Lab Exercise 3: Reading Files
CS 2334

January 31, 2019

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 and write data to a 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. Create a new project called *lab3*
2. Download code from Zylabs
3. Import into your Eclipse Workspace (dragging into eclipse or copy and paste work the best)
4. Add a Todo.java file at the top level of your project, following the instructions for lab plans posted on canvas. You will need to fill a todo list out for each lab.
5. (Optional, but recommended) add the TodoChecker.java file to your default package. Follow the instructions on canvas to properly configure your project to run the program.

Lab 3

For this lab, you will be parsing a file that contains a list of Equipment stored in Batman's Utility Belt. This file contains information about each of the Equipment. 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 (option/right click) 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 Equipment object. Each Equipment is described using five distinct values. These are:

1. The identification information about the equipment, broken into the name and count. Several copies of an item may appear in the Utility belt (e.g. 5 batarangs), so the Equipment class keeps a count variable tracking the number of copies of the item.
2. The totalWeight (double) of the Equipment in pounds.
3. The totalPrice (double) of the Player in USD.
4. The description of the equipment.

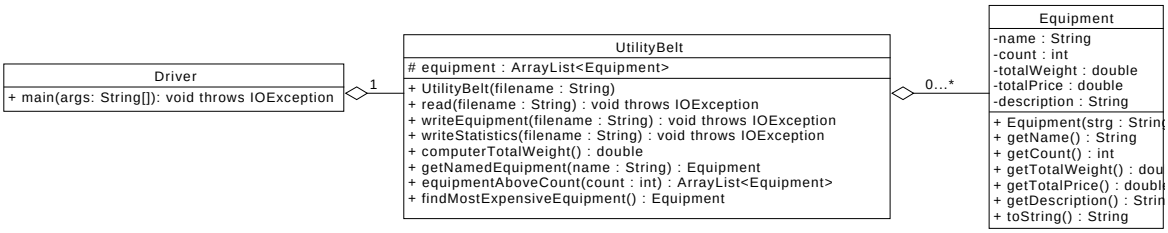
Thus, a line of the csv file representing a player would be formatted as:

```
name/count,totalWeight,totalPrice,description
```

Your task for this lab is to load these Equipment objects from a specified file, store this list of Equipment and then computes some statistics about the Equipment from this list. You will also write out information about the Equipment and the statistics to some files.

Classes

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


Driver
+ main(args: String[]): void throws IOException
1
UtilityBelt
equipment : ArrayList<Equipment>
+ UtilityBelt(filename : String)
+ read(filename : String) : void throws IOException
+ writeEquipment(filename : String) : void throws IOException
+ writeStatistics(filename : String) : void throws IOException
+ computerTotalWeight() : double
+ getNamedEquipment(name : String) : Equipment
+ equipmentAboveCount(count : int) : ArrayList<Equipment>
+ findMostExpensiveEquipment() : Equipment
0...*
Equipment
-name : String
-count : int
-totalWeight : double
-totalPrice : double
-description : String
+ Equipment(strg : String
+ getName() : String
+ getCount() : int
+ getTotalWeight() : dou
+ getTotalPrice() : doubl
+ getDescription() : Strin
+ toString() : String

The *Equipment* class is responsible for representing copies of an item in the utility belt (e.g. 5 batarangs is 1 equipment object). Here are the methods:

- The *Equipment* constructor takes a String as input and parses it to populate the instance variables for this class. The input string will be of the format: "name/count,totalWeight,totalPrice,description"

  Notice that the instance variables of a Player match with the parts of the constructor input String. You should use String.split() to help you separate the values. You will need to do some conversions to turn the string into a double and int (for weight/price and count, respectively). Look Integer.parseInt() and Double.parseDouble().

- *toString()*. This method returns a String formatted, as specified in the javadoc above the method.

- A full set of getters. Use Eclipse to generate these for you. See Source ¿ Generate Getters and Setters. You should write javadoc documentation for these generated methods.

