CS 2334
Project 3: Java Collections Framework

October 15, 2017

Due: 6:00 pm on Friday, October 27, 2017

Introduction

For the last two projects, you have been using data files that are well-structured. In particular, you could assume ahead of time that you knew which variables would be encoded in each file, and you knew the specific column in which each variable was placed. In this project, we will break both of these assumptions. In particular, the header row of each data file will inform you of the encoded variables and the order in which they occur. Given this information, your program will create the data structures necessary to load the infant motion data and to compute statistics over the individual measurements. Different data files may contain different data and the order of the columns may differ.

Learning Objectives

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

1. Make use of **HashMaps** and **TreeMaps** to flexibly store data in a structure that is efficient to access,
2. Compute statistics over the stored data in a manner that does not rely on *a priori* knowledge of the specifics of the data, and
3. Continue to exercise good coding practices for Javadocs and for unit testing.

Proper Academic Conduct

This project is to be done in the groups of two that we have assigned. You are to work together to design the data structures and solution, and to implement and test this design. You will turn in a single copy of your solution. Do not look at or discuss solutions with anyone other than the instructor, TAs or your assigned team. Do not copy or look at specific solutions from the net.

Strategies for Success

- The UML specification constitutes the interface that we will rely on during our testing. Do not make changes to this interface.
- When you are implementing a class or a method, focus on just what that class/method should be doing. Try your best to put the larger problem out of your mind.
- We encourage you to work closely with your other team member, meeting in person when possible.
- Start this project early. In most cases, it cannot be completed in a day or two.
- Implement and test your project components incrementally. Don't wait until your entire implementation is done to start the testing process.
- Write your documentation as you go. Don't wait until the end of the implementation process to add documentation. It is often a good strategy to write your documentation **before** you begin your implementation.

Preparation

- Copy your project 2 implementation into a new *project3* project.
- Import the data for project3 into your workspace:
  http://www.cs.ou.edu/~fagg/classes/cs2334/projects/project3/project3-data.zip
- Place these files into your project's *data* directory. You may also copy your project 2 origdata directory contents to the *data* directory.

- You may place your own private CSV files for testing into your *mydata* directory.

Data Representation and Computing Statistics

Representing Tuples

In projects 1 and 2, two types were used to represent information about a State. *Scalars*, such as time, captured a single value and **Point3D** objects captured a triple of values. In this project, we are going to unify these two different representations, as well as generalize the representation to allow for any number of component values.

The class **PointND** thus supersedes both of these representations, and will represent the values associated with a specific field. For the field *time*, the **PointND** object will represent a single value, whereas a field such as *left_wrist* will contain three values, indexed by the *subfield* Strings "x", "y" and "z".

PointND
-values:TreeMap<String,GeneralValue>
+PointND() +add(subFieldName:String, value:GeneralValue) +getValue(subFieldName:String):GeneralValue +size():int +iterator():Iterator<String> +toString():String

A single instance of a **PointND** represents the individual values through a map. Specifically, this map goes from **String**s describing the subfield names to the corresponding **GeneralValue**. When a **PointND** represents only a single value, we will typically use the empty **String** as the subfield name.

- *toString()* returns a single line **String** with one component for each subfield (the subfield strings are appended together). The format for a single subfield is as follows:

```
"SUBFIELDNAME = VALUE; "
```

  where SUBFIELDNAME is the name of the subfield and VALUE is the **String** that describes the value of the corresponding **GeneralValue**. Note the single space at the end of the string.

- *getValue(String subFieldName)* will return the **GeneralValue** that corresponds to the subfield. If *subFieldName* does not exist in the map, then this method must return an invalid **GeneralValue**.

Representing States

Your code will not know ahead of time what information will be used to describe a **State**. Instead, this information will be inferred from the Strings in the header row of each CSV file. We use the term *field* to describe the individual tuples, such as "left_wrist" or "right_shoulder"; and *subfield* is used to describe the individual components of the tuple (e.g., "x", "y" and "z").

The **State** class must acknowledge our lack of prior knowledge about the set of fields that it must represent. We therefore store the set of fields in a map that takes us from the field name to the corresponding **PointND** object.

State
-variables:TreeMap<String,PointND> -trial:Trial
+State() +State(trial:Trial, fieldMapper:FieldMapper, values:String) +getTrial():Trial +getPoint(fieldName:String):PointND +getValue(fieldName:String, subFieldName:String):GeneralValue +getMaxState(fieldName:String, subFieldName:String):State +getMinState(fieldName:String, subFieldName:String):State +getAverageValue(fieldName:String, subFieldName:String):GeneralValue +iterator():Iterator<String> +toString():String

The key methods include:

- The default constructor creates an object with no fields
- The main (non-default) constructor takes as input the **Trial** to which the **State** belongs, a **FieldMapper** that will inform the constructor about how to create fields from the data (described below) and a **String** that describes all of the values associated with the **State** (a CSV row).
- *getPoint* returns the **PointND** object corresponding to a *field name*. If the field is not a component of the **State**, then *null* is returned.

