Lab 9: HashMaps and Enums
CS 2334

March 13, 2017

Introduction

Rock-paper-scissors common game used to settle small disputes. In some circles, it is also a competitive sport. In this lab, we will provide code that will enable the **Master of the Rock-Paper-Scissors Arena** to set up battles between pairs of contestants.

In your implementation, you will experiment with using HashMaps and Enumerated data types in Java. You will implement a few different classes of enums. One enum has a custom class that is associated with each enum value. Another enum contains HashMaps that map between different values of the enum. Your implementation will also experiment with iterating over the elements contained within a Hashmap.

Learning Objectives

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

1. Create an enumerated data type and initialize parts of the type using a *static initializer block*
2. Create and add items to a HashMap
3. Pull values out of a HashMap using a key
4. Iterate over a HashMap in order to print out information

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. Import the existing lab 9 implementation into your eclipse workspace.

   (a) Download the lab 9 implementation from canvas.

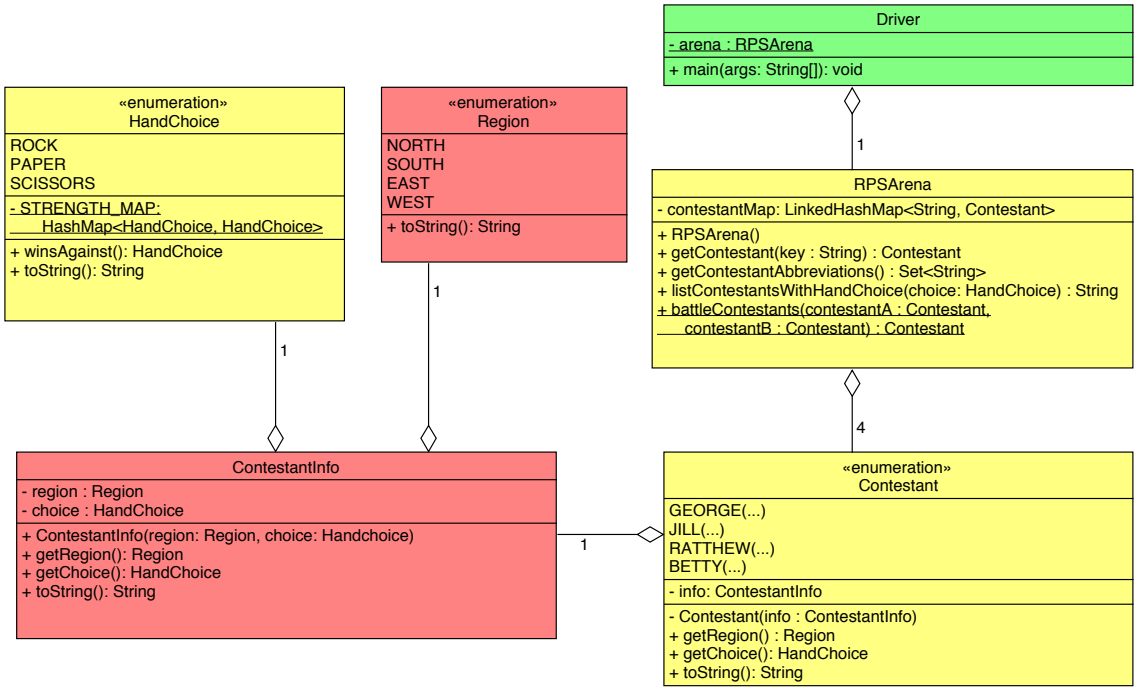
   (b) In Eclipse, select *File/Import*

   (c) Select *General/Existing projects into workspace*. Click *Next*

   (d) Select *Select archive file*. Browse to the lab9.zip file. Click *Finish*

Representing Different Rock Paper Scissors Players

Below is the UML representation of the lab. Your task will be to implement this set of classes and an associated set of JUnit test procedures.



We will only be using the following Contestants for this lab: *George*, *Jill*, *Ratthew*, and *Betty*. The keys used in **RPSArena** with the contestantMap HashMap are three letter abbreviations for the names of the Contestant: *GEO*, *JIL*, *RAT*, and *BET* respectively. The properties of the three Contestant that we will represent in this lab are as follows:

- GEORGE: region - NORTH, hand choice - ROCK
- JILL: region - SOUTH, hand choice - ROCK
- RATTHEW: region - EAST, hand choice - SCISSORS
- BETTY: region - WEST, hand choice - PAPER

A Contestant's hand choice affects how well s/he does in combat with other Contestants. Which hand wins against which other hand is defined by the following table:

Type	Effective Against
ROCK	SCISSORS
PAPER	ROCK
SCISSORS	PAPER

Table 1: Type effectiveness table

Lab 9: General Instructions

1. In lab9.zip, we have provided a full implementation of the **Driver** class.
2. We have provided partial implementations of the **RPSArena**, **Contestant**, and **HandChoice** classes. Create the remaining classes from the UML diagram below (**ContestantInfo** and **Region**).
   - Be sure that the class name is exactly as shown
   - You must use the default package, meaning that the package field must be left blank
3. Implement the attributes and methods for each class
   - We suggest that you start at the "bottom" of the class hierarchy: start by implementing classes that do not depend on other classes
   - Use the same spelling for instance variables and method names as shown in the UML
   - Do not add functionality to the classes beyond what has been specified
   - Don't forget to document as you go!
4. Create test classes and use JUnit tests to thoroughly test all of your code (except the **Driver** class).
   - You need to convince yourself that everything is working properly
   - Make sure that you cover all the classes and methods while creating your test. Keep in mind that we have our own tests that we will use for grading.

Lab 9: Specific Instructions

Region Enum

This enumeration should have the following members: *NORTH*, *EAST*, *SOUTH*, *WEST*.

- *toString()*: This method should return the name, in lowercase, of the particular member of the enum.

HandChoice Enum

This enumeration has the following members: *ROCK*, *SCISSORS*, *PAPER*.

The HandChoice enum contains a subset of the possible types of hand choices that a Contestant can have.

As previously mentioned, a Contestant's hand choice effects how well it does in combat with other Contestant. In order for the members of the enumeration to express the hand choices they win against, we have introduced a HashMap to store this relations: *strengthMap*. This HashMap maps a **HandChoice** value to the **HandChoice** value that it wins against.

In order to only create and populate this map once, while still being able to access them from references to the enumeration's members, we need to make them **static**—these properties need to also be marked **final** to indicate that they are constant and should not be changed.

Initializing and populating the *strengthMap* HashMap should be done in a **static initializer**. A static initializer is a block of code that runs only once when a class declaration is first loaded by the Java Virtual Machine. This code runs after initialization of static variables at their declaration, and before anything else—you can think of it as a constructor for the class itself, instead of a constructor for individual instances of the class. Since a static initializer is only run once, and is only used by Java internally, you don't give it a name as you would with other methods, as it cannot (and should not!) be callable by any other piece of code.

A static initializer is a method that looks like the following:

```
class Foo // Some class
{
    static // This is the static initializer for class Foo
    {
        . . . // Code goes here.
    }
}
```

See table 1 on page 4 for the mappings you need to setup inside the HashMaps.

The instance methods to be implemented are as follows:

- *winsAgainst()*: This method should return the HandChoice that this particular HandChoice wins against.
- *toString()*: This method should return, in lowercase, the name of the particular member of the enum.

ContestantInfo Class

This class contains information about a particular Contestant. This information includes the following: the Contestant **Region** and the Contestant's **HandChoice**.

- The **ContestantInfo** constructor takes in a **Region** and a **HandChoice**, and assigns them to the appropriate instance variables.
- *getRegion()*: This method returns the **Region** stored in this particular instance of **ContestantInfo**.
- *getHandChoice()*: This method returns the **HandChoice** stored in this particular instance of **ContestantInfo**.
- *toString()*: This method should return the information stored in this particular instance of **ContestantInfo** in the following format:

```
contestant from <REGION> throwing <HAND CHOICE>
```

where <*REGION*> is the **Region** stored in this instance, and <*HAND CHOICE*> is the **HandChoice** stored in this instance. If the **Region** were to be *NORTH* and the **HandChoice** to be *SCISSORS*, then the output would look like the following:

```
contestant from north throwing scissors
```

Note: the string ends with the 's' at the end of *hand choice* (there is no newline character).

Contestant Enum

The **Contestant** enum contains a subset of the very large number of Contestants that exist. This enum has the following members: *GEORGE* (NORTH region, ROCK hand choice), *JILL* (SOUTH region, ROCK hand choice), *RATTHEW* (EAST region, SCISSORS hand choice), and *BETTY* (WEST region, PAPER hand choice).

- The **Contestant** constructor takes in an instance of **ContestantInfo** and stores it in the appropriate instance variable.
- *getRegion()*: This method returns the **Region** of the **Contestant**.
- *getHandChoice()*: This method returns the **HandChoice** of the **Contestant**.
- *toString()*: This method returns a descriptive string of the **Contestant**. The string should be in the following format:

```
<NAME>: contestant from <REGION> throwing <HAND CHOICE>
```

  where <*NAME*> is the name of the Contestant in lowercase, and <*REGION*> and <*HAND CHOICE*> are covered in the above section detailing the **Contestant** enum. For the Contestant *JILL*, with a *SOUTH REGION* and a *HAND CHOICE* of *ROCK*, the returned string would look like:

```
jill: contestant from south throwing rock
```

