Java - elements of OOP (`II`)

Working environment setup

1. Download and unzip `lab02` source code

    1. Download `lab02.zip` from the course site (moodle)
    2. Unzip it (you get `lab02` directory)
    3. Move `lab02` to `programming-in-java` directory, i.e.,
        - `programming-in-java`
            - `lab00`
            - `lab01`
            - `lab02` <--
            - `gradle`
            - ...

2. [`IntelliJ`] Add `lab02` module to the `programming-in-java` project

    1. In the *Project* window click `settings.gradle` file to open it

    2. Modify its content to the following:

```
rootProject.name = 'programming-in-java'
include 'lab00'
include 'lab01'
include 'lab02'
```

    3. Save the file

    4. Click `Load Gradle Changes` (a small box in the top right corner)

1) Final variables, methods, and classes

Analyse the source code in packages:

- `lst02_01` (final variables)
- `lst02_02` (final methods)
- `lst02_03` (final classes)

Exercises

1. Look briefly at the chapters of `Java Language Specification` related to *final variables*, *methods*, and *classes*
2. Explain the benefits of using constants in programming
3. Explain the meaning of keyword `final` when used for:
    - local variables
    - instance variables
    - static constants
    - methods
    - classes
4. Answer the following questions:
    - If a class contains only private data fields and no setter methods, is the class immutable?
    - If all the data fields in a class are private and of primitive types, and the class does not t contain any setter methods, is the class immutable?

5. Explain why the following class is not immutable

```
public class C {
   private int[] values;

   public int[] getValues() {
      return values;
   }
}
```

6. [`c`] Refactor the source code to `one file-one class` structure

2) Enumeration classes (enums)

Analyse the source code in package `lst02_04`

Exercises

1. Look briefly at the chapters of `Java Language Specification` related to *enum classes*
2. Familiarise yourself with `java.lang.Enum` class

3. Analyse the following implementation of the singleton design pattern

```
enum EnumBasedSingleton {
   INSTANCE;
   int value;

   public int getValue() {
      return value;
   }

   public void setValue(int value) {
      this.value = value;
   }
}
```

Is this correct? Thread-safe?

4. [c] Refactor the source code to `one file-one class` structure

3) Nested classes

Analyse the source code in package `lst02_05`

Note: nested classes: static and non-static member classes, local classes, and anonymous classes

Exercises

1. Look briefly at the chapters of `Java Language Specification` related to *Nested classes* (a *static* and *non-static member classes*, *local classes*, and *anonymous classes*)
2. Explain the main differences between the types of nested classes available in Java
3. Answer the following questions:
   - Can an inner class be used in a class other than the class in which it nests?
   - Can the modifiers public, protected, private, and static be used for inner classes?
4. [c] Refactor the source code to `one file-one class` structure

4) Abstract Data Types (abstract classes and interfaces)

Analyse the source code in packages:

- `lst02_06` (abstract classes)
- `lst02_07` , `lst02_08` , `lst02_09` (interfaces)

Exercises

1. Look briefly at the chapters of `Java Language Specification` related to *abstract classes* and *interfaces*
2. Explain the main differences between abstract classes and interfaces in Java
3. List possible types of members of Java interfaces
4. From the following list select the correct definition of an abstract class:

```
class A { abstract void m1() {} }
public class abstract B { abstract void m1(); }
class C { abstract void m1(); }
abstract class D { protected void m1(); }
abstract class E { abstract void m1(); }
```

5. From the following list select the correct definition of an interface:

```
interface A { void m1() {} }
abstract interface B { m1(); }
abstract interface C { abstract void m1() {} }
interface D { void m1(); }
```

6. Explain the output of running the class *Main*:

```
interface A {}
class C {}
class D extends C {}
class B extends D implements A {}

public class Main {
   public static void main(String[] args) {
      B b = new B();
      if (b instanceof A)
         System.out.println("b is an instance of A");
      if (b instanceof C)
         System.out.println("b is an instance of C");
   }
}
```

7. [c] Refactor the source code to `one file-one class` structure

5) Functional interfaces and lambda expressions

Analyse the source code in packages `lst02_10` and `lst02_12`

Exercises

1. Look briefly at the chapters of `Java Language Specification` related to *functional interfaces* and *lambda expressions*
2. Explain the relationship between lambda expressions and functional interfaces
3. Write functional interfaces that correspond to the following function types:
    - $void \rightarrow int$
    - $int \rightarrow void$
    - $int \rightarrow int$
    - $(int, int) \rightarrow void$

   and then implement them (any implementation that compiles is good) using:

    - anonymous classes
    - lambda expressions

   Note: any implementation that compiles is good
4. [c] Refactor the source code to `one file-one class` structure

6) Mini project 02_01 (`exc02_01`)

[c] The implementation of interface `StackOfInts` :

1. Add JavaDoc comments to the interface and all its methods
2. Add JavaDoc comments to `LinkedListBasedImpl` (the class itself and all its methods)
3. Complete the linked list based implementation - `LinkedListBasedImpl` .
   Notes:
    - use nested class `Node` as the linked list building block
    - use the simplest possible implementation of the linked list, i.e,
        - it can be *unidirectional*
        - only two operations are requried: adding and removing an element from the *front of the list*
4. Write unit tests for different cases
5. Add JavaDoc comments to `ArrayBasedImpl` (to the class itself and all its methods)
6. Write unit tests for different cases (i.e. apply a TDD-like approach)
7. Complete the array based implementation - `ArrayBasedImpl` .
   Notes:
    - it should be an *array of integers* (`int`), and not, for instance, `ArrayList<Integer>`)
    - the size of the array should grow and shrink (according to some strategy) as elements are `pushed` and `popped`

7) Push the commits to the remote repository