Lab 8: HashMaps and Enums
CS 2334

March 14, 2018

Introduction

Board games are a common form of entertainment. In them, players are often associated with some trinket - a game piece. Pieces may take the form of various real world objects, such as race cars and boots (e.g. Monopoly). Players can also often move their pieces around a game board.

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 and add items to a HashMap
2. Pull values out of a HashMap using a key
3. Iterate over a HashMap in order to calculate 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.

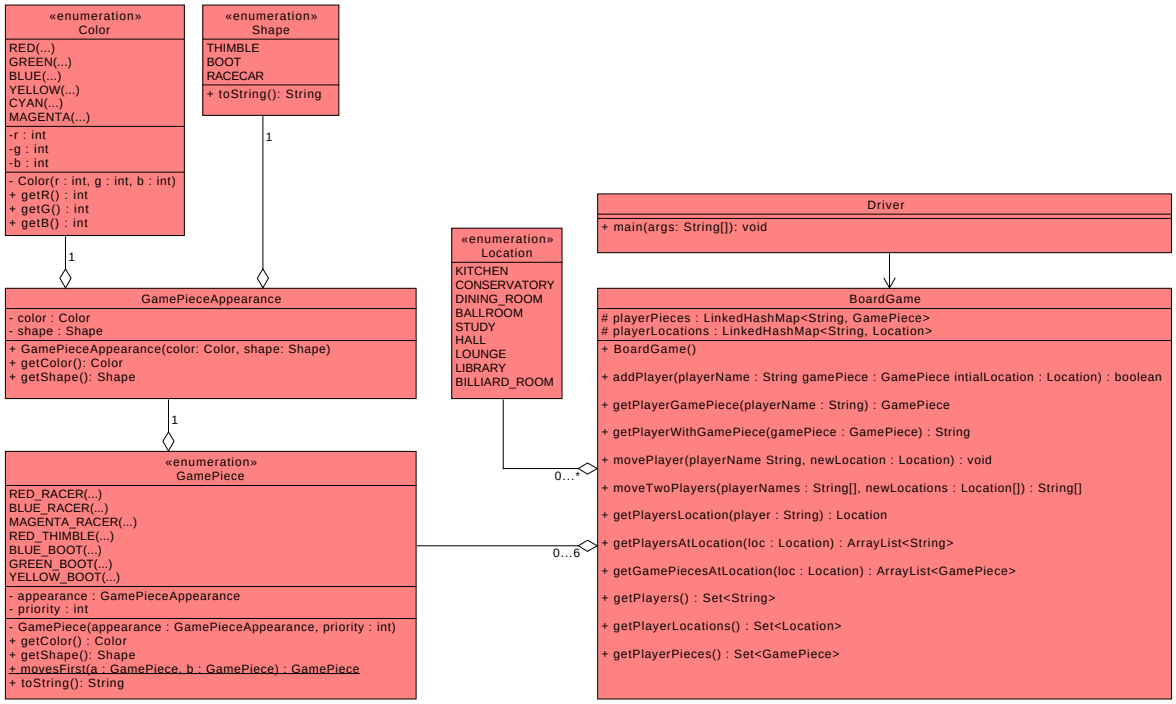
Preparation

1. Download and read through the javadocs for this lab from eclipse. There is no skeleton code given.

2. Read through the canvas instruction on the GCP portion of your lab.

Representing The Board Game

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. The lab may seem like a lot of code, but several methods will be close to single-line and the enums should be fairly simple overall.



The coloring of the classes indicates what work you need to do to complete the lab assignment. This is done for your convenience; you can also follow TODOS and the other lab instructions to understand what you need to do. The colors indicate as follows:

1. Green: A class/interface that is completely implemented. You do not need to edit these.

2. Yellow: A class/interface that is partially implemented. You will need to look for TODOs and read the writeup to determine what else is needed to complete the class.

3. Red: A class/interface that is completely unimplemented. You will need to create this class and ensure that it is fully functional.

4. Purple: This is a class/interface provided in standard Java. You can look online for documentation for these classes. You should never attempt to create these classes. Doing so will cause many errors.

Lab 8: General Instructions

1. Format your classes correctly
    - 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
2. 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!
3. 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 8: Specific Instructions

As with your previous lab, the specifics of how to implement your lab are within the javadoc given to you. Refer to this and the UML to complete your lab. The documentation here simply gives a general overview of each class so you can intuitively understand what they do. The values that the enums can take are also specified in the javadoc.

Color Enum

Specifies a set of colors. Each color has an rgb value (representing the red, green, and blue components of the color). All game pieces have a color.

Shape Enum

Specifies a set of shapes that they game pieces can take. Game pieces can be shaped as (similar to monopoly):

1. A race car
2. A boot
3. A thimble

GamePieceAppearance Class

A simple class that acts as a composite data structure of a Color and Shape. In effect, holds information on the appearance of a game piece. It has no further code.

GamePiece Enum

Specifies the game pieces available for players to choose to play the board game. Each game piece has a GamePieceAppearance to specify its Color and Shape. Each game piece also has a priority. Only one piece may be moved in the board game at a time. Thus, if two players wish to make a move at the same time, the game has to resolve who goes first. The player who has the game piece with the lower value priority goes first (e.g. 2 vs. 4 -¿ priority 2 goes first). Each player must use a game piece to play the board game, and no two players can have the same game piece (they cannot share it).

