Lab Exercise 5: Exceptions
CS 2334

February 13, 2018

Introduction

This lab focuses on the use of Exceptions to catch a variety of errors that can occur, allowing your program to take appropriate corrective action. You will implement a simple bank program that allows the user to specify an bank command (deposit, withdraw, transfer). Your program will parse these inputs, perform the operation and print out the result. If an error occurs during any of these steps, your program will catch the errors and provide appropriate feedback to the user.

Learning Objectives

By the end of this laboratory exercise, you should be able to:

1. Create a program that interacts with a user through text
2. Implement and throw a custom Exception
3. Robustly handle Exceptions with a try/catch/finally block

Proper Academic Conduct

This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Preparation

1. Download the lab5 partial implementation from canvas.
2. Make sure all classes are in the default package. You may need to drag and drop the classes into the src folder to make this happen.
3. Make sure that JUnit 4 or 5 is on the build path. It should appear in your project folder under the package manager. In addition, your tests should have errors if JUnit is not present. To resolve this, go to Project > Properties > Java Build Path > Add Library > JUnit and select the correct JUnit version and click finish.

User Interaction

Each line that is typed by the user is interpreted as a potential expression. Valid expressions consist of a sequence of one, two or four tokens (each token is separated from the preceding token by one space), and may take on one of the following forms:

- 1 token: [**quit**]. The program responds by exiting
- 2 tokens: [**deposit N**], where N is an integer. The program responds by adding N to the variable accountOneMoney.
- 2 tokens: [**withdraw N**], where N is an integer. The program responds by subtracting N from the variable accountOneMoney. It is an error if N > accountOneMoney (try to withdraw more than is available)
- 4 tokens: [**transfer A B N**], where [A B] is either [one two] or [two one]. Transfer N from account A to account B. It is an error if N > amount in A (try to transfer more than is available)

Inputs resulting in an illegal integer operation or not following one of these formats result in the display of a specific error message.

Below is an example interaction with a user. Note that both the user's input and the program's response are shown.

```
Enter a command: quit, deposit, withdraw, transfer
deposit 5
Result: Input was: deposit 5
Account Balances:
Account One: 5
Account Two: 0
-------------

Enter a command: quit, deposit, withdraw, transfer
withdraw 4
Result: Input was: withdraw 4
Account Balances:
Account One: 1
Account Two: 0
-------------

Enter a command: quit, deposit, withdraw, transfer
withdraw 2
Result: Not Enough Money!
Account Balances:
Account One: 1
Account Two: 0
-------------

Enter a command: quit, deposit, withdraw, transfer
transfer two one 4
Result: Not Enough Money!
Account Balances:
Account One: 1
Account Two: 0
-------------

Enter a command: quit, deposit, withdraw, transfer
transfer one two 4
Result: Not Enough Money!
Account Balances:
Account One: 1
Account Two: 0
-------------

Enter a command: quit, deposit, withdraw, transfer
deposit 7
Result: Input was: deposit 7
Account Balances:
Account One: 8
Account Two: 0
-------------

Enter a command: quit, deposit, withdraw, transfer
transfer one two 4
Result: Input was: transfer one two 4
Account Balances:
Account One: 4
Account Two: 4
____________

Enter a command: quit, deposit, withdraw, transfer
transfer two one 3
Result: Input was: transfer two one 3
Account Balances:
Account One: 7
Account Two: 1
____________

Enter a command: quit, deposit, withdraw, transfer
transfer a b c
Result: Illegal input: Illegal Argument
Account Balances:
Account One: 7
Account Two: 1
____________

Enter a command: quit, deposit, withdraw, transfer
withdraw a
Result: Illegal input: Illegal Argument
Account Balances:
Account One: 7
Account Two: 1
____________

Enter a command: quit, deposit, withdraw, transfer
deposit a
Result: Illegal input: Illegal Argument
Account Balances:
Account One: 7
Account Two: 1
____________

Enter a command: quit, deposit, withdraw, transfer
transfer one two a
Result: Illegal input: Illegal Argument
Account Balances:
Account One: 7
Account Two: 1
____________

Enter a command: quit, deposit, withdraw, transfer
quit
Result: quit
Account Balances:
Account One: 7
Account Two: 1
____________
```

Class Design

Below is the UML representation of the set of classes that you are to implement for this lab. It is important that you adhere to the instance variables and method names provided in this diagram (we will be executing our own JUnit tests against your code). In this diagram, you are seeing some new notation: the dashed open arrow means that there is some loose relationship between the classes. It is not an *is-a* relationship (class inheritance), or even a *has-a* relationship (a class or instance variable referring to another class). This relationship is much more nebulous – here, we are acknowledging that one class has local variables that reference another class.

