Programming in Java (24/25)

– *Exercises Day 6* –

Learning goals

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

- Be able to declare and use constants.
- Understand what the "public interface" of a class is.
- Become familiarised with the use of JavaDoc and use JavaDoc consistently for your classes and `public` constructors and methods (i.e., their public interface).
- Understand when to use `.equals()` and when to use == for the comparison of two expressions.
- Understand how the `do ... while` loop structure works and when to use it instead of a `while` loop.
- Be able to create Java `interface` types.
- Be able to implement Java `interface` types in your own classes (using the `implements` keyword).
- Be able to write code that calls methods from Java `interface` types (it works just like calling methods from classes).
- Put classes into packages

1 Public interface of a class

Consider the following class:

```java
public class Employee {
    private long id;
    private String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public long getSalaryInPence() {
        long l = calculateSalary();
        return l;
    }

    private long calculateSalary() {
        // slightly unusual approach to implementing this functionality
        return name.length() + id;
    }

}
```

What is the public interface of this class?

2 `do { practice; } while (!understood);`

Make a class with a method that reads a list of marks between 0 and 100 from the user, one per line, and stops when the user introduces a -1. The program should output at the end (and only at the end) how many marks there were in total, how many were distinctions (70–100), how many were merits (60–69), how many were passes (50–59), how many failed (0–49), and how many were invalid (e.g., 150 or -3). **Use a method of class Scanner exactly once**. The output may look similar to this example:

```
Input a mark: 45
Input a mark: 63
Input a mark: 73
Input a mark: 101
Input a mark: 45
Input a mark: 18
Input a mark: 92
Input a mark: -1
There are 6 students: 2 distinctions, 1 merit, 0 passes, 3 fails (plus 1 invalid).
```

3 Practice with `interface` types, part 1

Write a Java `interface` called `Measurable` that specifies a method called `getMeasure()` that returns a `double` value as the "measure" of an object.

Document your code using JavaDoc comments as good practice (also in all future programming exercises).

4 Practice with `interface` types, part 2a

Continuing the previous exercise, write a new class `MeasureSummer` with a `static` method

```
public static double sumMeasures(Measurable[] measurables)
```

that returns the sum of all the measures of the objects in the given array.

If the array reference `measurables` is the `null` reference, or the array contains `null`, your method may have arbitrary behaviour (i.e., your code should compile, but it is allowed to crash with some error message when you run it with these inputs).

Does your class `MeasureSummer` compile with the code that you have written so far?

Can you test your class `MeasureSummer` just by writing a `main` method in a new class, say `MeasureSummerRunner`? If so, what are the limitations?

5 Practice with `interface` types, part 2b

Improve your method from the previous exercise so that it returns `0.0` if the given array reference is the `null` reference and so that it skips `null` *entries* in the array (which itself would not be `null`).

For example, if you are given an array that has just two entries, one object of type `Measurable` with a measure of `12.0` and one `null` reference, your method should return `12.0`.

6 Practice with `interface` types, part 3

Revisit class `Point` from today's notes and make it implement the interface `Measurable`. Here we define the measure of an object of class `Point` as:

$$\sqrt{x^2 + y^2}$$

where `x` and `y` are the fields of class `Point`. Recall that you can use the method `Math.sqrt(double)` to calculate the square root of a non-negative number.

Write a class `MeasureSummerRunner` with a `main` method that creates a few arrays of type `Measurable[]` and passes them to the `sumMeasures(Measurable[])` method. Some of these arrays should also contain instances of class `Point`.

7 Practice with `interface` types, part 4

Now that you have created an implementation of interface `Measurable` in class `Point`, consider the following code snippet:

```
1        Point p = new Point(3, 4);
2        System.out.println( p.getMeasure() );
3        System.out.println( p.getX() );
4        Measurable m = p;
5        System.out.println( m.getMeasure() );
6        System.out.println( m.getX() );
```

What do you expect will happen?

Now try to compile the code. What happens? Why?

8 Practice with `interface` types, part 5

Write another class `MeasurableString` that implements the interface `Measurable`. Its constructor should have one parameter, of type `String`. The method `getMeasure()` that is required by the implemented interface `Measurable` should return the length of the `String` that was passed to the constructor, converted to a **`double`** value. Other than that, you do not need to write any methods in class `MeasurableString`.

Now write code in your class `MeasureSummerRunner` so that you create an array of type `Measurable[]` that contains some instances of class `Point` and some instances of class `MeasurableString`. Run method `sumMeasures(Measurable[])` with your array with the "mixed" entries as its argument.

What happens? Why?

9 Wrapping up into packages

Create a package `pij.day6.measure` and put the interface `Measurable` and the classes `MeasureSummer`, `Point`, and `MeasurableString` that you created or modified in Exercises 3 – 8 into this package. Now create a second package `pij.day6.runner` and put your class `MeasureSummerRunner` that you wrote in Exercises 4 – 8 into this package. Compile your code. Does your class `MeasureSummerRunner` still run as before from its `main` method?

10 Constants vs magic numbers

In this exercise, we want to practise the *refactoring* of code, i.e., rewriting code so that is becomes more readable and maintainable for other programmers (and for us).

The railway company *ACME Railways* is operating a scheme under which passengers can get a partial or full refund when a train is delayed.

If the passenger did not pay for the journey, there is never a refund. If the delay is less than 15 minutes, there is no refund. If the delay is at least 15 minutes, but less than 30 minutes, there is a refund of 15% of the original ticket price. If the delay is at least 30 minutes, but less than 60 minutes, the refund is 30 percent of the original ticket price. If the delay is 60 minutes or more, the refund is the full ticket price.

Imagine that one of your classmates wrote the following class offering a **`public`** method to calculate the refund:

```java
/**
 * Provides a method to calculate the refund in pence as a function
 * of the original ticket price and the delay of the train service.
 */
public class RefundCalculator {

    /**
     * Calculates the refund in pence given the original ticket price
     * and the delay.
     *
     * @param priceInPence the original price of the ticket in pence
     * @param delayInMinutes the delay of the train service in minutes
     * @returns the refund awarded in pence
     */
    public static int getRefundInPence(int priceInPence, int delayInMinutes) {
        if (priceInPence <= 0) { // no refund if the customer
            return 0;            // did not pay for the ticket
        }
        int percentage = delayToPercentage(delayInMinutes);
        int refundInPenceTimesHundred = priceInPence * percentage;
        int refundInPence = refundInPenceTimesHundred / 100;
        if (refundInPenceTimesHundred % 100 != 0) {
            // round up, so instead of 123.2 pence, we refund 124 pence
            refundInPence++;
        }
        return refundInPence;
    }

    /**
     * Converts the delay to a percentage of the price.
     *
     * @param delayInMinutes the delay of the train service in minutes
     * @returns the percentage of the ticket price for the refund
     */
    private static int delayToPercentage(int delayInMinutes) {
        if (delayInMinutes < 15) {
            return 0;
        }
        if (delayInMinutes < 30) {
            return 15;
        }
        if (delayInMinutes < 60) {
            return 30;
        }
        return 100;
    }
}
```

Consider the concept of "magic numbers" in the sense of fixed numerical values in the code that have no name to explain their purpose:

https://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants

Magic numbers make code hard to read and to maintain (i.e., update as needed when the requirements change or a bug is found).

In the above code, your task is to introduce suitable constants and to replace the magic numbers in the code by these constants.