Location Enum

Specifies the locations that game pieces can move to in the board game (think of Clue, for example). Multiple game pieces may be at the same location.

BoardGame Class

Main class for the lab. Specifies a Board game (very simple - there's not even a winning condition). The board game has the ability to:

1. Add players to the game. Players are associated with a game piece, which is set at some initial location. Players are represented only by their name.

2. Move players to different locations. I.e. the game pieces move on the board to a different location. Since each player is associated with exactly one game piece, they can be treated as effectively the same object.

3. Find information out about the board game. E.g. all the game pieces in a location, or the game piece associated with a player.

Driver Class

You should create a driver class to do full-scale testing of your code. There is no input/output requirement for the driver class.

Example Testing for Enums and Sets

Testing for enums is not always as straightforward as testing normal objects. One key detail to keep in mind is that enums have a static set of values. As such, you cannot add enum values during runtime. When testing objects, you might have created an object with some new set of values. You cannot do this here, but you can test the enum values have the correct attributes. Additionally, you can still test methods in the same way. We give some example testing code for enums below:

```
@Test
public void testColorValues()
{
    Color col = null;

    // RED:
    col = Color.RED;
    Assert.assertEquals("Incorrect rgb value in color " + col.name(), 255, col.↩
        getR());
    Assert.assertEquals("Incorrect rgb value in color " + col.name(), 0, col.getG↩
        ());
    Assert.assertEquals("Incorrect rgb value in color " + col.name(), 0, col.getB↩
        ());

    // BLUE:
    col = Color.GREEN;
    Assert.assertEquals("Incorrect rgb value in color " + col.name(), 0, col.getR↩
        ());
    Assert.assertEquals("Incorrect rgb value in color " + col.name(), 255, col.↩
        getG());
    Assert.assertEquals("Incorrect rgb value in color " + col.name(), 0, col.getB↩
        ());

    ...

}

/**
 * Tests the Location enum.
 *
 * Note: you don't really need to test the enum like this, but
 * if you want to make sure that the enum value names are correct this kind of ↩
     test works well.
 */
@Test
public void testLocation()
{
    Assert.assertEquals("Location enum values incorrect.",
            Location.KITCHEN, Location.valueOf("KITCHEN"));
    Assert.assertEquals("Location enum values incorrect.",
            Location.CONSERVATORY, Location.valueOf("CONSERVATORY"));
    Assert.assertEquals("Location enum values incorrect.",
            Location.DINING_ROOM, Location.valueOf("DINING_ROOM"));
```

```
    ...

}

/**
 * Tests the Shape toString.
 */
@Test
public void testShapeToString()
{
    Assert.assertEquals("Shape toString incorrect.",
            "thimble", Shape.THIMBLE.toString());
    ...

}
```

Sets are collections for which no element is repeated. I.e. each element is unique. This comes from a mathematical formulation. Additionally, sets have no implicit notion of order. That is, there is no "first" element of a set. This creates some complexities for testing with sets. While you can compare two arrays and see that they share the same elements in the same order, when comparing sets you must check that they share the same elements, regardless of order. We give an example test below:

```
@Test
public void testSet()
{
    // Variables to test (You would realistically get the set by calling a method ↩
        on some class):
    int[] dataRaw = new int[] {1, 5, 7, 4, 10, 7, 8, 5};
    // This is complicated, but all it's doing is converting the array to a list:
    List<Integer> dataList = Arrays.stream(dataRaw).boxed().collect(Collectors.↩
        toList());
    // Set is an interface, HashSet is a class:
    Set<Integer> dataSet = new HashSet<Integer>(dataList);

    // Remember: converting to a set removes all duplicate elements.

    int[] expected = new int[] {1, 5, 7, 4, 10, 8};
    // We can't directly compare our array of data with the set, so we check that ↩
        every element
    // in our expected array is in the set:
    for (int index = 0; index < expected.length; index++)
    {
        Assert.assertTrue(dataSet.contains(expected[index]));
    }
}
```

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 *lab8/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, compiled documentation, and todo.java) are due at 11:59pm on Tuesday, March 26. **Submission must be done through the Web-Cat server.**
- Use the same submission process as you used in lab 6. You must submit your implementation to the *Lab 8: Enumerated Types and HashMaps* area on the Web-Cat server.
- Submit your github link on canvas.
- Submit your GCP link on canvas (details for the GCP submission are not elaborated here; check on canvas).

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/
- Note: the LinkedHashMap type is useful due to the nature of how it interacts with loops. Whenever you loop over a LinkedHashMap, the order of the elements remains the same. This is not guaranteed for a normal HashMap. This is really the only major difference.

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: 10 points

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard 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: 20 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).

GCP: 20 points

You will be creating a GCP instance and uploading your code (should be a copy-paste, effectively). Details for this submission are included on canvas.

Github: 5 points

You will need to use github when developing your project. The grader will check that you have made several commits (¿= 5) while developing your lab. Commits should have good messages. You will link your github repo on canvas.

Lab Plan: 5 points

Your todo.txt file will be checked 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).

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).

For labs, the server will continue to accept submissions 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.