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 complare 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.

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.

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 **Part 1** and build the project **USmap**.
2. Download the file of state capitals.
3. The project contains three implementations of the *Traversal* interface. The *InFileBufferedTraversal* allows you to read any *Stringable* data into

an *ArrayList*. The *OutFileTraversal* saves the *Stringable* data stored in an *ArrayList* into a file. The *Interactions* class contains the code that shows you how to do this.

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

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:

```
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 *getKey*.

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 behaves the same way as our *same* method, except for checking first whether the given object is of the type *City*.

   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:

```
/**
 * 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.

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.**

Part 3: Timing and Big-Oh

Download the provided zip file and unzip it. Create a new *Eclipse* project named *Lab11-sorting*. Add the given code to the project. You should have the following Java files:

- class *Examples* defines and runs all the tests.
- class *Algorithms* implements the insertion sort and the quicksort.
- class *IntComp* implements the *Comparator* for integers.
- class *Sorter* is a wrapper that enables us to print the timing results neatly.
- class *Timing* provides a simple way to interact with the system clock.

For this section of the lab we are going to quickly explore the differences between $O(n\hat{}2)$ and average $O(n \log n)$ sorting algorithms.

Insertion Sort:

As mentioned in class, the running time of insertion sort is approximately $O((n * (n + 1))/4) = O(n\hat{}2)$. This is because in order to insert each element into the sorted portion of the *List* we must compare $k/2$ items on average, where k is the size of the sorted portion.

In the *Algorithms* class from the zip for this section, you can see an implementation of *Insertion Sort* which sorts an *ArrayList*<*X*> in-place.

Quick Sort:

This algorithm is considered one of the best in-place sorting algorithms because it is easy to implement and runs pretty fast. Have a look at the implementation in the *Algorithms* class.

Your Task

If you try to run the *Examples* class you will notice there is a *RuntimeException* that's thrown. This is because there is a missing implementation. As further practice with *Comparator*s, you need to implement the *IntComp* class which compares two *Integers* using available functions.

You must then add a new instance of your class to the *Examples* main method (see where the *null* is?) so that the sorting tests will work.

Once you have implemented the class and created an instance, run the *Examples* class to see what it produces. **Check the output to see if it is indeed sorted... if not you will need to fix your comparator!**

When the sorts work correctly, run the *Examples* class again, but this time modify the source to run 3 or 4 timed sort tests by changing the variable loops appropriately. Note the loop which uses this variable.

Results

You should get some reasonable differences between the times of *Insertion* and *Quick Sort* even on these smaller *ArrayLists*.

Before you take-off, look over the interesting portions of the supplied code:

- *static* and *Generic* methods in the *Algorithms* class
- The *fillData*(...) method in the *Examples* class... try to understand what's going on there
- The abstract class *Sorter* and its implmentations that wrap calls to the *Algorithms* code (remember function Objects?) and the methods which use them in the *Example* class.

- And check out the *Timing* for a way to query the *System* for accurate time counts and what we can do with them.