UNIVERSITÉ DU LUXEMBOURG

Programming 1

Patrick Keller (patrick.keller@uni.lu)

Lab 4 – Switches, Arrays and Loops

Submission (Preliminary: 24/10, Final: 31/10)
The mandatory exercises for **Code Submission** are **1, 3, 5** (marked with an ❗). All other exercises on this sheet are optional but still highly recommended! The **Explainer Video** 🎥 for this sheet must be realized on **Exercise 5 ("TicTacToe")**. A Flipgrid invitation link will be posted on Moodle.

Exercise 1 – ❗ Beverage Selection

In **Beverages.java** you will find a program that suggests different types of beverages to clients of a restaurant depending on the selected menu and whether the client is an adult. This information is read from standard input. The existing program heavily relies on `if-then-else`-statements. To improve the readability and maintainability of the code, we will modify it in two different ways:

1° Create a new program **BeveragesClassicSwitch.java** based on the existing **Beverages.java**. Use a **single classic switch** statement to replace the convoluted `if-then-else` structures from the original code. Use ?: inside the switch if helpful.

2° Create another program **BeveragesSwitchExpression.java** again based on the existing **Beverages.java**. This time replace the `if-then-else` blocks by a **single switch expression**. Use ?: inside the switch expression if helpful.

Exercise 2 – Navigation

#StaticArrayInitialization

In Lab sheet 02 we have seen an exercise where we checked the driving speed based upon road type ids provided by the user.

Road Type	Type id	Speedlimit
motorway	1	130
ordinary	2	90
town	3	50
calmed	4	30

This time we assume we have a navigation software that provides a list of these road type ids for the suggested route to our destination. Each entry in the list gives the road type of a road segment from start to destination.

1° Create a program **Navigation.java** that uses static initialization to create an array storing the following list of road types for each road segment of our route:
4, 3, 1, 2, 1, 3, 3, 4, 2, 4

2° Extend the program to output the list of road types in a textual manner instead of the road type ids. Do so by using a for-loop and a **single `System.out.println()` combined with a switch-expression**.

3° Now assume that we are driving exactly for 1 km per road segment (i.e. per entry of the list). Extend the program to calculate and output the minimal driving time for that route. (assuming we drive exactly at the allowed speed per segment and do not face any traffic jams, red lights, etc. and do not need any time to speed up to the allowed speed limit.
Hint: Driving 1 km at 130km/h takes 1/130hours = 6/13 minutes.

Exercise 3 – ❗ Palindrome

A Palindrome is a word that is read forwards the same as backwards, e.g. noon, radar, level, ...

1° Write a program **Palindrome.java** that reads a sequence of chars from standard input, one-by-one, until the user enter 0 (zero, which is not a part of the char sequence!). After reading the sequence, the program verifies and outputs whether the input sequence represents a palindrome or not. (i.e. in this part NoOn is not considered a palindrome!)

2° Write another program **PalindromeCaseInsensitive.java** that extends your previous program such that you can also detect palindromes that have mixed case characters (e.g. we want to consider NoOn to be a palindrome too!)

Exercise 4 – Vectors and Matrices

In this exercise we consider an N-dimensional euclidean vector space. (the vector space you know from high-school)
Each of the following programs will **first read the dimension N** of our vector space.

1° Write a program **Inverse.java** that reads a N-dimensional vector v from standard input and calculates and outputs the inverse of the vector. $(-v)$

2° Write a program **Norm.java** that reads a vector and outputs the euclidean norm of the vector.

3° Write a program **DotProduct.java** that reads two vectors and outputs their dot product.

4° Write a program **MatrixVectorMultiplication.java** that reads an NxN-dimensional matrix m and an N-dimensional vector v and outputs their multiplication. (m $\times$ v)

Exercise 5 – 🎥❗ TicTacToe

#ArrayAccess #ArrayCreation #ArraySubscriptOperator #ArrayType

1° Write a program **TicTacToe3X3.java** that uses a (two-dimensional) array to represent the 3x3 game field. The type you use for the array is left to your choice but keep in mind you need to represent 3 different game field states: empty, cross & circle.

2° Extend the previous program and add the game logic that runs for 9 rounds. (i.e. until the game field is completely filled up) Player "circle" is starting and players will switch at each round. Each round, read from standard input the X and Y coordinates the player wants to place it's mark. After the round is finished, print the updated game board to standard output (use tabs "\t" to align the columns).

3° Now extend your program to check after each round whether a player has won already. Modify the game logic such that the game ends once a player has won instead of completing all 9 rounds. Additionally, announce the winner to standard output.

Programming 1

Patrick Keller (patrick.keller@uni.lu)

Lab 4 – Switches, Arrays and Loops

Exercise 6 – Sieve of Eratosthenes

The *sieve of Eratosthenes* is an ancient method to find all prime numbers up to a specified number. Create a program **PrimeSieve.java** that implements the algorithm as follows:

1° Create an array of 100 `booleans` that are all initialized to `true`.

2° Initially, let $p = 2$ (ignore 0 and 1), as 2 is the first prime number.

3° Set all fields of the array whose index is a multiple of p and greater than p to `false`.

4° Find the next index of the array, whose value is `true`, set p to this index.

5° Repeat steps 3 and 4 until $p^2 > 100$.

6° All of the remaining array indices with a `true` value are prime numbers. Write all of them to the console. The output should be rendered as a table. Also indicate the number of prime numbers less than 100.

Exercise 7 – TicTacToe AI

This time we will extend the TicTacToe game from the corresponding exercise in two ways. Create a new program **TicTacToeNXM.java** based on the **TicTacToe3X3.java** you wrote earlier.

1° Extend your solution of the previous exercise such that it requests to input the size of the playing field at the start of the program, rather than fixing it to 3×3. Note that the game field does not have to be square. The smaller dimension will define the number of signs in a line needed to win - alternatively you can also request the number of wining signs upon start.

2° Now extend the program such that the second player (cross) is played by an AI instead of a human player. To do that you can use the **nextInt(int bound)** method from the Random class provided by Java. (see JDK documentation)
You can let the AI play randomly or try to make it smarter. (e.g. checking if it can finalize a winning line or blocking you from not finalizing yours.)

Exercises marked with a light bulb (like this one!) are slightly more advanced and could be too complicated for some beginners. We encourage you to try them but do not get frustrated if you do not succeed to complete them.