Programming 1

Christian Grévisse (christian.grevisse@uni.lu)
Gilles Neyens (gilles.neyens@uni.lu)

Lab 5 – Arrays & Strings in Java

✎ Working with `Strings`

The `String` class from the `java.util` package represents character strings, i.e. a sequence of characters. Since your very first Java program back in Lab 1 – Exercise 1, you have already worked with `String` *literals*, such as `"Hello, World!"`.

String literals are surrounded by double quotes (`""`, e.g. `String favoriteCourse = "Programming 1"`), whereas `char` literals are surrounded by single quotes (`''`, e.g. `char favoriteLetter = 'c';`). The *empty string* is just a pair of double quotes with no character in between (i.e. `""`). As you have already seen during the passed labs, you can concatenate strings with the `+` operator, e.g. `"Hello, " + "World!"`. The backslash notation allows to escape a certain character in order to introduce a special character, such as a tab (`\t`) or a new line (`\n`).

There is quite a bunch of interesting methods defined in the `String` class:

- `int length()`: Returns the length of this string.
- `char charAt(int index)`: Returns the char value at the specified index.
- `boolean equals(Object anObject)`: Compares this string to the specified object.
- `String substring(int beginIndex, int endIndex)`: Returns a string that is a substring of this string.
- `int compareTo(String anotherString)`: Compares two strings lexicographically.
- `String toUpperCase()`: Converts all of the characters in this string to upper case using the rules of the default locale.
- `String trim()`: Returns a string whose value is this string, with any leading and trailing whitespace removed.
- `String replace(char oldChar, char newChar)`: Returns a string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
- `String[] split(String regex)`: Splits this string around matches of the given regular expression.
- `String[] valueOf(<primitiveType> param)`: Returns the string representation of the primitive-type argument.
- ... and many more![a]

All classes in Java inherently implement the `String toString()` method (defined at the level of the `Object` class, the superclass of all Java classes), which gives a textual representation of an object. If not explicitly implemented by a class, the `toString()` method invoked on an object returns by default the name of the class and the hash code of the object. When a class *overrides* this method, i.e. gives a custom implementation, it usually gives a concise but informative textual representation of the object.

```
1  public class Employee {
2    String name;
3
4    // constructor, getter, setter, ...
5
6    @Override
7    public String toString() {
8      return "This employee's name is " + name;
9    }
10 }
```

The `toString()` method is called whenever a `String`-based representation of an object is needed, e.g. in the well-known method `System.out.println();`.

[a] For further information, please check the Java 8 API Documentation.

Programming 1

Christian Grévisse (christian.grevisse@uni.lu)
Gilles Neyens (gilles.neyens@uni.lu)

Lab 5 – Arrays & Strings in Java

Part I – Arrays

Exercise 1 – Vector

1° Create a class `Vector` representing a vector in an n-dimensional Euclidean space $\mathbb{R}^n$. The vector holds an attribute `values`, which is an array of real numbers.

2° Add a constructor which takes as parameter the size of the aforementioned array. This value is only used to initialize the array, but is not stored in any attribute in this class.

3° Add a method `randomFill()` which takes as parameters a minimum and maximum bound in order to set the different coordinates to random numbers comprised between those limits.

4° Add a method `getDimension()` that returns the dimension n of the vector.

5° Add a method `getElement()` that takes as parameter an index and returns the coordinate of the vector at that index. Make sure the passed parameter is sanitized in order to avoid an `ArrayIndexOutOfBoundsException`.

6° Add a method `setElement()` that takes as parameters an index and a new value to be set as the coordinate at the given index. Same sanity check here please.

7° Add a method `getNorm()` which returns the norm of the vector. Recall that for a vector $\mathbf{x} = (x_1, x_2, \dots, x_n)$ in Euclidean space $\mathbb{R}^n$, the norm is given by

$$\|x\| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

8° Add a method `Vector add(Vector v)`. The purpose of this method is to add to the current vector the one given as parameter and return the resulting one. Take care that the dimension of both vectors is equal, otherwise the method shall return `null`. This method shall use the methods `getElement()` and `setElement` in order to access a vector's coordinates.

