P04 Exceptional Piggy Bank

Overview

In this assignment, we are going to develop an enhanced version of the elastic piggy bank (P03). You are going to add the following features to our famous program:

- throw exceptions to report bugs or misuse of a set of methods that we have already implemented in P03,
- load coins to the elastic piggy bank from a file,
- save the contents of the bank in a file too, with respect to a specific format,
- develop unit tests to check the correctness of the implementation of the different operations supported by our exceptional piggy bank.

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P04 pre-assignment quiz is accessible through Canvas before having access to this specification by **9:59PM on Sunday 02/16/2020**. Access to the pre-assignment quiz will be unavailable passing its deadline.
20 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is otherwise correct. To become more confident of this, you should run additional tests of your own.
15 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.
10 points	**Manual Grading Feedback:** After the deadline for an assignment has passed, the course staff will begin manually grading your submission. We will focus on looking at your algorithms, use of programming constructs, and the style and readability of your code. This grading usually takes about a week from the hard deadline, after which you will find feedback on Gradescope.

Learning Objectives

The goals of this assignment include:

- Gain more experience with creating classes and using objects.
- Develop your understanding of the difference between checked and unchecked exceptions, and get practice both throwing and catching exceptions of each kind.
- Get more practice writing tests, specifically tests that detect whether exceptions are thrown under the prescribed circumstances or not.

Additional Assignment Requirements

- All String comparisons in this assignment are CASE INSENSITIVE.
- You MUST NOT add any additional fields either instance or static, and any public methods either static or instance to your `ExceptionalBank` class, other than those defined in this write-up and these javadocs.
- You CAN define local variables that you may need to implement the methods defined in this program.
- You CAN define private helper methods to help implement the different public methods defined in this program, if needed.
- You CAN define private static helper methods to help implement the different public static methods defined in this write-up.
- All your test methods must be public static. Also, they must take zero arguments, return a boolean, and must be defined and implemented in your `ExceptionalBankTester.java`.
- Your `ExceptionalBankTester` class must implement at least the test methods defined in these javadocs with exactly the same signatures. Feel free to add additional test methods and/or consider additional test scenarios to further convince yourself of the correctness of your implementation.
- All implemented methods MUST have their own javadoc-style method headers with complete information, with accordance to the CS300 Course Style Guide.
- Feel free to reuse the information provided in these javadocs in your own java-doc style method headers.

1 Getting Started

Start by creating a new Java Project in eclipse called P04 Exceptional Piggy Bank, for instance. You have to ensure that your new project uses Java 11, by setting the "Use an execution environment JRE:" drop down setting to "JavaSE-11.0.X" within the new Java Project dialog box. In this assignment, we are going to provide you with a source code similar to your implementation of P03 Elastic Bank, but **using an enumeration to define the `Coin` type** rather than an instantiable class. To do so, download first these two source files Coin.java and ExceptionalBank.java. Then, add both of them to the default package of the src folder of your project. Then, create and add a third file named `ExceptionalBankTester.java` to the same project. Note that this file should include all your test methods and a main() method. We note also that Appendix A provides links to java API of the set of exception classes that you may use in this program. Appendix B presents a set of useful methods that you may use while developing this assignment.

2 Complete the implementation of ExceptionalBank class

Start first by reading the provided code. You should be familiar now with the elastic piggy bank application. Notice that there are additional methods not provided in P03 added to this project, for instance *getSpecificCoinCount()* and *getSummary()* methods. Notice also that none of the implemented methods use hard coding of the coin names or values. Our implementation should work appropriately regardless of the set of constants defined in the enum `Coin` and their values.

2.1 Update the constructor, addCoin(), and removeCoin() methods

Start by updating the implementation details of the constructor of the class `ExceptionalBank(int)`, and the instance methods `addCoin()` and `removeCoin()` according to their detailed javadocs description provided within these javadocs. These methods are expected to work as described in P03 when they terminate without errors. Read carefully the provided javadoc method headers, and pay close attention to the exceptions that should be thrown by the constructor and the public methods. Add appropriate `@throws` annotations to the javadoc style method header of the constructor of your ExceptionalBank class, addCoin(), and removeCoin() methods. Keep in mind to NOT catch an exception within the implementation details of a method if that exception has been declared to be thrown using `@throws` annotation in its javadoc method header. Let the exception propagate to the method's caller.

