Lab Exercise 8
CS 2334

October 13, 2016

Introduction

In this lab, you will experiment with using HashMaps and Enumerated data types in Java. You will implement a few different classes of enums, one that has a custom class as the value. In the Driver class, you will create several different HashMaps—one that maps from a String to one of the enumerated data types, and two that maps values from one of the enums to other values from itself. In addition, your implementation will iterate over a HashMap.

Learning Objectives

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

1. Create an enumerated data type
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 8 implementation into your eclipse workspace.

   (a) Download the lab 8 implementation:
   `http://www.cs.ou.edu/~fagg/classes/cs2334/labs/lab8/lab8.zip`

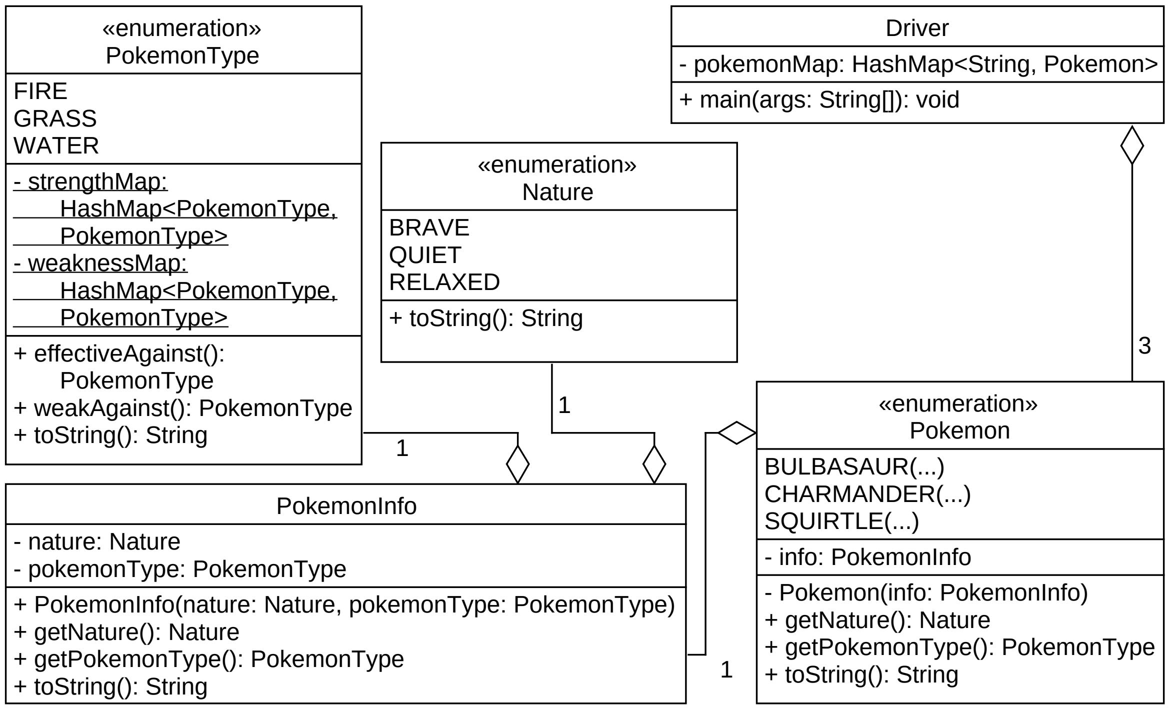
   (b) In Eclipse, select *File/Import*

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

   (d) Select *Select archive file*. Browse to the lab8.zip file. Click *Finish*

Representing Different Pokemon

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.



