CPE 102
Fall 2016
Lab 8- Comparator

Part A:
Objectives

- To gain experience developing several Comparator classes to use for sorting.
- To gain experience using Java Standard Library sort methods.

Resources

- The Java Standard Library

Orientation

You will be developing a class that represents a list of random **Integer** class objects. The class will support sorting using the appropriate Java Standard Library sort methods found in the **Arrays** or **Collections** classes. The class will support *natural ordering* by using the compareTo method supported by the Integer class and other ordering by implementing three different classes that implement the ***Comparator*** interface. The class will also have a method to determine whether or not the list is sorted in a particular order and will implement the toStringmethod so the contents of the list can be examined easily.

Specification

1. Implement a class called **IntegerList** that supports a specified number of randomly initialized Integer objects as follows:

    a. The class should have the minimally necessary instance variable(s) to support the specified functionality. You should only need an **ArrayList** of **Integer** objects to hold the actual numbers.

    b. Implement a *constructor* that accepts an int-parameter specifying the number of **Integer** objects the list should contain and a second int parameter specifying the maximum value (exclusive) to use when generating random numbers to populate the list with. The constructed **IntegerList** should contain an **ArrayList** that holds specified number of **Integer** objects that have been initialized with random integer values less than the specified max. *Hint: Remember that due to autoboxing (wrapping) you can add primitive ints to an **ArrayList** of **Integers***

    c. Implement a *second constructor* that accepts an int-parameter specifying the number of **Integer** objects the list should contain, a second int parameter specifying the maximum value (exclusive) to use when generating random numbers to populate the list with, and a third int parameter specifying a seed value for the Random number generator. The constructed **IntegerList** should contain an **ArrayList** that holds specified number of **Integer** objects that have been initialized with random integer values less than the specified max.

d. Implement a method called **sort** with no parameters and a return type of void. The method will sort the list of **Integer** objects using the appropriate method from the **Arrays** or **Collections** class.

e. Overload the **sort** method with a second **sort** that has one parameter of type ***Comparator<Integer>*** and a return type of void. The method will sort the list of **Integer** objects according to the specified **Comparator** and using the appropriate method from the **Arrays** or **Collections** class.

f. Implement a method called **isSorted** with no input parameters and a return type of boolean. The method will check to see if the list is in the natural order specified by the compareTo method of the **Integer** class. The method returns true if the list is in the specified order, otherwise false.

g. Overload the **isSorted** method to take a parameter of type ***Comparator<Integer>*** and a return type of boolean. The method will check to see if the list is in the order specified by the by the provided Comparator. The method returns true if the list is in the specified order, otherwise false.

h. Override the **toString** method inherited from **Object** so that it returns a **String** containing each integer value on its own line. If the list is empty the method should return an empty **String**.

i. Implement a **get** method that takes an int value representing an index and returns the **Integer** at that index.

2. Implement a class called **Descending** that implements the ***Comparator<Integer>*** interface so that it results in **Integers** being ordered in descending order.

3. Implement a class called **OddEvenAscending** that implement the ***Comparator<Integer>*** interface so that it results in **Integers** being ordered so that all odds come before all evens and within odds or evens the values are ascending.

4. Implement a class called **OddEvenDescending** that implement the ***Comparator<Integer>*** interface so that it results in **Integers** being ordered so that all odds come before all evens and within odds or evens the values are descending.

Testing

You are given a sample non-junit tester to play with and test your code. Your code **must** compile and run with this tester to receive any credit for the lab.

Submission:

In PolyLearn submit al your java files: IntegerList.java Descending.java OddEvenAscending.java OddEvenDescending.java

Lab 8-Part B

Objectives

- To gain experience writing a sort method using either the Selection or Insertion Sort algorithms.
- To gain experience writing a Binary Search method.

Resources

- The Java Standard Library

Orientation

First finish Part A of Lab 8. Then add the following specification to your working Lab 8.

Specification

1. Add the following to your implementation of **IntegerList** as follows:

    a. Add a public method **mySLOsort** to your **IntegerList** class that takes no parameters and has a return type of void. The **mySLOsort** method *must* sort the **Integer** objects using either the Selection Sort or Insertion Sort algorithm presented in class. The Integers should be sorted according the natural sort order of Integers. *Indicate in a comment which sorting algorithm you are using (Selection sort or Insertion sort).*

    b. Overload the public method **mySLOsort** so that it takes one parameter of type ***Comparator<Integer>*** and has a return type of void.
    The **mySLOsort** method *must* sort the **Integer** objects using either the Selection Sort or Insertion Sort algorithm presented in class. The Integers should be sorted according the **Comparator** passed to the method. *Indicate in a comment which sorting algorithm you are using (Selection sort or Insertion sort).*

    c. Add a public **binarySearch** method that takes an integer key as its *only* input parameter. It returns the integer index where the key is found in the **IntegerList** or -1 if the key is not in the list. You *must* implement this method iteratively. Use a while-loop. *Your method may assume the list is already sorted naturally. Do not call sort from within this method.*

    Require: `this.isSorted() == true (the list is already sorted naturally)`
    Parameters: **`key`** - the number to search for
    Returns: index of the key or -1 if key is not in the list

Testing

You are given a sample non-junit tester to play with and test your code. Your code **must** compile and run with this tester to receive any credit for the lab.

Submission:

```
Submit your java file in PolyLearn: IntegerList.java
```

Note: Your code will be hand checked, as well. You must implement Selection Sort or Insertion Sort to get any credit for the sorting tests. You must implement Binary Search to get any credit for searching tests.