CPSC250L Lab 3
`JOptionPane` and Exceptions

Fall 2016

1 Introduction

This lab will focus on a powerful way to handle errors known as *exceptions*. You will demonstrate the ability to both *throw* and *catch* exceptions. Additionally, you will get a soft introduction to GUI programming using `JOptionPane`. We recommend that you read about `JOptionPane` in the Java API documentation.

2 Exercises

2.1 Combination

In the lab repository, you should see a `Combination.java`. For this exercise, you will write a few methods for the `Combination` class. Read through the comments, and add your code at the `@todo` markers.

Exercise 1

Implement the following methods in the `Combination` class.

1. Create a constructor that receives three `int`s `a`, `b`, `c` and places them in the `numbers` array *in order*. This makes `a`, `b`, `c` the combination which this object represents.
2. A method named `getNumbers` which returns a *copy* of the combination in order. This method should not return `numbers`.
3. A method named `isWithinRange` which receives `int upper` which represents an upper bound and returns a `boolean`. This method returns `true` if each number in the combination is in the closed interval [0, `upper`] and `false` otherwise.

Test your code against `CombinationTest.java`.

Exercise 1 Complete

Run:

```
git add .
git commit -m "Completed exercise 1"
git push origin master
```

2.2 Lock

Create a class named `Lock` that uses the provided class `InvalidLockCombinationException`, which defines a new type of `RuntimeException`. We will be learning about inheritance and extending classes later in the semester. Review the code for

`InvalidLockCombinationException.java` to see how easy it is to create a specific exception type.

In this first exercise we gave you shell code with comments. In this exercise you will create the class from scratch; you should follow good style guidelines and add your own comments. At a minimum document your class with @author tag, and document all methods including parameters and return values. See the `Combination` class for example.

Exercise 2

The `Lock` class should have fields representing the combination, an upper limit for its dial, and an indicator which states whether or not the `Lock` is open. Implement the following methods in the `Lock` class.

1. Create a constructor that receives an `int` upper bound and a `Combination` object. If the `Combination` is within the range of the upper bound, then it should set its own combination to the input. Otherwise, it should throw an `InvalidLockCombinationException`. Furthermore, all locks should be open when created.
2. A method named `getDialLimit` which returns an `int` representing the dials upper limit.
3. A method named `open` which receives a `Combination` and returns a `boolean` representing whether or not the lock is open. If the received `Combination` equals the lock's `Combination`, then set the lock's state to open. If the lock is already open, then lock will remain open regardless of the received `Combination`. **Hint: use `Combination.equals(otherCombination)` to check if two combinations are equal.**
4. A method named `close` that sets the lock's state to closed and returns nothing.
5. A method named `isOpen` that returns a `boolean` that indicates whether or not the lock is open.

Test your code against `LockTest.java`.

Exercise 2 Complete

Run:

```
git add .
git commit -m "Completed exercise 2"
git push origin master
```

2.3 Lock with Reset

We will create two methods in the `Lock` class called `resetNaive` and `resetRetry`.

Exercise 3

Implement the following methods in the `Lock` class.

1. `void resetNaive()`

   Using `JOptionPane.showInputDialog`, get a `String` combination from the user and if it's not `null`, grab the 3 numbers. If the `String` is `null`, you don't want to do anything. (This allows the user to cancel!) Use a `Scanner` to scan the `String`, and extract the 3 integers. See `https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html#nextInt-int-` for a list of potential exceptions that may be thrown if the string is not formatted properly. You do NOT need to catch these here.

   Create a new `Combination` using the three numbers, and if the combination is within the dial limit, that becomes the `Lock`'s combination. If the combination is invalid, it throws an `InvalidLockCombinationException`. This method should throw the exception, but not catch it. Sometimes we want to handle exceptions, and sometimes we want them to pass a message to another method, which is the case here.

2. `void resetRetry()`

   While a good combination has not been found, keep trying to get one (**hint: use `resetNaive`**). If asking for a combination using `resetNaive` throws an `InvalidCombinationException`, use `JOptionPane.showMessageDialog` to display the message "Type 3 integers in the range [0..R]" where R is your dial limit set by the constructor. If asking for a combination throws any other exception error (e.g. one from Scanner), use `JOptionPane.showMessageDialog` to display the message "Type 3 integers separated by spaces". (These dialogues must be exact). Keep trying to get a good combination until a good combination is found, then that becomes the Locks combination.

Exercise 3 Complete

Run:

```
git add .
git commit -m "Completed exercise 3"
git push origin master
```

---

3 Common Mistakes

1. Ensure that the constructor for `Combination` stores the combination in the `numbers` array. Otherwise, this will break the `equals` method which will cause tests to fail.

2. Ensure that `InvalidLockCombinationException` extends `Exception`.

3. In `resetRetry`, be sure to invoke `resetNaive` as opposed to reimplementing `resetNaive`'s functionality in `resetRetry`.

4. Ensure that your messages are exactly as the lab specifies, and use the classes that the lab tells you to use!