Arrays class, part 2

1. `Arrays.fill()`

When working with arrays, Java programmers very often need to fill the array with the same value. You can, of course, write a loop and simply assign some value to each cell of the array in the loop:

```
int[] x = new int[100];
for (int i = 0; i < x.length; i++)
x[i] = 999;
```

Or you can simply call the `Arrays.fill()` method, which does exactly the same thing: it fills the passed array with the passed value. Here's how it looks:

```
Arrays.fill(name, value)
```

And the code in the example above can be made a little bit more compact and clearer:

```
int[] x = new int[100];
Arrays.fill(x, 999);
```

You can also use the `Arrays.fill()` method to fill not the entire array, but a part of it, with some value:

```
Arrays.fill(name, first, last, value)
```

Where first and last are the indices of the first and last cells to be filled.

> In accordance with Java's good old tradition, remember that the last element is not included in the range.

Example:

`int[] x = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};`
 	
`Arrays.fill(x, 3, 7, 999);`
 	
`String str = Arrays.toString(x);`
	We're filling the cells `x[3]`, `x[4]`, `x[5]`, and `x[6]` with the value `999`. Cells of an array are numbered starting from zero!
	
	The `str` variable contains the value:
	`"[1, 2, 3, 999, 999, 999, 999, 8, 9, 10]"`

The `Arrays.fill()` method only works with one-dimensional arrays. If you pass a two-dimensional array to the method, it will be treated as one-dimensional, with all the ensuing consequences.

2. `Arrays.copyOf()`

As you already know, you cannot resize an array after it has been created.

But what if you really want to?

Well, if you really want to, then you can!

- Create a new array of the desired length
- Copy all the elements from the first array into it.

By the way, this is exactly what the `Arrays.copyOf()` method does. This is what calling it looks like:

```
Type[] name2 = Arrays.copyOf(name, length);
```

This method **does not change the existing array**, but instead **creates a new array** and copies the elements of the old array into it.

If the elements don't fit (the length is less than the length of the existing array), then the extra values are ignored.

If the length of the new array is greater than the length of the old one, the cells are filled with zeros.

Example:

```java
int[] x = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int[] x2 = Arrays.copyOf(x, 5);
String str2 = Arrays.toString(x2);

int[] x3 = Arrays.copyOf(x, 15);
String str3 = Arrays.toString(x3);
```

The `str2` variable contains the value:
`"[1, 2, 3, 4, 5]"`

The `str3` variable contains the value:
`"[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0, 0, 0, 0]"`

3. `Arrays.copyOfRange()`

And what if you want to get an array of length 5 from an array of length 10, but you need it to contain the last 5 elements rather than the first 5?

In this case, another method of the `Arrays` class will come in handy — the `Arrays.copyOfRange()`. Here's what it looks like when we call it:

```java
Type[] name2 = Arrays.copyOfRange(name, first, last);
```

This method also creates a new array, but fills it with elements from an arbitrary place in the original array. Where first and last are the indices of the first and last elements that should be put into the new array.

> In accordance with Java's good old tradition, remember that the last element is not included in the range.

Example:

```java
int[] x = {11, 12, 13, 14, 15, 16, 17, 18, 19, 20};

int[] x2 = Arrays.copyOfRange(x, 5, 10);
String str2 = Arrays.toString(x2);

int[] x3 = Arrays.copyOfRange(x, 5, 15);
String str3 = Arrays.toString(x3);
```

The `str2` variable contains the value:
`"[16, 17, 18, 19, 20]"`

The `str3` variable contains the value:
`"[16, 17, 18, 19, 20, 0, 0, 0, 0, 0]"`

4. `Arrays.sort()`

Ah, the most delicious treat: **sorting**. Arrays are sorted quite often in programming. The three most common actions when working with arrays are:

- Sorting an array
- Finding the minimum (or maximum) element of an array

- Determining the index of an element in an array (finding an element in an array)

This is precisely why Java's creators included the `sort()` method in the `Arrays` class. This is what calling it looks like:

```
Arrays.sort(name);
```

This method sorts the passed array in ascending order.

Example:

```
int[] x = {11, -2, 3, 0, 999, -20, 8, -20, 99, -20};

Arrays.sort(x);

String str = Arrays.toString(x);
```

The `str` variable contains the value:

`"[-20, -20, -20, -2, 0, 3, 8, 11, 99, 999]"`

Excellent, right? You called one method, and now you have a sorted array. Beautiful.

By the way, you can use this method to sort not only the entire array, but just part of it. This is what calling it looks like:

```
Arrays.sort(name, first, last);
```

Where first and last are the indices of the first and last cells that the sort should touch.

> In accordance with Java's good old tradition, remember that the last element is not included in the range.

Example:

```
int[] x = {11, -2, 3, 0, 999, -20, 8, -20, 99, -20};

Arrays.sort(x, 4, 8);
String str = Arrays.toString(x);
```

The `str` variable contains the value:

`"[11, -2, 3, 0, -20, -20, 8, 999, 99, -20]"`

To sort arrays, Java uses the fastest sorting algorithm — ***QuickSort***. Its computational complexity depends on the size of the array and is calculated using the formula N log(N).

Sorting an array of 1000 elements will involve about 3,000 comparisons of array elements. Sorting an array of one million elements will involve about 6 million comparisons.

5. `Arrays.binarySearch()`

Well, and the last of the most interesting methods of the `Arrays` class is able to search for a given value in an array. This is no ordinary search — it is the beloved *binary search*. It boils down to this:

- First, the array is sorted.
- Then the middle element of the array is compared with the one we are looking for.
- If the element is greater than the middle element, then the search continues in the right half of the array.
- If the element we are looking for is less than the middle element, then the search continues in the left half of the array.

Because the array is sorted, it is possible to eliminate half of it in a single comparison. Then in the next step, we toss out another half, and so on.

This approach makes binary search very fast. In an array of one million (!) elements, it can find the index of the desired element in just 20 comparisons. The approach's shortcoming is that the array must first be sorted, and sorting also takes time.

This is what calling it looks like:

```
int index = Arrays.binarySearch(name, value);
```

Where `name` is the name of the array, which must be passed already sorted (for example, using the `Arrays.sort()` method). And `value` is the element we are searching for in the array. The result returned by the method is the *index of the desired array element*.

Examples:

```java
int[] x = {11, -2, 3, 0, 999, -20, 8, -20, 99, -20};
Arrays.sort(x);

int index1 = Arrays.binarySearch(x, 0);
int index2 = Arrays.binarySearch(x, -20);
int index3 = Arrays.binarySearch(x, 99);
int index4 = Arrays.binarySearch(x, 5);
```

`x` is:
`{-20, -20, -20, -2, 0, 3, 8, 11, 99, 999}`

`4`
`1` (indices `0` and `2` are also acceptable)
`8`
`-7`

6. Link to Oracle documentation on the `Arrays` class

If you are super interested, you can read everything about the `Arrays` class and all its methods in the official documentation a href="https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Arrays.html">on the Oracle website.

For example, you can read about the `Arrays.mismatch()` and `Arrays.compare()` methods. Maybe you will find them useful somehow.

And don't be confused by the number of methods. Each method has 5-10 variants, which differ only in their parameter types.