Static data types

Arrays

Static data types store values so that the total size of these values does not change along the program execution. In this document we are going to explain what an array is and how to use them to store many values of the same type.

An array is a set of data of the same type (integers, texts, characters...) grouped in a single variable. Data is structured so that we can access each element by its index or position in the array, starting at 0. The total number of elements contained in the array must be known before creating the array. So the size of the array is constant, and can't be changed along the program execution.

1. Unidimensional arrays

Unidimensional arrays are the simplest type of arrays. They consist in a sequence of data with a single index to access each position. We use square brackets `[]` to declare a variable as an array, and the `new` operator to determine its size (number of positions). We can declare the array variable in one line and determine its size in another, different line (once we know the size that we need for the array) or do it all in the same line:

```
int[] numbers = new int[10];         // Array of 10 integers

float[] numbers;
int n;
Scanner sc = new Scanner(System.in);
System.out.println("Enter the array size:");
n = sc.nextInt();
numbers = new int[n];                // Array of n floats
```

Square brackets can also be placed after the variable name, although we recommend you to place them before, so we can easily see that this variable is an array:

```
int numbers[] ...
```

In order to access the elements of an array, we set the desired index inside the square brackets, starting at position 0. This way we can, for instance, specify a concrete value for each element in the array:

```
numbers[0] = 3;
numbers[1] = 6;
...
```

We can also use these indexes to check the concrete value of a position:

```
System.out.println(numbers[1]);
System.out.println(numbers[2] * numbers[4]);
```

In order to check the total number of elements of an array, we can use its `length` property. For instance, inside a `for` loop:

```
for (int i = 0; i < numbers.length; i++){
    System.out.println(numbers[i]);
}
```

Alternatively, we can use this `for` structure to explore the contents of the array. This structure is particularly useful if we just want to check (not change) the values of the array:

```
for (int n: numbers){
    System.out.println(n);
}
```

Finally, we can set the initial values of the array when we declare it, placing these values into curly braces, separated by commas:

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

1.1. Array search

If we want to search a given value inside an array, we usually explore the array from the beginning to the end, looking for that value. We may need to:

- Determine if a given value exists in the array. In this case, we can finish as soon as we find this value in the array. For instance, this code checks if number 15 exists in an integer array. Note that we use a `while` loop along with a *boolean* variable to stop checkin whenever we find the value:

```
boolean found = false;
int i = 0;
while (!found && i < numbers.length){
    if (numbers[i] == 15){
        found = true;
    }
    else{
        i++;
    }
}

if (found){
    System.out.println("Number found at position " + i);
}
else{
    System.out.println("Number not found";
}
```

- Check how many times a value exists in an array. In this second case, we need to explore the whole array. For instance, this code counts the occurrences of number 15 in an integer array. To do this, we can better use a `for` structure with a counter:

```
int counter = 0;
for (int i = 0; i < numbers.length; i++){
    if (numbers[i] == 15){
        counter++;
    }
}

System.out.println("Numbe 15 has been found " + counter + " times");
```

Looking for minimum / maximum

In some situations, we may need to find the maximum or minimum value in an array. An appropriate approach for this consists in:

1. Assign this minimum/maximum value to position 0. In other words, we suppose that this first number will be the minimum or maximum we are looking for.
2. Explore the rest of the array (from position 1) and check if there is a number that exceeds previous minimum/maximum. If so, we need to update this minimum/maximum.

This example finds the maximum value in an integer array called *numbers*:

```java
int maximum = numbers[0];
for (int i = 1; i < numbers.length; i++){
    if (numbers[i] > maximum){
        maximum = numbers[i];
    }
}

System.out.println("The maximum is " + maximum);
```

1.2. Array sort

Sorting arrays is a task that consists in placing each value in its appropriate position, according to a sorting criteria. For instance, if we want to sort an integer array in ascending order, we need to make sure that every number is placed before the numbers that are higher.

There are many different algorithms to face this task. One of the most common are the *bubble* algorithm and the *direct exchange* algorithm.

