Assignment 1

Goal: The goal of this assignment is to gain practical experience with data abstraction, unit testing and using the Java class libraries.

Due date: The assignment is due at **1pm on Thursday 11 April.** Late assignments will lose 20% of the total mark immediately, and a further 20% of the total mark for each day late.

Problem Overview: In this assignment, and the following two assignments in this course, you will develop the component classes of a program to be used to analyse the information leakage of a very important secret…

A world super-power is at an impasse: aliens have been detected in a remote part of the Delta Quadrant, but they can't decide whether or not to make contact. To solve their problem they are going to decide the future of the world with the simple flip of a biased coin. Regardless of how the coin falls, however, they want to keep the decision secret.

Unfortunately, the super-power knows all too well that the moment the coin has been flipped, and the decision made, spies from every nation around the world will set to work, trying desperately to learn as much as they can about their decision.

To begin with, each spy will be privy to the bias of the coin that has been used to make the decision, and they know that if it comes up heads then the aliens will be contacted (*true*), and if it falls tails, no communication will be initiated (*false*). In general, a distribution describing the likelihood that the secret is true (resp. false) is referred to as a *knowledge-state*. Such states are used to describe what a spy knows, at a given time, about the distribution of the secret.

Each spy will learn about the secret through a sequence of interactions with informants. One by one, informants will come to the spy. Each such informant has an agenda…

Informants are omnipotent, knowing the outcome of the decision, as well as the current knowledge-state of the spy. If and only if the knowledge-state of the spy matches one pre-determined by the informant (the *condition* under which information may be leaked), then the informant plays a little game with the spy using a *two-coin channel*. The game goes like this…

The informant has two biased coins, *coin1*, which he will flip if the value of the secret is true, and *coin2*, that he will flip if the secret is false. The informant shows both coins to the spy, so that she may determine their biases. Then, in private – out of view of the spy herself – he flips coin1 if the secret is true and coin2 if the secret is false, and reveals the outcome of the coin flip to the spy (but not which coin has been flipped).

After the informant has revealed to the spy the outcome of the coin flip, the spy can update her knowledge of the distribution of the secret, given the outcome of the coin.

For example, if the spy initially knows that the secret is true with probability 3/4, and false with probability 1/4, so that her *a priori* knowledge of the secret is `{true@3/4, false@1/4}`, and coin1 has heads-bias 1/3 and coin2 has heads-bias 1/2 then:

With probability

```
3/8 = 3/4*1/3 + 1/4*1/2
```

the outcome of the coin flip will be *heads* and the probability that the secret is true *given that the outcome was heads* will be:

```
2/3 = 3/4*1/3 ÷ 3/8
```

And with probability

```
5/8 = 3/4*2/3 + 1/4*1/2
```

the outcome of the coin flip will be *tails*, and the probability that the secret is true *given that the outcome was tails* will be:

```
4/5 = 3/4*2/3 ÷ 5/8
```

That is, with probability 3/8 the spy will witness the outcome heads and her *a posteriori* knowledge-state will be `{true@2/3, false@1/3}`, and with probability 5/8 she will witness the outcome tails and her *a posteriori* knowledge-state will be `{true@4/5, false@1/5}`.

Given the initial heads-bias of the decision-making coin, and a sequence of informants who will visit a spy, one at a time, in the order defined by the given sequence, the super-power can determine the likelihood that the spy will, after all the informants have done their dirty work, be in any given knowledge-state.

For example, imagine the decision-making coin is fair, and a spy will be visited by informant one, with condition `{true@1/2, false@1/2}`, and coin1 with heads-bias 3/4 and coin2 with heads-bias 1/4; and then by informant two, with condition `{true@3/4, false@1/4}` and coin1 heads-bias 1/3 and coin2 heads-bias 1/2.

Initially the spy will be in knowledge state `{true@1/2, false@1/2}` with probability one, so that the distribution of knowledge-states of the spy is:

```
{{true@1/2, false@1/2}@1}
```

After a visit by the first informant, she will be in knowledge-state `{true@1/4, false@3/4}` with probability 1/2 and knowledge state `{true@3/4, false@1/4}` with the remaining probability:

```
{{true@1/4, false@3/4}@1/2, {true@3/4, false@1/4}@1/2}
```

And after the visit by the second and final informant, her distribution of knowledge states will be updated further to:

```
{{true@1/4, false@3/4}@1/2, {true@2/3, false@1/3}@3/16,
{true@4/5, false@1/5}@5/16}
```

Task Overview:

In this assignment, you will be writing data types for a *TwoCoinChannel*, a *ConditionaTwoCoinChannel*, and a *KnowledgeDistribution*. If you are a CSSE7023 student you will also be required to write a JUnit4 test suite for the TwoCoinChannel class in the *TwoCoinChannelTest* class.

