Lab 3: Call to Order
CSCI 2101 – Fall 2017

Due: Part 1: Tuesday, Oct 3, 11:59 pm, Part 2: Wednesday, Oct 11, 11:59 pm
Collaboration Policy: Level 1
Group Policy: Part 1: Individual, Part 2: Pair-Optional

This week's two-part lab will explore the fundamentals of algorithm analysis and sorting. In the first part, you will answer some written questions about Big-O analysis. In the second part, you will write a Java class providing a sortable version of the list structure developed in class. You will then use your sortable list structure to process student directory data and sort it in interesting ways. In doing so, you will also gain experience working with `Strings` in Java.

1 Part 1: Algorithm Analysis

Provide written, scanned, or typeset solutions to the following questions.

1. Show that 2^{n+1} is $O(2^n)$ by finding c and n_0 to satisfy the big-O requirement. Explain why your chosen values work.

2. Show that 2^{2n} is *not* $O(2^n)$ by showing that it is not possible to find c and n_0 to satisfy the big-O requirement. Note that $2^{2n} = (2^n)^2$.

3. Give a big-O characterization (and brief justification) of the running time, in terms of n, of each of the following five loops. Think in terms of the number of loop iterations that will be required. Note that the sum of the arithmetic sequence $1, 2, 3, \cdots, k$ is $\frac{k}{2}(1 + k)$.

```
Algorithm Loop1 ($n$):
  $s \leftarrow 0$
  for $i \leftarrow 1$ to $n$ do
    $s \leftarrow s + i$
```

```
Algorithm Loop2 ($n$):
  $p \leftarrow 1$
  for $i \leftarrow 1$ to $2n$ do
    $p \leftarrow p * i$
```

```
Algorithm Loop3 ($n$):
  $p \leftarrow 1$
  for $i \leftarrow 1$ to $n^2$ do
    $p \leftarrow p * i$
```

```
Algorithm Loop4 ($n$):
  $s \leftarrow 0$
  for $i \leftarrow 1$ to $2n$ do
    for $j \leftarrow 1$ to $i$ do
    $s \leftarrow s + i$
```

```
Algorithm Loop5 ($n$):
  $s \leftarrow 0$
  for $i \leftarrow 1$ to $n^2$ do
    for $j \leftarrow 1$ to $i$ do
    $s \leftarrow s + i$
```

4. Explain whether the following statement is true or false:

   "If choosing between an $O(n \log n)$ time algorithm or an $O(n^2)$ time algorithm to solve a given problem, it is always better to use the $O(n \log n)$ algorithm."

   Assume that the two algorithms use equivalent space and that the algorithms are already implemented (so you do not need to worry about the difficulty of implementation, for instance).

Submit your individual answers by the Part 1 deadline.

2 Part 2: Sorting

In this part, you will build an extension of the `SimpleArrayList` class developed in 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: `SortableArrayList`, `Student`, and `DirectorySort`. The `Student` class will represent a single student's information (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. You will also need to copy/paste the `AbstractSimpleList` and `SimpleArrayList` classes into your project in order to use them as a base for the `SortableArrayList` class.

3 Sortable Lists

To start, you should copy-paste the contents of the `AbstractSimpleList` and `SimpleArrayList` classes from the class website into your new BlueJ project. These two classes are available from the class schedule page. **You should not make any changes to either of these classes**.

Once these are part of your project, your `SortableArrayList` class should extend the existing `SimpleArrayList` class, like so:

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

Note that subclasses do not inherit the constructor(s) of their parent classes; constructors must be explicitly defined. 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 is done is that the child constructor will begin by calling a constructor defined in the parent, which will initialize all the instance variables defined in the parent. Afterwards, the child constructor will initialize any extra instance variables defined in the child. Calling the parent constructor is done using the `super` keyword. 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:

```
public Point3D(int xVal, int yVal, int zVal) {
    super(xVal, yVal); // call the Point constructor taking two int arguments
    this.zCoord = zVal; // initialize the zCoord instance var defined in Point3D
}
```

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

[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.

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 we haven't seen before, but basically, all this 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 `String` objects, then the `sort` method could accept a comparator for either `Strings` or `Objects`, since `Object` is a superclass of `String`. You will need to import `java.util.Comparator` in order to use this declaration.

