Lab 3: Loops and Testing
12:00 PM, Feb 7, 2018

Contents

1 Loops! 1
2 Test Setup Methods 2
3 Testing, testing, 1 2 3 4
4 Just for Fun: Making an Oracle 5
5 Testing the Oracle 6
6 Testing our BST Makers 6

Objectives

By the end of this lab, you will know:

- how to use loops to do cool things
- how and when to use setup methods in testing

By the end of this lab, you will be able to:

- write oracle testing methods

1 Loops!

Now that you’ve learned a bit of imperative programming, it’s time you put your looping knowledge to good use!

Task: First, go ahead and copy the source code for this lab:

```bash
cp /course/cs0180/src/lab03/src/* ~/course/cs0180/workspace/javaproject/src/lab03/src
```

In this part of the lab, you’ll use loops to help an online shopper with an online catalog (represented as a LinkedList of Items). An Item is a class that only contains a String description and int price, and the Item class has already been provided to you.

You’ll be using a Java enhanced for loop. As a refresher, here’s an example of the syntax, where you might be iterating through a List<Integer> called myList:
for (int num : myList) {
    // do something with num
}

Task: Open up Eclipse and make a new class called Catalog, which has a List<Item> named catalog as the only class variable. Also create a constructor that takes in a List<Item> and sets catalog equal to that list. Implement the following four functions using enhanced for loops:

1. lookupPrice(): given an item description, look up and return the price of the item with that exact description in the catalog. If the item isn’t found, throw a RuntimeException.
2. countItems(): given an item description, return the number of items in the catalog that contain (as in, have as a substring) the given description. For example, an item with description “Fine Wooly Socks” should be counted if the input description is “Socks”, because “Socks” is contained in that item’s description.
3. findItems(): similar to the above task, search for items that contain the given description, except this time, rather than just counting them, return a list containing these matching items.
4. updatePrice(): given an Item and a new price, update the catalog so that item’s price now matches the new price. You’re looking for a particular instance of an Item, not all items matching the same information in the input Item, so you should use shallow equality (==, not .equals).

Hint: Remember, you can’t use == to test String equality! You must use .equals(). For example, if you wanted to test equality between Strings s1 and s2, you could do s1.equals(s2).

Hint: Try looking up the Java String contains() documentation.

Hint: To get checked off, you have to have at least readable code and full javadocs comments!

You’ve reached a checkpoint! Please call over a lab TA to review your work.

2 Test Setup Methods

In CS18, we’ve seen how some of our methods are designed to update values, rather than to return computed answers. Methods that update values pose particular challenges for testing. Why? Consider the following two tests for the methods you have just implemented:

```java
public class CartTester {
    Catalog testCat = new Catalog();
    Item radio = new Item("radio", 10.95);
    Item hat = new Item("green witch hat", 8.50);

    public void setUpCat(){
        testCat.addItem(radio);
        testCat.addItem(hat);
    }

    public boolean testGetPrice(Tester t) {
        t.checkExpect(testCat.lookupPrice(radio), 10.95);
    }

    public void testRemItem(Tester t) {
        testCat.addListItem(radio);
    }
}
```
testCat.addItem(hat);
testCat.remItem(radio);

t.checkExpect(testCat.findMatches(radio), new LinkedList<Item>());

public static void main(String[] args) {
    tester.run(new CartTester());
}

Will both of these tests pass, or not? If testGetPrice runs first, both tests should pass. But if testRemItem runs first, then testGetPrice will fail because there is no longer a radio in the catalog. This example points out something critical about mutation and how it impacts testing:

Once your program allow mutation, two identical calls to the same method (or function) may yield different results.

This cannot happen when you program functionally. It’s a new complexity that arises when methods or functions update values. This has significant impact on testing, because it implies that the order in which tests run changes the result of testing. This is a serious problem. Most testing frameworks don’t guarantee the order in which they will run your tests (precisely to force you to keep tests independent of each other). This makes sense, because in multi-programmer projects, people should be free to write tests without worrying about interference from tests written by others.

To prevent problems, we have to adapt how we write tests to be immune from such impacts. In particular, our tests need to perform Setup to make sure they start from a clean, predictable state.