Bubble algorithm it compares two consecutive numbers of the array and, if they are not ordered, it swaps their values. At the end of this process, the highest/lowest value will be placed at the end of the array (depending on whether we are sorting in ascending or descending order). We need to replace this process N times to sort the N values of the array. This is the algorithm in Java to sort an array in ascending order:

```java
for (int i = 0; i < numbers.length; i++){
    for (int j = 0; j < numbers.length - i - 1; j++){
        if (numbers[j] > numbers[j+1])º{
            int aux = numbers[i];
            numbers[i] = numbers[j];
            numbers[j] = aux;
        }
    }
}
```

Note that, in every iteration of *j* variable, we finish one position before previous iteration, because the latest positions of the array have already been sorted in earlier iterations.

The **direct exchange** algorithm explores the whole array for every position *i*, looking for for the lowest/highest element, and places this element in this *i* position. This is the algorithm in Java for ascending order:

```
for (int i = 0; i < numbers.length - 1; i++)
{
    for (int j = i + 1; j < numbers.length; j++)
    {
        if (numbers[i] > numbers[j])
        {
            int aux = numbers[i];
            numbers[i] = numbers[j];
            numbers[j] = aux;
        }
    }
}
```

Note that *j* index is always greater than *i* index, so if element at position *i* is greater than element at position *j*, we need to swap their values to keep the array sorted.

All these sorting algorithms have a complexity of **O(n^2)**, which means that, for an array of size *n* we need to do n^2 iterations, approximately, to sort the whole array. There are other approaches, such as the *quicksort* algorithm, which are faster, but much more difficult to understand and type.

Also, Java provides an automatic sorting method for arrays, through `Arrays.sort` instruction. If array is made of simple data (such as integers, strings or floats, for instance), this instruction automatically sorts the array in ascending order. We need to import `java.util` package in order to use this instruction.

```
import java.util.Arrays;
...
int[] numbers = { 4, 6, 2, 8, 3};
Arrays.sort(numbers);
```

2. Bidimensional arrays

Although we can use unidimensional arrays for most of the tasks that we can do with arrays, in some situations we may need to store the information in a table, for instance, with its rows and columns, or store different sequences of data in separate indexes. To do this, we need to use bidimensional arrays.

In bidimensional arrays we need two indexes: one to refer to the row we are interested, and another one to point at the column inside this row. We need two pairs of square brackets when we declare a bidimensional array. Inside each pair of brackets we specify the total number of rows and columns of the array, respectively:

```
int[][] table = new int[3][10];     // 3 rows, 10 columns
```

We refer to each column by its row number, and then column number (both starting at 0):

```
table[0][2] = 2;        // 1st row, 3rd column
table[2][8] = 12;       // 3rd row, 9th column
```

If we want to explore bidimensional arrays, we need a nested loop: In the outer loop we can explore rows/columns, and in the inner loop we explore the opposite. We can use the `length` property of each dimension to determine its size. This example shows the contents of an array row by row:

```
for (int i = 0; i < table.length; i++)
{
    for (int j = 0; j < table[i].length; j++)
    {
        System.out.print(table[i][j] + " ");
    }
    System.out.println();
}
```

Note that `table.length` stores the number of rows, and `table[i].length` the number of columns for row *i*.

Alternatively, we can also establish a different number of columns for each row. This is called an *array of arrays*. To do this, we need to leave empty the second pair of square brackets when we declare the array:

```
int[][] data = new int[3][];        // 3 rows
```

Then, we need to go row by row establishing the number of columns for this row:

```
data[0] = new int[3];       // 1st row: 3 columns
data[1] = new int[5];       // 2nd row: 5 columns
data[2] = new int[8];       // 3rd row: 8 columns
```

We can also establish the default values of the whole array in the same way that we did for unidimensional arrays. In this case, the data of each row is enclosed in curly braces as well:

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

Exercise 1:

Create a program called **MatrixAddition** that asks the user to enter two bidimensional matrices or tables of 3 rows and columns, and then prints the result of adding them. In order to add two matrices, you must do it cell by cell:

```
result[i][j] = matrixA[i][j] + matrixB[i][j]
```

Exercise 2:

Create a program called **MarkCount** that asks the user to enter 10 marks (integers between 0 and 10). The program must output how many marks of each type have been typed. For instance, if the user types these marks: 1, 7, 5, 7, 2, 6, 7, 3, 5, 8, then the program must output:

```
Marks per category:
0: 0 marks
1: 1 marks
2: 1 marks
3: 1 marks
4: 0 marks
5: 2 marks
6: 1 marks
7: 3 marks
8: 1 marks
9: 0 marks
10: 0 marks
```