Java Final Test

Monday 27th April 2020

09:00 – 12:00

THREE HOURS

(including 10 minutes planning time)

Please monitor 161 on Piazza for announcements

There are **TWO** sections: Section A and Section B, each worth 50 marks.

Credit will be awarded throughout for code that successfully compiles, which is clear, concise, usefully commented, and has any pre-conditions expressed with appropriate assertions.

Important note:

- In each section, the tasks are in increasing order of difficulty. Manage your time so that you attempt both sections. You may wish to solve the easier tasks in both sections before moving on to the harder tasks
- It is critical that your solution compiles for automated testing to be effective. Up to **TEN MARKS** will be deducted from solutions that do not compile (-5 marks per Section). Comment out any code that does not compile before you log out.
- You can use the terminal or an IDE like IDEA to compile and run your code. **Do not ask on Piazza for help on how to use an IDE.**
- Before your final commit/push, you **must** ensure that your source code is in the correct directory, otherwise your marks can suffer heavy penalties. Only code in the original directories provided will be checked. Please make sure that your folder structure for each Section is:

  ```
  ~/javafinaltestpartX_username/.git/
  ~/javafinaltestpartX_username/lib/
  ~/javafinaltestpartX_username/src/
  ~/javafinaltestpartX_username/test/
  ```

  where X is "a" or "b" and username is your login.
- Before the 12:00 deadline, you will need to **commit/push** your code to GitLab and **submit** it via LabTS to CATe. Code that is pushed to GitLab after the deadline will be capped at **zero**.

Useful commands

```
cd ~/javafinaltestparta_username/

javac
-g
-d out
-cp "./lib/*"
-sourcepath src:test
src/**/*.java test/**/*.java

java
-ea
-cp "./lib/*:out"
org.junit.runner.JUnitCore generators.Question#Tests

java
-ea
-cp "./lib/*:out"
generators.TestSuiteRunner

cd ~/javafinaltestpartb_username/

javac
-g
-d out
-cp "./lib/*"
-sourcepath src:test
src/**/*.java test/**/*.java

java
-ea
-cp "./lib/*:out"
org.junit.runner.JUnitCore datastructures.StockStressMonkey

java
-ea
-cp "./lib/*:out"
org.junit.runner.JUnitCore datastructures.StockTest

java
-ea
-cp "./lib/*:out"
org.junit.runner.JUnitCore market.MarketTest

java
-ea
-cp "./lib/*:out"
testutils.TestSuiteRunner
```

Section A

Problem Description

In the field of automated software testing, test cases for a system under test are created programmatically by *test case generators*.

When writing a test case generator, it is useful to have a source of data elements for various data types. For instance, when generating tests for a system under test that processes binary trees it could be useful to have a source of random binary trees, or a method for systematically enumerating all binary trees up to a particular size.

In this section you will create Java classes and interfaces for generating objects of various types: integers, strings and pairs.

Getting Started

The skeleton files are located in the `generators` package. If you have cloned your repo under your Home directory, then you can find these files at:

- `~/javafinaltestparta_zz4319/src/generators`

The `generators` package contains one incomplete class, `Pair.java`, that you will populate. Do not change the name of this class. You should add further Java classes as instructed below. All new Java files that you create should be placed in the `generators` package.

Testing

You are provided with a set of test classes under:

- `~/javafinaltestparta_zz4319/test/generators`

There is a test class `Question`i`Tests.java` for each question i. These contain initially commented-out JUnit tests to help you gauge your progress during Section A. **As you progress through the exercise you should un-comment the test class associated with each question in order to test your work.**

These tests are not exhaustive and are merely intended to guide you: while it is good if your solution passes these tests, your work will be assessed with respect to a larger, more thorough test suite. You should thus think carefully about whether your solution is complete, even if you pass all of the given tests.

What to do

1. **Integer generators.**

   To start with you will create an interface for generating integers, and two classes that implement this interface.

   - Create an interface, `IntegerGenerator`, that declares two methods, `next()` and `hasNext()`. The return type of `next()` should be `Integer`, while the return type of `hasNext()` should be `boolean`. The intention is that, for any class implementing

the interface, `hasNext()` indicates whether another integer can be generated, and `next()`, which should only be called if `hasNext()` holds, generates an integer value.

- Create a class, `FixedIntegerSequenceGenerator`, that implements the `IntegerGenerator` interface. An object of this class type should generate the integers in the range $[0, 30)$ in order. After 29 has been generated, `hasNext()` should return *false*, and an `UnsupportedOperationException` should be thrown if `next()` is called again.
- Create a class, `MissingPrimesGenerator`, that implements the `IntegerGenerator` interface. An object of this class type should generate the positive integers in order, except that an integer should be skipped if it is a multiple of 2 or a multiple of 5, unless it is a multiple of *both* 2 and 5. Your implementation of `hasNext()` should reflect the fact that there is an endless supply of integers. (You need not worry about the fact that in reality there are only a finite number of distinct `Integer` objects in Java.)

