Lab 3: Call to Order
CSCI 2101 – Fall 2021

Due (Section A): Monday, October 18, 11:59 pm
Due (Section B): Sunday, October 17, 11:59 pm
Collaboration Policy: Level 1 (review full policy for details)
Group Policy: Pair-optional (you may work in a group of 2 if you wish)

This week's lab will explore sorting, lists, and basic data analysis. In particular, you will build an extension of the `SimpleArrayList` class that supports a new method `sort`[1]. Using this new class, you will write a program to read in Bowdoin College student directory information and sort student data in a few different ways.

Your program will consist of three classes: `Student`, `SortableArrayList`, and `DirectorySort`. The `Student` class will represent a single student's information (as read from a Bowdoin College directory file) and the `DirectorySort` class will simply be a holder for your `main` method and any other helper methods you wish to write for the `main` method. The `SortableArrayList` class will a subclass of `SimpleArrayList` (developed in class) that extends it to support sorting.

1 Sortable Lists

To start, you should copy the `AbstractSimpleList` and `SimpleArrayList` classes from the website into your new project. **You should not make any changes to either of these classes at any point in this lab.** Once these classes are part of your project, your `SortableArrayList` class should extend the existing `SimpleArrayList` class, like so:

```
public class SortableArrayList<T> extends SimpleArrayList<T>
```

Although `SimpleArrayList` already has a defined constructor (two, actually), subclasses do not inherit the constructors of their parent classes. Therefore, subclasses must explicitly define their constructors. Remember that the primary role of a constructor is to initialize the instance variables defined in the class – but since classes also have all the instance variables defined in their parent class(es), constructors are responsible not only for initializing the instance variables defined in the class itself, but also those defined in the parent classes. The way this works is that the child constructor begins by calling a constructor defined in the parent (by calling `super`), which will initialize all instance variables defined in the parent. Afterwards, the child constructor will initialize any extra instance variables defined in the child. Here's an example: suppose that you're extending our two-dimensional `Point` class to provide a three-dimensional point called `Point3D`. You might create a constructor for such a class as follows, in which the X and Y coordinates are defined in the parent `Point` class, but the Z coordinate is defined in the child `Point3D class`:

```
public Point3D(int xVal, int yVal, int zVal) {
    super(xVal, yVal); // call Point constructor
    this.zCoord = zVal;
}
```

[1] You may have noticed that the `List` interface already specifies a `sort` method, but we have disabled it in the regular `SimpleArrayList` class via the `AbstractSimpleList` base class.

Returning to the current program, the existing `SimpleArrayList` class provides two constructors – one which takes no arguments and uses a default starting capacity, and one which takes one argument and uses the specified starting capacity. Your `SortableArrayList` should provide the same two constructors, and should use `super` to call the appropriate constructors in the parent `SimpleArrayList` class.

Next, you should write a `sort` method within `SortableArrayList`, which will reorder the list in ascending order. This method must have exactly the following declaration:

```
public void sort(Comparator<? super T> c)
```

This declaration uses a generic type syntax that we haven't seen before, but basically, all this declaration is saying is that the generic type of the given `Comparator` object must be either `T` or any superclass (parent class) of `T`. For example, if the list is storing `Point3D` objects, then the `sort` method could accept a comparator for `Point3D`, or `Point`, or `Object`, since `Point` and `Object` are both superclasses of `Point3D`. You will need to import `java.util.Comparator` in order to use this method declaration.

1.1 Comparators

The parameter to the `sort` method is of type `Comparator`, which is a pre-defined interface (not a class!) in the standard Java library. The role of the `Comparator` object is to determine how to actually order the objects stored in the list. If the list is storing numbers, then we have an intuitive sense of how to compare them when ordering (e.g., -2 comes before 7, 11 comes before 13, and so forth), and this determines what the final sorted list should look like. However, since the list could potentially store any type of non-numeric object, it may not be obvious how the objects should actually be ordered. For example, if you're storing a list of `Point` objects, how do you decide whether one point "comes before" another? A `Comparator` object addresses this potential ambiguity by specifying exactly how to order two objects of the specified type. Any `Comparator` object is parameterized for some generic type `T` (representing the type of objects that it can compare) and defines the single method `compare`, which compares two objects `a` and `b` (both of type `T`) as follows:

```
/* Returns:  < 0  if a is smaller than b
 *           0    if a equals b
 *           > 0  if a is larger than b
 */
public int compare(T a, T b)
```

The key point is that the definition of "smaller than" or "larger than" with respect to the type `T` is left to the implementation of the `compare` method (inside the `Comparator`) to decide. For example, some comparator for `Strings` might decide that a string is "smaller than" another string if it comes first alphabetically, while a different comparator for `Strings` might decide that a string is smaller if it contains fewer characters. Both are valid ways to produce an ordering, and the specific `Comparator` implementation would specify which one to use. By passing a specific `Comparator` to the `sort` method of `SortableArrayList`, you would then be able to sort the list according to whatever order is defined by that particular `Comparator`.

