Assignment 4: Advanced Java

Introduction
1. The assignment is due on 14.6.15 at 23:55.
2. The TA in charge of this assignment is Helal Assi. Questions should be sent to the mail helal.assi@gmail.com with the subject “236703 HW4”.
3. Objective of this assignment: Getting to know Java programming on a deeper level, using Reflection tools and Annotations.
4. Read the instructions carefully, both in this document and in the provided code.
5. Make sure to write code that is clear, readable and reasonably documented.
6. Execution speed is not a central issue in the homework assignments in this course. In any case of uncertainty between simplicity and performance, choose the simpler implementation.
7. Try to avoid code duplication – use methods you’ve already written as much as possible.
8. In order to avoid mistakes please read the FAQ section in the course website which will be updated on an ongoing basis.

Preface
In the tutorials we briefly covered the SUnit testing framework in Smalltalk and similarly, the JUnit framework in Java. These frameworks are implemented around the idea of Unit Testing, which had an important role in the development of the Test Driven Development process. Unit Tests are tests for small individual units of source code (usually classes), that verify correctness of the unit. The tests are generally run after creation of a class, or after the source code is changed, though there are also ways to run tests before the class is fully created. The idea is to create a group of test cases that cover all the intended functionality of the class, in such a way that running the tests successfully will prove to a reasonable degree that the class is correct.

Unit Tests for software have been used since the start of procedural programming paradigm. The programmer would often write testing software that was not part of the application in order to check the correctness of a procedure. The Unit Testing process could also be used for modular programming, and for every popular programming paradigm since then; however its main weakness was the difficulty in automatically running a comprehensive set of tests. Since the first decade of the 21st century the use of auxiliary packages for Unit Testing (testing frameworks) has grown, enabling creation of a common pattern for all unit tests. Additional auxiliary libraries enabled increasing isolation of each separate test, so that tests could be run completely independently of other tests, units, and external dependencies (example – a database).

Writing Unit Tests using auxiliary packages enables full automation of their execution. These packages offer ways to indicate whether a test has succeeded or failed. This makes it easy for a
programmer to run a set of tests that check the classes they wrote with a single command; and to keep track of the results.

In this assignment you will implement a Unit Testing package in Java by writing a new class called **OOPUnitCore** that will run the tests and keep track of the results.

### Section A – Defining the Annotations

You must define five new annotations: **OOPTestClass**, **OOPSetup**, **OOPBefore**, **OOPAfter** and **OOPTest**. Each annotation should be defined using the meta-annotations Target and Retention as needed. The definition of these meta-annotations must be *exact*, depending on the purpose of the new annotation.

**OOPTestClass**

An annotation that should be used only for *types*. The purpose of **OOPTestClass** is to indicate that a class contains unit tests. A class marked with this annotation will be called a “test class”. This annotation should have a single Boolean property called *value*. A ‘true’ value indicates that tests in the class should be run in an order that will be defined by the **OOPTest** annotation. A ‘false’ value indicates that the order of the tests is not important. The default should be ‘false’.

**OOPSetup**

An annotation that is only used to mark methods. The purpose of the annotation is to indicate that the marked method should only be run once, before running the tests. The purpose of a method marked with this annotation is to initialize fields that only need to be initialized once (used mostly for “heavy” operations such as initialization of a database and so on). A method marked with this annotation will be called a “setup method”.

**OOPBefore**

An annotation that is only used to mark methods. The purpose of the annotation is to indicate that the marked method should be run before all of the test methods whose names are held in a field in the annotation. The purpose of a method marked with this annotation is to enable tests to run independently of each other (used mostly to initialize fields read by one of the methods that will run, to ensure the value was not influenced by a previous test). The annotation should have a single property called *value* of type String[] that will hold the list of test methods before which this method should be run.

**OOPAfter**

An annotation that is only used to mark methods. The purpose of the annotation is to indicate that the marked method should be run after all of the test methods whose names are held in a field in the annotation. The purpose of a method marked with this annotation is to enable the release of resources that were used during the tests. The annotation should have a single
property called `value` of type `String[]` that will hold the list of test methods after which this method should be run.