9° Add a method `toString()` which returns a textual representation of the vector's coordinates in the form $(x_1, x_2, \dots, x_n)$. For your convenience, you might have a look at the method `Arrays.toString()`, maybe also `String.format()`.

10° Finally, write a main program that creates two vectors (the user can decide the dimension n), fills them with random coordinates between 0 and 100, calculates their sum and prints these 3 vectors as well as their norm.

Exercise 2 – Sieve of Eratosthenes

The *sieve of Eratosthenes* is an ancient method to find all prime numbers up to a specified number. Implement 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 3 – Matrix Transpose

1° Create a class `Matrix` with a two-dimensional array of integers, representing an $m \times n$ matrix $\mathcal{M}$. The dimensions are specified through constructor parameters.
2° Add a method `randomFill()` which fills all cells of the matrix with random numbers between 0 and 100 (inclusive).
3° Add a method `set(int i, int j, int value)` which allows to set the value of the cell $\mathcal{M}_{i,j}$.
4° Add a method `print()` that displays the matrix in the console.
5° Add a method `transpose()` that returns the transpose of the matrix. As a reminder:

$$B = A^T \implies \forall (i,j) \in \left\{1, \dots, m\right\} \times \left\{1, \dots, n\right\} \quad b_{i,j} = a_{j,i}$$

6° Write a main program to test your implementation.

Exercise 4 – Char Arrays

Consider the following char array:

```
1 char[] shortAlphabet = { 'a', 'b', 'c' };
```

1° Print to the console the sequence of chars **acba** using the above char array.
2° Add another element `'d'` at the end of the char array and then print it out. Remember that the length of an array cannot change, once it has been created. A statically initialized array is thus immutable, so you will need to create a new array, copy the content of the old one into it and add the new element.
3° Print out the *integer* value of the first element of either array.

Part II – Strings

Exercise 5 – Frequency Analysis

The purpose of this exercise is to count how many times characters from A to Z appear in a text. Frequency analysis is a simple technique used as an aid to break basic encryption algorithms[1].

1° Write a program that asks the user for a text. The `Scanner` class has a method `nextLine()` which returns a `String` of a whole line the user has entered (in opposition to the `next()` method, which tokenizes the input). As we are only assuming uppercase letters from the classical 26-letter Latin alphabet, you might need to clean the user's input:

```
1 String text = scanner.nextLine();
2 text = text.toUpperCase().replaceAll("[^A-Z]", "");
```

The second line transforms the entered text to uppercase and removes all characters not belonging to the set $\left\{A, B, \dots, Z\right\}$.
2° Create an integer array of length 26. Each element will represent the count of the corresponding letter (i.e. the first element holds the count of **A**, the second of **B**, ...). Establish the frequency analysis over the cleaned text.

[1] For further information, consult the related Wikipedia article. The course Introduction to IT Security in the 5th semester of the Binfo will cover a lot more details.

3° Print a table mapping letters to their frequency in the given text.

Exercise 6 – Caesar Cipher

In cryptography, substitution ciphers are a very basic way of encoding an original message (*plaintext*) into an encrypted message (*ciphertext*) by replacing letters through other letters. A particular example of substitution ciphers are Caesar Ciphers, where the alphabet gets shifted by a constant number. For instance, the Roman emperor Julius Caesar used a shift of 3, i.e. $a \rightarrow d, b \rightarrow e, \ldots, x \rightarrow a, y \rightarrow b, z \rightarrow c$. Note that "overflows" are avoided by restarting at a after z[2].

Write a program that reads a line of plaintext from the user, who also indicates the shift (e.g. 3)[3]. For obvious reasons, a shift that is a multiple of 26 should be avoided. Clean the entered plaintext as in the exercise before. Determine the corresponding ciphertext and print it to the console.

