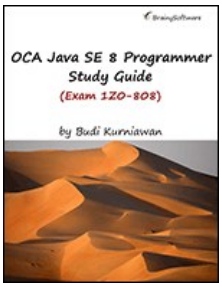

OCA Java SE 8 Programmer Study Guide (Exam 1Z0-808)

by Budi Kurniawan

Chapter 5: Arrays

Overview

In Java you can use an array to group primitives or objects of the same type. The entities belonging to an array is called the elements or components of the array. In this chapter you will learn how to create, initialize and iterate over an array as well as manipulate its elements. This chapter also features the **java.util.Arrays** and **java.util.ArrayList** classes.

5.1 Array Overview

In the background, every time you create an array, the compiler creates an object which allows you to:

- get the number of elements in the array through the **length** field. The length or size of an array is the number of elements in it.
- access each element by specifying an index. This indexing is zero-based. Index 0 refers to the first element, 1 to the second element, etc.

All the elements of an array have the same type, called the element type of the array. An array is not resizable and an array with zero element is called an empty array.

An array is a Java object. Therefore, an array variable behaves like other reference variables. For example, you can compare an array variable with **null**.

```
String[] names;
if (names == null) // evaluates to true
```

If an array is a Java object, shouldn't there be a class that gets instantiated when you create an array? May be something like **java.lang.Array?** The truth is, no. Arrays are indeed special Java objects whose class is not documented and is not meant to be extended.

To use an array, first you need to declare one. You can use this syntax to declare an array:

```
type[] arrayName;
```

or

```
type arrayName[]
```

For example, the following declares an array of **long**s named **numbers:**

```
long[] numbers;
```

Declaring an array does not create an array or allocate space for its elements, the compiler simply creates an object reference. One way to create an array is by using the **new** keyword. You must specify the size of the array you are creating.

```
new type[size]
```

As an example, the following code creates an array of four **int**s:

```
new int[4]
```

Alternatively, you can declare and create an array in the same line.

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

After an array is created, its elements are either **null** (if the element type is a reference type) or the default value of the element type (if the array contains primitives). For example, an array of **int**s contain zeros by default.

To reference an array element, use an index. If the size of an array is *n*, then the valid indexes are all integers between 0 and n-1. For example, if an array has four elements, the valid indexes are 0, 1, 2 and 3. The following snippet creates an array of four **String** objects and assigns a value to its first element.

```
String[] names = new String[4];
names[0] = "Hello World";
```

Using a negative index or a positive integer equal to or greater than the array size will throw a **java.lang.ArrayIndexOutOfBoundsException**. See Chapter 8, "Error Handling" for information about exceptions.

Since an array is an object, you can call the **getClass** method on an array. The string representation of the **Class** object of an array has the following format:

```
[type
```

where *type* is the object type. Calling **getClass().getName()** on a **String** array returns **[Ljava.lang.String**. The class name of a primitive array, however, is harder to decipher. Calling **getClass().getName()** on an **int** array returns **[I** and on a long array returns **[J**.

You can create and initialize an array without using the **new** keyword. Java allows you to create an array by grouping values within a pair of braces. For example, the following code creates an array of three **String** objects.

```
String[] names = { "John", "Mary", "Paul" };
```

The following code creates an array of four **int**s and assign the array to the variable **matrix**.

```
int[] matrix = { 1, 2, 3, 10 };
```

Be careful when passing an array to a method because the following is illegal even though the method **average** takes an array of **int**s.

```
int avg = average( { 1, 2, 3, 10 } ); // illegal
```

Instead, you have to instantiate the array separately.

```
int[] numbers = { 1, 2, 3, 10 };
int avg = average(numbers);
```

or you can do this

```
int avg = average(new int[] { 1, 2, 3, 10 });
```

5.2 Iterating over an Array

Prior to Java 5, the only way to iterate the members of an array was to use a **for** loop and the array's indexes. For example, the following code iterates over a **String** array referenced by the variable **names:**

```
for (int i = 0; i < 3; i++) {
    System.out.println("\t- " + names[i]);
}
```

Java 5 enhanced the **for** statement. You can now use it to iterate over an array or a collection without the index. Use this syntax to iterate over an array:

```
for (elementType variable : arrayName)
```

Where *arrayName* is the reference to the array, *elementType* is the element type of the array, and *variable* is a variable that references each element of the array.