The *Exception* class is provided by the Java API. The *BankException* class is derived from *Exception*. Note that we do not add any extra functionality to this class. It exists simply so that our try/catch/finally blocks can check for the particular exception. By making a new *Exception* type, we enable the program to pass more detailed information about what errors have occurred, just by having a more specific class. To provide even further granularity to our errors, we also construct the following classes:

1. QuitException: thrown when the user inputs ”quit” to end the program (case insensitive).

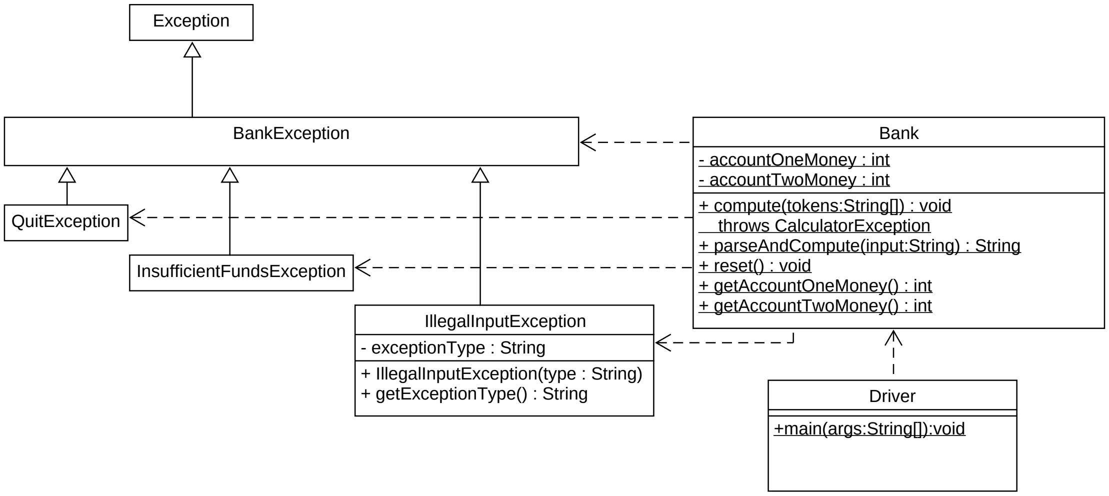
2. InsufficientFundsException: thrown when the program attempts to remove more money from an account than it currently has.

3. IllegalInputException: thrown when the input does not match with a format that we expect. This exception also has a private String variable *exceptionType* and a getter for the variable. *exceptionType* is set through the IllegalInputException constructor and gives some more detail about what kind of input error occurred. Valid strings to give are:

   (a) “Illegal Token Length”: when the number of tokens is less than 1, equal to 3, or greater than 4.

   (b) “Illegal Argument”: when a token does not match the type of token expected in its position. For example, in the input ”deposit aaa”, ”aaa” is an illegal argument, as it is not an int.

   (c) “Illegal Command”: when the first token is not supported by the program. The program accepts “deposit” and “withdraw” for the 2 token case, so “Juggle 5” would cause this exception.


Exception
BankException
QuitException
InsufficientFundsException
IllegalInputException
- exceptionType : String
+ IllegalInputException(type : String)
+ getExceptionType() : String
Bank
- accountOneMoney : int
- accountTwoMoney : int
+ compute(tokens:String[]) : void
throws CalculatorException
+ parseAndCompute(input:String) : String
+ reset() : void
+ getAccountOneMoney() : int
+ getAccountTwoMoney() : int
Driver
+main(args:String[]):void

The *Bank* class provides two methods for processing inputs. The following method is responsible for taking as input a single String that is to be interpreted as an expression:

```
public static String parseAndExecute(String input)
```

This method:

1. Separates the String into a set of tokens (substrings that are separated by one or more spaces)
2. Calls *execute()* to evaluate the expression
3. Returns a String based on code execution. Just returns the input if code executes without exception. If there is an error, what error message is returned is dependent on what exception type is caught. We have the following possibilities for return:
   (a) No Exception Caught: "The input was: " + input
   (b) QuitException Caught: "quit"
   (c) IllegalInputException Caught: "Illegal input: " + e.getExceptionType()
   (d) CalculatorException Caught (InsufficientFundsException is the only remaining case): "Not Enough Money!". We catch the more general exception here to make the code coverage measure happy.