Skeletons of all of these classes are included in the zip file. In brief:

The immutable class **TwoCoinChannel** can be used to keep track of the coins used by an informant, and to calculate, given a spy's a priori knowledge of the secret (i.e. the initial distribution of the secret): (a) The likelihood that the outcome of the coin flip will be heads or tails, and (b) The a posteriori distribution of the secret given the outcome is heads (or tails). (i.e. the likelihood that the secret is true or false given that the outcome is heads (or tails)).

This immutable class **ConditionalTwoCoinChannel** keeps track of a knowledge-state and a two-coin channel. It is used to describe the condition -- the knowledge-state -- under which an informant will use the aforementioned two-coin channel to communicate to a spy.

This **KnowledgeDistribution** class is a mutable representation of a discrete sub-distribution on knowledge-states. Such a sub-distribution represents a mapping from knowledge-states to probabilities, in which the weight of the distribution -- the sum of the probabilities of all the knowledge states in the distribution-- must be less than or equal to one. (We allow sub-distributions so that this class can be used to build total -- i.e. one-summing -- distributions.)

Practical considerations:

Skeletons for the classes *TwoCoinChannel*, a *ConditionaTwoCoinChannel*, and a *KnowledgeDistribution* are provided in the zip file in the `csse2002.security` package. These skeletons include Javadoc specifications of the constructors and methods that you need to complete. You must complete these class skeletons according to their specifications in the files. You will need also to declare variables, add import clauses, and add comments and javadocs as required.

If you are a CSSE7023 student, you will also need to complete systematic and understandable JUnit4 test suite for the TwoCoinChannel class in the skeleton of the *TwoCoinChannelTest* class in the `csse2002.security.test` package. You may write your unit tests assuming that the classes that TwoCoinChannel depends on (e.g. the BigFraction class and any of the Java libraries) are implemented and functioning correctly. That is, you don't need to create test stubs for these classes.

Copy or rename the skeleton files before you start. (Don't forget that the package and class names inside the files must correspond to the file and the directory names … otherwise you'll have difficulty compiling and running it.)

You must implement these classes as if other programmers were, at the same time, implementing the code that instantiate them and call their methods. Hence:

- The class names must be as above.
- Don't change the specifications provided, or alter the method names, parameter types or return types or the packages to which the files belong.

- You are encouraged to use Java 7 SE classes, but no third party libraries should be used. (It is not necessary, and makes marking hard.)
- Don't write any code that is operating-system specific (e.g. by hard-coding in newline characters etc.), since we will batch test your code on a Unix machine.

Implement the classes as if other programmers are going to be using and maintaining them. Hence:

- Your code should follow accepted Java naming conventions, be consistently indented, and use embedded whitespace consistently.
- Your code should use private methods and private instance variables and other means to hide implementation details and protect invariants.
- Your methods, fields and local variables (except for-loop variables) should have appropriate Javadoc comments or normal comments.
- Comments should be used to document your code's invariants, and to describe any particularly tricky sections. (You must provide an implementation invariant for each of the TwoCoinChannel, ConditionalTwoCoinChannel and KnowledgeDistribution classes). However, you should also strive to make your code understandable without reference to comments; e.g. by choosing sensible method and variable names, and by coding in a straight-forward way.
- The checkInv() method of each class (other than the test suite) should check that the implementation invariant you have specified in your comments is satisfied.

The Zip file for the assignment also includes some other code that you will need to compile your classes as well as some junit4 test classes to help you get started with testing your code.

Do not modify any of the files in packages csse2002.security or csse2002.math other than TwoCoinChannel, ConditionalTwoCoinChannel, KnowledgeDistribution, since we will test your code using our original versions of these other files. Do not add any new files either that your code for these classes depends upon, since you won't submit them and we won't be testing your code using them.