For example, the following code iterates over an array of **String**s.

```
String[] names = { "John", "Mary", "Paul" };
for (String name : names) {
    System.out.println(name);
}
```

The code prints this on the console.

```
John
Mary
Paul
```

5.3 The java.util.Arrays Class

The **Arrays** class provides static methods to manipulate arrays. Table 5.1 shows some of its methods.

Table 5.1: More important methods of java.util.Arrays

Method	Description
asList	Returns a fixed-size **List** backed by the array. No other elements can be added to the **List**.
binarySearch	Searches an array for the specified key. If the key is found, returns the index of the element. If there is no match, returns the negative value of the insertion point minus one. See the section "Searching An Array" for details.
copyOf	Creates a new array having the specified length. The new array will have the same elements as the original array. If the new length is not the same as the length of the original array, it pads the new array with null or default values or truncates the original array.
copyOfRange	Creates a new array based on the specified range of the original array.
equals	Compares the contents of two arrays.
fill	Assigns the specified value to each element of the specified array.
sort	Sorts the elements of the specified array.
parallelSort	Parallel sorts the elements of the specified array.
toString	Returns the string representation of the specified array.

Some of these methods are explained further in the next sections.

5.4 Changing an Array Size

Once an array is created, its size cannot be changed. If you want to change the size, you must create a new array and populates it using the values of the old array. For instance, the following code increases the size of **numbers**, an array of three **int**s, to 4.

```
int[] numbers = { 1, 2, 3 };
int[] temp = new int[4];
int length = numbers.length;
for (int j = 0; j < length; j++) {
    temp[j] = numbers[j];
}
numbers = temp;
```

A shorter way of doing this is by using the **copyOf** method of **java.util.Arrays**. For instance, this code creates a four-element array and copies the content of **numbers** to its first three elements.

```
int[] numbers = { 1, 2, 3 };
int[] newArray = Arrays.copyOf(numbers, 4);
```

Of course you can reassign the new array to the original variable:

```
numbers = Arrays.copyOf(numbers, 4);
```

The **copyOf** method comes with ten overloads, eight for each type of Java primitives and two for objects. Here are their signatures:

```
public static boolean[] copyOf(boolean[] original, int newLength)

public static byte[] copyOf(byte[] original, int newLength)

public static char[] copyOf(char[] original, int newLength)

public static double[] copyOf(double[] original, int newLength)

public static float[] copyOf(float[] original, int newLength)

public static int[] copyOf(int[] original, int newLength)

public static long[] copyOf(long[] original, int newLength)

public static short[] copyOf(short[] original, int newLength)

public static <T> T[] copyOf(T[] original, int newLength)

public static <T,U> T[] copyOf(U[] original, int newLength,
        java.lang.Class<? extends T[]> newType)
```

Each of these overloads may throw a **java.lang.NullPointerException** if *original* is null and a **java.lang.NegativeArraySizeException** if *newLength* is negative.

The *newLength* argument can be smaller, equal to, or larger than the length of the original array. If it is smaller, then only the first *newLength* elements will be included in the copy. If it is larger, the last few elements will have default values, i.e. 0 if it is an array of integers or **null** if it is an array of objects.

