Assignment 3 (0.5 points) - Deadline Aug 24 23:59 hrs

Learning goals

This assignment has two parts. The first part is about generics and the second part is about reflection. The two parts can be solved independently, and the learning goals are:

- Part 1:
  - Writing and using generic classes and interfaces in Java.
  - Using wildcard and bounded type parameters to handle covariance and contravariance.
- Part 2:
  - Introspecting classes and instances using reflection.
  - Dynamically invoking methods and changing fields using reflection.

1. Generics

In this part of the assignment, 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 `assignment3.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 `assignment3.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 `assignment3.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 assignment 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 `assignment3.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`.