Programming in Java (24/25)

– *Exercises Day 5* –

Learning goals

Before the next day, you should achieve the following learning goals:

- Become familiarised with the use of `private` and `public`. All your classes, fields, and methods should specify explicitly whether they are public or private according to the rules of thumb in the notes. If you make a decision of visibility that deviates from those rules, you should explain your reason in a comment.
- Related to the previous point, you should become used to write constructors in all your classes. The constructor(s) should be used to initialise the fields of any new object of that class.
- Understand what the keyword `static` means and how it works.
- Be able to create classes in their own `.java` file and to compile them using `javac`.
- Building on the previous point, create full Java programs. These programs must be completely object-oriented and launch from the `main` method of one of the classes.
- Be able to cast simple types from one type to another.
- Be able to create and use arrays in one or more dimensions.

Remember that star exercises are more difficult. Do not try them unless the normal ones are clear to you.

1 Dividing integers

Create a Java class called `Calculator`. The class should implement the following instance methods, each of them returning the result as a **`double`**.

- `add(int, int)`
- `subtract(int, int)`
- `multiply(int, int)`
- `divide(int, int)`
- `modulus(int, int)`

Note that you will need to cast the parameters into **`double`** to perform exact division.

Write a small Java program in a separate class with a `main` method that uses all the methods of `Calculator` and prints the results on the screen. (You do not need to provide a menu as in Exercise 3 of Day 3.)

2 Checking arrays

Create a new Java class called `ArrayChecker` with two instance methods:

`isSymmetrical(int[])`: a method that returns true if the array of **`int`** provided as argument is symmetrical and false otherwise. An array is symmetrical if the element at [0] is the same as the element at [length-1], the element at [1] is the same as the element at [length-2], etc.

`reverse(int[])`: a method that takes an array of **`int`** and returns another array of **`int`** of the same size and with the same numbers, but in opposite order. The method should not modify the contents of the input array.

Write a Java class with a `main` method that creates an object of class `ArrayChecker` and uses its methods to check whether a few arrays are symmetrical and, if they are not, reverses them.

3 Copying arrays

Create a new Java class called `ArrayCopier` with an instance method called copy that takes two arrays of integers as parameters. The method should copy the elements of the first array (you can call it `src`, from "source") to the second one (`dst`, from "destination") as much as possible.

If the second array is smaller, then only those elements that fit will be copied. If the second array is larger, it will be filled with zeroes.

Write a Java class with a `main` method that creates an object of class `ArrayCopier` and uses its copy method to copy some arrays in all three cases:

- Both arrays are of the same size.
- The source array is longer.
- The source array is shorter.

For this exercise, please do not use existing methods for copying arrays from the Java API.

4 Creating matrices

Create a class `Matrix` that has a 2-D array of integers as a field (which should of course have **`private`** visibility as a best practice). The class should have the following constructors and instance methods:

- a constructor `Matrix(int,int)` setting the size of the array as two integers (not necessarily the same). All elements in the matrix should be initialised to 1.
- a method `setElement(int,int,int)` to modify one element of the array, given its position (the first two integers) and the new value to be put in that position (the third integer). The method must check that the indices are valid before modifying the array to avoid an `ArrayIndexOutOfBoundsException`. If the indices are invalid, the method will do nothing and the third argument will be ignored.
- a method `setRow(int,String)` that modifies one whole row of the array, given its position as an integer and the list of numbers as a `String` like `"1,2,3"`. The method must check that the index is valid and the numbers are correct (i.e., if the array has three columns, the String contains three numbers). If the index or the String is invalid, the method will do nothing.
- a method `setColumn(int,String)` that modifies one whole column of the array, given its position as an integer and the list of numbers as a `String` like `"1,2,3"`. The method must check that the index is valid and the numbers are correct (i.e., if the array has four rows, the `String` contains four numbers). If the index or the `String` is invalid, the method will do nothing.
- a method `toString()` that returns the values in the array as a `String` using square brackets, commas, and semicolons, e.g., `"[1,2,3;4,5,6;3,2,1]"`.
- A method `prettyPrint()` that prints the values of the matrix on screen in a legible format. Hint: you can use the special character `'\t'` (backslash-t) to mark a tabulator so that all numbers are placed in the same column regardless of their size. You can think of a tabulator character as a move-to-the-next-column command when printing on the screen.

