CSC 352, Fall 2015
Assignment 5
Due: Wednesday, September 30 at 23:59:59

Introduction

The `a2` write-up started with 3+ pages of various information about assignments. None of that is repeated here, but all of it applies to this assignment, too.

Be sure to make the appropriate a*N* symlink for this assignment.

CMD && *CMD*

When working with Java it's easy to make a change and accidentally run the old version by forgetting to do `javac`. I often do this:

```
% javac x.java && java x
```

That compiles `x.java` and if there are no errors, it runs `java x`. After editing, an up-arrow (or `!javac`) compiles and runs again. I prefer the `&&` over `javac x.java; java x` because with `&&` a compilation error stops things but with a semicolon it always runs `java x`, sometimes rolling the error off the screen.

The Tester runs `javac` first

The Tester always compiles your code before running the tests. There's no need to run `javac` before you run the Tester.

Problem 1. (12 points) `decode.java`

For this problem you are to write a Java program that takes a series of numeric codes as arguments and outputs the character that corresponds to each, according to one of several character coding systems.

One of the character coding systems `decode` supports is ASCII. If `-a` is `decode`'s first argument, it uses the coding specified by the ASCII standard to decode. Example:

```
% java decode -a 72 101 108 108 111 33 LF
Hello!
```

The argument `LF` stands for "linefeed". Simply call `System.out.println()` whenever `"LF"` is encountered in the arguments.

Here's an example with multiple `LF`s:

```
 % java decode -a 51 LF 50 LF 49 46 46 46 108 105 102 116 111 102
102 33 LF   (note: command line wrapped around)
3
2
1...liftoff!
```

The full ASCII character set is supported with `-a`:

```
% java decode -a $(seq 0 127) | od -td1 -w10
0000000    0    1    2    3    4    5    6    7    8    9
0000012   10   11   12   13   14   15   16   17   18   19
0000024   20   21   22   23   24   25   26   27   28   29
0000036   30   31   32   33   34   35   36   37   38   39
0000050   40   41   42   43   44   45   46   47   48   49
0000062   50   51   52   53   54   55   56   57   58   59
0000074   60   61   62   63   64   65   66   67   68   69
0000106   70   71   72   73   74   75   76   77   78   79
0000120   80   81   82   83   84   85   86   87   88   89
0000132   90   91   92   93   94   95   96   97   98   99
0000144  100  101  102  103  104  105  106  107  108  109
0000156  110  111  112  113  114  115  116  117  118  119
0000170  120  121  122  123  124  125  126  127
0000200
```

The `-e` option causes EBCDIC coding to be used, but only a limited set of characters is supported: `a-z`, `0-9`, space (code 64), and two more that are described further below. As you heard in lecture, "EBCDIC" is pronounced "ebb-suh-dik".

Example:

```
% java decode -e $(seq 129 137) LF $(seq 145 153) LF $(seq 162 169)
LF 64 $(seq 240 249) LF   (note: command line wrapped around)
abcdefghi
jklmnopqr
stuvwxyz
 0123456789
```

There's "`man ascii`" but there's no "`man ebcdic`". https://en.wikipedia.org/wiki/EBCDIC is a good place to get started with EBCDIC. First, note that the above example demonstrates that `129` is `'a'` in EBCDIC. On that Wikipedia page, search for `129`. You'll find a cell that shows `a`, `0061`, and `129`. That cell shows that `a`'s code is `129`. That `0061` is the corresponding Unicode character. Try hitting http://www.fileformat.info/info/unicode/char/61/index.htm

The example above also demonstrates that `145` is `'j'`, `163` is `'t'`, and `249` is `'9'`. Search the Wikipedia chart to confirm those values for yourself.

There are two additional characters that `-e` must support. In Unicode they are called "NOT SIGN" (http://www.fileformat.info/info/unicode/char/ac/index.htm) and "PLUS-MINUS SIGN" (http://www.fileformat.info/info/unicode/char/b1/index.htm) Search for `00AC` and `00B1` to find them on the Wikipedia chart. Printing those symbols would raise interesting challenges so we're taking a shortcut with them: instead of printing them directly, `decode` prints a multi-character string to represent each. Here's a demonstration:

```
% java decode -e 95 64 143 LF
<NOT> <+/->
```

You can see that "`<NOT>`" is output for `95` and "`<+/->`" is output for `143`.

Specifying `-r50` causes RADIX-50 coding to be used. RADIX-50 is a system created by the long gone Digital Equipment Corporation (DEC). DEC manufactured the machine I've most loved, the VAX-11/780.

Again, Wikipedia has a good reference: https://en.wikipedia.org/wiki/DEC_Radix-50. Here's a long, tedious invocation that demonstrates the full coding shown on that Wikipedia page:

```
% java decode -r50 $(seq 0 7) LF $(seq 8 15) LF $(seq 16 23) LF
$(seq 24 31) LF $(seq 32 39) LF
 0123456
789ABCDE
FGHIJKLM
NOPQRSTU
VWXYZ.$%
```

Finally, `-h` causes a recently-invented "Hawaiian" coding to be used. This shows the full set of the Hawaiian coding's 14 characters:

```
% java decode -h $(seq 1 12) 14 13 LF
aeiouhklmnpw '
```

Note that `14` is a space.