Test your solution using (at least) the tests in `Question1Tests`.

[10 marks]

2. **String generators.**

   Next, you will create an interface for generating strings, and one class that implements this interface.

   - Create an interface, `StringGenerator`, whose methods are identical to those of `IntegerGenerator` except that `next()` has `String` return type.
   - Create a class, `DigitCombinationsGenerator`, that implements the `StringGenerator` interface. A `DigitCombinationsGenerator` object should produce all strings comprised of zero or more digits from the set $\{5, 6, 7, 8\}$, in the following order: `""`, `"5"`, `"6"`, `"7"`, `"8"`, `"55"`, `"56"`, `"57"`, `"58"`, `"65"`, `"66"`, `"67"`, `"68"`, `"75"`, `"76"`, `"77"`, `"78"`, `"85"`, `"86"`, `"87"`, `"88"`, `"555"`, `"556"`, ... (skipping ahead a bit) ... `"8766"`, `"8767"`, `"8768"`, `"8775"`, etc. Because the set of such strings is infinite, this generator never runs out of elements.

   Note: if you get stuck on the algorithm to produce all digit combinations, consider moving on to other parts of the test and coming back to it later.

   Test your solution using (at least) the tests in `Question2Tests`.

   [15 marks]

3. **Pairs.**

   The provided file `Pair.java` is a skeleton for a class to represent a pair, and is generic with respect to two type parameters `S` and `T`. Complete the implementation of `Pair` as follows:

   - `Pair` should have fields elem1 and elem2 of types `S` and `T` respectively, a constructor that accepts a parameter corresponding to each of these fields, and methods `getElem1()` and `getElem2()` that provide access to the values of these fields.
   - `Pair` should be immutable and it should not be possible to create subclasses of `Pair`.
   - The string representation of a `Pair` should consist of the character '{', followed by the string representation of `elem1`, followed by a comma and a space, followed by the string representation of `elem2`, followed by the character '}'.

- Two `Pair`s should be regarded as equal if their elem1 fields are regarded as equal and their elem2 fields are regarded as equal.

Test your solution using (at least) the tests in `Question3Tests`.

[8 marks]

4. **Generic data generators.**

   Make sure you consider getting started on Section B before attempting this question.

   Notice that the `IntegerGenerator` and `StringGenerator` interfaces differ only in the type of data elements that are generated. You will now write a *generic* data generation interface, adapt your existing interfaces to extend this interface, and create a data generator for pairs.

   - Create an interface, `DataGenerator`, that is generic with respect to a type parameter `T`. The type `T` is the type of data that classes implementing this interface will generate. Like `IntegerGenerator` and `StringGenerator`, `DataGenerator` should have a `hasNext()` method with `boolean` return type. It should also have a `next()` method with return type `T`.
   - Modify your `IntegerGenerator` and `StringGenerator` interfaces so that they *extend* the `DataGenerator` interface, providing `Integer` and `String`, respectively, in place of the generic parameter `T`. These modifications should be small, and do not require changing any of the methods in these interfaces.
   - Annotate the `next()` and `hasNext()` methods in `IntegerGenerator` and `StringGenerator` to indicate to the Java compiler that they match the signature of corresponding methods in `DataGenerator`.
   - Create a class, `PairGenerator`, that is generic with respect to two type parameters `S` and `T`. `PairGenerator` should implement the `DataGenerator` interface, providing `Pair<S, T>` in place of `DataGenerator`'s generic parameter `T`. A `PairGenerator<S, T>` object should consist of two data generators, one of type `DataGenerator<S>` and one of type `DataGenerator<T>`, and should have a constructor that accepts a parameter for each of these fields. The `hasNext()` method should return true if and only if `hasNext()` holds for both data generators. The `next()` method should return a `Pair<S, T>` whose `S` component is generated from the `DataGenerator<S>` field, and whose `T` component is generated from the `DataGenerator<T>` field.

   Test your solution using (at least) the tests in `Question4Tests`.

   [7 marks]

5. **Compound data generators.**

   Your final task for Section A is to write a class for a data generator that is composed of a list of other data generators. The idea is that this *compound* generator will use the first generator in its list of generators to generate elements until that generator is exhausted, and will then move on to use the second generator in the list, and so on.