To check the correctness of your ExceptionalBank class so far, implement the `testAddCoin()`, `testExceptionalBankConstructor()`, `testGoodExceptionalBankConstructor()`, and `testRemoveCoinEmptyBank()` test methods with accordance to the details provided in their javadoc method headers. We provide you in the following with an example of implementation of the `testExceptionalBankConstructor()` test method. Feel free to reuse the provided code in your own submission. You can define additional test scenarios to convince yourself that your test methods are able to detect defects of any wrong implementation, and pass otherwise.

```java
/**
 * This method checks whether the ExceptionalBank constructor throws an
 * IllegalArgumentException with appropriate error message, when it is passed
 * a zero or a negative capacity. This test must fail if another kind of exception
 * is thrown for such test scenario.
 *
 * @return true when this test verifies a correct functionality, and false otherwise
 */
public static boolean testExceptionalBankConstructor() {
  try {
    // create an exceptional bank with a negative capacity
    ExceptionalBank bank = new ExceptionalBank(-10);
    System.out.println(
        "Problem detected. The constructor call of the ExceptionalBank class did not "
            + "throw an IllegalArgumentException when it is passed a negative capacity.");
    return false; // return false if no exception has been thrown
  } catch (IllegalArgumentException e1) {
    // check that the caught IllegalArgumentException includes
    // an appropriate error message
    if (e1.getMessage() == null // your test method should not throw
        // a NullPointerException,but must return false if e1.getMessage is null
       || !e1.getMessage().toLowerCase().contains("must be a non-zero positive integer")) {
      System.out.println(
        "Problem detected. The IllegalArgumentException thrown by the constructor "
            + "call of the ExceptionalBank class when it is passed a negative capacity "
            + "does not contain an appropriate error message.");
      return false;
    }

  } catch (Exception e2) {
    // an exception other than IllegalArgumentException has been thrown
    System.out.println(
        "Problem detected. An unexpected exception has been thrown when calling the "
         + "constructor of the ExceptionBank class with a negative argument. "
         + "An IllegalArgumentException was expected to be thrown. "
              + "But, it was NOT the case.");
    e2.printStackTrace(); // to help locate the error within the bad ExceptionalBank
                          // constructor code.
    return false;
  }
  return true; // test passed
}
```

None of your unit test methods should ever throw any exception. Your test method should catch any exceptions that may be thrown by the call of the constructor or the methods defined in the `ExceptionalBank` class, and returns true if the expected behavior has been satisfied, and false otherwise.

2.2 Implement and test addCoins(String) method

The `addCoins(String)` is one of the new methods that we are going to add to this expanded version of the elastic piggy bank project. This method takes a String as input parameter which represents a command line to add new coins to the exceptional bank. You have to implement now this method with accordance to the details provided in its javadoc style method header comments available in these javadocs. If correctly formatted, the string provided to addCoins() method should consist of two parts separated by a colon ":". The first part must refer to one of the constants defined in the `enum Coin`, which represent a coin's name, based on a case insensitive comparison. The second part must be a positive integer. It represents the number of coins with that name that must be added to the exceptional bank as result of running addCoins() method. No coin will be added to the bank if the format of the provided string is incorrect.

Do not forget to implement the test methods related to check the correctness of the `addCoins()` defined in the `ExceptionalBankTester` class according to these javadocsfirst. The following are examples of valid string addCoins commands to add 2 quarters for instance:

- "QUARTER:2"
- "quarter:2"
- " Quarter : 2 "
- "quARTEr: 2"

Whereas, the following are examples of invalid (mal-formatted) string objects to add 2 quarters for instance:

- "QUARTERS:2"
- "quarter:two"
- "quarter:-2"
- " Quart : 2 "
- "quarter 2"
- "25 : 2"
- "quarter:2 :penny:3"

Notice also carefully that DataFormatException extends directly the `Exception` class. It is not a RuntimeException. It must be a **checked exception**.

2.3 Implement and test loadCoins() and saveBankSummary() methods

Now, implement the `loadCoins()` and `saveBankSummary()` methods in your `ExceptionalBank` class and their test methods in your `ExceptionalBankTester` class with accordance to the details provided in these javadocs. These are examples of text files whose File objects can be passed as input parameters to your loadCoins() method for instance sample1.txt, sample2.txt, and sample3.txt. The sample3.txt file contains lines mal-formatted and a line including an incorrect coin name. Those lines must be skipped. Only lines with correct format and coin names defined in the enum Coin will result in adding coins to our exceptional bank. The correct format of lines is the same as the correct format of the string provided as input to *addCoins()*method described in the previous section.

Note that ExceptionalBank.loadCoins() and ExceptionalBank.saveBankSummary() methods take a reference to an object of type java.io.File as an input parameter. In your test methods which may call these methods, create the file object using the constructor of the java.io.File class that takes a path name String as input parameter.

Note also that your *loadCoins()* method should be able to load a file saved by your *saveBankSummary()* method. Notice also that the *saveBankSummary()* method should not throw any exception. If an IOException may be thrown in the *saveBankSummary()*, you have to catch it.

3 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through gradescope.com. The only 2 files that you must submit include: `ExceptionalBank.java` and `ExceptionalBankTester.java`. Your score for this assignment will be based on your "active" submission made prior to the hard deadline of Due: **9:59PM on February 19th**. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline. Finally, the third portion of your grade for your submission will be determined by humans looking for organization, clarity, commenting, and adherence to the CS300 Course Style Guide.

Appendices

A Appendix A

java.util.Scanner
Random
IllegalStateException
NumberFormatException
FileNotFoundException
PrintWriter
java.io.File
IllegalArgumentException
NoSuchElementException
DataFormatException
IOException
PrintStream

B Appendix B

The following list of methods may be useful while completing this assignment.

- String.split()
- String.equals()
- String.equalsIgnoreCase()
- Integer.valueOf()