«enumeration» PokemonType
FIRE
GRASS
WATER
- strengthMap: HashMap<PokemonType, PokemonType>
- weaknessMap: HashMap<PokemonType, PokemonType>
+ effectiveAgainst(): PokemonType
+ weakAgainst(): PokemonType
+ toString(): String
«enumeration» Nature
BRAVE
QUIET
RELAXED
+ toString(): String
Driver
- pokemonMap: HashMap<String, Pokemon>
+ main(args: String[]): void
3
1
1
PokemonInfo
- nature: Nature
- pokemonType: PokemonType
+ PokemonInfo(nature: Nature, pokemonType: PokemonType)
+ getNature(): Nature
+ getPokemonType(): PokemonType
+ toString(): String
1
«enumeration» Pokemon
BULBASAUR(...)
CHARMANDER(...)
SQUIRTLE(...)
- info: PokemonInfo
- Pokemon(info: PokemonInfo)
+ getNature(): Nature
+ getPokemonType(): PokemonType
+ toString(): String

We will only be using the following Pokémon for this lab: *Bulbasaur*, *Charmander*, and *Squirtle*. The keys used with the pokemonMap HashMap are three letter abbreviations for the names of the Pokémon: *BLB*, *CHR*, and *SQR* respectively. The properties of the three Pokémon that we will represent in this lab are as follows:

- BULBASAUR: nature - RELAXED, type - GRASS
- CHARMANDER: nature - BRAVE, type - FIRE
- SQUIRTLE: nature - QUIET, type - WATER

A Pokémon's type affects how well it does in combat with other Pokémon. Certain types are especially effective against others, as defined by the following table:

Type	Effective Against
Fire	Grass
Water	Fire
Grass	Water

Table 1: Type effectiveness table

Lab 8: General Instructions

1. If not already defined (by our zip file), create Java classes for *Nature*, *Pokemon*, *PokemonType*, *PokemonInfo*, and *Driver* as described in the UML diagram
    - 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
    - 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

Nature Enum

This enumeration should have the following members: *BRAVE*, *QUIET*, *RELAXED*.

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

PokemonType Enum

This enumeration has the following members: *FIRE*, *GRASS*, *WATER*.

The PokemonType enum contains a subset of the possible types that a Pokémon can be (there are many more types than the three we are using in the lab).

As previously mentioned, a Pokémon's type effects how well it does in combat with other Pokémon. Certain types are especially effective against others. In order for the members of the enumeration to express the types they are weak/strong against we have two HashMaps to store these relations: *strengthMap* and *weaknessMap*. The *strengthMap* HashMap maps a type to the type that it is strong against. The *weaknessMap* maps the type to the type that it is weak against.

In order to only create and populate these maps 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* and *weaknessMap* HashMaps 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 Java. 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 3 for the mappings you need to setup inside the HashMaps.

The instance methods to be implemented are as follows:

- *effectiveAgainst()*: This method should return the PokemonType that this particular PokemonType is effective against.
- *weakAgainst()*: This method should return the PokemonType that this particular PokemonType is weak against.
- *toString()*: This method should return, in lowercase, the name of the particular member of the enum.

PokemonInfo Class

This class contains information about a particular Pokémon. This information includes the following: the Pokémon's *Nature* and the Pokémon's *PokemonType*.

- *PokemonInfo constructor*: The constructor takes in a *Nature* and a *PokemonType*, and assigns them to the appropriate instance variables.
- *getNature()*: This method returns the *Nature* stored in this particular instance of *PokemonInfo*.
- *getPokemonType()*: This method returns the *PokemonType* stored in this particular instance of *PokemonInfo*.
- *toString()*: This method should return the information stored in this particular instance of *PokemonInfo* in the following format:

```
a <NATURE> <POKEMON_TYPE> type
```

where <*NATURE*> is the *Nature* stored in this instance, and <*POKEMON_TYPE*> is the *PokemonType* stored in this instance. If the *Nature* were to be *BRAVE* and the *PokemonType* to be *WATER*, then the output would look like the following:

```
a brave water type
```

Note: the string ends with the 'e' at the end of *type* (there is no newline character).

Pokemon Enum

The *Pokemon* enum contains a subset of the very large number of Pokémon that exist. This enum has the following members: *BULBASAUR*, *CHARMANDER*, and *SQUIRTLE*.