Let’s change our CartTester class to have a method that creates a catalog with our two items:

Now, inside each test method, we first use initCatalog to make sure we know the contents of our testing data:

public static boolean testGetPrice(Tester t) {
    cat = this.initCatalog();
    return t.checkExpect(cat.lookupPrice(radio), 10.95);
}

public static void testRemItem(Tester t) {
    cat = this.initCatalog();
    cat.remItem(radio);
    return t.checkExpect(cat.findMatches(radio), new LinkedList<Item>())
    ;
}
Note here that we don't use the class-wide `testCat` catalog in testing – we use the fresh catalog that we set up for the tests in that method.

What if we want to test the effect of multiple methods though? Just call them all in the same method. For example:

```java
public static boolean testAddItem(Tester t) {
    newCat = new Catalog();
    newCat.addItem(radio);
    return t.checkExpect(cat.lookupPrice(radio), 10.95);
}
```

You can even put multiple `checkExpect`s in the same test method, as long as they build on the same initial data. The order of statements within a method is fixed (top to bottom), so there is no need to worry about these tests running out of order.

**Task:** Write tests for the four functions you wrote involving catalogs above, using setup methods as needed. In particular, you should have some test methods that would otherwise interfere with each other if they all worked off of the same class-wide catalog.

**Task:** In general, pairs of methods that can affect each others' output should be tested together (such as `addItem` and `lookupPrice`). What combinations of methods do you think need to be checked to make sure method interactions are as expected?

---

3 Testing, testing, 1 2 3

So far in CS 18 (and throughout your CS experience at Brown), you've primarily written unit tests. This is a test for functions where one particular input can only produce one correct output, i.e. there are not multiple solutions.

In this part of the lab, you'll learn how to test functions where there are many correct solutions. Let's suppose, for example, that we are sorting a list of pairs of numbers by the first number:

```plaintext
{(2, 3), (2, 2), (5, 1), (0, 5)}
```

This could produce either of the following:

- `{(0, 5), (2, 3), (2, 2), (5, 1)}`
- `{(0, 5), (2, 2), (2, 3), (5, 1)}`

Both are correctly sorted! So using the testing methods we've worked with so far, you could (wrongly!) fail the test. Let's say our sorter gives us the first list, but our tester checks if the given list was the second list. If this is the case, then we would fail the test even though we had a correct answer. We need to change how we think about testing to account for multiple solutions.

In comes oracle testing. Oracle testing, rather than comparing your method’s output with one correct solution, checks certain characteristics of your method’s output and ensures they all hold true.

To create an oracle, we first need to know what characteristics make for a correct output. You should try to find the minimum number of characteristics, without allowing for any wrong output to fulfill those. Let's take our sorting example. The two characteristics of a correct output are:
• The output list contains exactly the same elements as the input list (including the correct count if there are duplicate elements).
• The output list is sorted according to our sorting scheme.

If these characteristics hold, then we have a correct output, and if either one does not hold, our output is wrong by definition.

For this lab, you will be writing an oracle to test whether a program that claims to generates binary search trees (BSTs) from a list of numbers produces a valid answer. We have provided you with two implementations (in different classes) of a method called `makeBST`. The method takes a list of Integers and produces a binary search tree of Integers. You will write a function that takes the original list and the output of `makeBST` and produces a boolean indicating whether the output is a valid answer.

**Task:** Write down (on paper or in a text editor) the characteristics of a correct output, i.e. a correct BST created from an input list.

**Hint:** Feel free to look up the definition of a BST on Wikipedia to refresh yourself on its invariants!

You’ve reached a checkpoint! Please call over a lab TA to review your work.

