COSC 1P03 Lab 2
Jan. 27-31, 2014

Exercise 1
String Tokenization

Estimated Time: 30 min

Tokenization is the process of taking a long string of text and breaking it up into *tokens* — smaller sequences of text — using breakpoints defined by *delimiters*.

For example, if the text were `hello world`, if we were to assume a delimiter of the *space* character, then the two tokens would be `hello` and `world`. Tokenization can be manually implemented, but because it's so common Java provides multiple tools for doing it automatically.

A String Tokenizer is an object that accepts a string and a list of delimiters (also typically defined in a string), performs the splits, and returns one token at a time when asked. You might have noticed that this behaviour is part of what ASCIIDataFile can do.

For this exercise, you'll be using Java's `StringTokenizer` class, provided in the `java.util` package (so remember to `import java.util.*`). Refer to the API specifications for details on use (http://docs.oracle.com/javase/7/docs/api/index.html?java/util/StringTokenizer.html).

All you need to do for this task is to use an ASCIIPrompter to keep accepting strings from the user until they enter an empty string (i.e. until they simply hit OK without typing anything). Each entered string should be broken up by a StringTokenizer using delimiters of "`,.\t ^`". Display each token on a new line (whether you use ASCIIDisplayer or System.out.println is up to you).
e.g. for `hi.there+1P^03`, the delimiters above would yield:

```
hi
there+1P
03
```

Important: Notice that you'll need a new instance of StringTokenizer for each new string! Since you're using a prompter, this is simple. But consider how you would handle this if you were retrieving each token from a helper function, with the source of data being a text file. How would you trigger it creating a new tokenizer? How would you indicate that you'd reached the end of the file?

Exercise 2
Linked Structures

Estimated Time: 15 min

For this task, we're going to go over a very gentle refresher on sequentially linked structures. Once you're comfortable with the premise, we can then move on to specific tasks.

Start by downloading `Example.java` and `Node.java`. Compile and run it as-is.

Next, add the following to the code:

```
System.out.println("Adding \"c\" the wrong way:");
myList.next=new Node("c",null);
System.out.println(myList.item+"->"+myList.next.item+
"->"+myList.next.next);
```

Compile and re-execute. Why is that the "wrong" way? What happened to the list?
Once you understand what happened, try adding the following:

```
System.out.println("Adding \"b\" the right way:");
myList.next=new Node("b",myList.next);
System.out.println(myList.item+"->"+myList.next.item+
     "->"+myList.next.next.item);
```

Once you recompile and execute it again, can you see what was different between when you added "c" and when you added "b"?

Remembering that list manipulations can have special cases, add the following:

```
System.out.println("Adding \"z\" to the beginning:");
myList=new Node("z",myList);
System.out.println(myList.item+"->"+myList.next.item);
```

Once you're comfortable with what you have so far, take a brief look at using a pointer to traverse a list:

```
System.out.print("The last item is:");
Node ptr=myList;
while (ptr.next!=null) ptr=ptr.next;
System.out.println(ptr.item);
```

So long as you understood what's happening at each point, you're clear to move on to the next exercise. However, please do not hesitate to ask for an explanation of any points.

Important: You could simply get through this exercise by copying&pasting all of the code. However, if you don't understand what's happening at each step, then advancing doesn't help you at all. Remember that the point is to understand linked structures to prepare you for the assignment.

Exercise 3
Sorted Insertion

Estimated Time: 30 min

For this task, you'll be looking into a *sorted insertion*. You may feel free to use the SortedInsertion skeleton provided. Currently, all it does is to read from a file (`words.txt`) containing a list of tab-delimited words and dumps them onto the console.
When you have it able to perform a sorted insertion, add a method to dump the contents of the list onto the console (this will show all of the words in alphabetical order).

Suggested Plan of Action:

1. First write a method that simply adds a word to the list (e.g. Insertion at Front). Write the method to dump the contents of the list and verify that everything works so far.
2. Adapt the code to do a sorted insertion. Remember that you could encounter:
    1. An empty list (this should already be taken care of from Point 1)
    2. Insertion at the beginning of the list
    3. Insertion somewhere else
3. It's advisable to use two *walking pointers* (e.g. p and q)

Additional Tip: Don't forget that String has a `.compareTo` function, which allows you to easily compare two Strings. Refer to the Java API specifications for usage tips.

Final Note: This is not the same words file as included in the second assignment. It has had its delimiters simplified for easier reading

Exercise 4
Linked Structures and Lookups

Estimated Time: 30 min

Once you have Exercise 3 done, this part should be a piece of cake.
Your task has two parts: referencing the list from Ex. 3, and filtering out non-interesting terms.

Write a function, `valid`, that returns a `boolean` based on whether or not a provided String is valid. Validity is based on two criteria:

1. It must not be present in the list from Ex. 3.
2. It must not *be* a number, or *start with* a number.

It's advisable to first write a separate function that, when provided with String, returns a `boolean` indicating whether or not that String is currently within the list. It's suggested to use your Exercise 3 solution as your starting point for this part.
Next, write the `valid` function to first invoke the list-checking function, and then check for numbers.

Demonstrate by indicating which of the following words are valid:
for int dave10 size 2

(You should only be told that dave10 and size are valid)

Tips:
For the most part, this is simpler than Exercise 3. Rather than searching through values and manipulating the list, you're only doing the searching part. Note that, since you're not changing the list, you don't need to use two pointers for this one.

Final Note: If you were to use a more carefully-selected list of delimiters, notice that this could be used to extract the meaningful tokens from Java code. If you have time at the end of the lab, you're strongly encouraged to expand your code (including amending the list of delimiters) and using different tokens for input