# Hashcode, Equality, Maps, and Graphs

## Goals

In the first part of this lab we will learn how to define the `equals` and `hashCode` methods, and how to use the `HashMap` data structure from the *Java Collections* Framework.

In the second part we will start working on *Graph Traversal* algorithms, leveraging our knowledge of the Java libraries. You will finish this problem as a part of your programming assignment.

## 10.1 `HashMap` and JUnit

Ultimately, the goal of this part of the lab is to learn to use a professional test harness *JUnit*. It is completely separate from application code and is designed not only to report the cases when a result differs from an expected value, but also to report any exceptions the program would/should throw. The disadvantage of JUnit is that it uses Java `equals` methods to compare results, which by default only check for instance identity. In order to effectively use JUnit for testing we need to *override* `equals` correctly.

As discussed in lecture, each time we override the `equals` method we must make sure that the `hashCode` method is also overridden in a compatible way. That means if two instances are equal under our definition of `equals` then the `hashCode` method for both instances *must* produce the same value.

We start by learning to use `HashMap` class, then see how we can override the necessary `hashCode` method. Finally, we also override `equals` to implement an equality comparison that best suits our needs.

### Part 1: Using `HashMap`

Our goal is to design a program that will display the locations of the capitals of all 48 contiguous US states (on a map) and show us how we can travel between the cities.

The problem traveling between cities can be abstracted to the problem of finding a path in a ”network” of nodes connected with links — known in combinatorial mathematics as *graph traversal* problems. You have already seen some parts (`City`s) of this problem in earlier assignments.

### The Data

To provide real examples of data the provided code includes the (partly complete) definitions of `City` and `State` classes.

1. Download the files in `Lab10-HashCode.zip` and create a new project.
2. The Java files contain an implementation of the `Traversal` interface by `InFileCityTraversal` that allows you to read a file of `City` data. The code in the `Examples` class loads the city data generated by an `InFileCityTraversal` into an `ArrayList<City>`.

   Run the code with some of the city data files (`caps.txt`, `minicities.txt`, `smallcities.txt`).
3. The `Examples` class contains 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, but do not include the neighbors outside of the New England region.
4. Finally, we look at defining a `toString` method in both the `City` and `State` classes. `Object` defines a default implementation, but it is of little use (it prints the Object's class name and integer ID).

   Inspect the implementations of `toString` for `City` and `State`. Comment the method out (or just rename it) and rerun the tests... what happens?

   *Note*: In the future (i.e., for HWs) you should implement a `toString` method for each class you create. Make the result readable so the fields are clearly destinguished.

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

### Using a HashMap

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

```
HashMap<City, State> states;
```

The HashMap is defined 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 of keys for a* HashMap *— we do it to illustrate the issues 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 the six New England states to a given HashMap (see put).
3. Test the effect by verifying the size of a 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 into the table again using boston2 as the key. The size of the HashMap should now be 7.
2. Now define an equals method in the class City that compares the City name, state, zip. As we breifly discussed in lecture, start the method with:

   ```
   public boolean equals(Object o){
      City c = (City)o;
      ...
   }
   ```

   If the given object is of the type that cannot be cast to City at runtime, the cast will throw a ClassCastException.

   Now run the same experiment as above (adding MA with boston2). The resulting HashMap still has size 7: even though the two cities are equal, they still produce different hash codes.

3. Now hide the `equals` method (comment it out or rename it) and define a `hashCode` method that produces an integer that is the sum of the hash codes of all the fields in `City` class (ignoring `latitude` and `longitude`).

   Now run the experiment again. The resulting `HashMap` again has size seven. Even though the two cities produce the same `hashCode`, the `HashMap` thinks that they are not the same values.

4. Un-hide the `equals` method so that two `City` objects that we consider to be equal 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 "Effective Java" for a detailed tutorial on overriding `equals` and `hashCode`.*

## Part 2: Introducing JUnit

You will now rewrite all your tests using the `JUnit4`. In the **File** menu select **New** then **JUnitTestCase** and give your test class a name. The tests for each of the test methods will then become methods similar like:

```
/** Testing City toString */
public void testToString(){
    assertEquals(new City(3301, "Concord", "NH", 71.527734, 43.218525)
                 .toString(),
                 "new City(03301, Concord, NH, 71.527734, 43.218525)");
    assertEquals(new City(4330, "Augusta", "ME", 69.766548, 44.323228)
                 .toString(),
                 "new City(04330, Augusta, ME, 69.766548, 44.323228)");
}
```

We see that `assertEquals` calls are basically the same as the test methods for our `tester` library. Right click on the test class and select **Run As**, **JUnit Test**.

Try to see what happens when some of the tests fail, when a test throws an exception, and finally, make sure that in the end all tests succeed.

- Add a method that determines whether the city is South of the given latitude. Add and run tests using your JUnit test class.

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

*Ask for help and try things out — make sure you can use JUnit, so you will not run into problems when completing the assignment and your final project.*

### Warning

Try to get as much as possible during the lab. Ask questions when you do not understand something. *The first part of the next assignment asks you to hand in a complete solution to the next part of this lab.*

## 10.2 Stacks, Queues, Priority Queues, LinkedLists, and Vectors

Look up the documentation for the following Java classes and interfaces: `Stack`, `Queue`, `PriorityQueue`, `List`, `LinkedList`, and `Vector`. Identify which of them represent interfaces, which represent abstract classes, and which provide concrete implementations you can use in your programs.

