Programming assignment #1: CSE 12 Spring 2013

Here's a description of the first programming assignment for CSE 12, Spring 2013. In this assignment, you will develop a test plan for a class that implements the Java Collections interface `List<E>`, and implement the test plan using the JUnit framework.

>>> Due Fri Apr 12 5:00 PM

>>> Required solution file: List12Tester.java

This means that the file that contains your solution to the assignment must be turned in by the due deadline by using the bundleP1 script while logged into your cs12 account on ieng6. If your file is turned in 29 hours or more before the deadline, you will get 1 additional point added to your programming assignment score. The turnin procedure is described below in the section "Turning in your assignment".

It's a good idea to plan to do the assignment well before the deadlines; terminals and tutors can get very busy at the last minute.

In doing this assignment, it's your responsibility to understand the course rules for integrity of scholarship.

Getting started

Review the appropriate lecture notes, and optionally chapters 0, 1, sections 2.1, 2.2, 5.2 and 5.3 of the textbook. This assignment uses material introduced and discussed there. You will also find online documentation of the `java.util.List<E>` interface and the `java.util.LinkedList<E>` class useful; see http://docs.oracle.com/javase/7/docs/api/java/util/List.html and http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html.

Next, create a directory named "P1" under your cs12 home directory. Your solution to this assignment should be turned in from that directory by the due date and time. You will use the bundleP1 script to do that; see below.

The List<E> interface

In a later assignment, you will define a class named `List12` that implements the `java.util.List<E>` interface. How will it implement that interface? There are many possible ways to implement any one interface, and we will consider some of them in detail later. But first it is important to understand what that interface requires of *any* implementation.

One good way to make sure that you understand what an interface requires is to develop a test plan which can test any implementation of the interface to make sure it meets the requirements. In this assignment you will do that, and implement the test plan using the JUnit framework. This is an example of the strategy of test-driven development.

The `List<E>` interface in the java.util package requires 25 different public instance methods. A full test plan

should test all of them. However, for this assignment you need only to test these methods:

```
public boolean add(E o);
public void add(int index, E element);
public boolean contains(Object o);
public E get(int index);
public Iterator<E> iterator();
public boolean remove(Object o);
public E remove(int index);
public E set(int index, E element);
public int size();
```

Any class that implements the `List<E>` interface has to provide definitions for methods with those method headers (and the other methods required by the interface also). But what do those methods *do*? Their names suggest what they do, but to really understand what they are required to do, you need to read the full documentation of the interface. This is available online, and there is also a useful discussion in section 5.3 of the text.

Developing a test plan

When you have understood what the public interface to a class (in this case, a class named `List12` with type parameter `E`, that implements the `java.util.List<E>` interface) is supposed to do, you can develop a test plan for the class. Each public method should be checked, to make sure that if its preconditions are met when it is called, its postconditions (including class invariants), are met when it returns; and that any return value or thrown exception is what it should be. It is usually practically impossible to check a method with all possible valid and invalid inputs (there are too many), so you should think about what inputs are good ones to use. Chapter 2 and section 5.4 of the text give some good ideas to keep in mind when designing a test plan.

For example, the `boolean add(E e)` method of `java.util.List<E>` requires that `e` be added to the list, and furthermore, that it be added at the end of the list, and that it return `true` because the element was added successfully. So some possible tests in your plan might be:

- create an empty `List12`, call the `add` instance method several times to add some elements, and call the `contains` method to check that they are present. Also, call the `contains` method several more times with different arguments to check that other elements are not present!
- create an empty `List12`, call the `add` instance method several times to add some elements, and call the `get` method to check that the elements are present, in the correct order.
- create an empty `List12`, call the `add` instance method several times to add some elements, and check the return values.

Implementing your test plan: List12Tester.java

Once you have a test plan worked out, you are ready to implement it as a test program. The JUnit framework is the most popular one for writing Java test programs; we will use JUnit version 3.8.1. The basics for using JUnit are described in Chapter 2 of the text, the JUnit FAQ and the JUnit documentation.

Define a class named `List12Tester` that extends `junit.framework.TestCase`.

The test fixture will be an instance of the class `List12<E>` for some type parameter `E`; for example, you can use `String` or `Integer`, etc. The test fixture can be an instance variable in your `List12Tester` class, or a local variable in each of your "testXXX" methods.

You should define as many "testXXX" methods as your test plan suggests. As a simple and incomplete example, given the part of a test plan in the previous section, you might define some methods something along these lines:

```
public void testAddContains() {
  List<String> theL = new List12<String>();
  theL.add("A");
  theL.add("B");
  assertTrue(theL.contains("A"));
  assertTrue(theL.contains("B"));
  assertFalse(theL.contains("C"));
}

public void testAddGet() {
  List<Integer> theL = new List12<Integer>();
  for(int i=1; i<=1000; i++) {
    theL.add(i);
  }
  for(int i=0; i<1000; i++) {
    assertEquals(theL.get(i),new Integer(i+1));
  }
}

public void testAddReturn() {
  assertTrue(theL.add("A"));
  assertTrue(theL.add("B"));
  assertTrue(theL.add("A"));
}
```

