Advanced Java 2018
Exercise

Learning goals. This exercise has three parts. The first part is about generics. The second part is about reflection while the third part is about JDBC. The three parts can be solved independently, and the learning goals are:

- Generic:
  - Writing and using generic classes and interfaces in Java.
  - Using wildcard and bounded type parameters to handle covariance and contravariance.
- Reflection:
  - Introspecting classes and instances using reflection.
  - Dynamically invoking methods and changing fields using reflection.
- JDBC:
  - How to use the Derby database engine in embedded mode.
  - How to use the JDBC API to
    - Connect to the database.
    - Send queries to the database.
    - Retrieve and process the results received from the database.

Format. This exercise set consists of two parts:

1. A problem description on 3 numbered pages, including this page. Make sure you have them all, and read this page first.
2. A zip archive containing a stub implementation of the exercise.

The implementation portion of the exercise text has three parts. Each part contains a number of questions which you must solve by filling out the missing parts of the stub implementation. Your only deliverable is the solution code, no report is required.

Hand in. This exercise doesn't count any point. No submission is required and the solution is released.

Happy hacking!

1. Generics

In this part of the exercise, you are asked to make the sensor dataflow network from assignment 2 generic. In assignment 2, you implemented a network consisting of sensors, sensor monitors and sensor monitor subscribers. In this assignment you must abstract sensor data and discomfort level warnings and make the data that flows to subscribers generic. Furthermore, sensors, monitors and subscribers will be instantiations of the same generic class, but with different type arguments.

A generic *dataflow node* is a type-indexed family of classes `Node<I, O>` parametrized by two type parameters:

`I` The input type of the node. Data of this type can be pushed to the node.

`O` The output type of the node. The node will push data of this type to the subscribers of the node.

For simplification reasons, a dataflow node is defined to be exactly one thread. Therefore, the dataflow node class extends `Thread`. We have supplied a dynamically checked implementation using raw types. See `exercise1.dataflow.DynamicCheckedNode`.

To define the processing logic of a node, each dataflow node contains a *processor* which is responsible for consuming inputs and producing output. Each input can produce zero or more outputs. This one-to-many relationship is modeled by returning an iterator of output elements for each input element. See the supplied interface `exercise1.processors.Processor` as well as the predefined processors in the same package.

a. Finish the implementation of `Node<I, O>` by supplying the missing generic type arguments inside the class. It should behave as `DynamicCheckedNode`, but should be statically checked using generics. Your implementation should have no type-casts and no generics-related compiler warnings.

b. Implement a harness like the one in assignment 2 using `Node<I, O>`. Use the following types:

   – Sensor type: `Node<StartSignal, SensorReading>`.
   – Monitor type: `Node<SensorReading, DiscomfortWarning>`.
   – Subscriber type: `Node<DiscomfortWarning, Object>`.

   You can use the predefined processors to make your life easy. See the `exercise1.dataflow.DataflowHarness` for inspiration.

c. Consider an extended sensor that produces `SensorReadingExtended`. If a monitor can subscribe to a sensor (of type `Node<StartSignal, SensorReading>`), it should also be able to subscribe to an extended sensor (of type `Node<StartSignal, SensorReadingExtended>`). Make sure this is true in your implementation. Hint: Use bounded wildcards.

d. Similar to question (c), make sure that two subscribers of type `Node<Object, Object>` and type `Node<DiscomfortWarning, Object>` can subscribe to the same monitor (producing `DiscomfortWarnings`).

2. Reflection

In this part of the exercise, we disregard all static type checking in Java and use it as a purely dynamically type checked language by using reflection. We will use a single class called `Box` which will represent any classes. See `exercise1.djava.Box`. You are only expected to support public fields and methods with no other modifiers (such as final and static) and only non-primitive types. Furthermore, you are not expected to handle generics in any way.

a. Implement `Box` using reflection by filling out the stubs. Each method has comments that shows you how it should be implemented. None of the methods should throw an exception. If an exception occurs, either return it in a new `Box` or update the boxed object to be the exception. If you are unfamiliar with the `...` notation, Google "Java varargs".

b. Play around with writing programs using only literals (string constant, integer constants) and your `Box` implementation. A minimum acceptable answer requires you to invoke all the different methods of `Box` at least once.

- For example, create your own employee class with a couple of different fields and methods, and use it using `Box`.

3. Persistent Employee Database

In this part of the exercise, you will have to implement the *exercise1.jdbc.EmployeeDB* interface. In order to implement the interface you will use Derby database engine in embedded mode[1]. You will have to implement the stubbed out methods in *exercise1.jdbc.DerbyEmployeeDB* class. We have also provided you with the test class *exercise1.jdbc.DerbyEmployeeDBTest* to implement the test cases. Tty to implement test cases to test the interfaces defined in EmployeeDB.

Implement the exercise1.jdbc.EmployeeDB interface

- You should use the Derby database engine (using JDBC) to implement the EmployeeDB interface. The *exercise1.jdbc.DerbyEmployeeDB* class loads the Derby database engine and creates a database called employeeDB (if it does not exist). Implement the following stubbed out methods in *exercise1.jdbc.DerbyEmployeeDB* class using JDBC:

  1. `addEmployee`: Adds an employee record in the employee database (employeeDB). The method throws an exception if an employee record is being added for a department that does not exist in the employee database. The list of departments which are handled by the database is passed when the database is instantiated in the constructor.
  2. `listEmployeesInDept`: Returns the list of employee records who belong to the departments passed as argument to this method in the employee database (employeeDB).
  3. `incrementSalaryOfDepartment`: Increments the salaries of the employees in the departments passed as argument by the amounts passed as argument. Ensure all/nothing semantics[2]. Possible causes of failure can be:

     a) One or many departments passed as argument do not exist in the employee database (employeeDB).

     b) The amount by which the salary is to be incremented is negative.

     This method throws exceptions to signal which of the above errors happened.
  4. `cleanupDB`: Resets the state of the employee database to an empty state (newly created employee database).

- Remember you will have to create the database schema (table layout/s) to store the employee records in the database (employeeDB). You are free to architect that as you wish. We have not implemented that for you.

Implement the test cases

- Implement the test cases for exercise1.jdbc.EmployeeDB interface in the exercise1.jdbc.DerbyEmployeeDBTest class.

[1]Derby is an Open source relational database engine which can be run inside the JVM in which the application executes. http://db.apache.org/derby/. The Derby jar files (http://dk.mirrors.quenda.co/apache//db/derby/db-derby-10.14.2.0/db-derby-10.14.2.0-bin.zip)have been provided in the lib directory of the exercise handout. Make sure you add the lib directory in your classpath and you have a database :-)

[2]Either all records are incremented or none are incremented in the employee database (employeeDB)