### Stacks and Queues: Finding a Path

The goal of this exercise is to use the *Java* libraries to do the work for us. We want to compute a path from one city to another, in a graph that represents the 48 contiguous US states. Start a new project *GraphAlgorithms*. You will be able to reuse some of what you have done before for the problems that refered to the US cities, but we are starting anew with more effective use of the Java libraries and a better organization of the data.

Download `Lab10-Graphs.zip` into a new project and run the tests. To help you focus on the interesting parts, we have given you the following classes:

1. `City` that represents a capital of a state. It includes the location given as latitude and longitude, as well as methods that compute the location of the city on a `Scene` of size 400*x*400.
2. `State` that represent a state. Its fields are the name of the state (the two letter abbreviation, the capital `City` and an `ArrayList` of the names of the neighboring states.
3. `USMap` that represents the whole graph - the 48 US capitals and the connections to the neighboring states. This class already has the code that will initialize it with the necessary data.

1. Start by looking at the representation of the graph of the US. It represents the graph of states as a `HashMap<String, State>`, that makes it very easy to find a state and its neighbors.

   *Note:* **(this is not important)** The method `makeStates` uses a different technique for initializing an entire `ArrayList` to the given list of data. You do not need to understand how it is done. At some later time you may want to trace through *JavaDoc*s to understand how this is accomplished,

2. In looking for a path from one city to another we keep track of the visited States. For each state we visit we also remember the state we came from and the distance we have traveled so far. Design a class `FromTo` that will represent this information. Because all information about the capitals of all states is already recorded in the class `USMap`, you only need to record the names of the states. However, include the distance we have traveled from the origin, not the distance between the two states represented.

3. We now start defining the classes we will need to implement the *Graph Traversal Algorithms*. We need to keep track of the USMap, the visited states, and a *To-Do-List* of states to visit. We start with the visited states:

   Define the class `Path` that keeps track of the visited states using a `HashMap`. Use the visited state's name as the `Key` and the instance of your `FromTo` class as the `Value`. So, for example, we may have the following information about states and traveling between them:

   ```
   MA - visited first: came from "", distance 0
   NY - we came from MA, distance 130
   NH - we came from MA, distance 60
   VT - we came from NH, distance 60 + 70
   NJ - we came from NY, distance 130 + 100
   PA - we came from NJ, distance 130 + 100 + 90
   ```

   Make sure you include the above example in your tests. (The distances you get may be different from the ones we gave you — the given classes implement the computation of distances and your program should use it.

   The class `Path` should have a constructor that consumes the `String` that is the name of the origin and adds the first item to its record of visited states. This first `FromTo` object should have the origin set to the empty `String`, the distance set to 0, and the destination to be the given *origin*.

4. In the class `Path` design the method `fromToDist` that consumes two `String`s that represent the *beginning* and *ending* states for one leg of the journey, and the instance of the `USMap` and produces the distance between them.

5. In the class `Path` design the method `add` that consumes two `String`s that represent the *beginning* and *ending* states for one leg of the journey, and the instance of the `USMap` and adds to the `Path` the appropriate `FromTo` object: using the *ending* state as the key, and adding the current distance to the distance we already traveled to get to the *beginning* state.

6. In the class `Path` design the method `pathTo` that produces an `ArrayList` of `FromTo`-s we need to go through to get to the given `City`. So, for the above example, we would expect the following results:

```
pathTo(MA) --> [MA distance 0]
pathTo(NY) --> [MA distance 0;
                  NY distance 130]
pathTo(PA) --> [MA distance 0;
                 NY distance 0 + 130;
                  NJ distance 0 + 130 + 100;
                   PA distance 0 + 130 + 100 + 90]
```

7. In the class `Path` design the method `contains` that determines whether the state given as `String` is in this `Path`.

8. In the class `Path` design the method `directionsFromTo` that consumes the state of origin and our desired destination (as two `String`s) and produces the travel directions as a `String`. For example,

```
directions("MA", "MA") produces:
   "from MA go to traveling a total of 0 miles"

directionso("MA", "PA") produces:
   "from MA go to traveling a total of 0 miles
    from MA go to NY traveling a total of 130 miles
    from NY go to NJ traveling a total of 230 miles
    from NJ go to PA traveling a total of 320 miles"
```

We now want to keep track of the neighbors of the states we plan to visit soon (the `ToDo` checklist). So, for example, if we visit `MA`, we will add to the `ToDo` checklist all of its neighboring states. However, there are some restrictions. We do not add a neighbor to the checklist if it is already in the `Path`.

The interface `ToDo` describes the desired behavior:

```
interface ToDo{
  /** Add a new neighbor to the ToDo checklist
   * @param state the state whose neighbors we should add
   * @param path the path that has been already traveled
   */
  public void add(String state, Path path);

  /** Remove a state from the ToDo checklist
   *    throw an exception if the checklist is empty
   * @return next state to be visited
   */
  public String remove();

  /** Is this ToDo list empty?
   * @return true if there are no more states to visit
   */
  public boolean isEmpty();
}
```

9. Define the class `ToDoStack` that keeps track of the neighbors to visit soon that uses the `Java Stack` class to implement the `ToDo` interface as a `stack`.

10. Define the class `ToDoQueue` that keeps track of the neighbors to visit soon that uses the `Java LinkedList` class to implement the `ToDo` interface as a `queue`.

**The ground work you have done here provides all the parts you need for implementing two different graph traversal algortihms *Breadth-First Search* and *Depth-First Search*.** You will finish this work in the assignment for this week.