11 Working with HashMap: Overriding ‘equals’

The goal of this lab is to learn to use the professional test harness JUnit. It is completely separated from the application code. It is designed to report not only the cases when the result of the test differs from the expected value, but also to report any exceptions the program would throw. The slight disadvantage is that it uses the Java equals method that by default only checks for the instance identity. To use the JUnit for the method tests similar to those we have done before we need to override the equals any time we wish to compare two instances of a class in a manner different from the strict instance identity.

However, each time we override the equals method we should make sure that the hashCode method is changed in a compatible way. That means that if two instances are equal under our definition of equals then the hash-Code method for both instances must produce the same value.

We start with learning to use HashMap class. We then see how we can override the needed hashCode method. Finally, we also override the equals method to implement the equality comparison that best suits our problem.

The last part of the lab shows you how you can measure the algorithm performance (timing) to see concretely the differences between the running times of different algorithms that have been designed to perform the same tasks.

Part 1: Using the HashMap

Our goal is to design a program that would show us on a map the locations of the capitals of all 48 contiguous US states and show us how we can travel from any capital to another.

This problem can be abstracted to finding a path in a network of nodes connected with links — known in the combinatorial mathematics as a graph traversal problem. You have already seen this problem in your assignments at least once.

The Data

To provide real examples of data the provided code includes the (incomplete) definitions of the class City and the class State.

1. Download the code for Lab 11 and build the project USmap.
2. Download the file of state capitals caps.txt.

3. The project contains an implementation of the Traversal interface by the class InFileCityTraversal that allows you to read a file of City data. The code in the Examples class saves the city data generated by the InFileCityTraversal into an ArrayList.

   Run the code with some of the city data files.

4. The Examples class contains examples of the data for three New England states (ME, VT, MA) and their capitals. Add the data for the remaining three states: CT, NH, RI. Initialize the lists of neighboring states for each of these states. Do not include the neighbors outside of the New England region.

5. Finally, look at the definition of the method toString both in the City class and in the State class. The class Object defines such method for all classes, but it is of little use. Comment out the toString method in the class City and see what happens when you run the code.

   From now on, you should define a toString method for every class you define, making sure the resulting String is readable and the fields are clearly identifiable.

   We now have all the data we need to proceed with learning about hash codes, equals, and JUnit.

Using HashMap

The class USmap contains only one field and a constructor. The field is defined as:

```java
HashMap<City, State> states = new HashMap<City, State>();
```

The HashMap is designed to store the values of the type State, each corresponding to a unique key, an instance of a City — its capital.

Note: In reality this would not be a good choice to the keys for a HashMap — we do it to illustrate the problems that may come up.

1. Go to Java documentation and read what is says about HashMap. The two methods you will use the most are put and get.
2. Define the method `initMap` in the class `Examples` that will add to the given `HashMap` the six New England states.

3. Test the effects by verifying the size of the `HashMap` and by checking that it contains at least three of the items you have added. Consult Javadocs to find the methods that allow you to inspect the contents and the size of the `HashMap`.

Understanding HashMap

We will now experiment with `HashMap` to understand how changes in the `equals` method and the `hashCode` method affect its behavior.

1. Define a new `City` instance `boston2` initialized with the same values as the original `boston`. Now put the state `MA` again into the table, using `boston2` as the key. The size of the `HashMap` should now be 7.

2. Now define the `equals` method in the class `City` that makes sure the two cities have the same name, state, zip code, and the same latitude and longitude. Use the given helper method `sameDouble` to compare the last two fields. Start the method with:

   ```java
   public boolean equals(Object obj) {
       City temp = (City)obj; ...
   }
   ```

   If the given object is of the type that cannot be cast to `City` the method will fail at runtime with the `ClassCastException`.

   Now run the same experiment as above. The resulting `HashMap` still has size seven. Even though we think the two cities are equal, they produce a different hash code.

3. Now hide the `equals` method (comment it out) and define a new `hashCode` method by producing an integer that is the sum of the hash codes of all the fields in the `City` class.

   Now run the same experiment as above. The resulting `HashMap` still has size seven. Even though the two cities produce the same hash code, the `HashMap` sees that they are not `equal` and does not confuse the two values.
4. Now un-hide the equals method so that two City objects that we consider to be the same produce the same hash code.

When you run the experiment again you will see that the size of the HashMap remains the same after we inserted Massachusetts with the boston2 key.

Note: Read in "Effective Java" a detailed tutorial on overriding equals and hashCode.

Part 2: Introducing JUnit

You will now rewrite all your tests using the JUnit. In the File menu select New then JUnitTestCase. When the wizard comes up, select to include the main method, the constructor, and the setup method. The tests for each of the methods will then become one test case similar to this one:

```java
/**
 * Testing the method toString
 */
public void testToString(){
    assertEquals("Hello: 1\n", this.hello1.toString());
    assertEquals("Hello: 3\n", this.hello3.toString());
}
```

We see that assertEquals calls are basically the same as the test methods for our test harnesses, they just don’t include the names of the tests. Try to see what happens when some of the tests fail, when a test throws an exception, and finally, make sure that at the end all tests succeed.

- Add a method that determines whether the city is South of the given latitude. Run the tests using the JUnit.

- Add a method that determines whether this city is in the same state as the given city. Run the tests using the JUnit.

Ask for help, try things — make sure you can use JUnit, so you will not run into problems when working on the assignment and the final project.
Warning

Try to get as much as possible during the lab. Ask questions when you do not understand something. Everything that you do in this lab will be used in the next assignment or in the final project.