Assignment #3: Transaction Broker, Due Friday, November 5, 12:00 PM

In this assignment you will apply the various concepts and programming constructs your learned in the first six lectures to write a Java class that processes a stream of financial transactions. **This is a big jump from assignments 1 and 2. Be sure to take time to map out your methods, logic, etc. in pseudocode before you start writing Java – follow all the steps of the Software Engineering process that we discussed in class! And, to state the obvious – don't procrastinate! Start working on this now! You have a little over two weeks!**

Assignment Goals

1) Practice application and use of: loops, conditions, methods, variables, operators.
2) Practice translating requirements into code – this document tells you what has to happen, and your job is to follow the software engineering process and translate it all into a working program
3) As always – very careful reading of, and adherence to, requirements!

Program Requirements

Class and Methods

- You will write a single Java class called `TransactionBroker`
- Your class must include the methods whose headers are shown below. Their behavior is as follows:
    - `isValidDouble` must return true if `str` can be parsed into double, false if it can't.
    - `isValidOperator` must return true if `operator` is a valid operator (more on operators below), false if it is not.
    - `operation` must return the result of the operation specified by `operator` being run with `op1` and `op2` as its operands
    - `exponentiate` must return the result of raising `base` to the `exponent` power
    - `main`, as always, is the method that will execute when your program is run

```
public static void main(String[] args)
public static boolean isValidDouble(String str)
public static boolean isValidOperator(String operator)
public static double operation(double op1, double op2, String operator)
public static double exponentiate(double base, double exponent)
```

Note: A slide about dealing with exceptions is copied at the end of this document.

Logic

The command line input to your program will be a series of changes to make to a financial account. The rules regarding the account, the transaction processing, and the format of the operations are as follows:

1. The account, as well as all operations/transactions, must deal with all numbers in a way that includes both dollars and cents. So, for example, it must be able to add 1.37 to the account balance.
2. When your program starts, the account has $500 in it before any transactions are processed.
3. If a transaction causes the account balance to drop below $500, the user must be charged a $20 penalty and be informed. For example, if the account had $500 and then $20 is subtracted, the user must then see the **exact** two lines shown below. Note that we are NOT dealing with formatting the cents to always have exactly 2 decimal places.

```
Your last transaction lowered your balance to $480.0
You have been charged a low-balance penalty of $20.0
```

4. After every time the balance in the account changes (including being charged the low-balance-penalty described above), the user must be informed of their new balance, as follows:

```
Your balance: $460.0
```

5. Transactions can be a single number, in which case that number should be added to the account balance (e.g., our example input shown below starts with 5 transactions that are made of a single number: `5 -25 10 25 5.25`.) Each of such number must be added, one at a time, to the account balance.

6. Transactions can also be mathematical operations. In the case of mathematical operations, the <u>result of the operation being done on its two operands</u> is added to the account balance. Our example input includes the following transactions that are mathematical operations: `10 sub 60, 2 mul 50, 100 div 5, 115 mod 10, 2 pow 3, 50 add 50`
   a. The mathematical operations that are supported, as shown in the example are subtraction (sub), multiplication (mul), division (div), modulo (mod), exponentiation (pow), and addition (add)
   b. **If a user enters any operator other than the 6 operators we just listed off (for example, "blah"), the program must output the following error message and then exit:**

      ```
      "blah" is not a valid operator. Transactions can't be processed.
      ```
   c. If an operator is <u>not</u> preceded by one operand <u>and</u> followed by another, you must print out the following error message and end your program (I'm using mod as an example – replace that with the operator in question in your actual error message):