3.1 Comparators

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, and 13 comes after 11, 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. A `Comparator` object addresses this potential ambiguity by specifying exactly how to order two objects of the specified type, using the single method defined in the `Comparator` interface, summarized as:

```
public interface Comparator<T> {
  /* 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 arbitrary type `T` is left to the implementation of the `Comparator` to decide (remember that `Comparator` itself is only an interface, and therefore has no existing implementation). For example, a comparator for `Strings` could decide that a string is "smaller than" another string if it is alphabetically first, or alternately if it has fewer characters. Moreover, you could write multiple different `Comparator` implementations for the same type `T` if you wanted to provide multiple different ways to order them. By passing the desired `Comparator` object to your `sort` method, you would then be able to produce a sorted list of the objects according to whatever ordering is imposed by the given comparator.

To actually use a comparator, you will need to to define a new class that implements the `Comparator` interface, and then pass an instance of that class to the `sort` method. Here's an example comparator for ints (i.e., the `Integer` type) that just orders them in the obvious way:

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

}
```

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:

```
myIntList.sort(new IntComparator());
```

3.2 Sorting

Before you actually write any comparators, you should implement your `sort` method. 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. In your implementation, you will use the given `Comparator` whenever you need to compare two elements. While there are many different sort algorithms you could choose to use, your implementation should use selection sort (discussed 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 since the instance variables of the `SimpleArrayList` class are marked `private`, you cannot actually access them directly from within your `sort` method. Although one approach here would be to change those instance variables to `protected` (which would allow them to be accessed in subclasses such as `SortableArrayList`), you do not need to do so, because you can still use all the public methods declared in `SimpleArrayList` to interact with the list elements (e.g., `this.get(5)`, or equivalently just `get(5)`, to get list element 5).

3.3 Testing

Before moving on, you should test your `sort` method. A good way to do this is to define a `main` method within your `SortableArrayList` class 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`, so you shouldn't need to write a new one in `SortableArrayList`.

One convenient feature you can use here is Java's ability to let you define a class within an existing class (these are called "inner classes"). A typical use case is when you have a very short class (such as the `IntComparator`) that doesn't really merit its own file. You can put this class directly in your `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 `IntComparator` to be `static` – i.e.:

```
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`.

4 Directory Sorting

At this point, you should have a `SortableArrayList` class with a functioning `sort` method that accepts a `Comparator` object. Now you're going to write a program that reads a Bowdoin College student directory file, constructors 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 all 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:

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

Note that some of the fields are missing for some students; they are indicated by '?' characters where the corresponding data would normally be.

You should create your `Student` class representing a single student from the directory and holding all the directory information for that student (this should be in a separate file). Next, 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.

Your program should then calculate and print out answers to each of the following questions. The output for each question should include the complete directory information for the specified student(s) in some reasonable format (e.g., defined by your `toString` method in the `Student` class).

(a) Which student has the smallest SU box? Largest?

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

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

(d) **(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.

Each question can be answered by implementing an appropriate `Comparator` for `Student` objects and using it to sort the directory list. Note that some of these questions may have multiple possible answers (in the event of ties).

5 Implementation Tips

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

5.1 Reading Files

Last week we had a method to read an entire file into a single `String`, but reading line-by-line is much more helpful here. You can use a `Scanner` to read in a file line-by-line. Here is a snippet of code demonstrating how:

```
Scanner scan;
try {
    scan = new Scanner(new File(someFilename));
} catch (Exception e) {
    // failed to read file -- probably print 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 the above.

5.2 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, etc of all 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 is actually a regular expression (might ring a few 1101 bells...) 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.
- `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 a different interface called `Comparable`, which classes can implement to provide a "built-in" ordering, as opposed to requiring an explicit `Comparator` object to do so. The `String` class is one such class that implements `Comparable`, and therefore provides a "natural" ordering that is alphabetical. You don't need to have your `Student` class implement `Comparable`, but you could if you wanted to provide some default ordering for `Student` objects.

5.3 String Immutability

One important general principle to remember 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 brand-new string. A common beginner mistake stemming from forgetting this principle is to write something like the following:

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

The above code is wrong because the second line isn't actually doing anything useful – it's creating and returning a lowercase version of the string (which is being thrown away, since it's not assigned to anything), but the original string that's printed out is still "ABC", not "abc". The correct way to write this code is the following:

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

Note that you're still not 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 (still uppercase) string object – which is fine, but it's important to be aware of what's happening.

6 Evaluation

As usual, your completed program will be graded on correctness, design, and style. Make sure that your program is documented appropriately and is in compliance with the CSCI 2101 Java Style Guide. Lastly, make sure that your name (and the name of your partner, if applicable) is included in all of your Java files.

7 Submitting Your Program

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

1. Save your program and **quit BlueJ** (this is necessary because BlueJ gets confused if you perform step 2 – renaming your project directory – while the project is open).

2. Rename your project folder (which is the folder that contains your `.java` files, `package.bluej`, and possibly a few other files) so that it is named `username-lab3` (with your actual username). For example, I would rename my folder `sbarker-lab3`. For groups, use both usernames separate by a dash (e.g., `sbowdoin-jbowdoin-lab3`).

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 `sbarker-lab3.zip` (with your actual username).

4. Open a web browser and go to the course's Blackboard page, then browse to `Lab Submissions`. Click on `Lab 3` and then `Start New Submission`. In Section 2, you can, but do not need to, provide any comments. Then select `Browse My Computer` and browse to the `.zip` file you created in step 3. Select that file, then click on `Submit`.

5. If working in a group, submit your group report to me by email.

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 Lab 1 writeup for details on connecting to `microwave`).