CMPS 12M
Introduction to Data Structures Lab
Lab Assignment 2

The goal of this assignment is to practice using command line arguments, file input-output, and manipulation of Strings in java. File input-output and command line arguments will be necessary to complete some future assignments.

Command Line Arguments
In a Java program, function `main()` reads the command line from which it was called, then stores the tokens on that line in the `args` array. Use the following Java program to create an executable `jar` file called `CommandLineArguments` (see lab1 to learn how to do this)

```
// CommandLineArguments.java
class CommandLineArguments{
   public static void main(String[] args){
      int n = args.length;
      System.out.println("args.length = " + n);
      for(int i=0; i<n; i++) System.out.println(args[i]);
   }
}
```

then do `%CommandLineArguments zero one two three four` and observe the output. Run it with several other sets of tokens on the command line. These tokens are called command line arguments, and can be used within a program to specify and modify the program's behavior. Typically command line arguments will be either strings specifying optional behavior, text to be processed by the program directly, or names of files to be processed in some way.

File Input-Output
The java.util package contains the Scanner class, and the java.io package contains classes PrintWriter and FileWriter. These classes perform simple input and output operations on text files. Their usage is illustrated in the program `FileCopy.java` below, which merely copies one file to another, i.e. it provides essentially the same functionality as the Unix command `cp` (with respect to text files only.)

```
// FileCopy.java
// Illustrates file IO

import java.io.*;
import java.util.Scanner;

class FileCopy{

   public static void main(String[] args) throws IOException{

      Scanner in = null;
      PrintWriter out = null;
      String line = null;
      int n;

      // check number of command line arguments is at least 2
      if(args.length < 2){
         System.out.println("Usage: FileCopy <input file> <output file>");
```

```
            System.exit(1);
        }

        // open files
        in = new Scanner(new File(args[0]));
        out = new PrintWriter(new FileWriter(args[1]));

        // read lines from in, write lines to out
        while( in.hasNextLine() ){
            line = in.nextLine();
            out.println( line );
        }

        // close files
        in.close();
        out.close();
    }
}
```

As you can see, the Scanner constructor takes a File object for initialization, which is itself initialized by a String giving the name of an input file. The Scanner class contains (among others) methods called `hasNextLine()` and `nextLine()`. Read the documentation for Scanner at

http://docs.oracle.com/javase/8/docs/api/

to learn about the proper usage of these methods. The PrintWriter constructor takes a FileWriter object for initialization, which is in turn initialized by a String giving the name of an output file. PrintWriter contains methods `print()` and `println()` that behave identically to the corresponding methods in `System.out`, except that output goes to a file instead of `stdout`. Note that the FileWriter initialization can fail if no file named `args[1]` exists in the current directory. If it fails, it will throw an IOException. This is a checked exception, which cannot be ignored, and therefore function `main()` must either catch the exception, or throw it up the chain of function calls. (In the case of function `main()`, the "calling function" is the operating system). In this example, we deal with this by declaring `main()` to throw an IOException, causing the program to quit if the exception is encountered. Similar comments apply to the initialization of the Scanner object. See the java documentation for more details.

Compile and run `FileCopy.java`, and observe that a `Usage` statement is printed if the user does not provide at least two command line arguments. This `Usage` statement assumes that the program is being run from an executable jar file called `FileCopy`. All of your programs that take command line arguments should include such a usage statement.

String Tokenization

A common task in text processing is to parse a string by deleting the surrounding whitespace characters, keeping just the discrete words or “tokens” that remain. A token is a maximal substring containing no whitespace characters. For instance, consider the preceding sentence to be a string. The 10 tokens in this string are: "A", "token", "is", "a", "maximal", "substring", "containing", "no", "whitespace", "characters.". Whitespace here is defined to mean spaces, newlines, and tab characters. This is one of the first tasks that a compiler for any language such as Java or C must perform. The source file is broken up into tokens, each of which is then classified as: keyword, identifier, punctuation, etc. Java's String class contains a method called `split()` that decomposes a string into tokens, then returns a String array containing the tokens as its elements. Compile and run the following program `FileTokens.java` illustrating these operations.

```
// FileTokens.java
// Illustrates file IO and tokenization of strings.

import java.io.*;
import java.util.Scanner;

class FileTokens{
   public static void main(String[] args) throws IOException{

      Scanner in = null;
      PrintWriter out = null;
      String line = null;
      String[] token = null;
      int i, n, lineNumber = 0;

      // check number of command line arguments is at least 2
      if(args.length < 2){
         System.out.println("Usage: FileCopy <input file> <output file>");
         System.exit(1);
      }

      // open files
      in = new Scanner(new File(args[0]));
      out = new PrintWriter(new FileWriter(args[1]));

      // read lines from in, extract and print tokens from each line
      while( in.hasNextLine() ){
         lineNumber++;

         // trim leading and trailing spaces, then add one trailing space so
         // split works on blank lines
         line = in.nextLine().trim() + " ";

         // split line around white space
         token = line.split("\\s+");

         // print out tokens
         n = token.length;
         out.println("Line " + lineNumber + " contains " + n + " tokens:");
         for(i=0; i<n; i++){
            out.println("  "+token[i]);
         }
      }

      // close files
      in.close();
      out.close();
   }
}
```

What to turn in

Write a java program called `FileReverse.java` that takes two command line arguments giving the names of the input and output files respectively (following the preceding examples). Your program will read each

line of input, parse the tokens, then print each token backwards to the output file on a line by itself. For example given a file called `in` containing the lines:

```
abc defg
hi
jkl mnop q

rstu v
wxyz
```

the command `%FileReverse in out` will create a file called `out` containing the lines:

```
cba
gfed
ih
lkj
ponm
q
utsr
v
zyxw
```

Your program will contain a recursive method called `stringReverse()` with the following signature:

```
public static String stringReverse(String s, int n)
```

This function will return a String that is the reversal of the first *n* characters of `s`. Note that reversing a String recursively is very similar to reversing an array. Study the methods in the String class documented at http://docs.oracle.com/javase/8/docs/api/ to determine how this might be done. See especially the instance methods `charAt()` and `substring()`, as well as the static method `valueOf()`. Base your reversal strategy on one of the recursive methods `reverseArray1()` or `reverseArray2()` discussed in class. Use `stringReverse()` to perform the reversal of tokens from the input file.

Submit the files `README`, `Makefile`, and `FileReverse.java` to the assignment name `lab2`. This is not a difficult assignment, since most of the items mentioned here should be review, but please don't wait until the last minute to start.