Homework 1 – Exercise 3.2 – Hints

I wrote this document several semesters ago to help a student better understand how to solve Homework 1 – Exercise 3.2 and thought it would be helpful to share it with the class.

1. *java.util.ArrayList* is a "generic class" meaning that when you declare an *ArrayList* object (named *list*) you must specify the data type of the elements of *list* using this syntax,

```
ArrayList<type> list;
```

2. *type* cannot be a fundamental or basic data type such as **int** or **double**, but rather it must be a class type. Consequently, one cannot create *an ArrayList* of **ints** or **doubles**. This is where wrapper classes and autoboxing comes to the rescue. *Integer* and *Double* are classes that encapsulate a basic **int** or **double** variable within objects of the the *Integer* and *Double* classes. Therefore, if we wish to create an *ArrayList* that contains integers, we use the *Integer* wrapper class and write,

```
ArrayList<Integer> list;  // Declare an object named list which is an instance of the
                          // ArrayList<Integer> class.
```

3. The next step—which is a common mistake by being omitted—is to create or instantiate *list* by writing,

```
list = new ArrayList<>();  // Create an object named list which is an instance of
                           // the ArrayList<Integer> class.
```

Remember that in Java, to "make" an object that can be used, we first have to **declare** the object and second, we have to **create** the object. These two steps can be combined into one Java statement,

```
ArrayList<Integer> list = new ArrayList<>();
```

4. Note the parentheses following **new ArrayList<>** in the statement above. Since *list* is an object that has now been created, what happens next is that a constructor (ctor) must be called on *list* and the empty pair of parentheses indicate that we are calling the **default ctor** on *list*, i.e., the ctor that has no arguments. The default ctor—and ctor's in general—do some internal housekeeping on the object being created to initialize the object. Generally, this involves initializing the data members of the object.

5. Now that *list* has been declared, created, and initialized, we can begin to access the elements of *list*. To do that, we use the *ArrayList* class *get*() and *set*() accessor and mutator methods, i.e., we cannot access the elements of an *ArrayList* using brackets the way we access the elements of a primitive array.

But before we can access the elements using *get*() and *set*(), we first have to add some elements to *list*. Now, the default ctor inits *list* to be empty so it has 0 elements. *size*() is one of the several *ArrayList* methods that can be called to manipulate *list*. What *size*() does is return (as an **int**) the size of *list*, i.e., the number of elements in *list*. The default ctor that got called initialized *list* to be empty, so at this time if we called *list.size*() it would return 0. It would be a run-time error to call *list.get*(3) or *list.set*(5, 4) at this time because *list* is empty.

6. We can add elements to *list* by calling the *ArrayList* method named *add*(). For example, *list.add*(1) would add 1 to *list* so now *list* would contain one value, e.g., *list* would be { 1 } and calling *list.size*() would return 1. Next, calling *list.add*(5) would add 5 to *list*—at the end—so now *list* would be { 1, 5 } and *list.size*() would return 2. For testing purposes for Ex. 3.2, you can create and initialize your list this way,

```
import java.util.ArrayList;

public class H1_32 {

    // main() is the starting point of execution for all Java programs.
    public static void main(String[] pArgs) {
        new H1_32().run();
    }
```

```
    // Default ctor for the H1_32 class. Since we have no instance data to initialize, our
    // ctor does nothing.
    public H1_32() {
    }

    public void run() {
        // Declare and instantiate a list named list.
        ArrayList<Integer> list = new ArrayList<>();

        // Initialize it by calling initList().
        initList(list);

        // For debugging purposes, let's ensure that list contains the values we think it
        // should. We can do this by calling our helper method printList().
        printList(list);

        // Now it is time to perform the operations on list as specified in Ex. 3.2
        for (int i = 1; i < 10; ++i) {
            list.set(i, list.get(i) + list.get(i-1));
        }

        // And now that we have modified list, let's print it out again to see the result.
        printList(list);
    }

    // Initialize pList (which refers to the same ArrayList<Integer> object that list declared
    // in run() does). Consequently, any changes we make to pList are really changes to list.
    // After we return from this method to run(), list will be { 1, 2, 3, 4, 5, 4, 3, 2, 1,
    // 0 }. By the way, I sometimes preface my methods parameters with a lowercase 'p' to
    // indicate to the reader that pList is a parameter variable.
    private void initList(ArrayList<Integer> pList) {
        for (int i = 1; i < 6; ++i) {
            pList.add(i);
        }
        // Now, pList should be { 1, 2, 3, 4, 5 }
        for (int i = 4; i >= 0; --i) {
            pList.add(i);
        }
        // Now, pList should be { 1, 2, 3, 4, 5, 4, 3, 2, 1, 0 }
    }

    // This is a helper method that simply prints the values in pList to the terminal window
    // on separate lines (because we called System.out.println() rather than
    // System.out.print()).
    private void printList(ArrayList<Integer> pList) {
        for (int i = 0; i < pList.size(); ++i) {
            System.out.println(pList.get(i));
        }
    }
}
```

And there you have it, a complete program that can be used to verify that your solution to Ex. 3.2 is correct—although I hope you derived your solution by hand before you wrote this program.