Lab Exercise 3: Reading Files
CS 2334

September 7, 2017

Introduction

There are many reasons to bring data from a file into a program. Once the data from the file are represented within a data structure, they can be analyzed or presented to a user, and can even be used to take other external actions. Importing data from a file is, therefore, a useful skill that you will employ for the rest of your academic and professional career as a computer (or related) scientist.

In this laboratory, we will load data from a file into a data structure, analyze it, and then display the information in a structured form to the user. We have provided a specification for three classes. Your task is to complete the implementation of the classes according to the specification that we provide and create the tests to make sure that these classes are implemented correctly.

Learning Objectives

By the end of this laboratory exercise, you should be able to:

1. read and understand method-level specifications (including from UML diagrams),
2. read data from the file,
3. complete the implementation of a class containing multiple instance variables and methods,
4. use the information in the file to create an array of objects, and

5. scan through an array of objects and display the items that match a given criterion.

Proper Academic Conduct

This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Preparation

1. Download:
   `http://www.cs.ou.edu/~fagg/classes/cs2334/labs/lab3/lab3.zip`

2. Import into your Eclipse Workspace:

   (a) *File* menu: *Import*

   (b) Select *General/Existing Projects into Workspace* and then click *Next*

   (c) *Select archive file*: browse to the lab3.zip file

   (d) Click *Finish*

3. Once you create the new project, it may not initially know where to find the standard Java libraries (it varies depending on your configuration).

   (a) *Project* menu: Select *properties*

   (b) Select *Java Build Path / Libraries*

   (c) If the *JRE System Library* is not listed, then select *Add library: JRE System Library* and click *Next*. Select *Workspace default*. Click *OK*

   (d) If the *JUnit Library* is not listed, then select *Add library: JUnit* and click *Next*. Select *JUnit 4*. Click *Finish*

   (e) Click *Apply* and then *OK*

Lab 3

For this lab, you will be parsing a file that contains a list of Snacks present at a Tailgate party. This file contains information about each of the Snacks. The file encodes this information in a table using the *comma separated values* (CSV) format. If you double click on this file from within Eclipse, it will attempt to open the file in a spreadsheet program, such as Excel. Alternatively, you can select the file and open with a text editor. This will open the raw file in Eclipse.

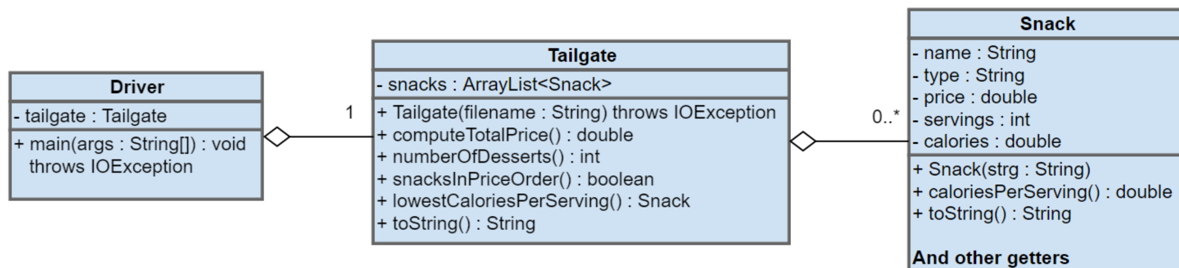
Each line of this file encodes information about exactly one Snack. Each Snack is described using five distinct values that are separated by commas in the file. These are:

1. The name (String) of the Snack.
2. The type (String) of the snack. This will be one of the following Strings: “Main”, “Side”, or “Dessert”.
3. The price (double) of the snack.
4. The number (int) of servings of the snack.
5. The total calories (double) in the snack.

Your task for this lab is to load these Snacks from a specified file, store this list of Snacks and then computes some statistics about the Snacks from this list.

Classes

The following UML diagram summarizes the classes that are to be implemented for this laboratory exercise.



The *Snack* class is responsible for representing a single Snack. Here are the methods that need to be implemented:

- The *Snack* constructor takes a String as input that corresponds to one Snack, and parses it to populate the instance variables for this class.
- *caloriesPerServing()*: returns a double equal to the totalCalories in a Snack divided by the number of servings in a Snack.
- *toString()*. This method returns a String in the format shown below.
- A full set of getters. Use Eclipse to generate these for you.

The *Tailgate* class represents a collection of Snacks at a Tailgating party. Here are the methods that need to be implemented:

- *Tailgate* constructor. This method takes as input a file name, reads through the file, creates one Snack object from each line in the file, and adds each to the ArrayList.
- *computeTotalPrice()*: sums the prices of all snacks at the tailgate. Returns a double.
- *numberOfDesserts()*: returns the number (int) of snacks at the tailgate with type = "Dessert".
- *snacksInPriceOrder()*: returns true if the snacks at the tailgate (stored in the ArrayList⟨Snack⟩) are stored in increasing price order from index 0 to the last index. Returns false otherwise.
- *lowestCaloriesPerServing()*: finds and returns the snack with the lowest calories per serving.
- *toString()*. This method returns a String in the format shown below.