**OOPTest**

An annotation that is only used to mark methods. The purpose of the annotation is to indicate that the marked method is a test method. The annotation can receive an instance of the class `Class` that represents an exception as an argument. In this case the test passes only if the exception was thrown by the test. The **annotation should have three properties: one called `test_throws`, of type `Boolean`, that will hold true if the test is expected to throw an exception and false otherwise. The second property is called `exc`, of type `Class< ?>`, that will hold the exception the test should throw; the third is called `order`, of type `int`. The third property indicates the order in which the test methods should run. By default, test methods should not throw exceptions in order to succeed.**

**The Annotations must be defined EXACTLY as they are described above!**

**Section B – Defining the Annotations**

2.1 **OOPResult**

We will use the interface `OOPResult` to represent the result of a single test method. (The interface is provided to you with the assignment)

The enum `OOPTestResult` is defined in `OOPResult`. `OOPTestResult` has three possible instances: `SUCCESS`, `FAILURE`, `ERROR`

You are required to implement `OOPResult` in a class of your own. You must implement the following methods:

- **public OOPTestResult getResultType()**
  This method returns the enum described above, representing the result of the test method.

- **public String getMessage()**
  This method returns a message according to the result of the test.

- **public Boolean equals(Object obj)**
  You must implement the equals method. The equality should be defined according to the return values of the methods `getResultType()` and `getMessage()`.

**How is the result of the test decided?**

When a test method is run, we say that it **succeeded** if the test method did not throw an exception (and was not supposed to throw an exception), or if an exception declared in the `OOPTest` annotation was thrown. In this case, an object implementing `OOPResult` that describes the results of the test will hold `OOPTestResult.SUCCESS`, and its message will be null.
If an (unexpected) exception was thrown, check the type of the exception:

- If the exception was of type `OOPAssertionError` (from the provided code), an object implementing `OOPResult` that describes the result of the test will hold `OOPTestResult.FAILURE`, and its message will be the message of the exception (accessed using the method `getMessage()`).
- If the exception was not of type `OOPAssertionError`, an object implementing `OOPResult` that describes the result of the test will hold `OOPTestResult.ERROR`, and its message will be the name of the exception’s class.

**Example:**

If a method test was run, and an exception `e` of type `OOPAssertionError` was thrown, the matching message will be the message returned by `e.getMessage()`.

### 2.2 OOPTestSummary

The class `OOPTestSummary` will be used to hold the results of running a number of test methods. `OOPTestSummary` contains a mapping that matches each test method to its result. This mapping is a map from the method name to an object that implements `OOPResult`, which describes the method’s result. You must define the class `OOPTestSummary` with the following methods:

- `OOPTestSummary (Map<String, OOPResult> testMap)`
  A constructor that receives a map between method names and the results of running these methods.
- `int getNumSuccesses()`
  Returns the number of successful test methods.
- `int getNumFailures()`
  Returns the number of test methods that failed because of an `OOPAssertionError` exception.
- `int getNumErrors()`
  Returns the number of test methods that failed because of a different exception.

### Section C – Defining OOPUnitCore

In this section, you will define the central class, `OOPUnitCore`. This class contains only static methods, which define the required functionality for running unit tests. Define the following two methods in `OOPUnitCore`:

- `void assertEquals(Object expected, Object actual)`
  This method must check whether it got two objects that are equal by value. If not, it should throw `OOPAssertionError` (From the provided code).
- `OOPTestSummary runClass(Class<?> testClass)`
This method receives a class. If the parameter is null, or the class is not a test class (The class is not tagged with the OOPTestClass annotation you defined) then an IllegalArgumentException should be thrown.

