9 Using Iterators; ArrayList Class; Designing Tests

Goals

In this lab you will learn to work in a more complex Java environment, using a library, using a simple test harness, and using classes from the Java Collections Frameworks. You will also encounter a RuntimeException, though no work is necessary on your part.

9.1 Organizing your work.

Download the lab zip file and unzip it. Start a new Project in eclipse with the name Lab9. Add all .java files from the unzipped lab folder to your project. You should have the following files:

- **interface** *ISelect* to represent a selection method
- **interface** *IRange* to represent an iterator
- **class** *ArrayListRange* to represent an iterator for ArrayList
- **class** *Algorithms* a container for methods filter, *filterWhile*, and *orMap*, as well as a source of sample data.
- **class** *Examples* that contains the test suite and methods you will write. It extends *SimpleTestHarness*.
- **class** *SimpleTestHarness* that manages the test evaluation and reporting.

  class *Interactions* that provides a framework for user interactions.

The Test Harness: use it to define data and write tests.

When designing a program it may be appropriate to see the results of all the tests. However, as the program size increases, the number and complexity of tests grows as well and it becomes increasingly more difficult to read all the results and process the information the results represent. A test harness is a program that provides for the user a way to organize the tests so that all the relevant results are reported in an organized manner. Additionally, a test harness may verify that all the paths through the program were tested (every clause of every statement was executed).

Out very simple test harness provides the following methods:

```
// reporting only the result of a test
test(String testname, boolean result);

// test verifying expected.equals(actual)
test(String testname, Object expected, Object actual);

// test verifying expected.same(actual)
test(String testname, ISame expected, ISame actual);

// test verifying expected == actual
test(String testname, int  expected, int actual);

...
// and one of the above for all primitive types except:

// test verifying expected same as actual within epsilon
test(String testname,
      double  expected, double actual, double epsilon);

...

// produce a report of all failed tests
void testReport()

// produce a report of all tests
void fullTestReport()
```

The programmer then collects all tests in the **class** *Examples* that extends the *SimpleTestHarness* and designs a method *runTests* that runs all the tests and reports the results.

The test harness keeps a record of all tests, and of all failed tests and provides methods for reporting the results.

For example the following would provide one test for each of the methods *size*() and *get*(0) when invoked by the object *ArrayList* object *data*:

```
public void runTests(){
  // two sample tests
  test("Sample data length", 57, data.size());
  test("First word", "envision", data.get(0),);
  // add more tests here ...

  // produce a report of the failed tests
  testReport();
  // produce a report of all test results
  fullTestReport();
}
```

The method *runTests* can then be invoked by the
`public static void main(String[] argv)`
anywhere, or in our *Interactions* class.

The JPT Library: supporting user interactions.

You project will use the jpt.jar library. It is saved in a folder that has the name **edu.neu.ccs.jpt** - following the Java conventions for naming packages of library files. Your lab instructor will guide you through the instructions for adding the library to your project.

Run the project. The GUI window with the buttons for each public method that takes no arguments and produces *void* is generated automatically from the *Interactions* class. In the *Interactions* class you can use the **JPT** console. You can use the console for the following kinds of interactions:

```
// display the given String in the console
console.out.println(String s);

// display the prompt and await user’s input: must be given!
String s = console.in.demandString("prompt ");

// display the prompt and await user’s input:
// hiting return with nothing typed in cancels the request
String s = console.in.requestString("prompt ");


// display the prompt and await user’s input: must be given!
// if input does not represent a number, error is reported
// in the console and a new input is demanded (till success)
int n = console.in.demandInt("prompt ");

// display the prompt and await user’s input
// if input does not represent a number, error is reported
// in the console and a new input is requested
// (till success, or until hitting return with no input
// cancels the request
int n = console.in.requestInt("prompt ");
```

These commands exist for all primitive types (double, boolean, etc.).

To Do: Warmup with interactions and examples.

- Add a method to the *Interactions* class that requests an integer, performs some calculation and display the output and see how it works.
- Add a couple of test cases to the *Examples* class, both successful and unsuccessful ones and observe what is reported.
- Define a String *s* in the class *Examples* and write a test case to test the *startsWith*(String *s*) method for the class String.
- Make examples of an *ArrayList* of Strings to be used as data.

For the rest of the lab you can add methods to the *Interactions* class if you wish to explore your program's behavior, or interact with the program through the console.

Algorithms: contains code to use and test; is not modified here.

The **class** *Algorithms* is a collection of methods that can be used with several different data sets and with a variety of function objects - a collection of reusable methods. For the start it only has one variant of *orMap* and two variants of a filter.

We also added two helper methods to provide the data for a challenge problem.

9.2 Designing and Using Function Objects.

To Do: Designing functions objects.

- Design the class that implements the *ISelect* interface with a method that determines whether the given *Object* is a String shorter than 4 letters. Your tests should be in the *Examples* class. You may add some interaction to the *Interactions* class.
- Design the class that implements the *ISelect* interface with a method that determines whether the given *Object* is a String that starts with the given prefix. Again, you will need tests in the *Examples* class.

9.3 Working with the ArrayList.

ArrayList is a class that represents a collection of data that can be accessed in order. Additionally, every element can be accessed directly by specifying its position (index) in the ordering. The first item in this collection is at index 0.

Here are some of the methods defined in the class *ArrayList*:

```
// how many items are in the collection
int size();

// add the given Object at the end of this collection
// false if no space is available
boolean add(Object obj);

// return the object at the given index
Object get(int index);

// replace the Object at the given index
// with the given element
void set(int index, Object obj);
```

The methods you design here should be added to the *Examples* class, together with all the necessary tests.

To Do: ArrayList direct access

- Design the method that determines whether the word at the given position in the given ArrayList is a short word.
- Design the method that determines whether the word at the given position starts with the given prefix.
- Design the method that swaps the *Object*s at the two given positions.

9.4 ArrayList Traversal Using the Iterator.

There are several different ways a programmer can use to traverse an ArrayList. We use an implementation of our IRange iterator.

Writing methods that use the *IRange* iterator for traversal is identical regardless of the underlying structure. The first two problems are simple and straightforward. For the second two you should use one of the two given filters. Again, implement these methods in teh *Examples* class.

To do: Using the iterator directly

- produce a String that combines all the Strings in the structure that can be traversed with the given *IRange*.
- count the number of short words in the collection that is traveresed with the given *IRange*.

To do: Using loops and iterators

- produce a list of all short Strings from the given *IRange*.
- produce a list of the Strings that start with the given prefix.

9.5 Challenge Problem.

The sample data provided in the class *Algorithms* is encoded. To decode it, you need to remove from the list all Strings that do not start with one of the *allowedPrefixes* defined in the *Algorithms* class.

Design the method or methods that perform the decoding.

Hint: Produce a function object that checks whether the given String starts with one of the *allowedPrefix*-es.