Exercise 7 – Uni.lu Student Number Verification

The student numbers at the University of Luxembourg respect the following schema:

$$\underbrace{0}_{\text{Semester}}\ \underbrace{12}_{\text{Year}}\ \underbrace{34567}_{\text{Identifier}}\ \underbrace{20}_{\text{Checksum}}$$

where the semester (either 0 or 1) stands for the first semester of enrollment (0 for winter semester, 1 for summer semester), the year is the first year of enrollment, followed by an actual identifier and a checksum. The checksum is a hexadecimal number, which explains why some student numbers end in a letter from `A` to `F`.

1° Create a class `StudentID` which holds an attribute `id`, initialized through the constructor via parameter. Add a getter for this attribute.

2° Add a method `getSemester()` which returns the first enrollment semester (as an `int`, either 0 or 1). You might need the method `Integer.parseInt()`.

3° Add a similar method `getYear()`. The returned `int` shall be greater than 2003 (the year of birth of the University of Luxembourg).

4° Add a method `getChecksum()` which returns the substring of the last two characters of the student ID.

5° Add a method `checkValidity()` which checks whether the student number is valid:

a) Take the first 8 digits of the student number and append a 1 in front of it.

b) Calculate the rest of this number modulo 97.

c) Transform this rest into a hexadecimal representation of 2 digits (respectively a digit and a letter):

```
1 String.format("%02x", rest)
```

d) The student number is valid iff the resulting checksum matches with the given one. You might need to transform one of them to lowercase (respectively uppercase).

6° **Bonus:** Improve the validation by not only checking the checksum, but also verifying that the semester and year are correct respectively realistic.

7° Write a main program that lets the user enter a student ID, which is then checked upon validity. An appropriate message is shown. If the ID is valid, also show the semester and academic year of first enrollment.

[2] Indeed, Caesar ciphers can be seen as a mathematical function mapping letters to the ring $\mathbb{Z}$ mod 26. Again, please stay tuned for the Introduction to IT Security course for further information.

[3] N.B.: Please read first the line, then the integer. Otherwise, you will need two calls to the `nextLine()` method. The reason for that will be explained in the second part of the course, on the topic of I/O streams.

✎ Varargs & Command-line arguments

In the *Java Language Specification*, we can read that:

> *The method `main` must be declared `public`, `static`, and `void`. It must specify a formal parameter (§8.4.1) whose declared type is* ***array of `String`****. Therefore, either of the following declarations is acceptable:*

```
1 public static void main(String[] args)
2 public static void main(String... args)
```

Latter declaration introduces *varargs*, i.e. the three periods (...) after `String`. Varargs[a] in general can be used to allow the user to specify an arbitrary number of arguments. However, varargs in a method declaration can only be put as the final parameter. They can be iterated over as a regular array.

The `args` parameter of the `main` method in a Java program allows the user to pass *command-line arguments* (cf. Operating Systems 1) to the program, which get passed to the `main` method in the `String[] args` parameter.

[a] For further information, consult https://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html

Exercise 8 – Command-line arguments

Write a program that calculates and displays the sum for the numbers passed as command-line arguments. We assume that only integer numbers are passed, so you may need to rely again on the `Integer.parseInt()` method. You do not need any `Scanner` object, only rely on the varargs parameter of the `main` method. You should be able to launch the program from the command line by typing

```
$ javac Sum.java
$ java Sum 42 16 9 10 25 29 11
142
```

Alternatively, you can also pass command-line arguments to a Java program in Eclipse by right-clicking on the Java class in the *Package Explorer*, selecting Run As > Run Configurations... . If you already ran the program beforehand, a dedicated run configuration should already exist, if not, select Java Application in the left list and press the New button. Under the tab Arguments, in the textfield Program arguments, enter a list of space-separated numbers, press Apply and Run[4].

[4] For further information on run configurations, consult the Eclipse documentation.