1 Objectives

- Use streams and exceptions together
- Learn the `wc` Linux command
- Use stream processing to determine the number or words, lines and bytes in a file
- Create your own exception.
- Use Javadoc

2 Preliminaries

Make a new project for today's lab. Import the files for this lab.

3 Exercise 1: Streams and Exceptions

An important ADT in computer science is the stream. A *stream* carries an ordered sequence of data of undetermined length. For flexibility, Java has many streams. Streams are used to read and write files as well as to communicate over a network.

By now, you are familiar with the Scanner class. It uses a stream to input text and interprets it as integers, booleans, string, etc as requested. Up till now, you have had to assume that a user entered an integer when your program asked for one and not some other type such as a boolean. Now you can use exceptions to take charge and handle bad user input.

Create a Java class named `StubbornInput` that will prompt for user input. This class will contain methods similar to the Scanner class but these methods will stubbornly ask the user for input again and again until data of the correct type is entered.

This class needs methods for

```
int     nextInt(String prompt, String errorMessage);
boolean nextBool(String prompt, String errorMessage);
```

The first parameter of each method is a string to print when requesting information from the user. The second parameter is a string to print when yelling at an unruly user. Using the parameters, each method should repeatedly ask the user for input until the correct type of input is given. You will need to catch and handle the exceptions that are generated when bad user input is given. Question: what is the easiest way to find out what exception will be thrown?

There are two ways to setup your exception handling:

- You can use recursion and have your catch block call your function again.
- You can use iteration via a loop and have the try-catch block inside your loop with a boolean for "am I dont yet". Simply return the data from inside the loop when you finally get it. Question: which form of loop is appropriate for a situation where the number of repetitions is unknown but the loop body will run at least once?

In both cases, you may need to put a call to the nextLine of Scanner in your catch block. This tells the Scanner to move along to the next possible input in case it threw the exception in mid-inputting. If you run your program on one bad input and it goes into an infinite loop, you may need this line. However, it is possible to solve this problem without it.

Run `TestStubbornInput` to test your class.

4 Exercise 2: Streams and the wc Linux Utility

4.1 Learn to use the wc program

`wc` is a UNIX utility that reads one or more input files and, by default, writes the number of newline characters, words, and bytes contained in each input file to the standard output. The utility also writes a total count of all named files if more than one input file is specified. `wc` considers a word to be a nonzero length string of characters delimited by white space (space, tab, newline, or carriage return). Open up a terminal window. In it, launch the Linux manual for `wc` by typing

```
man wc
```

Try out the `wc` command a few times in the terminal window to see what you get. For example,

```
wc *.java
wc ~/*.*
```

Running `wc *.java` in my current directory generates something like this

```
% wc WordCount.java
     103     232    2074 WordCount.java
```

which means the file `WordCount.java` has 103 lines, 232 words, and 2074 bytes (characters) in it.

4.2 Write your own wc program

`wc` is actually just a program installed in Linux. Write your own `wc` in Java. Write a Java class called `WordCount` that takes a file name as the command line input and prints out the information similar to that of `wc`, i.e., the number of lines, the number of words, and the number of bytes in the given file. You don't have to deal with a list of files.

The first thing you'll want to do is use a Scanner to ask the user for a filename. Then, your program must use the BufferedInputStream to read the file (it has a method we want). Look at the BufferedInputStream constructor in the Java API. It take an InputStream. You have a filename (a String). Look at the InputStream constructor. Not very helpful... Then look at the known subclasses of InputStream. Find one which takes a String filename and is named in a way which makes you think of getting input from files (aka, this class was well named).

Look at the Java API methods for BufferedInputStream. You need one which reads in data one character at a time, aka one byte at a time. Be aware that this method reads the next character in the file, moves onwards, and returns that character. The Java API tells you how to know when you have read the whole file.

A key piece of technology needed here is to be able to recognize the new line character which is used to count lines, and the white space characters which are used to count words. In Java (and any programming language that supports Unicode or ASCII code), the new line character is represented as an integer of value 10, a tab key is an integer of value 9, and a space key is an integer of value 32. You might want to declare a list of constants similar to the following in your program.

```
final static int TAB     = 9;
final static int NEWLINE = 10;
final static int SPACE   = 32;
```

Question: what do the keywords final and static mean here?

When your program reads a byte of input, compare it with the above constants. If the data is a new line character, then the program should increment the line counter by one, if it is a space or a new line or a tab, increment the word count by one. Collectively the tab, the space bar, and the new line characters are called *white space* characters. Note that if a sequence of consecutive *white space* characters precedes a non-*white space* character, they should not all increment the word count. For example, if a file content is

```
abc      xyz
```

although there are six space characters in between `abc` and `xyz`, this file contains only two words. On Windows systems, the appearance of both new line and char-

acter return constitutes a new line. In your program, you really only need to count the new line character for a line.

You also should note that special characters such as the new line character, the tab character, and the space character are all counted as bytes in a file.

To run your WordCount program, make a file and place it in your project for today's lab. (In the outer project, not the src or bin directories). Try your program on that file.

Your program must produce output in the **same** format as the Linux `wc` utility when it is run one one file.

5 Excercise 3: Create your own exception

For this exercise you are to create your very own exception. When you create a new exception, the convention is to end the name with "Exception". Let's name your new class `MyException`. For each new exception you need to create a new class which extends (inherits from) an existing exception. In general, you need to consult the Java exception hierarchy (see page 504 in Big Java) to determine which exception your new exception should inherit. See chapter 11.6 for instructions on how to create an exception. Make sure it is an unchecked exception.

Next finish the class `TestMyException`. In this class, main calls the method `testing`. You need to make changes to this method so that it throws your exception. Do not use any try-catch blocks in main. Instead let your exception end the program when it is uncaught. You should not have to edit main.

Run `TestMyException` to see your exception in action.

6 Exercise 4: Javadoc

Look at the Java API (linked off the course website). These pages are a form of documentation called a Javadoc. They show the public and protected portions of the classes in the Java libraries. In this format, they are part of the User's Manual for Java.

It is also possible to have Javadoc show the private member data and methods for classes. That would be part of the technical specification.

In this section, you will learn how to use Javadoc. First, let's have eclipse make some Javadoc. Select `Program` and `Generate Javadoc` from the Eclipse menus. The source should be your project for this lab. Pick just the `TestJavadoc` class. The destination should be `lab05/doc`. Note that you have options to include public, protected, and private information. The default is public.

Then hit finish. Many black lines of test will zip by in the console. You should not see any red ones.

In your new doc directory, you will be able to find the index.html file. Double click on it. If you get an error message, say ok and wait a minute. Either a new window will appear or double click it again to get the new window. It is looking for firefox in the wrong place. Erase the line with firefox in it and just write firefox and hit ok. It should bring up your Javadoc from then on. You should not have to do this again.

The index.html file should show a fairly empty Javadoc for `TestJavadoc`.

Look at the "A quick example of things Javadoc can do and the Java file." link on the course website.

1. Add a comment to each member data of your `TestJavadoc` class.
2. Add a comment to each method.
3. Play with multiline coments.
4. Make sure that you can control what shows in the field summary, constructor summary, method summary, field detail, constructor detail, and method detail.
5. Use the @param and @return tags

You will need to regenerate the Javadoc to update your changes.

7 Upon Completion

Commit your files to your repository.