Lab 00: Recursion Warmup
COSC 290 - Spring ’21

<table>
<thead>
<tr>
<th>Due</th>
<th>Feb 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starter File(s)</td>
<td>Lab00.zip (2 .java files)</td>
</tr>
<tr>
<td>Teaming Up</td>
<td>Pairs are OK</td>
</tr>
<tr>
<td>Submission</td>
<td>We will go over solutions to these problems on Friday, January 29th from 1:00 – 2:00PM</td>
</tr>
</tbody>
</table>

Welcome! Our goals for this initial warmup lab are the following:

- setup and test your Java working environment (JDK and editor)
- refresh your Java syntax knowledge and coding skills
- practice working with recursion, a programmatic design we will be utilizing very often this semester

1 Initial Setup

The steps below will refresh you on compiling and running Java code, while also showing you how you will submit your work. Before you begin, you want to make sure you have the following:

- an installation of the Java runtime and development kit (JDK)
- an editor to help you with writing (and, ideally, also compiling and running) Java code. Either Atom or JEdit are recommended.

Video tutorials for installing the Java runtime/JDK on Mac and Windows, as well as installing and configuring Atom and JEdit are provided on your lab Moodle page

With the above completed, follow the below steps to verify your Java installation:

- download the provided Lab00.zip and extract its contents, the two files: Lab00.java and Lab00Tester.java
- attempt to compile (using the javac command) the provided files. They should compile with no errors.
- attempt to run (using the java command) the provided file. It all goes well, you should see:
  ```
  Exception in thread "main" java.lang.UnsupportedOperationException: implement me!
  at Lab00Tester.main(Lab00Tester.java:11)
  ```
- You will be seeing a lot of the above exception in our future labs – these exceptions will be inserted into provided code to indicate what you need to implement!

2 Overview: Thinking Recursively

As mentioned previously, the goal of the programming portion of this lab is to help you get back in the mindset of writing recursive code. As a refresher, a recursive function is a function which has one or more calls to itself.

Below is a general strategy for designing a recursive solution to a problem:

1. Start with your base case – what sort of argument could be passed to this function where no recursion is necessary? In other words, you know exactly what you need to return without the use of any recursive calls. Remember - sometimes we need multiple base cases!
2. Next, move on to the recursive case(s). Remember, these recursive calls need to eventually stop, otherwise you will have infinite recursion. This means that, whatever argument you pass in to your recursive call should be incrementally working towards your base case(s).

3. When determining how to design your recursive call and what to do with its result, try the black-box strategy! Let’s treat our recursive call as a black box — in other words, assume that your recursive function works perfectly and gives you exactly what you intend. Given the argument you are passing to the recursive call, what will it return? What do you need to do with this result to get what you want overall given the original argument?

For example, imagine I’m writing a function `public int arraySum(int[] arr)` which returns the sum of all ints in an argument array, and I have the following:

- a base case saying if the length `arr` is 0, to return 0.
- a recursive case which recursively calls `arraySum` and passes it the argument `slicedArr`, which is a copy of `arr` with the first index removed.

If we just pretend my recursive case works exactly as I intended, what does it return to me? The sum of all ints in `slicedArr`, which is the sum of all ints in `arr` minus the first index. Thus, to get my overall result, I need to add the first index to the sum returned from my recursive call.

My recursive call would end up looking something like this:

```java
int[] slicedArr = ... // a copy of arr with the first index removed
return arr[0] + arraySum(slicedArr);
```

### 3 Your Task

Your task for this lab is to implement the following functions. Declarations (and, in some cases, some initial code) for each of the below functions are written for you in the provided `Lab00.java`.

To best flex our recursive muscles, your implementations must adhere to the following:

- each function should have at least one recursive call
- don’t create/use any global (i.e. instance or class) variables
- don’t modify or add any additional arguments
- don’t create any additional helper functions

#### 3.1 String Reverse

Write the implementation for the function `stringReverse(String str)` which accepts a `String` argument `str`, and returns the String reversed. A null argument can simply return `null`.

Examples:

- `stringReverse("cat")` should return "tac"
- `stringReverse("")` should return ""
- `stringReverse(null)` should return `null`
3.2 String Mirror

Similar to above, write the implementation for the function `stringMirror(String str)` which accepts a `String` argument `str`, and returns the `String` mirrored, meaning the `String` is repeated backwards. A `null` argument can simply return `null`.

Examples:
- `stringMirror("cat")` should return "catac"
- `stringMirror("Mirror me!")` should return "Mirror me!em rorrriM"
- `stringMirror(""")` should return ""

3.3 Stack Sum

Write the implementation for the function `stackSum(Stack<Integer> s)` which accepts a `Stack` argument `s` containing `Integer`s, and returns the sum of all values in the `Stack`

This argument `Stack` must not be changed upon completion of this function. In other words, its ok to pop things off the argument `Stack`, but you must ultimately put everything back in its original order before the function is done. A `null` argument can simply return zero.

3.4 Queue Reverse

Write the implementation for the function `queueReverse(Queue<E> q)` which accepts a `Queue` argument `q` containing objects of generic type `E`, and reverses the elements in `q` such that the first (front) element of `q` becomes the last (back), the second element becomes the second to last, the last becomes the new front, and so on.

This function modifies the state of the argument `q` directly; remember that objects are passed by reference in Java. A `null` argument can simply return null.

Examples (imagine below queues as being portrayed front to back):
- "cat" => "dog" => "bird" => "frog" would become "frog" => "bird" => "dog" => "cat"
- "cat" => "dog" would become "dog" => "cat"
- a singleton (i.e., size = 1) Queue would be unchanged

3.5 Map Inverter

Write the implementation for the function `mapInverter(HashMap<String, String> hm)` which accepts a `HashMap` argument `hm` where both its keys and values are `Strings`, and inverts it so that the values are now keys and the keys values.

This function modifies the state of the argument `hm` directly. Since keys are unique but values are not for `HashMap`s, if the argument `hm` contains duplicate values across its various keys, this function should throw an `IllegalArgumentException`.

Hint: take a look at the provided helper function `getFirstKey`, it will be useful in your implementation.

Examples:
- `{"cat":"Mittens", "dog":"Brownie", "parrot":"Polly"} should return{"Mittens":"cat", "Brownie":"dog", "Polly":"parrot"}`
• {"Colgate":"Raiders", "Dallas":"Cowboys", "Las Vegas":"Raiders"} should throw an IllegalArgumentException
• {"Buffalo":"Bills", "Bills":"Buffalo"} should return {"Buffalo":"Bills", "Bills":"Buffalo"}

4 Due date

This lab will be due Monday, Feb 1.