The *Driver* class contains a *main()* method that:

- Creates a *Tailgate* object with a specific file name (this name may be hard-coded into the method)
- Prints out the *Tailgate* object

Reading a File

Within the Tailgate constructor, a file must be read and then parsed. An example of the code needed to pull out individual lines and add them to a list of Strings is provided below:

```
1  // ArrayList of Strings
2  ArrayList<String> list = new ArrayList<String>();
3  // Open he file
4  BufferedReader br = new BufferedReader (new FileReader("filename.txt"));
5
6  // Read first line
7  String strg = br.readLine();
8
9  // Iterate as long as there is a next line
10 while (strg != null)
11 {
12   // Add the line to the ArrayList
13   list.add(strg);
14   // Attempt to get the next line
15   strg = br.readLine();
16 }
```

Note that in your assignment, you must create a Snack object instance for each line of the file and add this instance to the *ArrayList*.

A *BufferedReader* can take different input streams as a parameter. In Lab 1, an *InputStreamReader* was used in order to take input from *System.in* (the keyboard). In this lab, a *FileReader* will be used. A *FileReader* takes a file name as a parameter. Notice that the file name is a String[1]. Line 4 of this example opens the file and turns it into a *FileReader* object that can then be used by the *BufferedReader* object.

Automatically Generating Getters and Setters in Eclipse

Eclipse is able to generate the getters and setters for a class for you. While editing the class, the steps are:

- Declare all the instance variables
- Select the *Source* menu
- Select *Generate Getters and Setters...* from the dropdown menu

[1]Remember that String literals must have quotes around them.

From this menu, you can select the getters and setters you would like generated from the dropdown associated with each class variable. You can also select all getters, all setters or both. For this lab, remember that you are creating immutable classes.

Example Input/Output

Snack Class

Consider the following code:

```
Snack s = new Snack("Banana,Side,1.7,10,10.5");
System.out.println(s)
```

The output to the console is:

```
Name: Banana
Type: Side
Price: 1.7
Servings: 10
Total Calories: 10.5
```

Note: all lines except the last are followed by a newline character ("\n").

Tailgate Class

The output of Tailgate.toString() for TailgatePractice.csv is:

```
Total Cost Of Tailgate:

14.7

Number Of Desserts:

3

Snacks In Price Order:

false

Snack With Lowest Calories Per Serving:

Name: Banana
Type: Side
Price: 1.7
Servings: 10
Total Calories: 10.5
```

The output of Tailgate.toString() for TailgateInPriceOrderPractice.csv is:

```
Total Cost Of Tailgate:

14.7

Number Of Desserts:

3

Snacks In Price Order:

true

Snack With Lowest Calories Per Serving:

Name: Banana
Type: Side
Price: 1.7
Servings: 10
Total Calories: 10.5
```

Note: in both of these cases, the last line is **not** followed by a newline character.

Final Steps

1. Generate Javadoc using Eclipse.
   - Select *Project/Generate Javadoc...*
   - Make sure that your **lab3** project is selected, as are the Driver, Snack, Snacktest, Tailgate and TailgateTest java files
   - Select your *doc* directory
   - Select *Private* visibility
   - Use the default destination directory
   - Click *Finish*
2. Open the *lab3/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that all of your classes are listed and that all of your documented methods have the necessary documentation.
3. If you complete the above instructions during lab, you may have your implementation checked by one of the TAs.

Submission instructions

Before submission, finish testing your program by executing your unit tests. If your program passes all tests and the *Snack* and *Tailgate* classes are covered completely by your test classes, then you are ready to attempt a submission. Here are the details:

- All required components (source code and compiled documentation) are due 48 hours from the end of your lab section (on Saturday, September 9). **Submission must be done through the Web-Cat server.**
- Use the same submission process as you used in lab 1. You must submit your implementation to the *Lab 3: Reading Files* area on the Web-Cat server.

Hints

- See the API for *ArrayList*

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Correctness/Testing: 45 points

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is a product of the fraction of **our tests** that your code passes and the fraction of **your code** that is covered by *your tests*. In other words, your submission must perform well on both metrics in order to receive a reasonable grade.

Style/Coding: 20 points

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

Design/Readability: 35 points

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation (up to 10 points)
- Missing or inappropriate inline documentation (2 points per violation; up to 10 points)
- Inappropriate choice of variable or method names (2 points per violation; up to 10 points)
- Inefficient implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incorrect implementation of an algorithm (minor errors: 2 points each; up to 10 points)

If you do not submit compiled javadoc for your project, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions(where points may be deducted).

Bonus: up to 5 points

You will earn one bonus point for every two hours that your assignment is submitted early.

Penalties: up to 100 points

You will lose ten points for every minute that your assignment is submitted late. For a submission to be considered *on time*, it must arrive at the server by the designated minute (and zero seconds). For a deadline of 9:00, a submission that arrives at 9:00:01 is considered late (in this context, it is one minute late).

After 15 submissions to Web-Cat, you will be penalized one point for every additional submission.

For labs, the server will continue to accept submissions for two days after the deadline. In these cases, you will still have the benefit of the automatic feedback. However, beyond ten minutes late, you will receive a score of zero.

The grader will make their best effort to select the submission that yields the highest score.