To actually use a comparator, you will need to to define a new class that implements the built-in `Comparator` interface. Here's an example comparator for integers (i.e., the `Integer` type, since we can't use the primitive `int` as a generic type) that just orders them in the obvious way:

```
public class IntComparator implements Comparator<Integer> {

    @Override
    public int compare(Integer a, Integer b) {
        if (a < b) {
            return -1; // note: magnitude of the result is irrelevant
        } else if (a == b) {
            return 0;
        } else {
            return 1;
        }
    }

}
```

Note that this class is an example of a non-generic class implementing a generic interface: while the interface `Comparator` can be applied to any type `T`, the `IntComparator` class only compares `Integer` objects, and therefore the `IntComparator` class is not generic (and implements `Comparator<Integer>` rather than `Comparator<T>`).

Clever programmers will note that since the magnitude of the return value of `compare` is not significant, the above implementation is actually unnecessarily complicated. This particular `compare` method can be equivalently implemented in just a single line:

```
    return a - b; // returns -, 0, or + if a is less/equal/greater than b
```

Now that the comparator is defined, it can be used to sort a `SortableArrayList` of integers using the `sort` method. To do so, we just need to create an instance of the `IntComparator` class (taking no arguments, since we're just using the default 0-parameter constructor that exists when no explicit constructor is written), and then pass that instance to the `sort` method, like so:

```
    someListOfInts.sort(new IntComparator()); // sort using an IntComparator
```

1.2 Sorting

Before you actually write any comparators, you should implement your `sort` method inside the `SortableArrayList` class. Since this method must be given some comparator object when called, you can implement the `sort` method without having written any `Comparator` classes yet. In your implementation, you will use the given `Comparator` whenever you need to compare two elements by calling `compare`. Your sorting implementation should use selection sort, as covered in class. Here is pseudocode for a selection sort:

```
for each unsorted list index, starting from the end:
    find the largest value in the unsorted part of the list
    swap this value with the rightmost unsorted list element
```

Note that the `sort` method changes the existing list (and is therefore `void`), as opposed to creating and returning a sorted copy of the list. However, since the instance variables of the `SimpleArrayList` class are marked `private`, you cannot actually access them directly from within your `sort` method. You could fix this by changing the instance variables to `protected`, but doing so would require changing the `SimpleArrayList` class (which we prohibited), so you should not do this. Instead, call the public methods defined in `SimpleArrayList` to interact with the list elements. For example, within the `SortableArrayList` class, you could call `this.get(5)` (or, equivalently, just `get(5)`), to get element 5 of the list.

1.3 Testing

Before moving on, you should test your `sort` method. A good way to do this is to define a `main` method inside `SortableArrayList` that just tests the behavior of the `sort` method. In this method, construct a short list of ints and sort it using the `IntComparator` given above, then print it before and after the sort. The base `SimpleArrayList` class already defines a useful `toString` method which `SortableArrayList` inherits, so you don't need to write a new one in `SortableArrayList`.

You will, however, need to write the `IntComparator` class somewhere. One convenient feature you can use is Java's ability to let you define a class within an existing class (these are called "inner classes") rather than creating a separate file for the class. A typical use case is when you have a very short class (such as `IntComparator`), or a bunch of short classes, that don't really merit their own files. You can define the `IntComparator` class **within** `SortableArrayList` (inside the class block but outside any method blocks) just to use with your test code. One small tweak you will need for this to work is declaring your comparator to be `static` – i.e. as below:

```
public static class IntComparator implements Comparator<Integer>
```

You can even make this class declaration `private`, since you don't need to expose it outside the `SortableArrayList` class. Don't worry too much about what `static` means when applied to a class – *all* top-level classes are implicitly static, while non-static inner classes are different in subtle ways from static inner classes. For now, just always declare any inner classes `static`. It will be convenient to use inner classes again later on when you need to write a bunch of comparator classes (and can do so all within the same file if you use inner classes).

2 Directory Sorting

You should now have a `SortableArrayList` class with a functioning `sort` method that accepts a `Comparator` object (along with some test code showing that the sorting code works). Now you're going to write a program that reads a Bowdoin College student directory file, constructs a list of `Student` objects from the file, and then sorts it in a few different ways to answer some questions.

