CSCI 204 – Introduction to Computer Science II

Lab 4 - I/O Streams and Exceptions

1. Objectives

In this lab, you will learn the following:

- Using streams and exceptions
- The wc Linux command
- Use stream processing to determine the number of words, lines and bytes in a file
- Use streams to read data from an HTTP connection to a web server

2. Getting Started

As usual, you should set up your Eclipse workspace for today's lab:

1. Start Eclipse and create a new project called **`lab04`**. (Select **File → New ▪ Java Project**)

2. Next, import all of the files from the lab folder for today's lab into your workspace. (Start by highlighting the **`src`** folder and select **File ▪ Import...**)

   ```
   ~csci204/2011-spring/student/labs/lab04

   You will get a copy of
   ```

   - HTMLTagCounter.java
   - TestStubbornInput.java
   - WordCount.java

3. Finally, add the files into your Subversion repository. In Eclipse, click on `lab04` so that it is highlighted, and then right click. A popup menu will appear. Select **Team ▪ Share Project...** Then, select **SVN** as the repository type. and select

your repository for your labs.

For all future labs, you will simply be told to make a new project, import files for the lab (if necessary), and then add the new lab project into subversion. This will also be true of assignments and team projects as well, though you will need to set up a new repository for assignments and for projects. (The assignment and project guidelines will remind you of this.)

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. Java has many streams. Streams are used to read and write files as well as to communicate over a network. Essentially, if there is data flowing in or out of your program, then there is a good chance that a stream is behind it.

One challenge with streams involves placing structure on it. For example, suppose you have a sequence of characters `'1'`, `'.'`, `'2'`, and `'3'` in an input stream. How should that sequence be interpreted? As a string `"1.23"`? Or is as floating point number `1.23`? Parsing and interpreting input streams into properly typed data that your program can make use of used to be much more difficult. Fortunately, the Java API developers provided a very useful class called `Scanner` that can help with this.

By now, you are hopefully familiar with the `Scanner` class. It provides numerous methods for *tokenizing* a text stream (a process of breaking up a stream of characters into individual *tokens*, where each *token* is a sequence of characters that has some significant meaning). The class provides a wide array of methods to aid in interpreting the individual *tokens* as integers, booleans, strings, etc.. The `Scanner` class does a lot for you behind the scenes by providing not only tokenizing capabilities, but interpretive capabilities by turning those tokens into useful typed representations.

Until now, we have not worried much about *domain checking*, or *input validation* – verifying that the user entered the correct information. You probably have simply assumed 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.

1. Your exercise

Look over the file `TestSubbornInput.java` that was imported into your project. Look at the `in` object declaration of type `StubbornInput`. It is essentially acting like a simple `Scanner`, but its methods come with some extra functionality to create a prompt for the user, as well as an error message if the input was not able to be extracted from the input stream. Your job is to create the class `StubbornInput`. It will handle processing input from the input stream `System.in` (the keyboard). The class will contain two methods, both of which 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. It will also contain a constructor that should set up the Scanner contained in the class. If I didn't tell you this, how would you know what to implement? Read on…

First, you should be able to examine `TestStubbonInput.java`, and see that your `StubbonInput` class needs to define methods for:

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

You also need to define one constructor. (What type of constructor? Default? Or explicit?)

For the methods, 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. Consult the Java API to determine what exceptions should be handled.

There are two ways to setup your exception handling:

- You can have your catch-block call your method again (an algorithmic method known as *recursion*, to be covered in lecture soon!)

  OR

- You can use iteration via a loop and have the `try-catch` block inside your loop. Set yourself up with a `boolean` variable that will represent "am I done yet". Simply return the data from inside the loop when you finally get it.

As you've been learning, there are numerous ways to solve any given problem. There is no one perfect answer here. One word of advice… focus on one method, and test it.

Depending on your approach, you may need to put a call to the `nextLine` method 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 midst of processing the input. If you run your program on one bad input and it goes into an infinite loop, you may need this line.

Run `TestStubbornInput` to test your class. Be sure to commit your work into SVN when complete.