The *execute()* method is responsible for interpreting the set of tokens and producing a result:

```
    public static void execute(String[] tokens) throws CalculatorException
```

If there are two or four tokens that make up a valid expression, then this method executes the bank command. In all other cases, this method throws a *CalculatorException*, selecting the appropriate type (*QuitException*, *InsufficientFundsException*, or *IllegalInputException*) to throw. When there is exactly one token that is equal to the String "quit" with any casing, a QuitException is thrown. The details for the appropriate exception message are given in the code skeleton that we provide.

Also in the *Bank* are the variables accountOneMoney and accountTwoMoney. These represent two accounts held in the bank. A more detailed explanation is given in the code. In addition, there is a getter method for either variable and a method reset() that sets both to 0. These methods are used for testing purposes.

You must implement your own JUnit test class called *BankTest*. We have provided *BankSampleTest* that gives an outline as to how to test with Exceptions.

The *Driver* class is provided and is responsible for opening an input stream from the user and repeatedly reading and evaluating lines of input until a *quit* has been received.

Implementation Steps

1. Complete the implementation of the *Bank* class.

2. Implement a JUnit test for the *Bank* class.

Final Steps

1. Generate Javadoc using Eclipse.

    - Select *Project/Generate Javadoc...*
    - Make sure that your **lab5** project is selected, as are all of your java files
    - Select your *doc* directory
    - Select *Private* visibility
    - Use the default destination directory

- Click *Finish*

2. Open the *lab5/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that all of your classes are listed and that all of your documented methods have the necessary documentation.

3. If you complete the above instructions during lab, you may have your implementation checked by one of the TAs.

Submission Instructions

Before submission, finish testing your program by executing your unit tests. If your program passes all tests and your classes are covered completely by your test classes, then you are ready to attempt a submission. Here are the details:

- All required components (source code and compiled documentation) are due at 11:59pm on Thursday, February 15) **Submission must be done through the Web-Cat server.**

- Use the same submission process as you used in lab 4. You must submit your implementation to the *Lab 5: Exceptions* area on the Web-Cat server.

Hints

- Recall that a try/catch/finally block can have multiple catch statements. This allows you to check for different errors and respond in kind.

- Be careful not to deviate from the specification. It is okay to have empty exception classes, as these exceptions when thrown give more specific information to the program and the user about what error occurred.

- It is bad coding style for a *catch* statement to catch all *Exceptions* (unless you really mean to catch all exceptions). Instead, you should only catch the specific exceptions that you expect to happen. This way, other, unexpected exceptions will still result in a halt of your program, making it easier to track down problems.

- Although Java allows *switch* statements to be used with Strings, code coverage computations do not work properly for these cases. Instead, you should use if/else cascades to implement a sequence of tests of this type.

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Correctness/Testing: 45 points

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is a product of the fraction of **our tests** that your code passes and the fraction of **your code** that is covered by *your tests*. In other words, your submission must perform well on both metrics in order to receive a reasonable grade.

Style/Coding: 20 points

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

Design/Readability: 35 points

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation (up to 10 points)
- Missing or inappropriate inline documentation (2 points per violation; up to 10 points)
- Inappropriate choice of variable or method names (2 points per violation; up to 10 points)
- Inefficient implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incorrect implementation of an algorithm (minor errors: 2 points each; up to 10 points)

If you do not submit compiled Javadoc for your project, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions(where points may be deducted).

Bonus: up to 5 points

You will earn one bonus point for every two hours that your assignment is submitted early.

Penalties: up to 100 points

You will lose ten points for every minute that your assignment is submitted late. For a submission to be considered *on time*, it must arrive at the server by the designated minute (and zero seconds). For a deadline of 9:00, a submission that arrives at 9:00:01 is considered late (in this context, it is one minute late).

After 15 submissions to Web-Cat, you will be penalized one point for every additional submission.

For labs, the server will continue to accept submissions for three days after the deadline. In these cases, you will still have the benefit of the automatic feedback. However, beyond ten minutes late, you will receive a score of zero.

The grader will make their best effort to select the submission that yields the highest score.