Lab 7

Exceptions and Exception Handling

Due: during your lab session on or before November 22nd

Description:

In this lab you will:

1. Explore exceptions and exception handling
2. Write code to throw and catch exceptions
3. Understand the role polymorphism plays in determining the order in which exceptions are caught
4. Write code based on a checksum algorithm

You will have completed this lab when you:

- ☐ Can successfully create and throw the exceptions indicated.
- ☐ Can answer questions such as:
  - What is the advantage of using exceptions over an *if…else…* structure to capture errors?
  - Where is the getMessage() method actually declared, and how do you set the string to be output by this message in your code for the BadAccountInputException constructor?
  - Why must superclass exceptions be caught *after* subclass exceptions?
  - What is the purpose of a check digit, and how would you use it to catch an incorrect value entered into code?

Worth

2.5%

of your total mark

Lab 7

Exceptions and Exception Handling

I. Load the code for this lab, available on Brightspace, into Eclipse.

a. Download the CST8284_19F_Lab07.zip file from Brightspace and import the file into Eclipse package explorer.

b. Inspect the contents of the files just imported. Note that:

i. `AccountLauncher` contains the `public static void main()` method that launches your program. This class is complete and should not be modified.

ii. The `Account` class contains 4 TODOs for you to complete. The description of these tasks, along with the output to be printed, is generated by the Lab 7 QuestionGenerator.jar file provided. Double click on the .jar to launch it, enter your student number in the box at the top, and follow the instructions.

iii. In the same package as the above classes, create a new class, `BadAccountInputException`, which must extend from `java.lang.RuntimeException`. Your class must contain two constructors: a one-arg constructor that loads a single string as a parameter, and a no-arg constructor that chains to the one-arg constructor and passes the default string "Bad input: value entered is incorrect". In the one-arg constructor, pass the string message up to the superclass using `super()`—that's all. But you'll need to be able to explain how the `BadAccountInputException` subclass does this. And for this, expect to do some research in the Oracle documentation for the Throwable class hierarchy. In particular, you should be able to answer the two questions: (1) where is the `getMessage()` method declared? And (2) how does the error message String get loaded into it?

iv. Finally, you'll need to add the line

```
private static final long
                serialVersionUID = 1L;
```

to `BadAccountInputExcepton`, to prevent serialization warnings and errors

II. Implement the code indicated in the TODOs

a. Each of the TODOs is relatively self-explanatory. In each case, you'll catch the error stated and throw a new `BadAccountInputException()`, passing the String message indicated so that ex.getMessage() outputs your error message (the code to catch and output this message is already in the AccountLauncher class.) Be sure you understand how exception handling works, and be prepared to insert breakpoints in your code if unsure about the program's execution.

The following String methods may be of use to you in testing the first name, last name, and account number for correctness. You'll want to research these first at https://docs.oracle.com/javase/8/docs/api/java/lang/String.html to find out how they work:

trim()	substring()
matches()	contains()
split()	valueOf()

Additionally, you may find Integer.parseInt() useful to convert `String` values to their equivalent `int` values.

Be sure to test your code with values designed to throw the particular exception your code is designed to detect.

b. TODO #2 requires that you write code to calculate a checksum value, which is always the very last digit in the account number string. If the checkdigit of the customer account number is correct, isCheckDigitCorrect() returns true, otherwise false. Your error message should be: "Checkdigit failed; possible bad account number." (See the first and last name setters for an indication of how to use isCheckDigitCorrect().)

Note: Each account number consists of a three digit branch code, followed by a hyphen ("-"), followed by a six-digit customer account number. It is the six-digit customer account number we are interested in for the checksum calculation: ignore the three-digit branch number, pass only the string after the "-" to isCheckDigitCorrect().

Your code must be written so that it can handle a checksum of any numbers of digits, regardless of the specific number of digits provided in the example shown in TODO #2. You may assume each customer account number is at least five digits long.

What is a checkdigit? It is a (usually single-valued) number used to check that a serial or identification number is correct. As an example, consider the ISBN number on the back of your textbook (*Dietel & Dietel,* 11th edition): 0-13-474335-0. The algorithm for this says: ignoring hyphens, multiply the first digit by 1, add it to the second digit multiplied by 2, plus the third digit multiple by 3, etc. Calculate the sum modulus 11. The last value, '0' in this case, should equal the result. So mathematically, for **013474335,** the calculation is

```
(1)0 + (2)1 + (3)3 + (4)4 + (5)7
+ (6)4 + (7)3 + (8)3 + (9)5 = 176
```

`(176 % 11) == 0`, i.e. the last digit in 0-13-474335-0: hence the ISBN is correct. (Recall that % is the modulus operator; it returns the remainder of a division. Also note: (1) 'X' is used to indicate the number 10 in the standard ISBN format, which we'll ignore here; (2) the new ISBN-13 format follows a different algorithm, but the general principle remains the same: the last digit verifies that the other digits are correct.)

III. Demonstrate your program, and your understanding of the code, to the lab professor

Have the QuestionGenerator up and running when you present your code to the lab professor. Execute your program before stepping up to be evaluated, and have the account number provided in the example already entered to show that your check digit method works (i.e. the customer account number returns true from isCheckDigitCorrect()). For the other TODOs, load faulty date that demonstrates your code catches the exceptions, according to the TODO specified. Be prepared to answer any questions about your program's operation.

Marking

Requirement	***Mark***
BadAccountInputException written correctly	**2**
TODO #1 Bad account number code is caught correctly	**3**
TODO #2: Checkdigit calculation is correct	**10**
TODO #3 Name input exception is caught correctly	**3**
TODO #4 Account start date exception is caught correctly	**3**
Can answer questions about your code indicating you understand its operation, along with the questions listed at the start of this lab	**4**
Total:	**25**