CS61B, Fall 2009 **HW #3** **P. N. Hilfinger**

Due: Monday, 21 September 2009

Create a directory to hold your answers. There is a skeleton for your solutions in the repository under `staff/hw3`, and also in the directory `~cs61b/code/hw3`. Use the usual command sequence to copy your final solution to a `hw3-`N entry in your tags repository directory.

1. The standard abstract class `java.io.Reader` is described in the on-line documentation. It is a general interface to "types of object that have a 'read' operation defined on them." The idea is that each time you read from a `Reader`, it gives you the next character (or characters) from some source; just what source depends on what subtype of `Reader` you have. A program defined to take a `Reader` as a parameter doesn't *have* to know what subtype its getting; it just reads from it.

Create a class that extends `Reader`, and provides a new kind of Reader, a `TrReader`, that translates the characters from another `Reader`. That is, a `TrReader`'s source of characters is some other `Reader`, which was given to the `TrReader`'s constructor. The `TrReader`'s read routine simply passes on this other Reader's characters, after first translating them.

```
public class TrReader extends Reader {
    /** A new TrReader that produces the stream of characters produced
     *  by STR, converting all characters that occur in FROM to the
     *  corresponding characters in TO.  That is, change occurrences of
     *  FROM.charAt(0) to TO.charAt(0), etc., leaving other characters
     *  unchanged.  FROM and TO must have the same length. */
    public TrReader (Reader str, String from, String to) {
       // FILL IN
    }

    // FILL IN
}
```

For example, we can define

```
Reader in = new InputStreamReader (System.in);
```

which causes `in` to point to a `Reader` whose source of characters is the standard input (i.e., by default, what you type on your terminal, although you can make it come from a file if desired). This means that

```
while (true) {
   int c = in.read ();
   if (c == -1)
      break;
   System.out.print ((char) c);
}
```

would simply copy the standard input to the standard output.

However, if we write

```
TrReader translation = new TrReader (in, "abcd", "ABCD");
while (true) {
   int c = translation.read ();
   if (c == -1)
      break;
   System.out.print ((char) c);
}
```

then we will copy the standard input to the standard output after first capitalizing all occurrences of the letters a–d.

But if we have defined

```
/** A TrReader that does no translation. */
TrReader noTrans = new TrReader (someReader, "", "");
```

then a call such as `noTrans.read ()` simply has the same effect as `someReader.read ()`.

2. Using the `TrReader` class from problem #1, fill in the following function. You may use any number of '`new`' operations, *one* other (non-recursive) method call, and that's all. In addition to `String`, you are free to use any library classes whose names contain the word `Reader` (check the on-line documentation), but no others. See the template file `Translate.java`. Feel free to include unit tests of your `translate` method.

```
/** The String S, but with all characters that occur in FROM changed
 *  to the corresponding characters in TO. FROM and TO must have the
 *  same length. */
static String translate (String S, String from, String to)
{
    // NOTE: This try {...} catch is a technicality to keep Java happy.
    try {
    // FILL IN
    } catch (IOException e) { return null; }
}
```

3. Fill in the Java classes on the next page to agree with the comments. However, do *not* use any `if`, `switch`, `while`, `for`, `do`, or `try` statements, and do not use the '`?:`' operator. The `WeirdList` class may contain only private fields. The methods in `User` should *not* use recursion. *DO NOT FIGHT THE PROBLEM STATEMENT!* I really meant to impose all the restrictions I did in an effort to direct you into a solution that illustrates object-oriented features. You are going to have to think, but the answers are quite short. See the staff templates in `WeirdList.java` and `IntUnaryFunction.java`.

```
/** An IntUnaryFunction represents a function from
 *  integers to integers. */
public interface IntUnaryFunction {
    /** The result of applying this function to X. */
    int apply (int x);
}

/** A WeirdList holds a sequence of integers. */
public class WeirdList {
    /** The empty sequence of integers. */
    public static WeirdList EMPTY = // FILL IN;

    /** A new WeirdList whose head is HEAD and tail is
     *  TAIL. */
    public WeirdList (int head, WeirdList tail) { /* FILL IN */ }

    /** The number of elements in the sequence that
     *  starts with THIS. */
    public int length () { /* FILL IN */ }

    /** Apply func.apply to every element of THIS WeirdList in
     *  sequence, and return a WeirdList of the resulting values. */
    public WeirdList map (IntUnaryFunction x) { /* FILL IN */ }

    /** Print the contents of THIS WeirdList on the standard output
     *  (on one line, each followed by a blank).  Does not print
     *  an end-of-line. */
    public void print () { /* FILL IN */ }

    // FILL IN WITH *PRIVATE* FIELDS ONLY.
    // You should NOT need any more methods here.
}

// FILL IN OTHER CLASSES HERE (HINT, HINT).

class User {
    /** The result of adding N to each element of L. */
    static WeirdList add (WeirdList L, int n) { /* FILL IN */ }

    /** The sum of the elements in L */
    static int sum (WeirdList L) { /* FILL IN */ }
}
```