Another method similar to **copyOf** is **copyOfRange. copyOfRange** copies a range of elements to a new array. Like **copyOf, copyOfRange** also provides overrides for each Java data type. Here are their signatures:

```
public static boolean[] copyOfRange(boolean[] original,
        int from, int to)

public static byte[] copyOfRange(byte[] original,
        int from, int to)

public static char[] copyOfRange(char[] original,
        int from, int to)

public static double[] copyOfRange(double[] original,
        int from, int to)

public static float[] copyOfRange(float[] original,
        int from, int to)

public static int[] copyOfRange(int[] original, int from, int to)

public static long[] copyOfRange(long[] original, int from, int to)

public static short[] copyOfRange(short[] original, int from,
        int to)

public static <T> T[] copyOfRange(T[] original, int from, int to)

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

You can also use **System.arraycopy()** to copy an array. However, **Arrays.copyOf()** is easier to use and internally it calls **System.arraycopy()**.

5.5 Searching An Array

You can use the **binarySearch** method of the **Arrays** class to search an array. This method comes with twenty overloads. Here are two of its overloads:

```
public static int binarySearch(int[] array, int key)

public static int binarySearch(java.lang.Object[] array,
        java.lang.Object key)
```

There are also overloads that restrict the search area.

```
public static int binarySearch(int[] array, int fromIndex,
        int toIndex, int key)

public static int binarySearch(java.lang.Object[] array,
        int fromIndex, int toIndex, java.lang.Object key)
```

The **binarySearch** method employs a binary search algorithm to do the search. Using this algorithm, the array is first sorted in ascending or descending order. It then compares the search key with the middle element of the array. If there is a match, the element index is returned. If there is no match, depending whether the search key is lower or higher than the index, the search continues in the first or second half of the array, repeating the same procedure until there is no or only one element left. If at the end of the search no match is found, the **binarySearch** method returns the negative value of the insertion point minus one. The example in Listing 5.1 will make this point clearer.

Listing 5.1: A binary search example

```
package app05;
import java.util.Arrays;

public class BinarySearchDemo {
    public static void main(String[] args) {
        int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
```

```
        int index = Arrays.binarySearch(primes, 13);
        System.out.println(index); // prints 5
        index = Arrays.binarySearch(primes, 4);
        System.out.println(index); // prints -3
    }
}
```

The **BinarySearchDemo** class in Listing 5.1 uses an **int** array containing the first eight prime numbers. Passing 13 as the search key returns 5 because 13 is the sixth element in the array, i.e. with index 5. Passing 4 does not find a match and the method returns -3, which is -2 minus one. If the key were to be inserted to the array, it would have the index 2.

5.6 Passing a String Array to main

The public static void method **main** that you use to invoke a Java class takes an array of **String**s. Here is the signature of **main:**

```
public static void main(String[] args)
```

You can pass arguments to **main** by typing them as arguments to the **java** program. The arguments should appear after the class name and two arguments are separated by a space. You use the following syntax:

```
java className arg1 arg2 arg3 ... arg-n
```

Listing 5.2 shows a class that iterates over the **main** method's **String** array argument.

Listing 5.2: Accessing the main method's arguments

```
package app05;
public class MainMethodTest {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

The following command invokes the class and passes two arguments to the **main** method.

```
java app05/MainMethodTest john mary
```

The **main** method will then print the arguments to the console.

```
john
mary
```

If no argument is passed to **main**, the **String** array **args** will be empty and not null.

5.7 Multidimensional Arrays

In Java a multidimensional array is an array whose elements are also arrays. As such, the rows can have different lengths, unlike multidimensional arrays in C language.

To declare a two dimensional array, use two pairs of brackets after the type:

```
int[][] numbers;
```

To create an array, pass the sizes for both dimensions:

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

Listing 5.3 shows a multidimensional array of **int**s.

Listing 5.3: A multidimensional array.

```
package app05;
import java.util.Arrays;
```

```
public class MultidimensionalDemo1 {
    public static void main(String[] args) {
        int[][] matrix = new int[2][3];
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 3; j++) {
                matrix[i][j] = j + i;
            }
        }

        for (int i = 0; i < 2; i++) {
            System.out.println(Arrays.toString(matrix[i]));
        }
    }
}
```

The following will be printed on the console if you run the class.

```
[0, 1, 2]

[1, 2, 3]
```

5.8 ArrayList

As mentioned previously, arrays cannot be resized. Oftentimes, however, you need the flexibility to be able to add elements to a container without constantly worrying about the size of the container. For this, you can use **ArrayList**.

