COSC 1P03 Lab 3
Feb. 24-28, 2014

Exercise 1
Exceptions, Packages, and Multiple Classes

Estimated Time: 20 min

This first exercise deals with two basic concepts: exceptions and using classes in different packages.

You already know from lecture that exceptions are used for reporting on contract violations, errors, exception events, etc. As a reminder, an exception is just any class that's a subtype of an `Exception`. For this exercise, we'll be using an *unchecked exception*. That means three things:

1. The assumption is that this wouldn't be thrown unless something was actually outright designed incorrectly (i.e. it isn't a typically-anticipated error)
2. The compiler won't force you to use the tool in a try/catch block
3. It's a subtype of `RuntimeException` (this is simply how we indicate points 1 and 2)

Download the accompanying files for this lab. You'll find three source files: two in the `wicker` package, and one in the `cage` package.

The `cage` package contains the client program (i.e. the class that acts as the main class, and makes use of the `wicker` classes).

All you need to do for this exercise is to ensure that you can compile and run the included files.
This is important, because you need to understand how packages and exceptions work for the other exercises (and for your assignment. And for Computer Scienceing).

So, open up all of the files in your IDE of choice, compile, run, and ensure that you understand what's going on before you continue.

Note: You should be able to tell from the code, but the last line of the program is an intentional crash (as it triggers a thrown exception, outside of a try/catch block). Take a quick look at the *stack trace* that's displayed when it crashes.

Exercise 2
Fun with Numbers!

Estimated Time: 30 min

For this exercise, all you're going to do is to write a method that accepts a number, displays the digits one-by-one, and then indicates the sign of the number.

Background: As you should know by now, numbers (or any primitive types) are represented by a fixed number of binary digits (or bits). The number of bits allocated defines the number of representable states. For example, a signed byte can store any of 256 different values (from -128 to +127). An integer in Java can represent between -2,147,483,648 and +2,147,483,647. A long, on the other hand, can represent between -9,223,372,036,854,775,808 and +9,223,372,036,854,775,807.
The point? There are two:

1. We have limitations on what we can represent with primitive types (the ADT in the assignment addresses this limitation)
2. The connection between the number of binary digits (always precisely 64 for a long) and base-ten digits (from 1 to 19 for a long) involves a bit of math

To assist you with your assignment, for this exercise, you'll simply be separating the 1's 10's, 100's, 1000's, etc. columns from some number. You'll be using a **`long`**, to allow for a good variety of digits.

All you need to do is to write your method to accept the long as a parameter, and display appropriate output. e.g.:

```
Data:     12345
Digits:   1    2    3    4    5
Sign:     Positive

Data:     0
Digits:   0
Sign:     Zero

Data:     -9876543210
Digits:   9    8    7    6    5    4    3    2    1    0
Sign:     Negative
```

Tips:

- Don't forget about the `%` operator!
- You'll also still need division
- Because you're simply displaying the values (rather than, say, storing them in some form of ADT for later use), feel free to just use String concatenation to build up a solution
- Depending on your approach, you may find the case of zero to be a bit more annoying; if so, get a general solution working first, and then come back to it
- When dealing with negatives, remember how the *fraction* ADT example in class handled it
- When specifying **`long`** values in Java, the convention is to add a capital L at the end (e.g. `12345L`). For values that could be represented as an **`int`** anyway, it doesn't really matter either way; but keep it in mind for values with too many digits for an integer (e.g. `9876543210L`)

Exercise 3
Fun with Number-Resembling Characters!

Estimated Time: 20 min

Exercise 2 should have been relatively straight-forward in concept: mathematical operations to break down a number makes sense. It's a number, so it's math-friendly.

On the other hand, there are other ways to think of (and thus represent) a number: it's a sequence of digits, right? That means it can be represented with a string. This is effectively what we're doing when we write (or type) a number out, or store it in plaintext format.

For this exercise, you'll be repeating the functionality of Exercise 2, except for two changes:

- This time the method receives a String
- Instead of printing each digit, print half the value of each digit
  - This is just to verify that you can get it into a number, if needed

e.g.

```
Data:     12345
Digits:   0    1    1    2    2
Sign:     Positive

Data:     0
Digits:   0
Sign:     Zero

Data:     -9876543210
Digits:   4    4    3    3    2    2    1    1    0    0
Sign:     Negative
```

Tips:

- While although you *can* get the numeric equivalent of a String via `Integer.parseInt`, since you're dealing with individual characters, it's probably easier to remember that `char`'s can be treated as numbers in Java (i.e. when you subtract 48 from a character representing a digit 0-9, the result is the integer equivalent of that character)
- Don't forget that the String class has a `.charAt` function that's pretty handy
- Since the division by two is really only intended to give you practice switching between numbers and characters as digits, it's suggested that you try repeating the functionality of Exercise 2 first, and *then* adding the division afterward

Exercise 4
I Take Exception to That!

Estimated Time: 30 min

Hypothetically, there are quite a few things that could have gone wrong with Exercise 3, aren't there? For example, what if the "number" provided to the method was "spinach"? Depending on how you coded your solution, it might have provided an answer (without realizing that such an answer would be entirely nonsensical), or it might have automatically thrown a NumberFormatException (but without your having explicitly planned on it doing so).

What we'd really like is to be able to explicitly notify that the method is being improperly used.
The logical approach would be to verify that only valid characters (i.e. 0-9; with an optional + or -, but only if it's the first character in the String) have been provided. However, that's a slightly complicated task, and one that you're expected to do for your assignment anyway.

Instead, let's assume that the client is afraid of bees, and anything resembling a bee (like a 'b' or a 'B'). Is this sounding familiar? (It better, if you've completed the first exercise!)

Your task for this exercise is to combine Exercises 3 and 1. That is, you should still have a method that receives a String, and displays the (divided) digits and sign, but it should also throw an exception if/when some character is the letter b.

Catch the thrown exception in the calling method (presumably either your *main* method or your constructor), and provide feedback to the screen/console somehow.
e.g.

```
Testing on:    12345     -9876543210    888B8     33

Data:     12345
Digits:   0    1    1    2    2
Sign:     Positive

Data:     -9876543210
Digits:   4    4    3    3    2    2    1    1    0    0
Sign:     Negative

Data:     888B8
Digits:   4    4    4Early Termination.
Reason:   Bees are not numbers!
```

Tips:

- You'll be reusing `MelissophobiaException`, but not `BeeBasket`; simply have your method throw the exception directly, to be called by whatever method invoked it. Of course, you can borrow a bit of code from BeeBasket, as it has a very similar test in it
- You'll need to either remove the package line from `MelissophobiaException`, or import from that package (I'd suggest the latter)