CPSC250L Lab 4
Exceptions

Spring 2018

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.

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. `public Combination(int a, int b, int c)`

This constructor receives three `int`s `a`, `b`, `c` and places them in the `numbers` array *in order*. That is, your `numbers` array should look like `{a,b,c}`. This makes `a`, `b`, `c` the combination which this object represents.

2. `public int[] getNumbers()`

   This method returns a *copy* of `numbers`. That is, this method returns an array containing the elements of `numbers` in the same order. This method should not return `numbers`.

3. `public boolean isWithinRange(int upper)`

   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 the 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. `public Lock(int aLimit, Combination myCombo)`

   This constructor receives an `int` upper bound and a `Combination` object, which will represent the initial lock combination. 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. `public int getDialLimit()`

   This method returns an `int` representing the dials upper limit.

3. `public boolean open(Combination testMe)`

   This method 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. `public void close()`

   This method sets the lock's state to closed.

5. `public boolean isOpen()`

   This method returns `true` if the lock is open or `false` if the lock is closed.

Test your code against `LockTest.java`.

Exercise 2 Complete

Run:

```
git add .
git commit -m "Completed exercise 2"
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. Ensure that your messages are exactly as the lab specifies, and use the classes that the lab tells you to use!