A member of the **java.util** package, **ArrayList** is an ordered collection. You can access its elements by using indices and you can insert an element into an exact location. Index 0 of an **ArrayList** references the first element, index 1 the second element, and so on.

The **add** method appends the specified element to the end of the list. Here is its signature.

```
public boolean add(java.lang.Object element)
```

This method returns **true** if the addition is successful. Otherwise, it returns **false. ArrayList** allows you to add null.

ArrayList also has another **add** method with the following signature:

```
public void add(int index, java.lang.Object element)
```

With this **add** method you can insert an element at any position.

In addition, you can replace and remove an element by using the **set** and **remove** methods, respectively.

```
public java.lang.Object set(int index, java.lang.Object element)

public java.lang.Object remove(int index)
```

The **set** method replaces the element at the position specified by *index* with *element* and returns the reference to the element inserted. The **remove** method removes the element at the specified position and returns a reference to the removed element.

To create an **ArrayList**, you specify the type of objects that it can contain. For example, to create an **ArrayList** that can store String object, you would write one of these:

```
ArrayList<String> myArrayList = new ArrayList<String>();

ArrayList<String> myArrayList = new ArrayList<>();
```

The no-argument constructor of **ArrayList** creates an **ArrayList** object with an initial capacity of ten elements. The size will grow automatically as you add more elements than its capacity. If you know that the number of elements in your **ArrayList** will be more than its capacity, you can use the second constructor:

```
public ArrayList(int initialCapacity)
```

This will result in a slightly faster **ArrayList** because the instance does not have to grow in capacity.

Listing 5.4 demonstrates the use of **ArrayList** and some of its methods.

Listing 5.4: Using ArrayList

```
package app05;
import java.util.ArrayList;

public class ArrayListDemo1 {
    public static void main(String[] args) {
        ArrayList<String> myList = new ArrayList<>();
        myList.add("Hello");
        myList.add(1, "World");
        myList.add(null);
        System.out.println("Size: " + myList.size());
        for (String word: myList) {
            System.out.println(word);
        }
    }
}
```

When run, here is the result on the console.

```
Size: 3
Hello
World
null
```

Self Test

1 Question ?

Which of the following statements about arrays is (are) true?

A. An array is an object

B. Only an array of objects is an object

C. An array of primitives is not an object

D. An array cannot be resized

2 Question ?

Here is an intriguing piece of code.

```
... numbers = {{1, 2, 3}, {4}};
```

What Java type can be used for **numbers?** (Choose all that apply)

A. int[][]

B. byte[][]

C. float[][]

D. Integer[][]

E. Object[][]

F. It is an attempt to declare a two-dimensional array, but the statement is invalid because each row in a two-dimensional array must have the same number of elements.

3 Question ?

Consider the following code snippet.

```
public class Container {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

What can be said about iterating the arguments in an enhanced for?

A. It will throw a NullPointerException if the class is invoked without an argument

B. It will throw a ArrayIndexOutOfBoundsException if the class is invoked without an argument

C. The code will not throw an exception

D. The code will print the arguments passed to the class

4 Question ?

Given

```
public class ArrayTest {
    public static void main(String[] args) {
        int[] numbers = new int[5];
        numbers[0] = 100;
        for (int i = 1; i < 5; i++) {
            numbers[i] = i + numbers[i - 1];
        }
        System.out.println(numbers[4]);
    }
}
```

What is the output of the program?

A. 108

B. 109

C. 110

D. 120

5 Question ?

Given

```
package com.brainysoftware.oca;
import java.util.ArrayList;
public class EmployeeRecord {
    public static void main(String[] args) {
        ArrayList<String> employees = new ArrayList<>();
        employees.add("Henry Higgins");
        employees.add("William Murdoch");
        employees.add(0, "Craig Thomas");
        for (int i = 0; i < 3; i++) {
            System.out.print(employees.get(i));
            if (i < 2) {
                System.out.print(", ");
            }
        }
    }
}
```

What is the output of this program?

A. Henry Higgins, William Murdoch, Craig Thomas

B. Craig Thomas, William Murdoch, Craig Thomas

C. Craig Thomas, Henry Higgins, William Murdoch

D. Henry Higgins, Craig Thomas, William Murdoch

6 Question ?

Which statement(s) creates an **ArrayList** of **Employee** object with an initial capacity of 10? (Choose all that apply)

A. ArrayList<Employee> list = new ArrayList<>();

B. ArrayList<Employee> list = new ArrayList<>(10);

C. ArrayList<Employee> list = new ArrayList<Employee>();

D. ArrayList<Employee> list = new ArrayList<Employee>(10);

7 Question ?

This code fragment attempts to convert an **ArrayList** to an array:

