Lab 3 - Sorting and Generics

due September 6th at 23:55

Objective

Today's lab will help you get familiar with

- Working with generics through using ArrayList
- The basics of sorting
- JavaDoc

ArrayList <E> Class

Up to this point you have been using arrays to store data, but these are static and inflexible. The **ArrayList** class is a dynamic array implementation that will grow the array as you add data. Plus it will allow you a variety of options. This class is also a particular type of class called a *generic*, which means that it will generically handle an array of any specified type. You can find the Class's API here.

```
import java.util.ArrayList;

public class ArrayListTest {
  public static void main(String[] args) {
    ArrayListTest r =new ArrayListTest();
    r.run();
  }

  public void run()
  {
    ArrayList<String> array = new ArrayList<String> ();
```

```
            array.add("one");
            array.add("three");
            array.add("two");

            for(int i = 0; i < array.size(); i++) {
              System.out.println(array.get(i));
            }
   }
}
```

In the example above an **ArrayList** is created, three strings are added, and then printed. Note, the size of the arraylist is not set, and this strange syntax ("**<String>**") is used. **ArrayList**s like all the other container structures that we will study this semester are *generic* structures, i.e., they are not restricted to a specific type. We specify the type at the time that we declare the data structure. In the example, we are creating an **ArrayList** that will hold objects of the **String** class.

Note that the type cannot be a primitive since the **ArrayList** stores references and not primitives. Luckily Java provides wrapper classes for primitives which are specific classes that you can use like primitives. More information is here. Therefore, the following are all legal to specify for an **ArrayList**

- **`<String>`**
- **`<Boolean>`**
- **`<Integer>`**
- **`<Double>`**

Commenting

Starting with this lab, we will be stricter about commenting of your code. In particular, each block of code (loop, if-else, etc) should be commented and each method should have a complete Javadoc header. BlueJ generates a Javadoc template for you when you create a new class. You can likely learn the formatting from that. Your header needs to include a description of what the method does, as well as tags and descriptions for parameters and return values. Javadoc will then be able to auto-generate documentation in the style of the API. The API is a good source of examples of how to write the descriptions.

Assignment:

1. Create a class **RandomStringContainer**. This class should have a private **ArrayList** to store data.
    a. Create a constructor for the class to create an empty array list.
    b. Create a public method (*addToFront*) that takes one parameter, a String. The String will be inserted as the first element of the ArrayList, moving all the other elements up by one position. **Hint**: Before you start writing code, read the descriptions of the relevant methods in the API for ArrayList.
    c. Create a public method (*addToBack*) that takes one parameter, a String. The String will be inserted as the last element of the ArrayList.
    d. Create a public method (*addSorted*) that takes one parameter, a String. Assuming the ArrayList is sorted, the String should be inserted in the correct location that would keep the array sorted (you do not need to check if the array is sorted. Just assume it is).
    e. Create a public method (*selectionSort*) that sorts the ArrayList using the selection sort algorithm (see here).
    f. Create a public method (toArray) returning a String array whose entries are equal to the ArrayList. This array should be used to perform your JUnit test using the assertArrayEquals method.

2. Create an **ExperimentController** class. Create a private static String array (*twoLetterWords*) initialized to contain the following words, one per element: aa ab ad ae ag ah ai al am an ar as at aw ax ay ba be bi bo by da de do ed ef eh el em en er es et ew ex fa fe gi go ha he hi hm ho id if in is it jo ka ki la li lo ma me mi mm mo mu my na ne no nu od oe of oh oi ok om on op or os ow ox oy pa pe pi po qi re sh si so ta te ti to uh um un up us ut we wo xi xu ya ye yo za

   The class will have the following methods:

   a. timeAddToFront(int numberOfItems, int seed):
      i. create an instance of a RandomStringContainer
      ii. for the specified *numberOfItems*, insert random words from *twoLetterWords* to the container using the *addToFront()* method.
      iii. The method will return the time taken to add all the items to the container using *addToFront()*.
   b. timeAddToBack(int numberOfItems, int seed):
      i. create an instance of a RandomStringContainer
      ii. for the specified *numberOfItems*, insert random words from *twoLetterWords* to the container using the *addToBack()* method.
      iii. The method will return the time taken to add all the items to the container using *addToBack()*.
   c. timeAddSorted(int numberOfItems, int seed):
      i. create an instance of a RandomStringContainer
      ii. for the specified *numberOfItems*, insert random words from *twoLetterWords* to the container using the *addSorted()* method.
      iii. The method will return the time taken to add all the items to the container using *addSorted()*.
   d. timeSortofUnsortedList(int numberOfItems, int seed):
      i. create an instance of a RandomStringContainer
      ii. for the specified *numberOfItems*, insert random words from *twoLetterWords* to the container using the *addToBack()* method.
      iii. The method will call `selectionSort()`.

        iv. The method will return the time it took to sort the array.
    e. timeSortOfSortedList(int numberOfItems, int seed):
        i. create an instance of a RandomStringContainer
        ii. for the specified *numberOfItems*, insert random words from *twoLetterWords* to the container using the *addToBack()* method.
        iii. The method will call `selectionSort()`.
        iv. The method will call `selectionSort()` again.
        v. The method will return the time it took to sort the already sorted array.

3. Unit test the RandomStringContainer class. Remember to include edge cases. Many methods will require multiple tests.
4. Run your program through ExperimentController so that you test your program for **various sizes of input**. For each amount of data you should run multiple trials with different seeds. You then can create graphs where the y axis signifies the amount of time, and the x axis is the number of elements. More specifically:
    a. Compare the average run time of inserting from the front, back and sorted for different amounts of data.
    b. Compare the average run time of adding the elements and the sorting the unsorted list vs. inserting the entries using the addSorted for different amount of data.
    c. Compare the average run of sorting an unsorted array and sorting a sorted array for different amount of data.

Submission:

In addition to your code you must submit lab notes (in PDF format). Save your notes in the project folder before you compress it and upload it.

Grading:

1. **ExperimentController** design and implementation. (2 pt)

2. **RandomStringContainer** and design and implementation (2 pt)
3. Unit testing (2 pt)
4. commenting+style(2 pt)
5. Notes (2 pt)