Download the file `directory.txt` from Blackboard, which contains a directory listing of Bowdoin students from 2012. Each line of the file contains information for a single student in the following format:

```
[firstname] [lastname] | [address] | [phone] | [email] | [SU box]
```

For example, a directory line might consist of the following:

```
James Bowdoin | 221 Coles Tower | 123-4567 | jbowdoin@bowdoin.edu | 523
```

You should create a `Student` class (in a separate file) representing a single student from the directory and holding all the directory information for that student. Exactly how you design the `Student` class is up to you, but it will probably need (at least) methods to get the various pieces of directory information about the student.

2.1 Directory Analysis Questions

Finally, you should write the `main` method in your `DirectorySort` class, which should read in the directory file and construct a `SortableArrayList` of all the student objects (i.e., creating a `Student` object from each line and then adding them to your sortable list). After creating the sortable list of students, you should calculate and print out answers to each of the following questions:

(a) Which student has the smallest SU box?

(b) Which student has the largest SU box?

(c) Which student appears first in a printed directory, assuming the directory is ordered by last name?

(d) Which student appears last in a printed directory, assuming the directory is ordered by last name?

(e) Which student has the most vowels in their full name? Vowels should include 'a', 'e', 'i', 'o', and 'u' only (in either upper or lower case).

(f) Which student has the least vowels in their full name?

(g) **(optional extra credit)** Which student has the most occurrences of any single digit in their phone number? E.g., the phone number 155-4351 has three occurrences of the number 5, which is more than 123-4546, which only has two occurrences of the number 4.

To answer each of these questions, you will need to implement a `Comparator` for `Student` objects and use it to sort the directory list. Since the ordering you are working with in each question is different, you will need a different `Comparator` class for each one.

Note that some of these questions may have multiple possible answers (in the event of ties).

The final output of your `main` method should be the answers to each question, one answer per line, in the following form:

```
(num) first last, SU box, phone, email
```

For example:

```
(a) Sarah Bowdoin, 100, 123-4567, sbowdoin@bowdoin.edu
(b) James Bowdoin, 200, 234-5678, jbowdoin@bowdoin.edu
... etc ...
```

3 Implementation Tips

Implementation tips about various parts of the program are given below.

3.1 Reading Files

In the last lab, we had a method to read an entire file into a single `String`, but here it is more appropriate to read the file line-by-line. You can use a `Scanner` to do so as shown below:

```
Scanner scan;
try {
    scan = new Scanner(new File(someFilename));
} catch (Exception e) {
    // failed to read file - good idea to print an error and exit/return
}
while (scan.hasNext()) { // while there's more of the file to read
    String line = scan.nextLine(); // read the next line
    // do something with line
}
scan.close(); // done reading the file, close the Scanner
```

You will need to import `java.io.File` and `java.util.Scanner` to use code like the above.

3.2 Useful String Methods

A large part of the directory processing program will be manipulating `Strings` in various ways (both when reading the student info and also when implementing some of the comparators). There are many useful `String` methods that you can use, but several particular ones that may be helpful (and that you should become familiar with) are listed below. Consult the `String` Javadoc for details on parameters, return values, and so forth of these methods.

- `indexOf` – Get the position of a specified substring inside the string (or -1 if not found).
- `substring` – Get a subsequence of the string going from a starting index up to an ending index (just like slicing a string in Python).
- `split` – Split a string on a specified delimiter (e.g., commas, spaces, etc) and returns an array of the components from doing so. A good way to parse the directory data is to first do a split on vertical bars (‘`|`’). Note, however, that the delimiter parameter is actually a regular expression (which you might or might not have seen before) and the vertical bar character has a special meaning in a regex. Therefore, to do a split on a literal vertical bar, you need to escape it like so:

  ```
  String[] parts = someStr.split("\\|");
  ```

- `trim` – Get a string with all whitespace (e.g., spaces, tabs, newlines) at the beginning and end of the original string removed.

- `toLowerCase` and `toUpperCase` – Get an all lowercase/uppercase copy of the string.
- `charAt` – Get the character at the specified position of the String. A related method is `toCharArray`, which gives you a `char[]` of the string's contents. Note that unlike in Python, you can't directly loop over the individual characters of a `String` in Java using a for-each style loop. If you want to loop over each character, you need to either use a regular for loop in conjunction with `charAt`, or convert to a character array first and then loop over that.
- `Integer.parseInt(s)` – this method can be used to convert strings to integers, like so:

  ```
  int val = Integer.parseInt("123");
  ```

  Note that this method isn't a method defined on `Strings` – it's a static method of the `Integer` class, hence why it is called on `Integer` rather than on a string object itself.
- `compareTo` – Similar to the `compare` method of a `Comparator`, the `compareTo` method allows comparing against another `String` object (and does so alphabetically). This method is actually defined in an interface called `Comparable` (a different interface from `Comparator`), which classes can implement to provide a "built-in" ordering, as opposed to requiring an external `Comparator` object to define the ordering. The `String` class is one such class that implements `Comparable`, and therefore provides a "natural" ordering that is alphabetical.

