**COMP 210** **Object-Oriented Programming**

# Project 1

*Assigned: February 28* *Due: March 31*

**Abstract**

Finally you are (hopefully) ready to implement a basic Huffman encoder/decoder!

## 1 Usage

For your first project you are to develop a CLI program that enables basic Huffman encoding and decoding for messages that contain alphabetic characters only. The program works exclusively through files and has three modes of use. The first mode is to compute a Huffman code for a given document and write that code to another file. This option is invoked by the argument `-c`. If the program executable were named huff, then typing

```
huff -c mymessage.txt mycode.txt
```

at the command line would read the message from the file `mymessage.txt`, compute the Huffman code for that document, and write the code to the file `mycode.txt`. The second mode doesn't stop at computing the Huffman code but instead goes on to write the complete message and its code to a file. The command

```
huff -e mymessage.txt compressedmsg.txt
```

would compute the Huffman code for the message found in `mymessage.txt` and then write that code and the compressed message to `compressedmsg.txt`. The final mode of operation decodes a previously encoded message. The command

```
huff -d compressedmsg.txt decodedmsg.txt
```

would read the code and encoded message from `compressedmsg.txt` and then reproduce the original message in `decodedmsg.txt`. These three command formats are summarized in figure 1.

```
huff -c MESSAGEFILE CODEFILE
huff -e MESSAGEFILE ENCODEDMESSAGEFILE
huff -d ENCODEDMESSAGE DECODEDMESSAGEFILE
```

Figure 1: Three Commands for the Huffman Program

### Running Your Program

Once you've written many, many JUnit tests, you may wish to run some manual tests with your program. I'll describe one way you can do this. First use the command line to navigate to your project folder. After you've used Eclipse to build your project, there should be a `bin` folder inside; `cd` into it. Now if your project is inside a package called, say, `proj1`, you can replace `huff` in the above examples with `java proj1.Huff`, where `Huff` is the name of the Java file (and class) that contains `public static main`. For example,

```
java proj1.Huff -d compressedmsg.txt decodedmsg.txt
```

## 2 File Format for the Compressed Message

In real life we'd write the code and compressed text as binary data. If you then tried to open such a file in a text editor you would get gibberish (Why?). This is a bit annoying (and hard for me to grade), so we're going to skip the actual compression part and instead simply write a string of the characters 1 and 0 that would get written to memory if we actually compressed the file. We'll refer to these strings as *binary strings*. The end result will be a larger file since we'll swap one character in the original message for multiple 1s and 0s in the "compressed" message. The smallest possible unit that Java will let you read/write from a file is 1 byte. If you want to attempt the actual compression, then you'd have to combine all the bits, possibly pad it to get an exact multiple of 8, and then break that message up on byte boundaries. You're more then welcome to give this a go, but it is not required. If you're the least bit interested in actual compression programs, this would be a good place to get started.

Writing the code itself to the file should be done by writing the tree. Your goal is to write a linear representation of the tree as a single line of text so that when decoding occurs you can first reconstruct the tree through a linear scan of that line of text. Given that our messages contain only alphabetic characters, we can use comma separated values to write our tree and differentiate leaf nodes from non-leaf nodes by writing a character like * for non-leaf nodes. The end result is a single line of * and letters separated by commas. We've discussed very similar algorithms for doing this with our trees in class: ultimately it boils down to choosing the right traversal pattern and working it through some recursive I/O.

All told, we have the file format for encoded/compressed messages shown in figure 2. This file consists of two lines, the first is the tree as comma separated characters and the second is the binary string that represents the compressed message. It may help to write out some examples and come check them with me.

```
COMMASEPARTEDCHARACTERS
BINARYSTRING
```

Figure 2: Encoded/Compressed Message format

## 3 Hints

Here are some bits and pieces that might be used in an implementation. Note, however, than many designs are possible, meaning that some of these may not apply to your design.

- Although we have talked a lot about Huffman Trees, there are lots of details about Java itself we have not covered. This project will require you to read the Java documentation!
- As in C++, when files are "opened" in Java, they need to be closed. In Java 7+ we have "try-with-resources" blocks, e.g.

```
try (BufferedReader buffered = new BufferedReader(new FileReader(inFilename))) {
  // ... read from "buffered"
   }
```

  This code creates a `BufferedReader` object (see the Java documentation!) out of a filename. At the end of the `try` block, Java makes sure that `buffered` is closed *no matter what*, regardless of any exceptions thrown. `BufferedReader` is not necessarily the best stream class to use here, it's just an example. You'll need something similar for the output.

- Stream object (e.g. `BufferedReader`) methods can throw an `IOException` if something goes wrong. Any functions that use such options must be prepared to catch these exceptions with `try { ... } catch (IOException e){ ... }` or they must be specially marked, e.g.

```
returnType foo(BufferedReader in) throws IOException {
  // ... read from "in"
}
```

  Alternatively, you can catch the exceptions using try/catch blocks around each method that may throw an exception.

- You will need to find the frequencies of each character in a given file. The easiest way to do this is to loop through each character in a message and add to a certain "count." You could keep an array of counts, though I think you'll find this prone to bugs. It would be better to Java's `Map` interface, along with a Map implementation such as `HashMap` (look them up!).

- You'll need to loop through characters in a given file. The common idiom for doing this is just a Google search away. Alternatively, you may find ways to directly read a file and return a `String`. Then you can just loop over the characters in the `String`.

# 4 Logistics

Before you begin the code portion, draw a UML diagram for all the classes your program will require. If you change your design while coding, change the UML diagram accordingly. Submit the source code via *handin* as assignment *proj1* and submit a hard copy of the UML diagram for your code.

Grades are based on the completeness and correctness of the program, the documentation, the tests, style, and the overall quality of the design. Tests should minimally cover the main public facing interface of the program. In the event of an incomplete project, submissions that are more thoroughly tested with jUnit tests can expect to receive better grades than those with minimal testing. At this point you should be following "proper" Java style — classes start with upper-case, local and instance variables with lower-case, all names use camel-case, etc. At this point in your career, if you're still writing code that is inconsistently indented you need to spend some effort fixing this bad habit.

The project is **due no later than class time on Wednesday 3/21**.