**What does `runClass` do?**

a. The method creates a new instance of the received class. You may assume that the class has a constructor that receives no parameters (not necessarily public).

b. The method should run all setup methods (OOPSetup) in the class and in its parent classes. The object on which these methods should be executed is the object created in step a.

c. The method should run all of the test methods in the class and in its parent classes. The object on which these methods should be executed is the object created in step a.

d. Before running each test method, this method should run all methods in the class and its parent classes that are marked with OOPBefore, for which the method’s name appears in the list of methods in OOPBefore’s value field (as described above).

e. After running each test method, this method should run all methods in the class and its parent classes that are marked with OOPAfter, for which the method’s name appears in the list of methods in OOPAfter’s value field (as described above).

f. The method should return an object of type OOPTestSummary that describes the run results for all of the method tests, as described above.

**Backup and Restoration of the Object State When Needed**

In order to prevent the case where OOPBefore\OOPAfter methods throw exceptions and so leave the object in an incorrect state, we will create a backup of the object fields before the test method (before running “before” methods) and will restore it if necessary. For simplicity’s sake, the backup need only be of the fields of the object itself, and not of its inherited fields. In order to maximize the chance that the backup will hold values that won’t be influenced by the method, it must be constructed using the following priority:

1. If the object in a field supports “clone”, the backup will hold a clone of the object.
2. If the object in a field has a copy constructor (a constructor that receives an object of the same type as the only parameter), that constructor will be used to create a new object of the same type, and the new object will be saved in the backup.
3. If neither of the previous holds, the object in the field itself will be used in the backup.

For example, if the object on which the test method is being run has a field of type java.util.Date (which implements Cloneable), the backup will hold a copy created using a call to clone. If it holds a field of type String, the backup will hold a copy created using the String(String s) constructor.
The result of restoring the object from the backup is that all fields for which one of the first two cases hold will point to different objects than those they pointed to before running the method, but the state of the pointed-to objects will be the same.

If a method tagged with OOPBefore throws an exception you must restore the state of the object from before the OOPBefore method was called, and then go on to run the next test. The same holds for the case where a method tagged with OOPAfter throws an exception (you must restore the state of the object from before the OOPBefore method was called). In the case where an exception is thrown in before\after methods, the result of the test should be OOPTestResult.ERROR.

**Additional Requirements**

- In the case of inheritance, you must run the setup method of each of the classes, starting from the highest defining parent and down to the current class. Before methods should also be run in this order (from the parent down to the child). As opposed to this, the after methods should be run in the opposite order, from the current class and up through the inheritance chain to the highest parent.
- There is no specific order in which the OOPBefore methods in the same class must run. The same holds for the OOPAfter methods.
- You may assume that all of the methods in the test class do not receive parameters and have a void return value.
- You may assume that methods tagged with OOPSetup will not throw exceptions.
- Note that a method tagged with OOPBefore can run several times – it will run once before every test method in its method list. The same for OOPAfter.
- You may assume that every method is tagged with at most one of the defined annotations.
- You may assume that there will be at most one method in each class tagged with the OOPSetup annotation.

**General Notes:**

1. Make sure to use method/class/field names exactly as they are defined in the assignment and in the given files.
2. Avoid code duplication. Some of the functionality repeats itself, and can be encapsulated in a separate method.
3. Java’s Reflection API is simple and user-friendly. You will find getters for almost every type of data you wish to use. Use Eclipse’s automatic code completion to see what your options are at each point.
4. You are encouraged to use JUnit to test your solution – in the provided tests you can find an example of an exception being thrown. You are not required to use JUnit, and you may not submit your test code.
Submission Details

- Requests for postponement, for any reason, must be sent to the TA in charge of the course (Nurit). Note that the course has a late-submission policy, meaning you can submit your assignment late without a postponement approval. Details can be found in the course site under “General Info”.
- The assignment must be submitted electronically. (Save the submission confirmation).
- Your submission should consist of a zip file with the name format:
  
  OOP4_<ID1>__<ID2>.zip containing:
  
  - A file called readme.txt with the following format:
    
    name1 id1 email1
    name2 id2 email2
  
  - The zip file should hold all code files in the package "Solution".
- Do not use folders inside the zip, and do not hand in files from other packages.
- Points will be deducted for not conforming to the submission requirements (rar instead of zip, extra files, a readme file with the wrong name, etc.).

Good Luck!