  You don't need to use `Comparable` in this lab, but it's easy to mix up `Comparator` with `Comparable`, so here's an easy way to remember the difference: a `Comparator` is a standalone object that exists to order some **other** type of object (e.g., `Student` objects), while `Comparable` is an interface that a class can implement so that the class knows how to order itself without needing to use an external `Comparator`. However, a benefit of using `Comparators`, and the reason we use them here, is that they allow us to specify multiple different ways to order the same type of object (by using multiple different `Comparators`).

3.3 String Immutability

One important general principle to note about `Strings` is that they are *immutable* – this means once created, a `String` object can never itself be changed. Therefore, all of the `String` manipulation methods don't actually *change* the called-upon string. Instead, they simply create and return a new string object. A common beginner mistake when working with strings is to write something like the following:

```
String str = "ABC";
str.toLowerCase();
System.out.println(str); // still prints "ABC"
```

The above code is wrong because the second line isn't actually doing anything useful – it's creating a lowercase version of the string (a second string object), but then throwing that lowercase string away rather than assigning it to a variable. Thus, the original string object `str` is still "ABC", not "abc". The correct way to write this code is the following:

```
String str = "ABC";
str = str.toLowerCase();
System.out.println(str); // now prints "abc"
```

Note that this code still isn't actually changing the original `String` object – you're just creating a second object (the lowercase string) and reassigning it to the existing variable name. Reassigning the variable effectively throws away the original string object (which is still uppercase), but that's okay here if we no longer need the uppercase string.

4 Evaluation

Your completed program will be evaluated along the usual three criteria: correctness, design, and style. In self-screening your program before submission, your primary reference for correctness-related issues should be your lab writeup (i.e., this document). For design and style issues, your primary references should be the Coding Design & Style Guide, your own good sense, and any specific guidance provided by your instructor (e.g., feedback provided on past labs). Also make sure that your name (and the name of your partner, if applicable) is included in all of your Java files, and that your project directory is properly named (see below for naming instructions).

Group Evaluations

For groups, only one group member should submit the final program (but make sure that both names are indicated). In addition to your group's single submission, **each group member must individually submit a group report to your instructor over Slack**. Your group report (which will not be shared with your partner) should summarize your contributions to the lab as well as those of your partner. Your report could be as simple as "we both worked on the entirety of the lab together in front of one machine" if that is the case. Remember that the general expectation is that all group members participate fully in most or all parts of the lab (i.e., not "divide up the work"). Your group report does not need to be long (a few lines is fine), but must be received for your lab to be considered submitted. Submit your group reports over Slack; do not include them in your project submission to Blackboard. Group submissions will normally receive a single grade, but we reserve the right to adjust individual grades up or down in the event of clear inequities.

5 Submitting Your Program

Once your program is finished, you should follow the following steps to submit:

1. Save your program and quit your IDE (e.g., BlueJ or Eclipse).

2. **Rename your project folder (which is the folder containing your `.java` files and any associated files) so that it is named `username-lab3` (if working individually) or `username1-username2-lab3` (if working in a group, such as `sbowdoin-jbowdoin-lab3`). Do not include anything else in the folder name!**

3. Create a single, compressed `.zip` archive of your project folder. On a Mac, right-click (or, if you have no right mouse button, control-click) on your project folder and select "Compress your-folder-name" from the menu that appears. On a Windows machine, right-click on the folder, select "Send To," and then select "Compressed (zipped) Folder." In either case, you should now have a `.zip` file that contains your project, named something like `jdoe-lab3.zip` (with your actual username(s)).

4. Open a web browser and go to the course's Blackboard page, then browse to `Lab Submissions`. Click on `Lab 3` and then `Browse Local Files` to locate and attach your `.zip` archive. Don't write any comments in the comment section, as your instructor will not see them. Once you've attached your `.zip` archive, click on `Submit` to complete your submission.

5. **If working in a group, submit your group reports to your instructor by Slack**.

After submitting your lab, remember to save a copy of your project folder somewhere other than on the desktop of the machine you are working on (if you're on a lab machine). If you just leave it on the desktop, it will only be available on that machine – if you log into any other machine on campus, it will not be there. You can also store your projects in Dropbox (or any similar service) or in your folder on the `microwave` server (see the Lab 1 writeup for details on connecting to `microwave`).