CS 61B
Spring 2019

Extra Practice Worksheet
Discussion 6: Java Misc.

Worksheet 6 Solutions

1 Logging and Error Handling

As briefly introduced in the last extra practice worksheet, logging is a common software engineering strategy that helps programmers debug their code through log files. At his internship, Joey writes the code below to test his `writeToLog` method which writes the given argument to some log.

```
public class LogFile {
    public static void printTenth(int[] moneyArray) {
        try {
            writeToLog(a[10]);
            if (a[10] == 11) {
                throw(new Exception("eleven"));
            }
        } catch (IndexOutOfBoundsException e) {
            writeToLog("No tenth item!");
            throw(e);
        } catch (Exception e) {
            writeToLog("The tenth item was an eleven!");
        }
        writeToLog("done");
    }
    public static void main(String[] args) {
        printTenth(new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
        printTenth(new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11});
        printTenth(new int[]{0, 1, 2, 3, 4});
        printTenth(new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
    }
}
```

Solution:

10

done

11

The tenth item was an eleven!

done

No tenth item!

`ArrayIndexOutOfBoundsException`

Notice "done" gets logged unless you preemptively return from the function, for example by throwing an `Exception`.

2 String Comparisons

Suppose you are a freelance writer, and one of your clients asks you to write them a short thought piece about unusual applications of computer science. They have quite generously offered to pay you by the word, with the limitation that you may not exceed 10 pages. Obviously, you would now like to find the shortest words in the English language so that you can use as many of them as possible. To this end, you decide to write a little Java.

```
public interface StringComparator {
     /** Returns a negative number if s1 is 'less than' s2, 0 if 'equal,'
       * and a positive number otherwise. Null is considered less than
       * any String. If both inputs are null, return 0. */
     public int compare(String s1, String s2);
}
```

1. Write a new class `LengthComparator` that implements the `StringComparator` interface provided above. The `LengthComparator` should compare strings based on their lengths.

   Solution:

   ```
   public class LengthComparator implements StringComparator {
       public int compare(String s1, String s2) {
           if ((s1 == null) && (s2 == null)) {
               return 0;
           }
           if (s1 == null) {
               return -1;
           }
           if (s2 == null) {
               return 1;
           }
           return s1.length() - s2.length();
       }
   }
   ```

2. You obtain a String array containing all the words in the English dictionary, and now you want to find every word that is no longer than "computer." Complete the method `lesserStrings`, which takes as arguments an array `arr`, a String `str`, and a `StringComparator sc`, and returns a `List` of all the strings in arr that are 'less than' or 'equal to' `str` according to `sc`.

   Solution:

   ```
   public static List<String> lesserStrings(String[] arr, String str,
           StringComparator sc) {
       ArrayList<String> result = new ArrayList<>();
       for (int i = 0; i < arr.length; i++) {
           if (sc.compare(arr[i], str) <= 0) {
   ```

```
                result.add(arr[i]);
            }
        }
        return result;
    }
```

3. Now suppose your client says that they will also pay you more for using words that occur later in the dictionary. Complete `DoubleComparator` to order strings first by length, and then in reverse alphabetical order. *Hint:* Use the `LengthComparator` class and Java Strings' `compareTo` method.

   Solution:

```
public class DoubleComparator implements StringComparator {
    public int compare(String s1, String s2) {
        LengthComparator lc = new LengthComparator();
        if (lc.compare(s1, s2) == 0) {
            return s2.compareTo(s1);
        } else {
            return lc.compare(s1, s2);
        }
    }
}
```

3 Peeking Iterator

Fill in the code for `PeekingIterator` support the `peek()` operation. The `peek()` returns the element that will be returned by the next call to `next()` without advancing the iterator.

Solution:

```
public class PeekingIterator implements Iterator<Integer> {
    //Add instance variables here.
    Integer next;
    Iterator<Integer> iter;
    boolean noSuchElement;


    public PeekingIterator(Iterator<Integer> iterator){
        iter = iterator;
        advanceIter();
    }
    public Integer peek() {
        return next;
    }
    public boolean hasNext() {
        return !noSuchElement;
```

```
    }
    public Integer next() {
        if (noSuchElement) {
            throw new NoSuchElementException();
        }
        Integer res = next;
        advanceIter();
        return res;
    }
    private void advanceIter() {
        if (iter.hasNext()) {
            next = iter.next();
        } else {
            noSuchElement = true;
        }
    }
}
```