You can optionally define a `setUp()` and a `tearDown()` method to initialize (and "de-initialze") the test fixture, if it is an instance variable. Defining a `main` method then will let you run your `List12Tester` class as a program, once you have a `List12<E>` class to test with it! (See the next section for that.) To run your `List12Tester` as a text-output program, define `main` as:

```
public static void main(String args[]){
  junit.textui.TestRunner.main(new String[] {"List12Tester"});
}
```

Or, to run it as a GUI program, define `main` as:

```
public static void main(String args[]){
  junit.swingui.TestRunner.main(new String[] {"List12Tester"});
}
```

Testing your tester

How can you test your List12Tester class? You want to make sure that all your List12Tester tests will pass when testing a version of the List12<E> class that correctly implements the List<E> interface, and at least one test will fail if the version does not correctly implement the interface.

You need to define the List12<E> class before you can do that. (In fact, you need one before you can even compile your List12Tester class, since it references List12<E>!)

Normally, after you have written your test program, you would start defining your List12<E> class, and each time you add some functionality to it, compile it and use your List12Tester program to test it; it should pass more tests each time. However, for this assignment, it is easy to define some versions of the List12<E> class that you can use for testing, taking advantage of software reuse.

The Java standard class libraries contain a class java.util.LinkedList<E> that correctly implements the java.util.List<E> interface. So using inheritance, it is very easy to define a class List12<E> that correctly implements the interface:

```
public class List12<E> extends java.util.LinkedList<E> {}
```

Similarly, you can define versions of List12<E> that fail to implement the interface in various ways, by overriding a method from java.util.LinkedList<E> and providing an incorrect defintion of it. Whatever List12.class file is in the same directory as your List12Tester program is the one that will be tested when that program runs.

A correct version of List12<E> is available in the directory `~/../public/P1/GoodL12/` on ieng6. It should pass all your List12Tester tests. One incorrect version (you will also need to create some of your own) is available in the directory `~/../public/P1/BadL12/`; it should fail at least one of your tests.

Turning in your assignment

After you are done with the assignment, you can turn it in. To turn in your assignment when logged into ieng6 or a lab workstation, cd to the directory containing your solution file and run the command

```
bundleP1
```

and carefully follow the dialog. If it is before the due deadline and all required assignment files are present, this will turn in those required files. (There is no way to turn in any additional files.) You can run bundleP1 more than once if you want (and it is a good idea to try bundleP1 well before the deadline, even if your solution files aren't your final version, just to test it); only the last one will count.

To find out the current "official" time, run the command "date" on ieng6. After you have turned in your files, your assignment will be graded, and your grade emailed to **your cs12 account, on ieng6.ucsd.edu**. (Grading may take up to 7 days after the due deadline; please be patient until then.)

Further hints and suggestions

1. In reading the documentation of the `List<E>` interface, you will see that some methods are described as an "optional operation". In addition, some aspects of some methods are documented as optional; for example, the `add()` method may allow adding `null` elements, or it may not. For this assignment, assume that all such options are to be implemented in the way that the `java.util.LinkedList<E>` class implements them. Refer to the online documentation for details.
2. Also, pay close attention to the `List<E>` interface to make sure that your `List12Tester` does not depend on anything *not* specified by that interface. This is important because we will be testing your tester

with our own `List12` implementation, and it will not provide anything beyond what is specified in that interface. If you are making use of other constructors or instance or static methods or variables in your tester, it will not compile or run at all when we use our implementation. This is an important consideration for all the assignments in CSE 12, not just this one.

Grading

There are 10 possible points on this assignment, not including early-turnin points. We will test your solution as follows:

- We will copy `~/../public/P1/GoodL12/List12.class` and your `List12Tester.java` file into an otherwise empty directory.
- We will then compile your solution with the command:

  ```
  javac List12Tester.java
  ```

  (If your solution file does not compile, you will recieve 0 points for this assignment.)
- We will run the tests in your `List12Tester` class to make sure the correct `List12<E>` implementation fails none of your tests.
- We will then copy in various incorrect versions of `List12.class` and run the tests in your `List12Tester` on each of them, to make sure that each one fails at least one of your tests.

We will also consider your coding style, including commenting.

If you have any questions about what constitutes good commenting or coding style, see a tutor or TA. At least you should include a javadoc comment for each public variable, method, and class. In the javadoc comment for the public class in a source code file, include an `@author` tag to state your name and course login name. (To generate the actual hyperlinked documentation, you can run the command `javadoc *.java` in your P1 directory; to view it, visit the file `index.html` it creates with your browser.)

In addition, each function body should typically contain some comments explaining the basic flow of control and the logic of the code. But don't overcomment; put them where they are useful for understanding. You can assume that your audience understands Java, but they may not understand the particular logic you are relying on in your algorithm.

Readability is important: The code you turn in is a finished product that will be read by others, so make it readable. Your code should be indented in a consistent and readable way. Indent 2 to 4 spaces at each level (8 is too much). Try to keep each line of code less than 80 characters long, so can be viewed nicely in a standard terminal window. If you do development under another environment, make sure the source code files you turn in do not contain control characters that make it hard to read.

Good luck!