RPSArena Class

The RPSArena will create a map of abbreviated contestant names to the contestants. It will provide an API with several options for getting information about these contestants. The RPSArena is used by the **Driver** to interact with the user via the following methods:

- The **RPSArena** constructor will initialize the contestantMap.
- getContestant(String key): this method attempts to return the contestant in the contestantMap associated with the key string parameter. If an associated contestant does not exist, the method returns null.
- getContestantAbbreviations(): this method returns the set of abbreviation for all of the supercontestants. This is equivalent to the keyset of the contestantMap HashMap.
- listContestantsWithHandChoice(HandChoice choice): this method returns a **String** containing information about the contestants in the arena. It iterates through the list of contestants; for each contestant that has a HandChoice equal to the choice input to this method, the output String includes a line with the format:

```
<Abbreviation> - <Contestant Description>.
```

  Take note that the contestantMap HashSet is actually a LinkedHashSet. This is done to ensure that the order of accessing the list iteratively stays the same every time. A standard HashSet will be in a random order when accessing iteratively.
- battleContestantes(Contestant contestantA, Contestant contestantB): this method is used to determine which contestant would win in a fight. We select the victor by comparing hand choice types. If contestantA's hand choice is strong against contestantB's hand choice, then contestantA will win (and contestantA will be returned). Likewise, if contestantB's hand choice is strong against contestantA's, then contestantB will win. If neither power is strong against the other (powers are the same in this case), the method will return null.

Driver Class

The Driver class will create a **RPSArena** object and use it to process the user's queries. It will present the user with an option to choose a pair of Contestants to battle against one-another or to print information about Contestants with a given hand choice. If the user opts to choose specific Contestants to fight, then your program will print the information for the result of that battle. If the user opts for the list, then all of the Contestants in the RPSArena are presented to the user.

- Main menu:

```
As master of the Rock-Paper-Scissors arena, you may choose two Contestants ↩
    to fightagainst each other. You may also view their information.
Please select an option:
1: Choose two Contestants to battle
2: List all Contestants with a specified hand choice
```

- Contestant Battling selection menu:

```
Please choose from the following Contestant: [GEO, JIL, RAT, BET]
```

  Note: Sets are unordered. However, the arena class uses a LinkedHashSet, which ensures that the order is the same every time that elements are accessed iteratively.

- Contestant Information selection menu:

```
Select hand choice: ROCK, PAPER, SCISSORS
```

The program uses a *BufferedReader* to take in the input. The code must be able to handle any input that the user could choose, e.g. numbers other than 1 and 2, letters, Contestants not listed, etc. If incorrect input is given, then your program must re-prompt until a correct input is given.

Once all of the necessary information is obtained from the user, your program must print the report about the results of the battle between the chosen Contestants (or all Contestants with a given hand choice) and exit.

The drive is provided for you and handles this.

Example Interactions

```
As master of the Rock-Paper-Scissors arena, you may choose two Contestants to ↩
    fightagainst each other. You may also view their information.
Please select an option:
1: Choose two Contestants to battle
2: List all Contestants with a specified hand choice
1
Please choose from the following Contestant: [GEO, JIL, RAT, BET]
JIL
You choose jill: contestant from south throwing rock.
Please choose another Contestant from the following Contestants: [GEO, JIL, RAT, ↩
    BET]
RAT
You choose ratthew: contestant from east throwing scissors.
jill was the victor!
```

```
As master of the Rock-Paper-Scissors arena, you may choose two Contestants to ↩
    fightagainst each other. You may also view their information.
Please select an option:
1: Choose two Contestants to battle
2: List all Contestants with a specified hand choice
2
Select hand choice: ROCK, PAPER, SCISSORS
rock
GEO - george: contestant from north throwing rock
JIL - jill: contestant from south throwing rock
```

Final Steps

1. Generate Javadoc using Eclipse.
   - Select *Project/Generate Javadoc...*
   - Make sure that your project (and all classes within it) is selected
   - Select *Private* visibility
   - Use the default destination folder
   - Click *Finish*.
2. Open the *lab9/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 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 your 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 at 11:59pm on Friday, March 16. **Submission must be done through the Web-Cat server.**
- Use the same submission process as you used in lab 4. You must submit your implementation to the *Lab 9: Enumerated Types and Hashmaps* area on the Web-Cat server.

Hints

- In Eclipse, EclEmma will flag your enums as being incomplete in their coverage. If it is flagging the very top of your enum files (and nothing else), this is okay. Web-Cat will not be this strict in its code coverage testing.
- Link to Java tutorial on enumerated types. The Planets enum is very helpful for understanding this lab:

  https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
- Link for the LinkedHashMap datatype:

  https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html
- Information on how to iterate over a hashmap (works the same with a linked-hashmap):

  https://www.geeksforgeeks.org/iterate-map-java/

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 three 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.