8 Understanding Libraries

8.1 Reading JavaDocs

The documentation for Java projects can be quite extensive. Reading just the comments in the code is difficult. Furthermore, when a library is distributed, one does not always have access to source on hand. In those cases, it become imperative that quality documentation is accessible in a human readable format.

To make it possible to generate readable and searchable documentation for Java programs, the programmers write specially formatted comments in *the Javadoc style*. Sun, the developer of Java, provides a program that reads the documentation and generates nicely formatted web pages that contain all "well-formatted" comments provided by the programmer.

On the main web page find the link to the documentation for the `tester` package. You will see right away that it consists of three interfaces, six regular classes and two `Exception` classes. There is a comment next to each of these. On the left is a list of all classes, interfaces, and exceptions and each name is a link to the detailed description of that particular item.

Follow the link to the `interface ISame`. The code for this interface has been written as follows:

```
package tester;

/**
 * An interface to represent a method that compares
 * two objects for user-defined equality.
 *
 * @author Viera K. Proulx
 * @since 30 May 2007
 */
public interface ISame<T>{

  /**
   * Is this element the same as that?
   * @param that element
   * @return true is the two elements are the same
   *    (by our definition)
   */
  public boolean same(T that);
}
```

Now look at the `class IllegalUseOfTraversalException`. It shows you that programmers can define a new class of exceptions, specific to the situations that may be encountered in their programs. The content of a class that extends `java.lang.RuntimeException` is quite standard and there is not much to see there.

We will now look at where this exception is needed. Follow the link to the `interface Traversal`. Did you notice that the names of interfaces on the left hand side bar are written in *italics*?

Here is the code for the `interface Traversal`:

```
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
   * <CODE>{@link Traversal Traversal}</CODE>
   * represents an empty dataset
   *
   * @return true if the dataset is empty
   */
   public boolean isEmpty();

   /**
     * <P>Produce the first element in the dataset represented
     * by this <CODE>{@link Traversal Traversal}</CODE> </P>
     * <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();

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

Next week you can use this as a guide for writing your own *JavaDoc* documentation.

8.2 Annotations and `main` Methods

As many of you have already seen from demos given by Weston Jossey, annotations can be used to leverage the Tester without the need for you to put all of your test cases within a single @Example class. Annotations in Java (and other languages such as C#), allow programmers to use meta-data to describe the behavior or nature of a given class, method, or variable. A real world example of humans using annotations would be when doctors and nurses will use "annotations" in the form of different symbols to mark mild, critical, and fatal injuries during a crisis scenario.

The Tester provides a programmer two annotations: @Example and @TestMethod. Utilizing these annotations is quite simple.

```
@Example
public class NumExamples{

  public NumExamples(){}

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

When the Tester is run via tester.Main (as we have been doing all semester), the Tester is able to find any classes that have the @Example annotation, exmaine them for any methods with the @TestMethod annotation, and then run them accordingly. It is important to note that because of this flexibility, we now have the ability to test private fields by writing test cases within our respective classes.

```
import tester.*;
//Represents a non-empty list
@Example
public class ConsList<T> implements IList<T> {
    private T first;
    private IList<T> rest;

    public ConsList(T first, IList<T> rest) {
        this.first = first;
        this.rest = rest;
    }

    //Adds a new element to the list
    public IList<T> add(T t) {
        return new ConsList<T>(t, this);
    }

    //Checks to see if this.first is equal to the
    //given element, or if this.rest contains the
    //given element.
    public boolean contains(T t){
        ISame<T> comp = (ISame<T>) first;
        if(comp.isSame(t))
          return true;
        else
            return this.rest.contains(t);
    }

    @TestMethod
    private void testFirstAndRest(Tester t){
        ConsList<Integer> cons =
            new ConsList<Integer>(
                        5, new MtList<Integer>());
        t.checkExpect(cons.first, 5);
        t.checkExpect(cons.rest,
                      new MtList<Integer>());
    }
}
```

