10 Java Collections; Equality, JUnit

Activities

1. Familiarize yourself with some of the Java Collections Framework.
2. Learn the basics of overriding the *hashCode* and *equals* methods.
3. Learn the basics of working with the *JUnit*.

Warning

In the final project you will find useful almost everything from this lab. Concretely, your porject should use some of the *Java Collections Framework* classes and interfaces — showing that you can build on the work already done. You will also be required to write your tests using JUnit, which, in turn, requires that you override the *equals* method, and the *hashCode* method as well.

10.1 Activity: Reading JavaDocs

Go to the Java API at *http://java.sun.com/j2se/1.5.0/docs/api/*. Bookmark this page! When coding you will often use classes that are provided for you by Java. The Java API describes these classes and lists all of the fields and methods of these classes that are available to you.

The front page of the Java API lists all of the packages provided by Java. A package is a collection of related interfaces and classes.

Tips For Quickly Finding Class Specifications

The left frame of the API page lists all classes alphabetically. If you want the specifications for a specific class you can click in this frame and use your web browsers search function to find that class. For example, find the *ArrayList* class. Another way to quickly find Java API specifications is to search Google for "java api class x", where x is the name of the class you're searching for. For example, the search "java api class arraylist" returns the specifications for class *ArrayList* as the first result.

The Anatomy of a JavaDoc

All of the specifications are in a JavaDoc format. JavaDocs are automatically generated from source code based on specifically formated comments that the programmer adds for each class and each method. We will look at the format of such comments shortly.

Lets use the *ArrayList* JavaDoc as an example.

The top of the JavaDoc lists the other classes that *ArrayList* extends and implements. In this case, *ArrayList* extends from the classes *Object*, *AbstractCollection*, *AbstractList*, and implements the interfaces *Cloneable*, *Collection*, *List*, *RandomAccess*, *Serializable*.

Next is a general description of the class. In this case, the JavaDoc says that *ArrayList* is a ”Resizable-array implementation of the List interface.”

Following this is a summary of fields, constructors, and methods provided by *ArrayList*. In general, classes will provide very few public fields and the JavaDoc will contain mostly specifications of methods. Look over some of the methods provided by *ArrayList*.

The method summaries provide headers (return type, name, and arguments) and a short description of the method's functionality. More detailed descriptions are linked from these summaries and appear farther down on the same page.

Stack, Queue, Priority Queue, LinkedList; Vector

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 a complete implementation that you can use in your program. Draw a class diagram that shows the relationship between these classes and interfaces.

Generating JavaDoc-s

For detailed description of how to write documentation for the automatic Javadoc generator see *http://java.sun.com/j2se/javadoc/writingdoccomments*: How to Write Doc Comments for the Javadoc.

To see how to write the commets in the *javadoc* style, look at the code for the previous lab.

Look first at the code for the class *Traversal*. Notice the special format of the comments. Notice also that they are shown in a different color than the comments we have seen so far.

When the comments for Java programs are written using this special format, the documentation web pages can be generated automatically — with all the cross-references necessary.

The comment always starts with /∗∗ and usually spans several lines. Each line then starts with a ∗ and the last line has only ∗/ in it. When you start typing such comment in *Eclipse* the color of the comment changes and the ∗ at the beginning of the line is generated automatically. In addition the beginnings of some of the special comment commands are also generated for you.

To see how to write the comments while designing a program, start by adding to the *Examples* class a *stub* of a method *reverse1* with the header below. (A *stub* is a method with a complete header and the body that only produces the correct type of value, but does not perform the desired computation. We use the stubs as place-holders when designing a program just to make sure the program would compile. Later, we design the rest ofthe method.)

```
public <T> ArrayList<T> reverse1(ArrayList<T> alist){
  return alist;
}
```

then start the comment above. You will see that it generates the following *template* for the comment:

```
/**
  *
  * @param <T>
  * @param alist
  * @return
  */
```

We complete the comment as follows:

```
/**
  * Reverse the elements in the given <code>ArrayList</code>
  * using a helper <code>ArrayList</code>
  *
  * @param <T> the datatype for the elements of <code>ArrayList</code>
  * @param alist the original <code>ArrayList</code>
  * @return <code>ArrayList</code> with elements in reverse order
  */
```

In **Project** menu select **Generate Javadoc...** and choose the *doc* folder for the documentation. Choose to make the documentation pages only for the class *Examples*.

When done, look at the pages in a browser. The *index.html* file will be in the folder you have selected.

This is enough for a start — experiment with Javadoc-s for a whole project at home. For the remainder of the semester, always write comments so that we can generate complete documentation from the program sources.

10.2 Activity: 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 checks first whether the given object is of the type *City*, then compares the two objects field by field. The implementation of the *ISame* interface already does most of what you will need.

   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.**