

Traversals; Testing private Methods; Direct Access Data Structures

In this lab we will practice abstracting over the type of data in a collection, see how Java documentation presents classes and methods, look at a couple of Java Collections Library classes, and work with the direct (random) access data structure that we've seen in Lecture.

8.1 Writing JavaDocs; User-Defined Exceptions

In the last lab we took a look at documentation generated from the special comments written in the *JavaDoc language*. Starting with this lab (i.e., on the next assignment) you should write all your purpose statments as *JavaDoc* comments. Start by looking at how the comments are defined in the `tester` library.

On the main lab page find the link to the documentation for the `tester` package. You can see that the library consists of three interfaces, six regular classes and two `Exception` classes. In the table there is a comment next to each class, interface, and exception, and each name is a link to a more detailed description.

Click on the link for the `IllegalUseOfTraversalException` class. The `tester` library defines a new class of exception specific to situations that may be encountered. Typically programmers developing a library will create special `RuntimeException` subclasses to signal special errors, unless a standard exception (e.g., `IllegalArgumentException`) suits their needs. The content of a class that extends `RuntimeException` usually accepts a `String` message which is passed to the **`super`** constructor, though you can customize (fields etc...) to provide more error information.

We now look at where our `IllegalUseOfTraversalException` is used/thrown. Follow the link to the `Traversal` interface. In the class list (lower left frame) you'll also notice that the names of interfaces are written in *italics*.

Here is the code for the `Traversal` interface, notice the format of the comments for each source element:

```
package tester;

/**
 *  An interface that defines a functional iterator
 *  for traversing datasets
 *
 * @author Viera K. Proulx
 * @since 30 May 2007
 */
public interface Traversal<T>{

   /** Produce true if this
    *  {@link Traversal Traversal}
    *  represents an empty dataset
    *
    *  @return true if the dataset is empty
    */
    public boolean isEmpty();

   /** Produce the first element in the dataset represented
    *  by this {@link Traversal Traversal}.
    *
    *  <p>Throws <code>IllegalUseOfTraversalException</code>
    *  if the dataset is empty.</p>
    *
    *  @return the first element if available -- otherwise
    *  throws <code>IllegalUseOfTraversalException</code>
    */
    public T getFirst();

   /** Produce a {@link Traversal Traversal}
    *  for the rest of the dataset
    *
    *  <p>Throws <code>IllegalUseOfTraversalException</code>
    *  if the dataset is empty.</p>
    *
    *  @return the {@link Traversal Traversal}
    *  for the rest of this dataset if available - otherwise
    *  throws <code>IllegalUseOfTraversalException</code>
    */
    public Traversal<T> getRest();
}
```

Each element of a Java file (class, interface, signature/header, method) can be preceeded by a multiline comment that begins with "/**" (note the

double star) and ends normally with "`*/`". Everything in the comment makes up the purpose statment for the element that follows.

You can use this example as a guide for writing your own *JavaDoc* comments, but see:

`http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`

for more technical information and examples of different `@` keywords/uses.

8.2 Annotations and `main` Methods

Java *annotations* can also be used to leverage the `tester` without the need for you to put all of your test cases within a single `Examples` class. Annotations in Java (and other languages such as C#), allow programmers to include meta-data to describe a given element (e.g., class, method, or variable).

The *tester* provides two kinds of annotations that clients/programmers can use on their classes and methods: `@Example` and `@TestMethod`. Using these annotations is quite simple.

```
@Example
public class NumExamples{

   public NumExamples(){}

   @TestMethod
   public void numberSanity(Tester t){
      t.checkExpect(5, 5);
      t.checkFail(10, 5);
      t.checkExpect(5 + 5, 10);
   }
}
```

When the `tester` is run via `tester.Main` the it searches for any classes that have the `@Example` annotation and examines them for methods with the `@TestMethod` annotation and runs them as tests accordingly. Using annotations gives us the flexibility necessary to test `private` fields and methods by writing test cases within our respective classes.

```
import tester.*;

/** Represents a non-empty list */
@Example
public class ConsLo<T> implements ILo<T>{
   /** The first item of this list */
   private T first;
```

```
    /** The rest of this list */
    private ILo<T> rest;

    /** The full constructor
     * @param first  The first item of this list
     * @param rest   The rest of this list
     */
    public ConsLo(T first, ILo<T> rest) {
       this.first = first;
       this.rest = rest;
    }

    /** Produce a list, adding the given item to this list
     * @param t the given object
     */
    public ILo<T> add(T t) {
       return new ConsLo<T>(t, this);
    }

    @TestMethod
    /** EFFECT: Test the constructor and the add
     *   method for this class */
    private void testFirstAndRest(Tester t){
       // Setup
       ConsLo<Integer> cons =
         new ConsLo<Integer>(5, new MtLo<Integer>());
       ConsLo<Integer> cons2 = cons.add(8);

       // Tests
       t.checkExpect(cons.first, 5);
       t.checkExpect(cons2.first, 8);
       t.checkExpect(cons.rest, new MtLo<Integer>());
       t.checkExpect(cons2.rest, cons);
    }
}
```

When we have multiple classes with test cases, we do not always wish to run our test cases all at once every time we run our project. In these scenarios, it becomes acceptable to use a `main` method to run individual `Example` classes with the tester.

```
public static void main(String[] args){
   // This will run with normal reporting enabled
   Tester.run(new ConsExamples());
   // This will run with verbose reporting enabled
```

```
    Tester.runFullReport(new ConsExamples());
  }
```

The main method must be defined in a public class. You can invoke the main method of a class in a number of different ways. In Eclipse it is easiest to right-click on the defining class and select
`Run As > Java Application`. You can also go through the `Run` menu: `Run > Run As > Java Application`. If the option does not exist then either your class is not public, or the `main` signature does not match.