Create a Java program in a separate class that uses all those methods from the `Matrix` class: creates matrices, modifies its elements (one-by-one, by rows, and by columns), and prints the matrix on the screen.

5 One-liners for matrices (*)

In your `Matrix` class, write a method `setMatrix(String)` that takes a `String` representing the numbers to be put in the elements of the array separated by commas, separating rows by semicolons, e.g., `"1,2,3;4,5,6;7,8,9"`.

6 Symmetry looks pretty

Make a class `MatrixChecker` with three instance methods:

- `isSymmetrical(int[])` takes an array of `int` and returns `true` if the array is symmetrical and `false` otherwise. An array is symmetrical if the element at [0] is the same as the element at [length-1], the element at [1] is the same as the element at [length-2], etc.
- `isSymmetrical(int[][])` takes a two-dimensional array of `int` and returns `true` if the matrix is symmetrical and `false` otherwise. A matrix is symmetrical if `m[i][j] == m[j][i]` for any value of `i` and `j`.
- `isTriangular(int[][])` takes a two-dimensional array of `int` and returns `true` if the matrix is triangular[1] and `false` otherwise. A matrix is triangular if `m[i][j] == 0` for any value of `i` that is greater than `j`.

The methods in class `MatrixChecker` do not know of our class `Matrix`. We want our class `Matrix` to be able to do the corresponding checks as well without having to write the same code again.

Add suitable methods to your `Matrix` class from the other exercises to allow the user of class `Matrix` to perform such checks on objects of type `Matrix`. Your methods should internally use the methods from `MatrixChecker`. (Hint: these methods will internally need to create objects of type `MatrixChecker`.)

7 From String to array of Strings

Create a `static` method split that takes a `String` `s` and an `int` `n` and returns an array of with entries of type `String`. The returned array should contain a split version of the input `String`, split into substrings of `s` that follow each other in `s` and that all have length `n`. The only exception to the length restriction is the last element of the array, which can have length at least 1 and at most `n` because the `String` does not necessarily have a length that is a multiple of `n`.

For example, the method call `split("Programming", 3)` returns an array `["Pro", "gra", "mmi", "ng"]`.

If `s` is `null` or has length 0 (two different cases!), or if `n` is not a positive number, the method should return the empty array.

[1] A matrix can be upper-triangular or lower-triangular, but just checking one of the two is fine for this exercise.

8 Instance counter

Note: This exercise is designed to help you practice the meaning of **`static`** fields. However, remember that adding static fields to your classes is usually a bad idea. Do not do it unless you have a very good reason. Hint: **never** do it in Programming in Java, unless you are defining constants (more on constants soon).

Complete the example given in the notes with a class called `Spy`. Your class must have:

- one and only one **`static`** variable, an **`int`** called `activeSpyCount`.
- an instance variable of type **`int`** for the spy's ID.
- an instance variable of type **`boolean`** that indicates whether the spy is active.
- a constructor that receives the ID of the spy as an argument, increases `activeSpyCount` by one, sets the spy's active status to **`true`**, and prints on the screen the ID of this spy plus the number of currently active spies.
- a `retire()` method that prints on the screen "Spy XX has retired" (where XX is the spy's ID), decrements the spy counter, sets the spy's active status accordingly, and prints on the screen the total number of active spies.
- a main method in which several objects of class `Spy` are created and some of them no longer active (their method `retire()` is called).

Observe how the static variable is accessed by different objects both to increment and to decrement it.

9 Big enough (*)

Write a small program that asks for the names and IDs of all employees in a small company, and store them in an array of integers and an array of Strings. (You will need to create a Java class to hold the arrays, and to access them).

This is similar to the example from the notes, but you do not know the number of employees in advance. Read the names and IDs of employees until the user enters an empty name (i.e., length 0) or an ID equal to 0.

Once you have finished reading employee data, go through the employee list and print the names and IDs of those employees whose ID is even or their names start with 'S'.

(Hint: As you do not know how many employees you need in advance, you will need a "growing" array. However, in Java an array cannot be resized after it has been created. The solution is first to create a small array. If it gets full, create a new array twice as big, copy all data to the new array, and assign the member variable that so far pointed to the old array to the new array, etc).