When we have multiple classes with test cases, we do not always wish to run all of our test cases all at once. In these scenarios, it becomes acceptable to use main methods 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());
}
```

You can now invoke your main method a number of different ways; however, if you are inside of Eclipse, you can either set your run configuration to point to your main class, or, while you have the class with your main method open, go to: Run – Run As – Java Application

What advantage or disadvantage does **runFullReport** give you? For more information, go to the Javalib website and check out the documentation.

8.3 Implementing Traversals

Create a new project in Eclipse called *Lab8*. Add to it an interface we used before, the `ISelector` interface. Also, add the `tester` library, just as you have done with every other project.

In the past we have designed classes that represent recursively constructed lists of arbitrary items. However, every time we wanted to add some functionality to these classes, we had to modify all three classes. This works well when we are the sole users of our program. If we want to distribute our program as a library, we need to equip the classes with methods that will allow the users that come later on to manipulate the data contained in this list.

The `Traversal` interface has been designed to supply the methods we may need for any program that needs to look in some orderly manner at the data contained in a list.

We would like to - again - implement our `filter` method. We will need again the `ISelector` interface:

```
// Our usual Selector interface
interface ISelector<T>{
  boolean pick(T t);
}
```

We can now design the classes that represent lists of data:

```
//Generic List Union
interface ILo<T> extends Traversal<T>{
  // Note that isEmpty(), getFirst() and getRest()
  //   are also added to this interface by extending
  //   the 'implementation' of Traversal
}

//Represents an Empty List of T
class MtLo<T> implements ILo<T>{
  // Basic Constructor
  public MtLo(){}

  // Traversal functions so that things like 'filter' can be
  //   written without disturbing the list classes
  public boolean isEmpty(){ return true; }

  public T getFirst(){
    throw new IllegalUseOfTraversalException(
    "No first element in an empty list"); }

  public Traversal<T> getRest(){
    throw new IllegalUseOfTraversalException(
    "No remaining elements in an empty list"); }
}


//Represents a non-empty list of T
class ConsLo<T> implements ILo<T>{
  private T first;
  private ILo<T> rest;

  // Basic Constructor
  public ConsLo(T first, ILo<T> rest){
    this.first = first;
    this.rest = rest;
  }

  // Traversal functions so that things like 'filter'
  // can also be written without disturbing the list classes
  public boolean isEmpty(){ return false; }

  public T getFirst(){ return this.first; }

  public Traversal<T> getRest(){ return this.rest; }
}
```

It looks like we have not achieved much. However, we can now define the `filter` method outside of the classes that represent the list of items. If we wish to build a library, we do not know what methods will the user need. We need to provide a clean interface for the user, so that the user can add new methods that will deal with the data contained in the list.

We place these methods in a separate class we call `Algorithms`.

```
//First attempt at a generic filter algorithm
class Algorithms{
  // Filter the Traversal based on the given Selector
  public <T> ILo<T> filter(Traversal<T> tr,
                           ISelector<T> choose){

    if(tr.isEmpty())
      return new MtLo<T>();
    else
      if(choose.pick(tr.getFirst()))
        return new ConsLo<T>(tr.getFirst(),
                             filter(tr.getRest(), choose));
      else
        return filter(tr.getRest(), choose);
  }
}
```

Add these classes and interfaces to your project. Make examples of lists of `String`s and design a couple of test cases for these methods. You do not have to complete all tests, but make sure you understand what is going on and how the method in the `Algorithms` class can be used.

8.4 F.I.F.O. Queue

For the past seven labs, we have primarily worked with only one type of data structure, lists. Now, it is time to take what we have learned about manipulating lists and apply them to other data structures (such as Doubly-Linked-Lists, Binary Search Trees, Queues, Stacks, etc.). You will begin to notice that many data structures are built on the top of other data structures, so it is important to build a strong foundation before moving on to more complicated structures.

A queue is something that can be found in every day life. When you wait in line at your favorite fast-food restaurant, you are a part of a real life first in first out (FIFO) queue. A FIFO queue behaves exactly as we would expect it to. The first item that enters the queue is the first item that will get to leave the queue.

We have provided you with the following files:

1. IQueue.java
2. Inspectable.java
3. Inspector.java

Each of these represents an interface which you will need to implement to make your code operational. The `IQueue` interface is your template for building queues. Read the comments to understand the purpose of each method.

Implement the `IQueue` interface using the same style we have used for lists (there should be an empty class, as well as a non-empty class). Write examples to fully exercise your methods. Remember that `deQueue` should return the least-recent element that has been added to the queue.

8.5 The Fuzz

The following problem is meant to challenge your problem solving skills. Examine the problem, discuss possible solutions with your partner(s), and agree on the best way to tackle the problem. Use your notes from Monday's lecture to help guide you with the visitor pattern.

Cedar Point is a roller coaster theme park located in Sandusky, Ohio. As "America's roller-coast", Cedar Point is recognized as one of the key spots for roller coaster enthusiasts to frequent. However, as with any other venue with tens of thousands of visitors, there are bound to be some "unsavory" characters.

Cedar Point has `ParkPolice` that monitor the `Queue`s for any potential felons that may have entered into the park. Unfortunately, a police officer has no way to actually go through the `Queue`s to examine the individuals, because the crowds are too thick to move through.

Step One: Extend the `IQueue` interface with the *Traversal* interface. Implement the methods as needed. This should be simple enough if your

`Queue` implementation was done in a clear manner. Write test cases to verify that your methods behave as expected.

Step Two:

```
Sometimes the easiest way to understand a problem
is to make up some "mock" conversation that simulates
the behavior of some action or sequence of actions.
The Visitor pattern is not the simplest of techniques
to understand,  but a little "dialogue" can turn that
confusion clear as mud.

This dialogue is a representations of steps two and
three.

   Police Chief:  Hey Queue Inspector!  Wake up.
                  We’ve got felons on the loose!
                  Inspect this queue and check
                  to see if any felons are queueing.
                  If they are, boot them out.  You
                  can tell that a Person is a felon
                  if they are wearing pinstripes.

Queue Inspector:  No problem Chief.  Hey Queue
                  (ok... he’s a delusional inspector),
                  give me some information about the
                  people in your queue.

          Queue:  Alright.  This person is at the
                  front of this queue, and here’s
                  the rest of this queue.  I
                  expect you to give me the result
                  of your search after you’re done.

Queue Inspector:  Thank you.  I see the first person
                  is a felon.  They will not be in the
                  queue after I’m done...  Hey, Rest
                  Of The Queue, give me some
                  information about the people in your
                  queue.
```

```
      ROT-Queue:  Alright.  This person is at the
                  front of this queue, and here's
                  the rest of this queue.  I
                  expect you to give me the result
                  of your search after you're done.

Queue Inspector:  Thank you.  Ah!  Someone who isn't
                  a felon!  I shall keep you in the
                  queue.   Hey, Rest Of The Rest Of
                  The Queue, give me some information
                  about the people in your queue...



I will leave it up to you to finish the story!
The conversation has to end at some point.  The
question is where!
```

Write a `ParkPolice` class that implements the `Inspector` interface. In addition, extend your `IQueue` with the `Inspectable` interface. Don't worry about your two `Queue` classes right now, we just want to extend the interface so we can see our method availability with auto-complete.

A `ParkPolice` class has no fields, and two public methods. The role of a `ParkPolice` instance is to uncover any `Person`s in the park that may be wearing pinstripes (which is a sure sign they are a steroid user!). In our Visitor pattern, our `ParkPolice` instance will be visiting a `Queue` instance.

Question... Can you name at least five cheaters who have played for the Yankees in the past decade?

Take note that your `Person` class can be structured and handled however you want, so long as a `ParkPolice` officer can discern that the `Person` is not a felon.

Step Three: Finish implementing your `Queue` classes so that it satisfies the `Inspectable` interface. When you do this, use your `Traversal` methods to obtain the proper "first" and "rest". While this may seem redundant inside of a `Queue`, think about why this added step might save headaches if you were to implement `Inspectable` for a Binary Search Tree. *Hmm.... That sounds like a nice homework problem...*

As always, make sure you have examples that fully exercise your code. If you are confused about how to test steps two or three, make sure to ask a TA or tutor for assistance in understanding the visitor pattern.

8.6 Javadocs

If you have some time left, convert all documentation for the classes you designed into the *Javadoc* style and generate the web pages of documentation. In the *Project* menu select *Generate Javadoc* and then select which files should be used to generate the documentation. See where you have warnings and fix the problems.