4 Just for Fun: Making an Oracle

Now that you know what constitutes a correct output, it’s time to write a method to check each of these conditions. Each method should take in any parameters it needs to test its particular characteristic, and return a boolean indicating if the characteristic holds true. For example, if we were creating methods to test our sorting method, we would have:

```java
public static boolean testSameElements(List<Integer> original, List<Integer> sorted) {
    //code goes here
    return isSame;
}
```

We would also have a test to see if a list is correctly sorted:

```java
public static boolean testSorted(List<Integer> sorted) {
    //code goes here
    return isSorted;
}
```

**Task:** We have also provided you with all of the solution files you’ll be modifying today, so go ahead and copy those over as well, as follows:

```bash
cp /course/cs0180/sol/lab03/sol/* ~/course/cs0180/workspace/javaproject/sol/lab03/sol
```

Before we explain what we have provided, we need you to double-check that your files from Lab02 are all in the right folders and named correctly! Specifically, we want you to open the Lab02 files and check at the top that they are all in the `lab02.sol` package. This just means ensuring they all have this line at the top of the file: `package lab02.sol;`. You should have:

• IBST.java
- Node.java
- Leaf.java
- A test class for IBST (doesn’t matter what it’s named)

**Task:** Ensure you have the above files and that they’re in the `lab02.sol` package. If they are not, you should first make sure you have a `lab02` package (you can see this under `javaproject` on the file explorer pane in Eclipse). If not, right-click `javaproject` and make a `New > package`, and name it `lab02`. Once you have that, right-click the files in the wrong package, and select Refactor > Move. Then, select the `javaproject/sol/lab02.sol` package and click finish. Make sure it was updated properly!

Now, we’ll explain the class hierarchy we’ve created for you. In the source folder, you have `IBSTMaker`, an interface for a BST maker, which has the method `makeBST`. Implementing this interface are `BSTMaker1` and `BSTMaker2`.

Additionally, there are four BST-based classes: `Node1` and `Node2`, which both extend your `Node` class from last lab, and `Leaf1` and `Leaf2`, which both extend your `Leaf` class from last lab. `BSTMaker1` makes trees of type `Node1` and `Leaf1`, and `BSTMaker2` makes trees of type `Node2` and `Leaf2`.

Also in the solution folder, we have a class `Oracle`, containing the method `sameElems`, which takes in an `IBST` and a list of `Integers` and determines if they contain exactly the same elements (including correct duplicate counts). We also have provided a stub for the `bstOracle` method, which you’ll be writing in this next part!

That’s all you’ll need to know, for now! It’ll be your job to implement two methods: `isBST`, which takes in an `IBST` and returns a boolean indicating if it is a valid binary search tree, and `toList`, which makes a list of a tree’s elements. You can find information on how to approach this task in the previous lab.

**Task:** Write `isBST` and `toList` in your files from `lab02`. You should open the `Lab02` PDF for help with this task! And as always, don’t forget to test!

**Task:** Now, inside `Oracle`, fill in the method called `bstOracle`, which takes in a `List<Integer>` and the `IBST` created from the list, and calls both `isBST` and `sameElems` with the necessary arguments. `bstOracle` should return a boolean indicating if both tests passed.

---

| You’ve reached a checkpoint! Please call over a lab TA to review your work.

### 5 Testing the Oracle

Now, you should have your `isBST` and oracle methods written. However, we aren’t quite ready to test our BST implementations with this oracle. First, we have to make sure the oracle itself works!

**Task:** In the `OracleTest` class we’ve provided, write the `checkExpect` for each test method to make sure that various binary trees are (or are not) indeed BSTs. Our class provides several `IBSTs` that you can use for testing. Note that not all of our provided trees are actually BSTs; be sure to write the `checkExpect` correctly to reflect this!

### 6 Testing our BST Makers

Now that you have your working oracle, it’s time to use it to check whether the `BSTMaker` implementations we provided are buggy.

**Task:** On a piece of paper, write down the various situations and edge cases that you think you should test. Remember that the input to the `makeBST` methods is a list of numbers, so your “edge cases” here should be various kinds of lists that you think make for a good collection of tests.
Task: In the BSTMakerTest class we provided, create checkExpect tests for several of your edge cases. Your goal should be to write enough tests to determine whether each of the two makeBST methods produces valid BSTs (if one does not, can you detect what is wrong with it?). Your tests should check both of the makeBST implementations.

Hint: Look at our example test! You can add an optional String argument to your checkExpect to name your tests so that when they fail, you can tell which test it was.

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)