ARRAYS

Exercises: -

1. Get a Random Value from an Array

By using the *java.util.Random* object we can easily get any value from our array:

Ans.

```
int anyValue = array[new Random().nextInt(array.length)];
```

2. Compare Two Arrays

Even though arrays are *Object*s and therefore provide an *equals* method, they use the default implementation of it, relying only on reference equality.

We can anyhow invoke the *java.util.Arrays*' *equals* method to check if two array objects contain the same values:

Ans.

```
boolean areEqual = Arrays.equals(array1, array2);
```

Note: this method is not effective for jagged arrays. The appropriate method to verify multi-dimensional structures' equality is the *Arrays.deepEquals* one.

3. Check if an Array Is Empty

This is an uncomplicated assignment having in mind that we can use the *length* attribute of arrays:

Ans.
```
boolean isEmpty = array == null || array.length == 0;
```

Moreover, we also have a null-safe method in the *ArrayUtils* helper class that we can use:

Ans.
```
boolean isEmpty = ArrayUtils.isEmpty(array);
```

This function still depends on the length of the data structure, which considers nulls and empty sub-arrays as valid values too, so we'll have to keep an eye on these edge cases:

```
// These are empty arrays
Integer[] array1 = {};
Integer[] array2 = null;
Integer[] array3 = new Integer[0];

// All these will NOT be considered empty
Integer[] array3 = { null, null, null };
Integer[][] array4 = { {}, {}, {} };
Integer[] array5 = new Integer[3];
```

4. How to Shuffle the Elements of an Array

In order to shuffle the items in an array, we can use the *ArrayUtil*'s feature:

Ans.
```
ArrayUtils.shuffle(array);
```

5. Box and Unbox Arrays

We often come across methods that support only *Object*-based arrays.

Again the *ArrayUtils* helper class comes in handy to get a boxed version of our primitive array:

Ans.
```
Integer[] list = ArrayUtils.toObject(array);
```

The inverse operation is also possible:

Ans.
```
Integer[] objectArray = { 3, 5, 2, 5, 14, 4 };
int[] array = ArrayUtils.toPrimitive(objectArray);
```

6. Remove Duplicates from an Array

The easiest way of removing duplicates is by converting the array to a *Set* implementation.

As we may know, *Collection*s use Generics and hence don't support primitive types.

For this reason, if we're not handling object-based arrays as in our example, we'll first need to box our values:

Ans.
```
// Box
Integer[] list = ArrayUtils.toObject(array);
// Remove duplicates
Set<Integer> set = new HashSet<Integer>(Arrays.asList(list));
// Create array and unbox
return ArrayUtils.toPrimitive(set.toArray(new Integer[set.size()]));
```

Note: we can use other techniques to convert between an array and a *Set* object as well.

Also, if we need to preserve the order of our elements, we must use a different *Set* implementation, such as a *LinkedHashSet*.

7. Map an Array to Another Type

It's often useful to apply operations on all array items, possibly converting them to another type of object.

With this objective in mind, **we'll try to create a flexible helper method using Generics:**

```
public static <T, U> U[] mapObjectArray(
  T[] array, Function<T, U> function,
  Class<U> targetClazz) {
    U[] newArray = (U[]) Array.newInstance(targetClazz, array.length);
    for (int i = 0; i < array.length; i++) {
        newArray[i] = function.apply(array[i]);
    }
    return newArray;
}
```

If we don't use Java 8 in our project, we can discard the *Function* argument, and create a method for each mapping that we need to carry out.

We can now reuse our generic method for different operations. Let's create two test cases to illustrate this:

```
@Test
public void whenMapArrayMultiplyingValues_thenReturnMultipliedArray() {
    Integer[] multipliedExpectedArray = new Integer[] { 6, 10, 4, 10, 28, 8
};
    Integer[] output =
      MyHelperClass.mapObjectArray(array, value -> value * 2,
Integer.class);

    assertThat(output).containsExactly(multipliedExpectedArray);
}

@Test
public void whenMapDividingObjectArray_thenReturnMultipliedArray() {
    Double[] multipliedExpectedArray = new Double[] { 1.5, 2.5, 1.0, 2.5,
7.0, 2.0 };
    Double[] output =
      MyHelperClass.mapObjectArray(array, value -> value / 2.0,
Double.class);

    assertThat(output).containsExactly(multipliedExpectedArray);
}
```

For primitive types, we'll have to box our values first.

As an alternative, we can turn to Java 8's Streams to carry out the mapping for us.

We'll need to transform the array into a *Stream* of *Object*s first. We can do so with the *Arrays.stream* method.

For example, if we want to map our *int* values to a custom *String* representation, we'll implement this:

```
String[] stringArray = Arrays.stream(array)
  .mapToObj(value -> String.format("Value: %s", value))
  .toArray(String[]::new);
```

8. Filter Values in an Array

Filtering out values from a collection is a common task that we might have to perform in more than one occasion.

This is because at the time we create the array that will receive the values, we can't be sure of its final size. Therefore, **we'll rely on the *Stream*s approach again.**

Imagine we want to remove all odd numbers from an array:

```
int[] evenArray = Arrays.stream(array)
  .filter(value -> value % 2 == 0)
  .toArray();
```

9. Other Common Array Operations

There are, of course, plenty of other array operations that we might need to perform.

Apart from the ones shown in this tutorial,
we've extensively covered other operations in the dedicated posts:

- Check if a Java Array Contains a Value
- How to Copy an Array in Java
- Removing the First Element of an Array
- Finding the Min and Max in an Array with Java
- Find Sum and Average in a Java Array
- How to Invert an Array in Java
- Join and Split Arrays and Collections in Java
- Combining Different Types of Collections in Java
- Find All Pairs of Numbers in an Array That Add Up to a Given Sum
- Sorting in Java
- Efficient Word Frequency Calculator in Java
- Insertion Sort in Java