Arrays.stream(boxedArray).mapToInt(Integer::intValue).toArray();
```

Version < Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};
int[] primitiveArray = new int[boxedArray.length];
for (int i = 0; i < boxedArray.length; ++i) {
    primitiveArray[i] = boxedArray[i]; // Each element is outboxed here
}
```

Section 40.15: Remove an element from an array

Java doesn't provide a direct method in `java.util.Arrays` to remove an element from an array. To perform it, you can either copy the original array to a new one without the element to remove or convert your array to another structure allowing the removal.

Using ArrayList

You can convert the array to a `java.util.List`, remove the element and convert the list back to an array as follows:

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

List<String> list = new ArrayList<>(Arrays.asList(array));
list.remove("foo");

// Creates a new array with the same size as the list and copies the list
// elements to it.
array = list.toArray(new String[list.size()]);

System.out.println(Arrays.toString(array)); //[bar, baz]
```

Using System.arraycopy

`System.arraycopy()` can be used to make a copy of the original array and remove the element you want. Below an example:

```
int[] array = new int[] { 1, 2, 3, 4 }; // Original array.
int[] result = new int[array.length - 1]; // Array which will contain the result.
int index = 1; // Remove the value "2".
```

```
// Copy the elements at the left of the index.
System.arraycopy(array, 0, result, 0, index);
// Copy the elements at the right of the index.
System.arraycopy(array, index + 1, result, index, array.length - index - 1);

System.out.println(Arrays.toString(result)); //[1, 3, 4]
```

Using Apache Commons Lang

To easily remove an element, you can use the Apache Commons Lang library and especially the static method `removeElement()` of the class `ArrayUtils`. Below an example:

```
int[] array = new int[]{1,2,3,4};
array = ArrayUtils.removeElement(array, 2); //remove first occurrence of 2
System.out.println(Arrays.toString(array)); //[1, 3, 4]
```

Section 40.16: Comparing arrays for equality

Array types inherit their `equals()` (and `hashCode()`) implementations from java.lang.Object, so `equals()` will only return true when comparing against the exact same array object. To compare arrays for equality based on their values, use `java.util.Arrays.equals`, which is overloaded for all array types.

```
int[] a = new int[]{1, 2, 3};
int[] b = new int[]{1, 2, 3};
System.out.println(a.equals(b)); //prints "false" because a and b refer to different objects
System.out.println(Arrays.equals(a, b)); //prints "true" because the elements of a and b have the
same values
```

When the element type is a reference type, `Arrays.equals()` calls `equals()` on the array elements to determine equality. In particular, if the element type is itself an array type, identity comparison will be used. To compare multidimensional arrays for equality, use `Arrays.deepEquals()` instead as below:

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

Object[] aObject = { a }; // aObject contains one element
Object[] bObject = { b }; // bObject contains one element

System.out.println(Arrays.equals(aObject, bObject)); // false
System.out.println(Arrays.deepEquals(aObject, bObject));// true
```

Because sets and maps use `equals()` and `hashCode()`, arrays are generally not useful as set elements or map keys. Either wrap them in a helper class that implements `equals()` and `hashCode()` in terms of the array elements, or convert them to `List` instances and store the lists.

Section 40.17: Copying arrays

Java provides several ways to copy an array.

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.

Object.clone()

Since arrays are `Objects` in Java, you can use `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.

Arrays.copyOf()

`java.util.Arrays` provides an easy way to perform the copy of an array to another. Here is the basic usage:

```
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);
```

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]
```

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]
```

Section 40.18: Casting Arrays

Arrays are objects, but their type is defined by the type of the contained objects. Therefore, one cannot just cast `A[]` to `T[]`, but each A member of the specific `A[]` must be cast to a `T` object. Generic example:

```
public static <T, A> T[] castArray(T[] target, A[] array) {
    for (int i = 0; i < array.length; i++) {
        target[i] = (T) array[i];
```

```
    }
    return target;
}
```

Thus, given an `A[]` array:

```
T[] target = new T[array.Length];
target = castArray(target, array);
```

Java SE provides the method `Arrays.copyOf(original, newLength, newType)` for this purpose:

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```