See the `tester` documentation for more information on difference between `run` and `runFullReport` methods.

8.3 Implementing Traversals

Create a new project in Eclipse called *Lab8*. Add an interface, `ISelector`, similar to what we've used for predicate function-classes. Don't forget to add the `tester` library to the project's build-path. Something like this should work:

```
/** Represents a predicate for the data of any type */
interface ISelector<T>{
  /** Returns <code>true</code> if this predicate
   *  holds for the given item
   *
   * @param t  The given item
   */
  public boolean pick(T t);
}
```

In the past we have designed interface/classes to represent recursively constructed lists of arbitrary items, but in order to add new functionality we had to modify each of our interfaces/classes. This works well when we are the only developer/user of our classes, but if we want to distribute our program as a library, we need to provide methods that allow the clients to later manipulate data in the list *from the outside*.

The `Traversal` interface we saw in lecture was specifically designed for this purpose, so we can design the classes that represent lists of data, and add the methods needed to implement the `Traversal` interface. This way, methods defined outside of our list classes (and interface) can also have access to the list data.

Add the `Lists.java` file from the main lab page to your project. These definitions should look familiar (remember the lecture notes?).

It doesn't look like we've achieved much, but we can now define methods that manupulate items in the list (like `filter`) from outside our list classes. When we build such classes/libraries there is no way to provide all the methods that clients will need, so instead we can provide a clean interface to allow clients access to the data.

As an example (like in Lecture) we can place these kinds of methods in a separate class we call `Algorithms`. Add the `Algorithms.java` file/-class to your project and make examples of lists of `String`s. Design a few test cases for `filter`; you do not have to completely test the method, but make sure you understand what is going on and how methods in the `Algorithms` class can be used.

Java Collections Framework

Go to the web site for Java libraries at:
`http://java.sun.com/javase/6/docs/api/`.
Scroll through the *All Classes* in the lower left frame to find the `Comparable` and `Comparator` interfaces. You can see from the descriptions that there is a lot of detail, much more than we would expect from such a simple function-object. We will address some of these issues in the lectures.

ArrayList

Scroll through the *All Classes* frame on the left again and find the `ArrayList` class. In lecture we discussed a number of methods we can use to modify/manupulate and instance of `ArrayList`. Use the online documentation, lecture notes, or your steel-trap of a memory as you work on this last part of the lab.

8.4 Direct Access Data Structures with Mutation

For this part of the lab download the `Balloons.zip`, which contains the following files:

- *Balloon.java*: a sample data class representing Balloons.
- *TopThree.java* will be used to practice working with `ArrayList` in imperative style (using mutation).
- The *Examples.java* file that defines examples of all data and defines all tests.

Create a new **Project** *Lab8ArrayLists* and import these files. Don't forget to add the `tester` library too. In this part of the lab we will work with `ArrayList`s of `Balloons`.

Here are some of the important methods defined in the `ArrayList<E>` class. See the JavaDoc for their purpose statments:

```
boolean isEmpty();
int size();
boolean add(E obj);
E get(int index);
E set(int index, E obj);
```

8.5 Using the ArrayList class

Note that in order to use an `ArrayList` we have added

```
import java.util.*;
```

To the beginning of our file(s).

1. The class `TopThree` now stores the values of the three elements in an `ArrayList`. Complete the definition of the `reorder` method. Have a close look at the documentation for the `ArrayList` and `Comparator` to decide which methods you should use.

2. In the `Examples` class, design the method `isSmallerThanIndex` that determines whether the radius of the `Balloon` at the given position (index) in the given `ArrayList` of `Balloons` is smaller than the given **`int`** `limit`.

3. Design the method `isSameAsIndex` that determines whether the balloon at the given position in the given `ArrayList` of `Balloons` has the same `size` and `location` as the given `Balloon`.

4. Design the method `inflateAtIndex` that increases the radius of a `Balloon` at the given index by 5. For this you should also design a helper method... where do you think it should be?

5. Design the method `swapAtIndices` that swaps the elements of the given `ArrayList` at the two given positions (indices). Make sure that this method works for any `ArrayList`... it should be parametrized as we did in Lecture.

Note 1: We have used the words *position* in the `ArrayList` and *index* in the `ArrayList` interchangeably. Both are commonly used, so we want to make sure you are comfortable with to both ways of describing locations of elements in an `ArrayList`.

Note 2: Of course, the tests for these methods will also appear in the `Examples` class. Make sure that every test can be run independently of all other tests. To do this, you must initialize (`reset`) the data at the begining of your test method, then evaluate the test by invoking the appropriate `check` method. Just to be safe, you may also reset the data after the tests are complete.

Note: Finish this lab and include your work in your portfolio.

8.6 Generating Javadocs

If you have time left, convert the purpose statements for your homework classes into `Javadoc` comments and generate web pages of your documentation. Under the `Project` menu select `Generate Javadoc` and then select the files that should be used to generate the documentation (likely your entire project). By convention we typically generate JavaDoc files into a directory named `docs` at the same level as the `src` directory. Be sure to fix any warnings and problems with the generation.