Here is an example of what you would see with an unruly user. The red, bold-faced text is the characters entered by the user.

```
Int please: a
No really, I mean an int.
Int please: yo8
No really, I mean an int.
Int please: @$%#^$^
No really, I mean an int.
Int please: 56
56
Bool please: nope
No really, I mean a boolean.
Bool please: 2
No really, I mean a boolean.
Bool please: falslala
No really, I mean a boolean.
Bool please: false
false
```

4. Exercise 2 – Streams and the `wc` Linux Utility

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 stream. 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:

```
prompt$ wc *.java
 69    292    1977 HTMLTagCounter.java
 29     84     609 TestStubbornInput.java
 56    181    1423 WordCount.java
154    557    4009 total
```

which means the file `TestStubbornInput.java` has 29 lines, 84 words, and 609 bytes (characters) in it.

1. Write your own wc program

wc is actually just a program installed in Linux. For this exercise, you will write your own wc implementation in Java. You will finish the implementation of the `WordCount` class supplied in `WordCount.java` you imported in the beginning. It has a main method for you already. You will need to implement the following methods:

- `public WordCount(File f)` – constructor, which handles opening the file `f` and counting the lines, words, and bytes. It does not print the results.
- `public void printResult()` – Print the results of processing the file.

Unlike the actual wc program, you do not have to deal with a list of files.

In your constructor, your program must use a `BufferedReader` to read the file. If you can't remember the correct constructor for `BufferedReader`, read the Java API. Recall that we are reading *character **files***. Thus, which class derived from `Reader` should we use to pass to `BufferedReader`?

Look at the Java API methods for `BufferedReader`. You need one that reads in data one character at a time. (i.e., you can not use any tokenizing classes such as Scanner.) 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.

You will need to use a `try/catch` block to handle some possible checked exceptions.

1.1. What are we reading in?

A key piece of technology needed here is to be able to recognize whitespace characters. Specifically, you will need to distinguish between the *newline* character, which is used to count lines, and the remaining white space characters (space and tab), which are used to count words. In Java (and any programming language that supports Unicode or ASCII code), the newline 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. (There is also a separate carriage return

character that has an integer value 13, but we will not worry about that here since most text file in Linux do not use it.) 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;
```

Take a moment to remind yourself what the purpose of final and static are here.

1.2. How do you count bytes?

Make a counter and increment it every time you read in another character from the file. Remember – do not the count the last character, which a special value to tell you that you are done reading the file.

1.3. How do you count lines?

When your program reads a byte of input, compare it with the above constants. On Windows systems, the appearance of both newline and carriage return simultaneously constitutes a newline. In your program, you only need to count the newline character for a line. Make a counter and everytime you see a NEWLINE, increment your counter.

1.4. How do you count words?

When your program reads a character of input, compare it with the above constants. Collectively the tab, the space bar, and the newline characters are called *whitespace characters*. Whitespace characters delimit words. There can be a lot of whitespace between words. For example, suppose a line in a file contains the following:

`abc  xyz`

Although there are six space characters in between `abc` and `xyz`, this file contains only two words.

How do you teach your program to count words? One way is to recognize that there are two possible states your reader can be in: **in a word**, and **not in a word**. It will be easiest to count words if you make a counter and increment it every time a word begins.

How do you know if you are in a word? The character you just read in is not a whitespace character.

How do you know when a word begins? If you see a non-whitespace character, and you were in the **not in a word** state, then you would increment your counter and become **in a word**.

When a word ends, you would enter the state **not in a word**. How do you know when a word ends? You see a whitespace character and you were in the **in a word** state.

NOTE – Do not worry about punctuation. A word for this purpose can be any token delimited by whitespace so “hello!” is one word not two.

2. When you are done

Create a test file called `test.txt` that contains some same text. (For example, you might just want to cut and paste this text into your program.) Save your test file in your project folder. Be sure to place the file in the outer project – do not place the file in 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 on one file. Modify the output in the program to make it generate the same output as `wc` Linux utility. You can space your output using tabs. A tab in Java is the character '\t'.

You must allow the user to enter a file name. We will run your code on our own code.

This is the output of running the `WordCount` program on `TestStubbornInput.java`:

```
29      84      609 TestStubbornInput.java
```

5. Connecting to a web server using streams

As we have observed, the Java API contains a very rich set of classes for dealing with streams. The API extends much of this functionality to make it easy to deal with network communications. It makes sense; after all, data coming into your program is a stream regardless of whether the source is a file, the keyboard, or a remote system on the Internet. When you open up your favorite web browser, and you type in a valid *Uniform Resource Locator* (*URL*), the browser sends a special command using the *Hypertext Transfer Protocol* (*HTTP*) to display the requested resource from the web server named in the URL. The browser retrieves the specified resource (which is usually a page written in *Hypertext Markup Language*, or *HTML*.) HTML is a special language designed to apply a wide range of structure based on semantic meaning to plain text. The result is a nice, standard way to make web pages not only nicely formatted, but also interactive through hyperlinks set up in the page. HTML specifies a series of special elements called *HTML tags* to allow the browser to format the page as specified. For example, the tag tells the browser that the text following the tag should be formatted as a paragraph, up to the point where a closing tag appears.

1. Your exercise

In Eclipse, open up the file `HTMLTagCounter.java` that was imported in your project. You will see a partially completed class called `HTMLTagCounter`. It is doing nothing more than grouping together useful, related static functions, one of which is the `main()` function. Examine the contents closely. There are two functions that need to be modified.

1.1. readURLFromUser()

First, look at `readURLFromUser()`. As it stands, it does no error checking / exception handling to verify that the string the user enters is a proper URL. This needs to be modified to allow the proper exception to be caught, report an error message to the user, and then have the user try again. You will need to set up a looping construct that does not terminate until the user enters a 'q' to quit, or a valid URL is entered. (Valid, meaning, the URL object can be created without throwing an exception. It has nothing to do with establishing a connection to the server yet.) Check the documentation for `java.net.URL` and handle the exception(s). How you do it is up to you, but it must be handled. You can not simply use a throws clause in the method header.

1.2. main()

Next, carefully read through the existing `main()` function. The comments are detailed. The code is there to open an HTTP connection to the specified URL. There is even code there that creates a `BufferedInputStream` to the remote web server.

Your job is to add the code to start reading data from the input stream, one character at a time, and look for two HTML tags -- and <ul>. You do not need to do anything but count the number of times each of these tags occurs, and print out the final tally.

NOTE: - an HTML tag may take on one of the following forms:

```
<p attr_name="blahblah">

<P>

<P attr_name=”blahblah”>
```

In other words, do not simply hard code checks for the string <p>, as you will miss a lot of tags. You need to check for both cases. Either the tag identifier will be surrounded by the open and close brackets with no space, or it will have an open bracket with a space immediately after it, indicating that there are one or more attributes associated with the tag.

2. Wrapping it up

Be sure you are catching all possible exceptions, including those that may occur because of a bad URL. Test out several URLs, and intentionally create some malformed URLs (for example, misspell something, leave a ':' out, use ftp://, etc.) Make sure that all cases are handled.

As you are working on the project, do not forget to use the debugger! Also, you should expect to have some test lines outputting various data to indicate that you are indeed matching the correct tags. However, before you check in your final work in SVN, remove that test code!

When you are done using the debugger, make absolutely certain you have stopped the debugger and the red square is not still available (the stop button).

3. Sample output on http://www.bucknell.edu

Data entered by the user is red and boldfaced.

```
Enter a valid URL [Or, 'b' for http://www.bucknell.edu, or 'q' to quit:
b
Connected to http://www.bucknell.edu
Scanning… finished.
<p>: 13
<ul>: 16
Closing connection.
```

These results might vary if the home page for bucknell.edu has been substantially modified. These results were generated on 9/20/2010.

7. Finishing up

Comment all of your code for these three exercises, and check in your work in SVN.