- *Pokemon constructor*: This constructor takes in an instance of *PokemonInfo* and stores it in the appropriate instance variable.
- *getNature()*: This method returns the *Nature* of the *Pokemon*.
- *getPokemonType()*: This method returns the *PokemonType* of the *Pokemon*.
- *toString()*: This method returns a descriptive string of the *Pokemon*. The string should be in the following format:

```
<NAME>: a <NATURE> <POKEMON_TYPE> type
```

where <*NAME*> is the name of the Pokémon in titlecase (the first letter is capitalized, the rest are lowercase), and <*NATURE*> and <*POKEMON_TYPE*> are covered in the above section detailing the *Pokemon* enum. For the Pokémon *BULBASAUR*, with a *QUIET Nature* and a *PokemonType* of *GRASS*, the returned string would look like:

```
Bulbasaur: a quiet grass type
```

Driver Class

The Driver class will create and populate a HashMap with Strings as keys and members of the *Pokemon* enum as values. It will also present the user with an option to choose a single Pokémon or all of the Pokémon in the HashMap about which to print information. If the user opts to choose a specific Pokémon, then your program will print information for that specific Pokémon. If the user opts for the list, then all of the Pokémon in the HashMap are presented to the user.

- Main menu:

```
Professor Oak, the Pokemon professor, presents you with three pokeballs     ↩
    and tells you that you can choose one as your first Pokemon!
Please select an option:
1: Choose a Pokemon
2: List all Pokemon
```

- Pokémon selection menu:

```
Please choose from the following Pokemon: [SQR, BLB, CHR]
```

  Note: Sets are unordered, so the list of Pokémon could be in any order. The formatting occurs when a Set is printed.

Use a *BufferedReader* to take in the input. Your code will need to be able to handle any input that the user could choose, e.g. numbers other than 1 and 2, letters, Pokémon 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 on chosen Pokémon (or all Pokémon) and exit.

Example Interactions

```
Professor Oak, the Pokemon professor, presents you with three pokeballs - and ↩
    tells you that you can choose one as your first Pokemon!
Please select an option:
1: Choose a Pokemon
2: List all Pokemon
1
Please choose from the following Pokemon: [SQR, BLB, CHR]
SQR
You choose Squirtle: a quiet water type.
Your water type Pokemon is weak against grass types and strong against fire types.
```

```
Professor Oak, the Pokemon professor, presents you with three pokeballs - and ↩
    tells you that you can choose one as your first Pokemon!
Please select an option:
1: Choose a Pokemon
2: List all Pokemon
2
SQR - Squirtle: a quiet water type.
BLB - Bulbasaur: a relaxed grass type.
CHR - Charmander: a brave fire type.
```

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

- All required components (source code and compiled documentation) are due at 11:59:00pm on Friday, October 14th.
- Method 1: Submit through Eclipse
  1. From the *Window* menu, select *Preferences/Configured Assignment.*
  2. Select your project.
  3. From the Project menu, select *Submit Assignment.*
  4. Under *Select the assignment to submit*, select *Lab 8: Enumerated Data Types and Hash Maps.*
  5. Click *Change Username or Password....* Enter your Web-Cat username and password. Click *OK.* You should only need to do this step once per session.
  6. Click *Finish.*
  7. Your browser should automatically open a Web-Cat page that shows your submission being graded. After a short wait, the page will show a report of your submission. See the main class web page for a link that describes the Web-Cat output.
- Method 2: Submit directly to the Web-Cat server
  1. From the File menu, select *Export.*
  2. Select *Java/JAR File.* Click *Next.*
  3. Select and expand your project folder.
  4. Select your *src* and *doc* folders.
  5. Select *Export Java source files and resources.*
  6. Select an export destination location (e.g., your *Documents* folder/directory). This file should end in *.jar*
  7. Select *Add directory entries.*
  8. Click *Finish.*
  9. In your web browser, login to the Web-Cat server.
  10. Click the *Submit* button.
  11. Browse to your jar file.

12. Click the *Upload Submission* button.
13. The next page will give you a list of all files that you are uploading. If you selected the correct jar file, then click the *Confirm* button.
14. Your browser will then open a Web-Cat page that shows your submission being graded. After a short wait, the page will show a report of your submission. See the main class web page for a link that describes the Web-Cat output.

Hints

- All enum types automatically provide a *name()* instance method that will return a String description of the enum value. Specifically, it is the same String that you use to declare the value in the enum class.

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 proportional to the fraction of tests that your code passes (so 22.5 points means that your code passed half of the tests)

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)
- Incomplete coverage of your Unit Tests. We expect that your unit tests will test all lines of your code (up to 15 points)

If you do not submit compiled Javadoc for your lab, 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.