   - Create a class, `CompoundDataGenerator`, that is generic with respect to a type parameter `T` and that implements the `DataGenerator<T>` interface.
   - A `CompoundDataGenerator` should hold a list of `DataGenerator<T>` object references and should take a list of such references on construction.

- Successive calls to `next()` should yield all the elements that the first of these generators can produce, until it runs out of elements, then all the elements that the second generator can produce, etc. The `hasNext()` method should return false if and only if all the data generators have been exhausted.

Test your solution using (at least) the tests in `Question5Tests`.

[10 marks]

Total for Section A: 50 marks

Section B

Problem Description: an unnecessarily complicated market

Overview

In this section you will create a simulated market where different agents sell and buy raw materials or finished products.

Every agent repeatedly performs an action depending on its role. There are four roles:

- *Supplier*: sells new raw materials
- *Manufacturer*: buys the necessary raw materials and sells finished products
- *Consumer*: buys products, use them, and then disposes of them
- *Recycle center*: collects disposed products, extract their raw material components, and sells recycled raw materials

The *market* is the place where all these operations happen. The market keeps *stocks* of each type of goods. When a manufacturer asks for a raw materials, if recycled items are available, they are sold before new items. You will have to implement the behaviors of the market and its agents for Q2.

To unnecessarily complicate all the process, each stock implements a peculiar data structure. Items are *pushed* in the stock labeled also with the identifier of the agent that sold them. Each agent has a unique integer id (`Agent.id`). An agent with a larger id is considered a more valuable stakeholder and its items are to be sold first. In the real world, this would be a priority queue, where the agent's id is the priority of a stored item. Such a priority queue may be implemented with a heap, for example, but not here :)

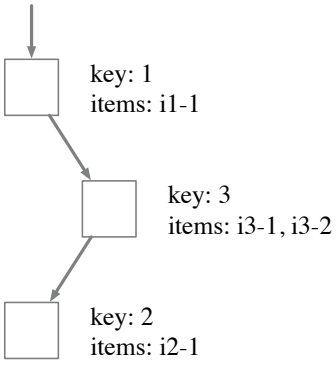
Here you need to implement the stocks as a variation of binary search trees (BSTs). Each node of the tree has an agent id as its key and contains a collection of all the items pushed by that agent. Items are *popped* out from the collections of the node with the largest id in the tree. When such collection becomes empty, the node is removed.

For example, consider the case of three agents (A1, A2, and A3) performing the following push operation, in the given order:

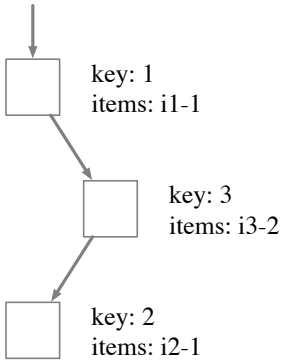
```
push(i1-1, A1)
push(i3-1, A3)
push(i2-1, A2)
push(i3-2, A3)
```

The resulting BST structure is shown in Figure 1a. Consider now a pop operation is performed. The highest priority node is 3 and it contains two items. One of them is returned (you can choose any of them, the order does not matter as long as they come from node 3). The resulting BST is shown in Figure 1b.

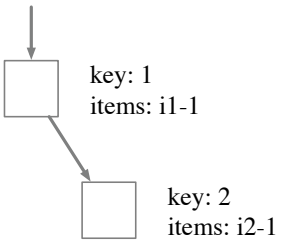
Now another pop is performed. The highest priority node is still 3 and it contains one item. The pop operation will return the item (`i3-2` in the example). Since Node 3 does not contain items anymore, it has to be removed, leaving the BST as shown in Figure 1c.



(a) Stock after the example push sequence.



(b) Stock after the first pop.



(c) Stock after the second pop.

Getting Started

If you have cloned your repo under your Home directory, then you can find these files at:

- `~/javafinaltestpartb_zz4319/src/`

During the test, you will modify **only** the files listed below among those provided in the skeleton source. You are free to add as many files as you see fit, but not to modify provided source files not listed below. You can add tests or otherwise modify the test suite.

- `datastructures/StockImpl.java`: where you will implement the BST-based data structure
- `market/MarketImpl.java`: where you will implement the functionalities of the market
- `agents/*`: where you will implement the agents specified below

Do not change the names or the contents of any other provided source class (in the folder `src`). Do not change the public interfaces of the provided sources (apart from possibly synchronizing methods as required in Question 3). You can add as many private methods, private fields, or additional classes as you see fit. Look also for comments in the code and at the tests for additional hints.