Error handling

Behavior is undefined if decode's first argument is not `-a`, `-e`, `-r50`, or `-h`. Behavior is undefined if any argument other than `LF` is not an integer in the range of the specified encoding.

Implementation notes

After you figure out what this problem is all about **you might imagine that you're going to be typing a bunch of data, or a bunch of code, but that's surely not my intention**.

First of all, you'll find that `Character.toChars()` makes handling ASCII a snap. You'll see that the documentation for `toChars(int codePoint)` reads, "*Converts the specified character (Unicode code point) to its UTF-16 representation stored in a char array*." The coding of the first 128 Unicode characters is the same as the ASCII coding. Start by trying `Character.toChars(65)` on `www.javarepl.com`. Also look at http://www.fileformat.info/info/unicode/char/41/index.htm and note that there's a `41` in the URL because `65` in decimal is `41` in hexadecimal. Be sure to read Spolsky's article, too, cited in the `required_reading` folder on Piazza.

For the non-ASCII encodings, let me suggest an unusual but interesting and educational approach: specify the coding as a string. For example, I use this Java string literal to implement the Hawaiian coding:

```
" 1=a 2=e 3=i 4=o 5=u 6=h 7=k 8=l 9=m 10=n 11=p 12=w 13=' 14= "
```

Instead of Java code that looks for `1` and produces an `'a'`, or using a `HashMap` that associates the key `7` with the value `'k'`, etc., my solution looks in the string just above to see that the code `12` corresponds to a `'w'`.

I'm intentionally being vague with "looks in the string" because I'd like you to see if you can come up with a solution yourself. Here's a BAD idea: use a for loop to loop through the string one character at time. Here's a not-great idea: split up the string. With those two ruled out, see what comes to mind, and do let us know if you need help. Look for an elegant, simple solution, even though it may seem a bit inefficient. (And remember that programmers are historically pretty bad at guessing what's going to be fast or slow.)

My solution has similarly structured strings for EBCDIC and RADIX-50. You might be thinking, "Ok,

but I've still got to type in those strings." Nope—let the computer do most of the work instead! Here's the first step I took to create the string literal that holds the RADIX-50 coding:

```
% java decode -a $(seq 48 57) $(seq 65 90) 46 36 37 | fold -1 |
      cat -n > x
```

I then used an Emacs "keyboard macro" to massage the lines in `x` into a format like that shown in the Hawaiian coding above. Then I read the contents of `x` into `decode.java`. Yes, I had to get `-a` working first—there's an example of a "bootstrapping" process there: I used a first version of the program to help build a later version.

An Emacs "keyboard macro" is simply a recording of keystrokes can that can be "played back" to repeatedly perform a series of editing operations. Lots of editors have a similar capability.

Here are some links that talk about "macros"/keystroke recording in various editors:

Notepad++: http://www.cathrinewilhelmsen.net/2013/10/27/notepad-macros-a-basic-example/

Sublime: http://docs.sublimetext.info/en/latest/extensibility/macros.html

Vim: http://vim.wikia.com/wiki/Macros

Emacs: http://www.emacswiki.org/emacs/KeyboardMacros#keyboard_macro

There's lots on YouTube, too. If you find something you really like on macros/keystroke recording, post it on Piazza!

With the approach I outline above your code should have few conditionals. My solution has one if/else and two switch statements. This approach can be characterized as a *data-driven* approach.

If you're among the students who have already studied object-oriented design, perhaps in 335 or an agressive 127B, you might consider an object-oriented solution. If you want to pursue that, I'll suggest a starting point:

```
abstract class Decoder
{
    public abstract String decode(String code);
}
```

I'll add a little more detail about an object-oriented approach in `a5/decode-hints-ood`. Remind me if that file doesn't appear promptly.

Problem 2. `questions.txt`

`a5/questions.txt` will soon be a plain text file with a number of free-response questions but as of press time, `a5/questions.txt` is not ready!

When `a5/questions.txt` appears, which will be no later than noon on Friday, 9/25, copy the file into your directory and edit your answers into it, leaving the questions intact. Add your answers after the "`Answer:`" lines. Answers may be multi-line. Per-problem points are specified in the file.

If you're unsure about the format for any of your answers, mail it to `352f15` and we'll take a look!

Problem 3. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple "`Hours:`" lines.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a5/turnin` to submit your work. Each run creates a time-stamped "tar file" in your `a`*`N`* directory with a name like `a`*`N.YYYYMMDD.HHMMSS`*`.tz` You can run `a5/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `352f15` if you need to recover a file—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a5/turnin -l` shows your submissions.

To give you an idea about the size of my solution, here's what I see as of press time:

```
% wc decode.java
  89  272 2480 decode.java
% grep -c \; decode.java
27
```

There are no comments in my code.

Miscellaneous

`a5/tester` is not in place as of press time but will be operational before noon on Friday, 9/25.

This assignment is based on the material in the UNIX slides and C slides 1-64.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point

problem would correspond to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that a student who has only taken CSC 127A and 127B but done well in them and has completed the previous assignments will need 7-9 hours to complete this assignment.

Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help. Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems. If you reach the eight-hour mark, regardless of whether you have specific questions, it's probably time to touch base with us. Give us a chance to speed you up! **Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.**

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, and more. (See the syllabus for the details.)