Lab 5 — Interfaces and ArrayLists

1. Download **lab05_files.zip** from the class web page and use it as your starting point for this lab.

2. Together we developed a Dataset class that works with Measurable objects. As an example, the program TestCircle creates a Dataset of Circle objects, and then reports their average and largest areas. A Circle implements the Measurable interface by providing the getMeasure() method, which returns the Circle's area. Review the code for Dataset, Circle, and TestCircle, then try running TestCircle.

3. A person has a name and a height in inches. Complete the **Person** class definition to represent this information. Person objects should be Measurable, and should work with the TestPerson program. This program creates a data set of Person objects and reports the average height of everyone in the data set, along with the name of the tallest person. (For this exercise, do not edit the code for TestPerson.)

4. The Rectangle class in the java.awt library can be used to represent 2-dimensional rectangular areas, similar to our Circle class. The constructor takes the width and height of the rectangle as integer values, which it stores in the instance variables `this.width` and `this.height` inside the Rectangle object. For example:

   ```
   Rectangle r = new Rectangle(300, 200);
   ```

   Suppose we wish to add Rectangle objects to a Dataset, so that we can compute their average area and determine the biggest rectangle in the set. The problem is that Rectangle objects need to be Measurable, but we cannot change the Rectangle class definition itself, since it is part of the java.awt library. How then can we make Rectangles implement the Measurable interface, so that they too can be used with a Dataset? The trick is to create a new class of our own called, say, MyRectangle, that *inherits* the properties of a Rectangle and also *implements* the Measurable interface. Using this idea, define the class **MyRectangle** and then complete **TestMyRectangle.java** by adding a few more rectangles to the data set and printing their average area along with the area of the biggest rectangle. Your MyRectangle class definition should be declared as follows (don't forget to import the java.awt.Rectangle class as well):

   ```
   public class MyRectangle extends Rectangle implements Measurable {
   ```

5. Examine **PrintCoins.java**, which prints out an unordered array of Coin objects, then compile and run it. The method `Arrays.sort` (from the class java.util.Arrays) can be used to sort any array of objects, as long as the objects implement the **Comparable** interface. You do not need to define this interface yourself (it is part of the core Java language), but you will need to modify the Coin class to implement it, like this:

   ```
   public class Coin implements Measurable, Comparable {
   ```

   You'll also need to add the following method, which is required by the Comparable interface:

   ```
   int compareTo(Object other)
   ```

   This method should return –1, +1, or 0, depending on whether the face value of this Coin is less than, greater than, or equal to (respectively) the face value of *other*, which you can assume will also be a Coin. In order to retrieve the face value of *other*, you will first need to cast it to type Coin. Finally, add the following line to PrintCoins before the call to `System.out.println` and rerun the program:

   ```
   Arrays.sort(coins);
   ```

   Do the coins get printed out in sorted order? What happens if you declare the *other* parameter to be of type Coin instead of Object? What is the cause of the problem?

6. The Dataset class allows you to add as many objects to the data set as you like, as long as they are Measurable. The catch, however, is that the objects themselves (other than the biggest one) are not stored, just the sum of their measures and the total object count. What if we wish to store the objects themselves? We could use a fixed-size array, but we may not know in advance how many objects we will end up adding to the data set, and we might run out of room in the array. Instead, it would be better to use an **ArrayList**, which you can think of as an array that starts out empty and automatically expands as new objects are added one at a time. An ArrayList stores the actual objects themselves, which can be referenced by their zero-based position number *i* by calling the ArrayList's method **`get(i)`**. You can find out how many elements are currently stored in an ArrayList by calling its **`size()`** method. To use the ArrayList class, you need to import it from java.util:

   ```
   import java.util.ArrayList;
   ```

   Add some code to TestPerson that creates a new empty ArrayList and adds several new Person objects to it. ArrayList is a *parameterized* or *generic* Java class, meaning that you must specify the type of objects to be stored. For an ArrayList of Person objects, the type is written **`ArrayList<Person>`**.

   ```
   ArrayList<Person> alist = new ArrayList<Person>();
   ```

   After adding the objects to your ArrayList, use a for-loop to print out the name and height of each person.

7. The ArrayList method **`add(i, x)`** can be used to insert an element *x* at position *i* in the list. For example, you can add *x* to the beginning of the list by calling **`add(0, x)`**. Change TestPerson so that the Person objects are repeatedly added to the *front* of the ArrayList instead of the end. How does this affect the order in which they appear when printed out?

8. We can remove the i^{th} element from an ArrayList using the method **`remove(i)`**. Experiment with this method by removing a few Person objects from your ArrayList and then printing it again to see the effect.

9. A *stack* is a special kind of list that lets you add and remove elements at only one end, traditionally called the *top* of the stack. Think of a stack of cafeteria trays. You can add or remove trays only from the top. Adding an object to the top of the stack is called *pushing* the object onto the stack. Removing and returning the top element is called *popping* the stack. Let's implement a class called **PersonStack** to represent a stack of Person objects, which will use an internal ArrayList<Person> to keep track of the objects. Your PersonStack class should support the following operations:

   - **`size()`** returns the number of objects currently on the stack
   - **`isEmpty()`** returns true or false indicating whether the stack is empty
   - **`push(p)`** adds the Person object *p* to the top of the stack
   - **`top()`** returns the topmost Person object without removing it from the stack. If the stack is empty, a NoSuchElementException should be thrown.
   - **`pop()`** removes and returns the topmost object. If the stack is empty, a NoSuchElementException should be thrown.
   - **`print()`** prints the current contents of the stack in order from top to bottom

   Calling these methods should update the internal ArrayList<Person> object accordingly. Finish the implementation of the PersonStack class and then test it with the TestStack program.