```
ArrayList<String> list = new ArrayList<>();
list.add("Benson");
String[] names = list.toArray(new String[0]);
for (String name : names) {
    System.out.println(name);
}
```

What is the output of this code fragment?

A. No output because the **toArray** method returns an empty String array

B. A runtime error because there is no place in the array to move the element in the **ArrayList** to.

C. Benson

D. null because the String array has not been initialized

8 Question ?

Given this class:

```
1. public class MyDB {
2.     public static void main(String[] args) {
3.         String name1 = "John Troy";
4.         String name2 = "Simba";
5.         String[] names = {name1, name2};
6.         name1 = null;
7.         name2 = null;
8.     }
9. }
```

Which of the following statements are correct?

A. One object is eligible for garbage collection at line 7

B. Two objects are eligible for garbage collection at line 8

C. No object is eligible for garbage collection at line 7

D. No object is eligible for garbage collection at line 8

9 Question ?

Given the following code snippet:

```
public class FriendManager {
    public static void main(String[] args) {
        String[][] friends = {{"James", "Gillis"}, {"Tony", "Bubba"},
                {"Alexis"}};
        System.out.println(friends[0][1]);
    }
}
```

What is the output of this program?

A. James

B. Gillis

C. null

D. Alexis

10 Question ?

Given the following class:

```
package test;
import java.util.ArrayList;
public class Entertainer {
    public static void main(String[] args) {
        ArrayList<int> cards = new ArrayList<>();
        cards.add(1);
        cards.add(2);
        cards.add(3);
        for (int i : cards) {
            System.out.print(i);
        }
    }
```

```
}
```

What is the output of this class?

A. 123

B. 1 2 3

C. The class will not compile.

D. The program will throw a NullPointerException

Answers

1 **A, D**.

An array is always an object, regardless of the type of the data it contains. Once created, an array cannot be resized.

2 **A, B, C, D, E**.

A and B are the most likely, but C, D and E are also valid. With regard to D, the array elements will be converted to Integer objects automatically. F is incorrect as each dimension in a multidimensional array may have a different number of elements in each of its rows.

3 **C, D**.

The String array passed to the **main** method will never be null. In the event the class is invoked without an argument, the JVM will create an empty array.

4 **C**.

- Focus on the loop. There are four iterations, from i = 1 to i = 4.
- numbers[1] = 1 + numbers[0] = 1 + 100 = 101
- numbers[2] = 2 + numbers[1] = 2 + 101 = 103
- numbers[3] = 3 + numbers[2] = 3 + 103 = 106
- numbers[4] = 4 + numbers[3] = 4 + 106 = 110

5 **C**.

The code added "Henry Higgins" and "William Murdoch" as the first and second elements of the ArrayList, respectively. Then, it inserted "Craig Thomas" at position 1, pushing the previous elements to positions 2 and 3, respectively.

6 **A, B, C, D**.

All of them. By default, an **ArrayList** is created with an initial capacity of 10 elements.

7 **C**.

The **toArray** method returns an array and copies all the elements of the **ArrayList** to the array. The array will have the same size as the **ArrayList**.

8 **C, D**.

The array holds references two the two String objects even after the reference variables name1 and name2 are set to null. Therefore, no object is eligible for garbage collection on line 7 and line 8.

9 **B**.

friends[0] refers to the first array in the two-dimensional array, which is {"James", "Gillis"}. **friends[0][1]** references the second element in the first array.

10 **C**.

An **ArrayList** may only contain objects and not primitives.