Learning Goals

During this lab, you will:

- Review some easy Java OO
- Learn how to write good test cases
- Experiment with your IDE's debugger
- (Bonus!) Review Java Generics

Testing

You first start off looking at the Sorter interface. It only has one method: sortList. You have an implementation file called BadInsertionSorter and an associated test file. Naturally, the first thing you do when handed these files is to go ahead and run the tests... Immediately you see something wrong. How should you approach this?

First, we should look at the test case we were provided. It's pretty much a catch-all solution, but we clearly can't identify what a problem is if we are just running this. In general, when you fail a unit test you should be able to pinpoint the problem – each unit test should be targeting one piece of logic. Of course, we can't achieve this 100% of the time, but we can try our best.

Let's first add some more test cases in bottom-up testing style.

We also see that we have a helper method that isn’t being explicitly tested. This is bad! You should always test your helper methods. Write some test cases for your swapItems method.

You might have written a simple method that tests swapItems on a list with two items in it. Here’s such a method:

```java
@Test
public void testEasySwap() {
    List<Integer> list = new ArrayList<>(Arrays.asList(2, 1));
    BadInsertionSorter<Integer> sorter = new BadInsertionSorter<>();
    sorter.swapItems(list, 0, 1);
    assertEquals(Arrays.asList(1, 2), list);
}
```

Because we used JUnit’s assertEquals method instead of assertTrue, the Failure Trace in Eclipse’s JUnit panel gives us some useful information when we run our test: java.lang.AssertionError: expected:[1,2] but was <[1,1]>. It looks like there’s a bug in our swapItems method! Using this and other test cases that you’ve written, identify and fix the bug now.

Debugging

Let’s return to the first test method that we wrote: testSort. Now that we fixed the problem with the helper method, we are at least getting slightly more reasonable results... We are getting the numbers back, but they are still not sorted. This is where we can jump into the debugger, which will allow us to step through our code slowly and inspect variables’ values closely.
First, we can set some breakpoints. A **breakpoint** is a signal that tells the debugger to temporarily suspend execution of your program at a certain point. In Eclipse, a breakpoint is represented by a small, blue dot to the left of a line in the editor panel (the place where you edit your Java files).

Let’s add a breakpoint at line 13 (\texttt{int j = i;}) in our \texttt{BadInsertionSorter}. To add a breakpoint here, double-click on the space to the left of line 13. A small, blue dot should appear. (To remove a breakpoint, double-click on the blue dot again.) Now, run your test suite in Debug mode.

There are 4 main operations in Debug mode:

- **Resume** - resume the execution of the currently suspended program. The program will continue running until the next breakpoint is hit or until program completion.

- **Step Into** - step into the next method call at the currently executing line of code.

- **Step Over** - step over the next method call (without entering it) at the currently executing line of code. Even though the method is never stepped into, the method will be executed normally.

- **Step Return** - return from a method which has been stepped into.

Using these four operations, step through \texttt{testSort} in Debug mode. What bugs can you find? **Hint:** How should insertion sort work on the list in \texttt{testSort}?

**Testing Due Diligence**

Our insertion sort method \texttt{sortList} seems to be working now; however, we should still write a few more tests to convince ourselves. What are some additional test cases that we might write for \texttt{sortList}?

- a singleton list
- a list with two elements
- a list in reverse-sorted order
- an already-sorted list

Can you think of some others?

**Writing Correct Test Cases**

Let’s look at a test case that we might write for a list with two elements:

```java
@Test
public void testBogusTwoElements() {
    List<Integer> list = new ArrayList<>();
    list.add(2);
    list.add(1);
    InsertionSorter<Integer> sorter = new InsertionSorter<>();
    List<Integer> sortedList = sorter.sortList(list);
    assertEquals(list, sortedList);
}
```

Is this a valid test case? Why or why not? **Hint:** Think about the conditions under which this test case will pass.

A best practice for unit testing is to make sure that your tests are as \textit{transparent and obvious as possible}, so that spotting bugs in your test cases is as easy as possible.
Generics in Java are very versatile. Rather than needing to write different Collections (for instance) for each type of possible class, or just having all Collections contain Objects, you can parameterize a class with an unknown type, and even specify some interface that that type implements. This allows the flexibility of being able to parameterize any type with another, while not sacrificing the safety that Java provides with strong, static type-checking.

```java
public class ArrayList {
    // Instance variables and return values must be Objects, or else class can only be used
    // for one kind of type.
    // Any kind of object can now be added into the array, so you may as well use a
    // language like Python with no static typing.
    Object[] arr;
    public void add(Object elem){...}
}
```

The way that Generics work in Java is when a class or interface is defined, you use diamond notation, like so:

```java
public class ArrayList<T> {
    // Instance variables and return values can now be defined as a specific type.
    T[] arr;
    public void add(T elem){...}
}
```

This defines some type, T, that will be used uniformly throughout the class, allowing return types, instance variables, etc, to be statically typed without knowing what the exact type the parameter will be. One note is that the parameterized type must extend Object, so you can’t parameterize a type with, say int or String[], but you can with Integer.

Issues can happen if you try to use parameters in both a parent and child class. For example, take this ArrayList class, with an inner ALIterator class to allow for the class to implement Iterable interface.

```java
import java.util.*

public class ArrayList<T> implements Iterable<T> {
    T[] entries;
    T get(int i) {...}
    // Rest of implementation here

    ALIterator<T> extends Iterator<T> {
        T current = entries[0];
        T next() {...}
        boolean hasNext(){...}
    }

    Iterator<T> iterator() {
        return new ALIterator<T>()
    }
}
```

This code contains an error on line 10, "Type mismatch: cannot convert from T to T." Logically, this doesn’t make sense. Where is this implementation causing problems for Java’s type-checker? **Hint:** Think about what the error means and why Java believes that T is not the same type as T.

Generics are even more powerful, however. Another advantage to using generic types is it is possible to declare that a generic type implements a certain interface. Take a look at the Sorter interface from the starter code for this recitation.

```java
import java.util.*

public class ArrayList<T> implements Iterable<T> {
    T[] entries;
    T get(int i) {...}
    // Rest of implementation here

    ALIterator<T> extends Iterator<T> {
        T current = entries[0];
        T next() {...}
        boolean hasNext(){...}
    }

    Iterator<T> iterator() {
        return new ALIterator<T>()
    }
}
```
public interface Sorter<E implements Comparable<E>> {
    List<E> sortList(List<E> inputList);
}

What value does declaring E as implementing Comparable<E> provide?