- *getValue()* returns the **GeneralValue** that corresponds to the field and subfield names. **If the State does not contain the field/subfield, then this method returns an invalid GeneralValue**.
- *getMinState()* and *getMaxState()* return the **State** that contains the specified minimum/maximum value.
- *iterator()* returns an **Iterator** over the field names.
- *toString()* returns a multi-line **String** (one line for each field, in alphabetical order of the field names). Each line has the following format:

```
"FIELDNAME(POINTND)\n"
```

  where FIELDNAME is the name of the field and POINTND is the value returned by *PointND.toString()*.

Statistics

Statistics are computed in the same way as in project 2, with two small differences:

- field and subfield names are used to specify the variable of interest, and
- *Min* and *Max* return the **State** that contains the minimum and maximum values. This will allows us to easily answer questions such as "which week contains the minimum value for the left wrist z."

SingleItemAbstract
+getMaxState(fieldName:String, subFieldName:String):State *+getMinState(fieldName:String, subFieldName:String):State* *+getAverageValue(fieldName:String, subFieldName:String):GeneralValue*

Mapping Files to Data Structures

In order to aid in the extraction of data from the CSV files, we are introducing two new classes: **FieldMapper** and **Field**. The goal here is to create a mechanism by

which it is easy to create a set of **PointND** objects from a single data row of the CSV file. This set will then become components of a single **State**.

FieldMapper
-fieldMap:Map<String,Field>
+FieldMapper(columnNames:String[]) +getField(fieldName:String):Field +extractPointND(stringValues:String[], fieldName:String):PointND +size():int +iterator():Iterator<String>

The **FieldMapper** class represents the set of fields that are contained within a single CSV file. The key details are as follows:

- The constructor takes as input the list of individual column names. Column names are related to field and subfield names as follows:
  - If the second to last character of the column name is "_" (underscore), then the characters leading up to the underscore become the field name, and the last character (after the underscore) is the subfield name. A column name of "left_wrist_z" corresponds to a field name of "left_wrist" and a subfield name of "z".
  - Otherwise, the entire column name is the field name, and the subfield name is the empty **String**. A column name of "time" corresponds to a field name of "time" and a subfield name of "".

  The job of this constructor is to create the entire data structure (including all component **Fields**).
- *getField()* returns the **Field** that corresponds to the **String** field name.
- *extractPointND()* takes as input the String array of values (all values from the CSV row) and a **String** field name. This method constructs and returns the **PointND** object for the specified field.
- *size()* returns the number of fields.
- *iterator()* returns a **String** iterator over all of the field names.

The **Field** class captures the set of subfields within one field, and it represents the column number in the CSV file that corresponds to each subfield.

Field
-subFields:Map<String,Integer>
+Field() +addSubField(subFieldName:String, columnIndex:int) +getIndex(subFieldName:String):Integer +size():int +iterator():Iterator<String> +toString():String

The key details are as follows:

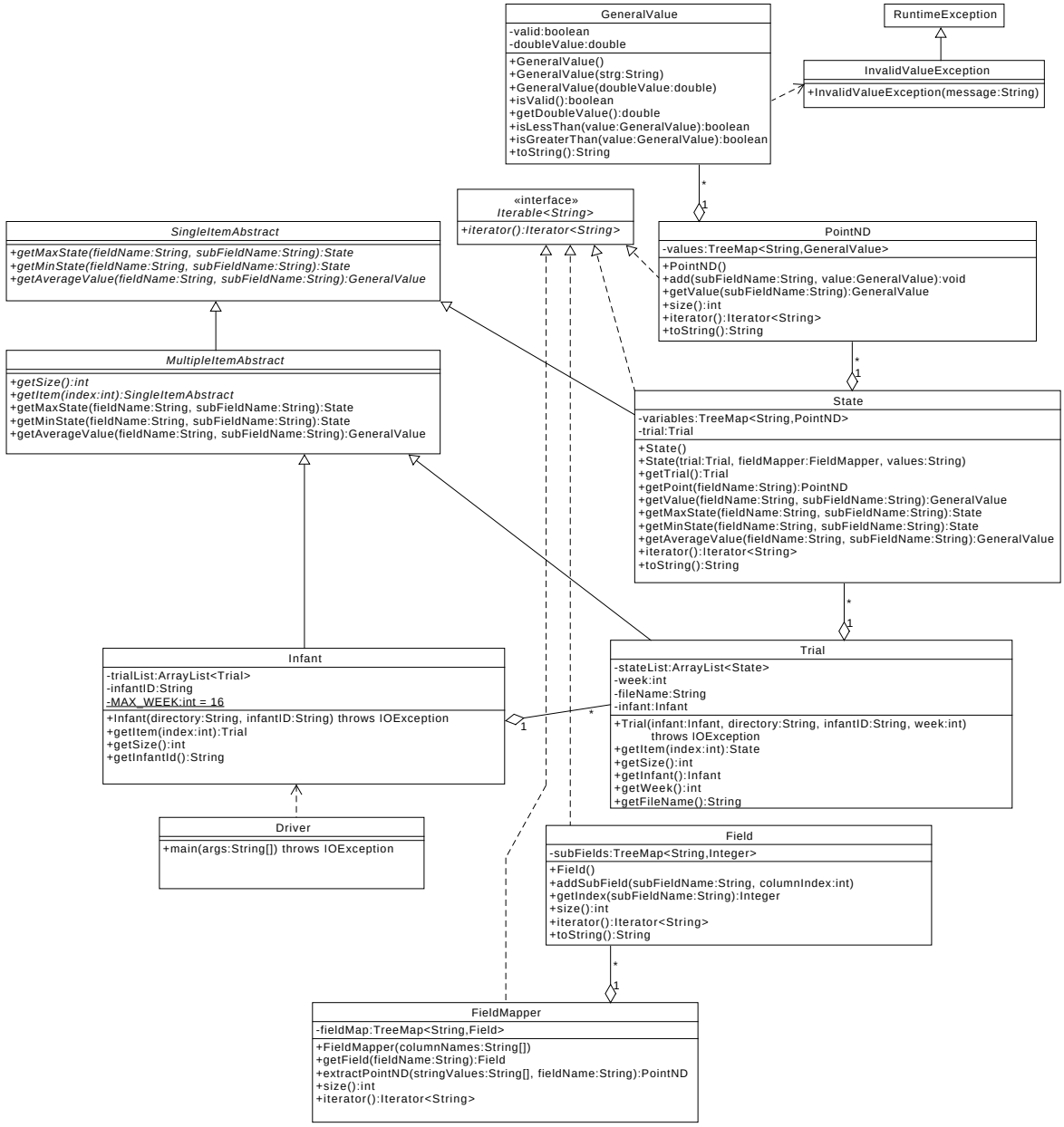
- The constructor creates an empty field.
- *addSubField()* takes as input a subfield name and the CSV column index for that subfield, adding this information to the *subFields* map.
- *getIndex()* returns the **Integer** column index for the specified subfield name. If the subfield is not contained in the **Field**, then *null* is returned.
- *size()* returns the number of subfields.
- *iterator()* returns a **String** iterator over all of the subfield names.
- *toString()* returns a single line **String** with components for each subfield (ordered alphabetically by subfield name). Each component has the following format:

```
"SUBFIELD(INDEX); "
```

  where SUBFIELD is the subfield name and INDEX is the integer column index of the subfield in the CSV file. Note the space after the semicolon.

Overall Design

Below is a complete UML diagram for our key classes.



Testing

Provide test classes for all of the classes in the above UML diagram (except for the **Driver** class. Note that your test implementations from project 2 can be copied into this project. Some of these classes will require modification. Note that you should make a point of testing the statistics for some of the new columns that occur in the new CSV files.

Final Steps

1. Generate Javadoc using Eclipse for all of your classes.
2. Open the *project3/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.

Submission Instructions

- All required components (source code and compiled documentation) are due at 6:00 pm on Friday, October 27.
- Submit your project to Web-Cat using one of the two procedures documented in the Lab 1 specification.

Grading: Code Review

All groups must attend a code review session in order to receive a grade for your project. The procedure is as follows:

- Submit your project for grading to the Web-Cat server.
- Any time following the submission, you may do the code review with the instructor or one of the TAs. For this, you have two options:
    1. Schedule a 15-minute time slot in which to do the code review. We will use Doodle to schedule these (a link will be posted on Canvas). You must attend the code review during your scheduled time. Failure to do

so will leave you only with option 2 (no rescheduling of code reviews is permitted). Note that schedule code review time **may not** be used for help with a lab or a project

2. "Walk-in" during an unscheduled office hour time. However, priority will be given to those needing assistance in the labs and project

- Both group members must be present for the code review
- During the code review, we will discuss all aspects of the rubric, including:

  1. The results of the tests that we have executed against your code
  2. The documentation that has been provided (all three levels of documentation will be examined)
  3. The implementation. Note that both group members must be able to answer questions about the entire solution that the group has produced

- If you complete your code review before the deadline, you have the option of going back to make changes and resubmitting (by the deadline). If you do this, you may need to return for another code review, as determined by the grader conducting the current code review
- The code review must be completed by Friday, November 3rd to receive credit for the project.

References

- The Java API: https://docs.oracle.com/javase/8/docs/api/
- The API of the *Assert* class can be found at: http://junit.sourceforge.net/javadoc/org/junit/Assert.html
- JUnit tutorial in Eclipse: https://dzone.com/articles/junit-tutorial-beginners

Rubric

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

Correctness/Testing: 40 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: 25 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 project 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 twelve hours that your assignment is submitted early.

Penalties: up to 100 points

You will lose five points for every twelve hours 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. Assignments arriving 48 hours after the deadline will receive zero credit.

After 30 submissions to Web-Cat, you will be penalized one point for every additional submission.

Web-Cat note: 24 hours before the deadline, the server will stop giving hints about any failures of your code against our unit tests. If you wish to use these hints for debugging, then you must complete your submissions 24 hours before the deadline.