SYSC 2004
Winter 2018
Lab 7

Objectives

1. Working with Files
2. Handling Exceptions
3. Non-Public Java Class (on our way to Inner Classes)

Provided: sampleCSV.csv

Submission Requirements: (Exact names are required by the submit program)

- **Without a submission you will not get a mark**
  Lab7a.java, Lab7b.java, Lab7b.properties

Part 1 –Comma-Separated Files

Comma-Separated Files (.csv) are a simple text format that maintains data in tabular-format. It is essentially a simple Excel file, in text format. It is a wonderful skill to know how to read in a CSV file because you can tap into all kinds of data.

Step-by-Step Instruction

1. First, understand what a CSV file is. Download the posted sample and open it up.
   - **Double-click** on the download file and probably, Excel will open it. Look at the headings so you get a clue as to what data is in the file.
   - Now, truly understand what a CSV file is: **Right-click** on the file and **Open-With** a text editor (e.g. Notepad). You will actually see the commas separating the fields.
2. Create a project for the lab called **lab7**, creating a single class called **Lab7a.java**. Add the call-signature for the static `main`() method.
3. CSV files are text files. You must use a `Reader` class. Our first step will be to simply open-and-close the file. This will be an exercise in exception-handling and will overcome the biggest hurdle with files … finding the file in its proper pathname.
   - Declare and initialize a `FileReader` object
   - The compiler will likely flag an error, saying the the `FileReader` constructor can throw an exception. Wrap this code in a `try/catch` statement. (Read the lecture notes on Exceptions).
   - Inside the same `try-block`, declare and initialize a `BufferedReader` object, wrapping the previous `FileReader` object.
   - Add a `finally-clause` to your `try/catch` statement in which you close the file objects.
   - First, run your program without putting the **sampleCSV.csv** in the proper folder. This will allow you to experience an exception occurring.

- Now, move the downloaded **sampleCSV.csv** file into the folder expected by NetBeans. Move the file to your project directory (**lab7**), at the same level as the `build`, `src` and `test` folders. Run your program again, and no exception should occur. If an exception occurs, check the exact filename, including uppercase/lowercase of the extension. Everything must be exact. That is why I/O is such great experience with exceptions.
- At this point, you have a program that correctly opens a file for reading.

4. It is now time to read in the data. The first line is special, because it holds the meta-data (i.e. the column headings). The remaining lines are the actual data. There are an unknown number of lines to be read.
    - Simply read the next line, using `readLine`(). Read the API for `BufferedReader`() to understand how to call the method.
    - Make sure that the End-of-File (EOF) was not reached (i.e. that the line read in is not `null`)
    - Print out the line, using the familiar `System.out.println().`
    - Run your program and be amazed at how many rows of data you have read.
5. Instead of just printing out the data on the screen, a useful program will internally store the data so that it can then be used and manipulated. We will store our data first in an `ArrayList`, because we have an unknown-and-variable number of rows to read in. Yet we also have an unknown-but-fixed number of columns for each row. We need a class to represent a row. Your first sub-task will be to create a `Row` class, according to the UML below. We are going to make one change in the procedure though; we will make Row a non-public class. The purpose is to expose you to other class formations, and pave the way for talking about inner-classes.
    - Do not create a separate class file! Instead, at the bottom of your existing file **Lab7a.java,** outside the final curly bracket, declare a class but do not make it public. Java only allows (and requires) one public class per file, but there can be any number of non-public classes at the bottom of the file.

      ```
      class Row {   }    // No public
      ```

    - Complete the code for the class `Row`, following the UML. It will be an exercise in array programming.
        - The constructor allocates an empty fixed-size array of column-values.
        - The `add`() method stores the `String value` in the array, using the `column` as an index. Assume arguments are valid.
        - The `toString`() method should printing out all columns in the row, separated by a comma. Essentially, you will turn it back into a comma-separated list, although we can be lazy and permit a final dangling comma at the end.

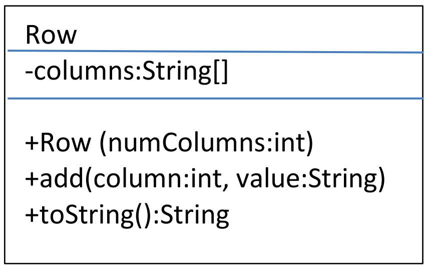
          Example: heading1, heading2, heading3,

- Go back to editing the `main`() method. Delete your previous printing of each line. Instead, as you read in each line, you will create a new `Row` object, and then add that Row object into an `ArrayList`. After all is done, then you will printout all the rows in the `ArrayList` using `Row`'s `toString`() method.
  - Declare and initialize an `arrayList` of `Rows`.
  - After reading in a `line`, split the `line` into each comma-separated parts

```
String[] columnHeadings = line.split(“,”);
```

  - Create a new `row` object
  - Iteratively (in a loop), add each of the split parts into the `row` object.
  - Add the `row` object to the `arrayList` of `Rows`.
  - After all rows are read in (after the finally-clause), iteratively print out the `rows`. The console output should look just like your previous printout, but now it is coming from a “database” implemented as an `ArrayList`.

.


Row
-columns:String[]
+Row (numColumns:int)
+add(column:int, value:String)
+toString():String