The junit4 test classes as provided in the `package csse2002.security.test` are *not intended to be an exhaustive test for your code*. Part of your task will be to expand on these tests to ensure that your code behaves as required by the javadoc comments. (Only if you are a CSSE7023 student will you be required to submit your test file TwoCoinChannelTest.java.) We will test your code using our own extensive suite of junit test cases. (Once again, this is intended to mirror what happens in real life. You write your code according to the "spec", and test it, and then hand it over to other people … who test and / or use it in ways that you may not have thought of.)

If you think there are things that are unclear about the problem, ask on the newsgroup, ask a tutor, or email the course coordinator to clarify the requirements. Real software projects have requirements that aren't entirely clear, too!

If necessary, there may be some small changes to the files that are provided, up to 1 week before the deadline, in order to make the requirements clearer, or to tweak test cases. These updates will be clearly announced on the Announcements page of Blackboard, and during the lectures.

Hints:

1. It will be easier to implement the TwoCoinChannel first, followed by the ConditionalTwoCoinChannel and then KnowledgeDistribution. The tests you may wish to write before you start coding these classes.
2. Read the specification comments carefully. They have details that affect how you need to implement and test your solution.

Submission: Submit your files `TwoCoinChannel.java`, `ConditionalTwoCoinChannel.java`, `KnowledgeDistribution.java` (and `TwoCoinChannelTest.java` if you are a CSSE7023 student) electronically using Blackboard:

https://learn.uq.edu.au/

You can submit your assignment multiple times before the assignment deadline but only the last submission will be saved by the system and marked. Only submit the source (i.e. .java) files for the **TwoCoinChannel, ConditionalTwoCoinChannel, KnowledgeDistribution (**and **TwoCoinChannelTest** for CSSE7023**)** classes. Any further instructions about the submission will be posted on the newsgroup.

Evaluation:

If you are a CSSE2002 student, your assignment will be given a mark out of 10, and if you are a CSSE7023 student, your assignment will be given a mark out of 13, according to the following marking criteria. (Overall the assignment is worth 10% for students from both courses.)

Testing (5 marks)

- All of our tests pass — 5 marks
- At least 3/4 of our tests pass — 4 marks
- At least 1/2 of our tests pass — 2 marks
- At least 1/4 of our tests pass — 1 mark
- Work with little or no academic merit — 0 marks

Note: code submitted with compilation errors will result in zero marks in this section.

Code quality (5 marks)

- Code that is clearly written and commented, and satisfies all specifications, style rules and rules of data abstraction — 5 marks
- Minor problems, e.g., lack of commenting or private methods — 3-4 marks
- Major problems, e.g., code that does not satisfy the

specification or rules of data abstraction, or is too complex, or is too difficult to read or understand 1-2 marks

- Work with little or no academic merit 0 marks

Note: you will lose marks for code quality

a) for breaking java naming conventions,

b) failing to comment tricky sections of code,

c) inconsistent indentation and / or embedded white-space, and

d) lines which are excessively long (lines over 80 characters long are not supported by some printers, and are problematic on small screens).

JUnit4 test – CSSE7023 ONLY (3 marks)

We will try to use your test suite TwoCoinChannelTest to test an implementation of TwoCoinChannel that contains some errors in an environment in which the other classes TwoCoinChannel.java depends exist and are correctly implemented.

Marks for the JUnit4 test suite in TwoCoinChannelTest.java will be allocated as follows:

- Clear and systematic tests that can easily be used to detect most of the (valid) errors in a sample implementation and does not erroneously find (invalid) errors in that implementation. 3 marks
- Minor problems, e.g., Can only be used easily to detect some of the (valid) errors in a sample implementation, or falsely detects a small number of (invalid) errors in that implementation, or is somewhat hard to read and understand. 2 marks
- Major problems, e.g., cannot be used easily to detect most of the (valid) errors in a sample implementation, or falsely detects many (invalid) errors in that implementation, or is too difficult to read or understand. 1 marks
- Work with little or no academic merit 0 marks

Note: code submitted with compilation errors will result in zero marks in this section.

School Policy on Student Misconduct: You are required to read and understand the School Statement on Misconduct, available on the School's website at:

http://ppl.app.uq.edu.au/content/3.60.04-student-integrity-and-misconduct

This is an individual assignment. If you are found guilty of misconduct (plagiarism or collusion) then penalties will be applied.

If you are under pressure to meet the assignment deadline, contact the course coordinator **as soon as possible**.