The *UtilityBelt* class represents a collection of Equipment held by Batman. See the sections on reading/writing from/to files below before starting to work on this class. Here are the methods that need to be implemented:

- *UtilityBelt* constructor. This method takes as input a file name, reads through the file, creates one Equipment object from each line in the file, and adds each to the ArrayList (*equipment*). This method handles some file I/O and you do not need to change it. The actual file reading work is done by read.

- *read(String filename)*: helper method for the constructor. Does the actual work of taking as input a file name, reading through the file, creating one Equipment object from each line in the file, and adding each to the ArrayList (*equipment*).

- *writeEquipment(String filename)*: creates a string representing the UtilityBelt and writes it out to a file (specified by the input parameter). Specifically, loop through the list of Equipment and write out each toString to the file. Newlines should be added after each toString. Note: to add newlines when writing out to a file with a BufferedWriter, you need to call bufferedwriter.newLine().

- *writeStatistics(String filename)*: very similar to writeEquipment, creates a string representing some statistics of the UtilityBelt and writes it out to a

file (specified by the input parameter). Specifically, write out information in the following format:

Total Weight: ⟨result of computeTotalWeight()⟩

Most Expensive Equipment: ⟨name of Equipment returned by findMostExpensiveEquipment()⟩

- *computeTotalWeight())*: sums and returns the total weight of all the Equipment.
- *getNamedEquipment(String name)*: returns the Equipment object with the matching name. Returns null if there is no match.
- *equipmentAboveCount(int count)*: returns an ArrayList containing all equipment with a count value higher than the input parameter. Order should be preserved, so if the original array order is [5, 2, 3, 4, 1] and 3,4, and 5 are returned, the returned array should be [5, 3, 4]
- *findMostExpensiveEquipment()*: Finds and returns the most expensive equipment (highest price value).

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

- Creates a *Equipment* object with a specific file name (this name may be hard-coded into the method - the only names you may use when submitting are "Input.csv" or "InputOfficial.csv". Other names may cause Zylabs errors.)
- Writes out the information of the *UtilityBelt* object to some files. (You should do this to test your code)
- Tests other parts of your code as necessary

Reading a File

Within the UtilityBelt 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  // Read the next line. This discards the first line read, which is generally not ↩
       actual data.
10 String strg = br.readLine();
11
12 // Iterate as long as there is a next line
13 while (strg != null)
14 {
15   // Add the line to the ArrayList
16   list.add(strg);
17   // Attempt to get the next line
18   strg = br.readLine();
19 }
```

Note that in your assignment, you must create a Equipment 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 some cases, an *InputStreamReader* may be used in order to take input from *System.in* (the keyboard). In this lab, a *FileReader* will be used to take input from a file. 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.

Writing to a File

Within the Equipment class there are two functions that write out some data. You can write a string to a file with the BufferedWriter's write(String str) method, and you can add a newline to the file with the newLine() method.

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

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

- Select *Generate Getters and Setters...* from the dropdown menu

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.

Final Steps

1. Update all time estimates in your TODO.json; run the checker to make sure that your TODO.json file is formatted correctly.

2. Make sure before submitting that you only initialize UtilityBelts in the Driver with "Input.csv" or "InputOfficial.csv". Failure to do so may result in a Zylabs error.

Submission Instructions

You will submit to Zylabs. Before submission, finish testing your program by editing and executing your main() method. It would also be good to add some data to your csv files (see above about file naming). You should submit when your program behaves as you expect it to.

Grading Rubric

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

Zylabs Submission: 80 points

The Zylabs grader 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 Zylabs server will inform you as to which tests your code passed/failed. You may submit up to 15 times to pass more tests and improve your score.

Lab Plan: 20 points

Your todo.txt file will be checked by the zylabs unit test to verify its format. A grader will also asses the quality of your lab plan. Your plan should have at least 5 objectives that should be relatively granular (e.g. an objective of "finish lab" is much too large and not a good tasks in a plan). The grader will ensure that your predicted and actual times are reasonable (e.g. not 20000 minutes).