Outlining and Stubs

One of the biggest problem-solving anti-patterns for programmers is to just dive into coding without a plan.

One of the most effective ways to plan a program is to judiciously use two techniques: *outlining* and *stubbing*.

Outlining your strategy

The essence of your program is your STRATEGY for CONVERTING the INPUT into the OUTPUT.

If you design that strategy well, you set yourself up to implement the raw code much more straightforwardly. It doesn't matter how solid your JavaScript knowledge is, if you have only a muddled, vague strategy for converting your expected input into your required output, then you'll confuse yourself quickly. You will have to rewrite your code over and over as you struggle to clarify what you're really doing.

Outlining your strategy in plain but precise English makes it easier to evaluate your strategy on its actual merits. Whereas if you go straight to raw code, you run a high risk of immediately getting immersed in syntactical details and not properly evaluating what you're planning to write.

If you keep your plan of attack at a higher level of abstraction, your written program "skeleton" will really help you keep your mind straight about what code to write next, and why.

Note: often people refer to English-language "outlining" of a program as PSEUDOCODE. You can read about pseudocode here . If you hear the term "pseudocode", just think of it as referring to the style of outlining-and-stubbing that you are learning in this article.

Outlining example

To illustrate outlining your program, let's first look at a sub-par example of outlining, then convert it to a good example of outlining.

Given this problem statement:

```
Write a function called "findFirstWordWithMostRepeatedChars" that accepts a string.
As its long but self-descriptive name suggests, it returns the first word in a given
string with the most repeated characters.




"Words" are understood to be space-delimited.
```

So-so example of outlining

Here's a not-very-actionable example of an English outline of a solution to the above problem.

```
// 1) Break up phrase into individual words.

// 2) Break up individual words into individual letters.

// 3) Return the word that has the highest count for a single letter.
```

Why is it only so-so? It's because step (3) is too vague to actually do much with.

If you handed the above outline to another programmer, they would have a hard time understanding what you were asking them to do, exactly.

A better example of outlining

```
  // Break input text into words, splitting on spaces.

  // For each word...

      // find the max repeat count in that word by adding every character into a
hash-bucket

      //  If that max repeat count is higher than the overall max repeat count, then

      //    keep track of the word with the max repeat count
```

Why is this example better? Because it offers two more-specific details:

1. how we find the max repeat count in a single word (by counting characters in an object)
2. the fact that we keep track of the word with the max repeat count as we go

Stubbing

We can do even better, though.

There's a complementary outlining technique called "stubbing". A "stub" function is a function that has been defined in code but its implementation is intentionally missing. It's a kind of placeholder. It helps to sketch out the "skeleton" of intent even further, and more precisely, without committing to a full implementation yet.

Let's illustrate this idea by extending the above outlining example to include a stubbing aspect:

```
function findMaxRepeatCountInWord(word) {

  // Break up individual words into individual letters.

  // Count the instances of each letter

  // Iterate all the counts and find the highest

  // Return this word's max repeat count

}




function findFirstWordWithMostRepeatedChars(text) {

  var maxRepeatCountOverall = 0;

  var wordWithMaxRepeatCount = '';




  // Break up input text into words (space-delimited).

  // For each word...

      var repeatCountForWord = findMaxRepeatCountInWord(word)
      //  If that max repeat count is higher than the overall max repeat count, then

      //    update maxRepeatCountOverall

      //    update wordWithMaxRepeatCount



  return wordWithMaxRepeatCount;

}
```

Things to note about this outline

This outline mixes stubbed-out variables and functions along with English-language commentary about the intended approach.

- A couple of key variables have been written directly as clearly-named code, rather than describing them in English.
- Similarly, some clearly-named functions have been written as code, not English.
- Each function declares its parameters. The parameters themselves have clear names too, just like the other variables.
- Each function has some English description of how that function processes its input into output. This isn't exactly pseudo-code, it's more of a bullet-point outline of intent.
- The return value of each function is explicitly declared.

This style of mixed stubs and English SAVES TIME. This style succinctly conveys your intended approach and minimizes the amount of re-typing you will have to do to convert your outline into working code.

That efficiency gain is especially important when you are performing under the time constraints of a live interview/admissions assessment.