COMP 2522
Object Oriented Programming 1
Lab 6

Yue Wang
`yue_wang@bcit.ca`

Due at or before 23:59:59 on Sunday, March 12, 2023

1 Introduction

This is a short lab that will show you how we can write and use code that generates a custom exception.

2 Submission requirements

This lab must be completed and pushed to GitHub Classroom at or before 23:59:59 on Sunday, March 12, 2023.

Don't wait until the last minute as you never know what might happen (computer issues, no Internet, power outage, ...). Submit often, and please make sure your comments are short and clear and specific.

You must not ignore Checkstyle warnings or the code style warnings produced by IntelliJ. **You must style your code so that it does not generate any warnings.**

When code is underlined in red, IntelliJ is telling us there is an error. Mouse over the error to (hopefully!) learn more.

When code is underlined in a different colour, we are being warned about something. We can click the warning triangle in the upper right hand corner of the editor window, or we can mouse over the warning to learn more. If the warning seems silly, tell me about it, and I will investigate and possibly ask the class to modify some settings to eliminate it once and for all!

3 Grading scheme

Figure 1: This lab, like all labs, will be graded out of 5

Your lab will be marked out of 5. Consult the rubric available on D2L. On a high level, expect that marks will be assigned for:

1. (3 point) Meeting the functional requirements in this lab, i.e., does the code do what it is supposed to do?
2. (1 point) Meeting non-functional Java OOP design requirements such as encapsulation, information hiding, minimizing mutability, maximizing cohesion and minimizing coupling, making good choices about associations and thinking about Demeter, etc.

3. (1 point) Writing code that is self-documenting and well-styled that generates no warnings from IntelliJ or Checkstyle, with correct and complete Javadocs, using short and atomic methods correctly encapsulated in classes that are well defined.

4 Setup

Please set up your lab in the following manner:

1. Visit this URL to copy the template I created to your personal GitHub account, and then clone the repository that gets generated to your laptop.:

   `https://classroom.github.com/SOMETHING`

2. After you accept the lab project, GitHub Classroom will clone the project and create a new code repository for you. Go ahead and visit it by clicking the URL.

3. Do you see the button at the top of the repository named Code? Please use that to acquire the URL of your new GitHub repo so you can clone it to your computer. Click Code. In the dropdown window that opens, copy the URL.

4. The welcome window for IntelliJ lets us create a project from version control (Get from VCS). Please create a new local clone of your fresh lab repo.

5 Style Requirements

Continuing with this lab, there are code style requirements that you must observe:

1. Your code must compile.
2. Your code must not generate any IntelliJ code warnings or problems.
3. Your code must execute without crashing.
4. Your code must not generate any Checkstyle complaints (unless they are complains I have specifically said you can ignore).
5. Don't squeeze your code together. Put a blank space on either side of your operands, for example. I will be assessing the readability and clarity of your code.
6. All of your program classes must be in package ca.bcit.comp2522.xxx (replace xxx as required by the assignment, lab, quiz, etc.). For example, today's work should go in the ca.bcit.comp2522.labs.lab0X package.
7. All classes require Javadoc comments including the @author tag AND the @version tag. Class comments go after the package and import statements. There should be no blank lines between a class comment and the class it describes.
8. Public constants require Javadoc comments, private constants do not. Constants should be the first thing in your class.
9. Constants should be static and final, are often public, and should be followed by instance variables.
10. Instance variables have private or protected visibility, never public or default visibility.
11. Public and protected methods require Javadoc comments including @param tag(s)the @return tag, and the @throws tag where needed (we won't worry about throws until we talk about exceptions in depth).
12. A method's comment must begin with verbs describing what the method does, i.e., Calculates, Returns, Sets, Prints, etc. Note that we use present tense in Java – Returns, not Return. Prints, not Print.

13. The @return and @param tags go AFTER the description.
14. Private methods require non-Javadoc comments (the comments that start with a slash and a single asterisk).
15. Do comment complicated logical blocks of code inside methods with sparse, clear inline comments.
16. Do not use magic numbers (you must use constants instead). Remember that a magic number is any numeric literal. A constant can be local in a method (use the final keyword with it) or class-level (make it static and ALL_CAPS).
17. All method parameters that are object references must be made final (so we don't forget parameters are passed by value):
    (a) Nice to prevent erroneous assignments, and necessary if parameter is referenced by inner class, but that is perhaps a little advanced for now.
    (b) References made final mean that the reference, once pointing to an object, cannot be changed to point at a different object.
18. Consider making your methods final:
    (a) Making a method final prevents subclasses (those that inherit the method) from changing its meaning.
    (b) Final methods are more efficient (the methods become inline, thus avoiding the stack and generating overhead).
19. Data and methods that work together must be encapsulated in the same class.
20. Code duplication must be minimized.
21. The values of local variables that are primitives are set when they are declared, and local variables are not declared until they are needed.
22. Every class that stores state needs an equals method, a hashCode method, and a toString method.
23. In general, we enforce a fairly hard maximum method length of 20 lines of code. Aim for 10 (excluding braces).

6 Implementation requirements

The following must be added:

1. **Create a NotAnIntegerException class**. Make it a checked exception. Checked exceptions extend java.lang.Exception. Why do we call it a checked exception? Because the compiler checks if we are adding a throws declaration to a method header when the method throws the checked exception. (As an aside, recall that an unchecked exception is not added to the throws declaration in a method header). Review the lecture slides for a solid template!
2. **Create an InputReader class**. InputReader contains:
    (a) a single instance variable of type Scanner
    (b) a zero-parameter constructor which instantiates a new Scanner object and assigns it to the instance variable
    (c) a single function called getNumber( ) that:
        i. accepts no parameters and returns an int
        ii. throws a NotAnIntegerException (put this in the method header)
        iii. wraps a call to the Scanner object's nextInt( ) method in a try-catch
        iv. catches any non-integer input by catching a java.util.InputMismatchException
        v. clears the Scanner's buffer in the catch clause by invoking the nextLine method on it and not assigning the return value to anything

   vi. throws a new NotAnIntegerException from the catch clause with the message “That’s not an integer!”
   vii. returns the user’s input.

3. **Create a NumberReader class**. NumberReader contains:

   (a) a single instance variable of type InputReader which should be initialized in a zero-parameter constructor

   (b) a method called public void guessNumber() that:

      i. uses an infinite loop to invoke the InputReader’s getNumber( ) method and acquire a value from the user
      ii. adds the value to a running total, or stops the loop if the user types a zero
      iii. displays the sum of the input numbers on the screen after the user types a zero
      iv. contains a try catch block to catch a NotAnIntegerException that getNumber( ) might throw, and displays an error message if the user types something that is not an integer.

   (c) a main method that declares and instantiates a NumberReader object and invokes the guessNumber() method on it.

Sample Run:

```
Enter an integer, 0 to stop: e
Invalid entry -- that is not a number!
Enter an integer, 0 to stop: 1
Enter an integer, 0 to stop: 2
Enter an integer, 0 to stop: 0
The sum of numbers entered is 3
```

Good luck, and have fun!