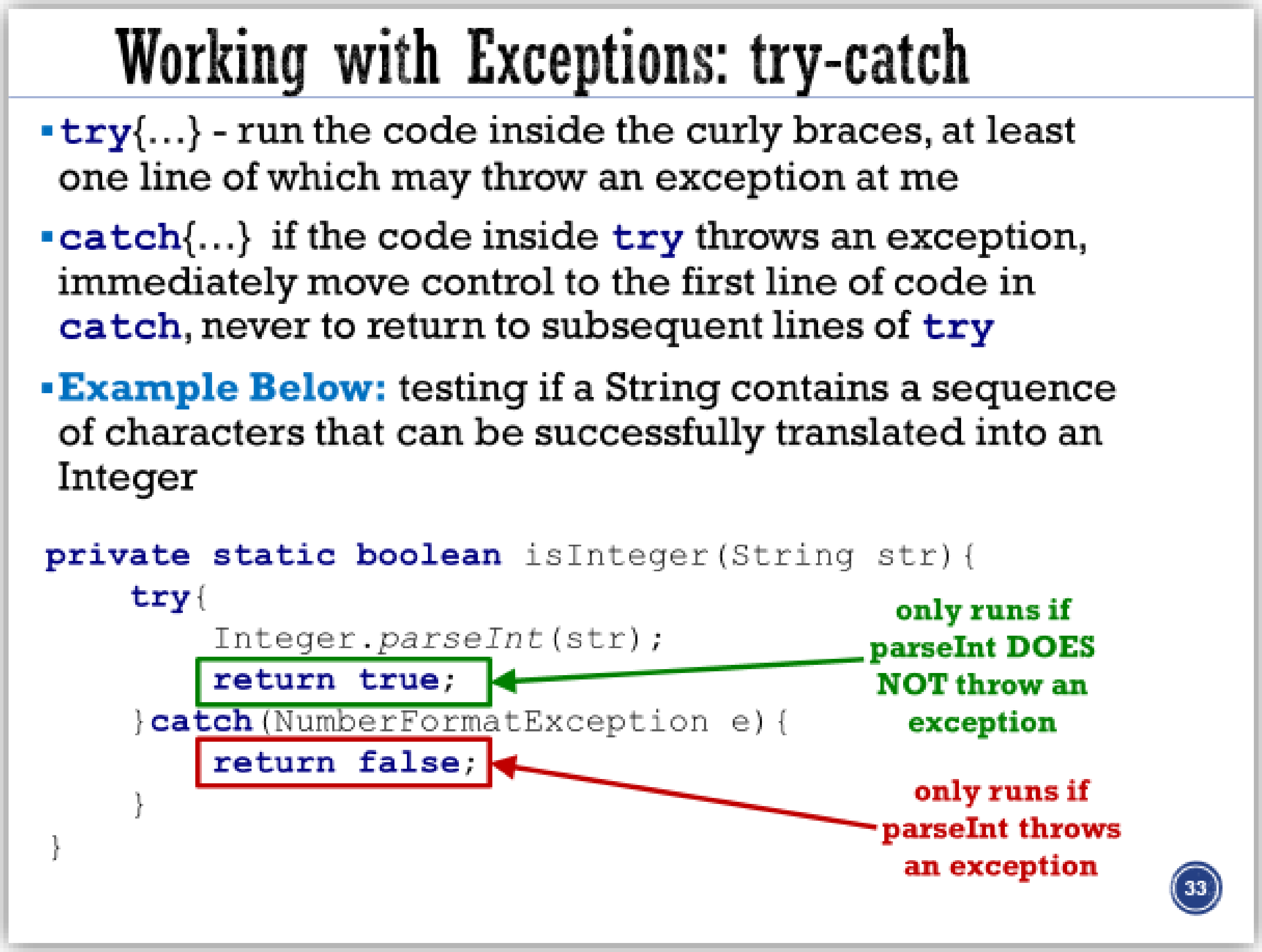
      ```
      The mod operator must be preceded by, and followed by, numeric operands
      ```

7. Note that all input values are (like all command line arguments) separated by a space only – there are no commas or any other special character indicating where one transaction ends and the next one begins, nor are there quote marks combining mathematical operations into a single command line argument.
8. After you have processed all of the transactions, you must print out a single line with twenty starts (*) on it, as shown in the example output. You will then print out the final balance, as well as the sum of all the low-balance-penalty charges that the user was charged during the series of transactions.
9. All messages must be formatted exactly as they are in these requirements and in the examples below. Changing anything about the messages will cause the tests to fail on your code.
10. **Your program will start, process the transactions specified on the command line, print out the required output, and then end.**

<u>Example Command line input:</u>
```
5 -25 10 25 5.25 10 sub 60 -10 2 mul 50 1.75 -2 20.37 100 div 5 115 mod 10 2 pow 3 50
add 50 20 -305
```

<u>Example Program Output:</u>
```
Your balance: $505.0
Your balance: $480.0
Your last transaction lowered your balance to $480.0
You have been charged a low-balance penalty of $20.0
Your balance: $460.0
Your balance: $470.0
Your balance: $495.0
Your balance: $500.25
Your balance: $450.25
Your last transaction lowered your balance to $450.25
You have been charged a low-balance penalty of $20.0
Your balance: $430.25
Your balance: $420.25
Your balance: $520.25
Your balance: $522.0
Your balance: $520.0
Your balance: $540.37
Your balance: $560.37
Your balance: $565.37
Your balance: $573.37
Your balance: $673.37
Your balance: $693.37
Your balance: $388.37
Your last transaction lowered your balance to $388.37
You have been charged a low-balance penalty of $20.0
Your balance: $368.37
********************
```

```
Your final balance: $368.37
The total you were charged in penalties: $60.0
```



Working with Exceptions: try-catch

- try{…} - run the code inside the curly braces, at least one line of which may throw an exception at me
- catch{…} if the code inside try throws an exception, immediately move control to the first line of code in catch, never to return to subsequent lines of try
- **Example Below:** testing if a String contains a sequence of characters that can be successfully translated into an Integer

```
private static boolean isInteger(String str){
     try{
          Integer.parseInt(str);
          return true;
     }catch(NumberFormatException e){
          return false;
     }
}
```

only runs if parseInt DOES NOT throw an exception
only runs if parseInt throws an exception
33