CS209 Lab4

Code Breaker - Part 2

You have recognized in the first part of this assignment what is presumably the word "the". To completely decode the secret message, we need a `HashMap<Character,Character>` that will associate to every symbol in the message (the key) the letter it represents (the value). Note that in a HashMap keys are unique (each key only appear once), but values can appear several times. In our case, as its a one-for-one matching, the values will also be unique. So far, we can populate the HashMap with three entries decoding *t*, *h* and *e*.

When you try to decode a message, it helps a lot to know what you are looking for. British decoders during WWII were very much helped by looking for standard formulas such as the Nazi official greeting ("Heil Hitler") and American decoders used similar techniques to break Japanese military codes.

In a message written by a pirate and presumably indicating where a treasure is located, we can expect direction indications (cardinal points, familiar to sailors, as well as words such as "degree"), as well as perhaps the words used on ships to say "left" ("port") and "right" ("starboard"), spelled out numbers and distance units. Not, obviously, meters, but units known by 18th century European seamen: (nautical) miles, yards, fathoms, feet, inches.

In part 2, the goal is to try so set up the conversion HashMap and populate it as much as we can.

After having identified "the", you will try to find the following words (which may or not be present) by looking for patterns that could match:

```
String[] likelyWords = { "seventeen", "starboard", "thirteen",
                "fourteen", "eighteen", "nineteen", "fifteen",
                "sixteen", "through","degree",
                "fathom","branch", "eleven", "twelve",
                "twenty", "thirty", "forty", "fifty", "seven",
                 "eight", "three", "north", "south",
                "right","left","from", "shop", "yard",
                "foot", "feet", "inch", "mile", "east",
                 "west", "port", "four", "five", "nine",
                "one", "two", "six", "ten"};
```

To help you find patterns in the code, you'll find - in the blackboard folder for the practical - a class called `CodeUtil` with two static methods:

- One that takes the secret message and a translation HashMap and returns the message with known symbols replaced by the corresponding letter and dots where the meaning is unknown so far.
- One that takes a word to search and returns it with the letters when the symbol matching the letter is known, and a dot otherwise.

There are several important points to note:

1. The words in array `likelyWords` are ordered by decreasing length. It's important, because a long match is far more likely to be correct than a short match.
2. We are looking for patterns, but nothing says that they are present in the message; and the same pattern might occur several times.
3. Many patterns may be ambiguous. For instance, if you only know the codes for *s* and *t* but not the codes for *e*, *a* or *w*, `..st` could match *east* or *west*. When you find a possible match, you must check if the implied symbol translation:
    a. Doesn't contradict what you know so far.
    b. Allows decrypting other words in the list. The more symbols it allows you to decipher in the message, the greater chance that the translation is correct.

Interface Specifications

Program Call:

As in part 1, the program must take one command-line argument, the name of the secret message file.

If there isn't the right number of arguments, or if the file name is wrong, the program should simply exit with a message (NOT prompt for correct file names).

Output:

The decoded secret message.