Testing

You are provided with a set of test cases in the `test` directory. The tests aim at exercising a variety of behaviors of your implementation and to further explain what your code is expected to do. The test suite is not exhaustive: even if your solution passes all the tests, your work will be assessed by the examiners, who may also use a different test suite to check your code.

What to do

1. **BST-based stocks.**

   In the package `datastructures` you will find `StockImpl`, which implements the interface `Stock`. This is the BST-based priority "queue" described in the overview. The tests in `datastructures.StockTest` will help you check your implementation.

   Heads up: for Q3, you will have to make this structure thread-safe. You may want to think whether to directly design a thread-safe solution or leave this extension for later.

   Your tasks are:

   - define a class for the BST nodes. Each node has a key used to keep the nodes in order. The key will be the `Agent.id` of the agent adding items to the BST via a push operation. Agent ids are unique by construction, so there should be no duplicates in the BST. Besides its key, each node has a collection of items. The collection can be anything you want (e.g., List or Set). You need to be able to pick an item from the collection, it does not matter which one, and to add an item to the collection.
   - implement the method `push(item, agent)`. This method adds an item to the tree. It traverses the tree to find the node with key equal to `agent.id` and adds the item to its collection. If such node does not exist, it adds it to the tree preserving the BST invariants, and stores the item in its collection of items.
   - implement the method `Optional<E> pop()`. This method traverses the tree to find the node with the largest key and returns an optional containing one of the items from the node's collection (the returned item is removed from the collection). If the node's collection becomes empty, the node has to be removed from the tree, preserving its BST invariants. If the tree is empty, the method returns `Optional.empty()`.
   - implement the method `size()`, which returns the total number of items stored in the BST.

   Hint: the highest priority node in a BST is always the rightmost one. Therefore, it can only be either a leaf or a node with a single child... Remember this for both Q1 and Q3.

   [20 marks]

2. **Market and agents.**

   You are required to implement the class `market.MarketImpl` and the agents in the package `agents`. All the agents inherit from `domain.Agent`, which takes care of assigning them a unique `id` upon creation. `Agent` also extends `Thread` and defines its run method: until interrupted, and agent repeatedly `doAction()` and then sleeps for a random time. For each agent, you only need to implement `doAction()` (possibly adding fields and private methods as you see fit). The tests in `market.MarketTest` will help you check your implementation and provide additional hints.

   Your tasks are:

   - implement the methods of `MarketImpl`. The method names are self-explanatory. The market keeps a *stock* of each product (including disposed ones) and pushes and pops from those stocks as needed. For raw materials, the market should serve recycled ones first, if available. (**Hint**: you may keep two separate stocks for each

raw material type, one for recycled items, and one for new items. This will make it easier to implement *buy* methods that return recycled items firsts, if available.)

- implement the method `doAction()` of `agents.RawGlassSupplier` and `agents.RawTungstenSuppli` Suppliers just sell to the market a new (`Origin.NEW`) raw material every time `doAction` is called. The classes for the new materials are specified in `goods`. Each supplier sells material of the type corresponding to its name, e.g., `XSupplier` produces `RawX`.
- Implement the method `doAction()` of `agents.BulbManufacturer`. At each invocation of `doAction`, this manufacturer repeatedly invokes *buy* operations until it secures two units of tungsten and one unit of glass. Then, it sells a new unit of `Bulb`, constructed with the set of raw materials it secured.
- implement the method `doAction()` of `Consumer`. At each invocation, the consumer tries to buy a `Bulb` from the market. If it gets one, it `think()` (method inherited from `Agent`), and then they `disposeBulb`.
- Implement the method `doAction()` of `RecycleCenter`. At each invocation, the recycle center collects a disposed product from the market. If it's available, it extracts its constituent materials (`Product.getConstituentMaterials()`) and, for each item of raw material with origin `Origin.NEW` it sells a new item of the same type with origin `Origin.RECYCLED` to the market. Extracted items of raw material already `Origin.RECYCLED` are ignored.

[15 marks]

3. **Thread-safe stocks.**

   For this question, you are require to make the `StockImpl` implementation (all three methods) you wrote for Q1 thread-safe. A coarse-grained access control is worth up to 5 marks. Any correct fine-grained access control is worth up to 10 marks.

   As for the interim test, before moving to the fine-grained implementation, make sure to save a copy of your coarse-grained one (e.g., in a class `StockImplCoarseGrained`), or to write clearly in a comment at the beginning of the class how you would implement such coarse-grained strategy.

   The tests in `datastructures.StockStressMonkey` may help you find errors in your solution (unfortunately, passing the tests does not imply the solution is correct).

   Hint: the size of the stock is just an integer that gets incremented or decremented by push and pop operations...

   [15 marks]

Total for Section B: 50 marks