You want to build a word cloud, an infographic where the size of a word corresponds to how often it appears in the body of text.

To do this, you'll need data. Write code that takes a long string and builds its word cloud data in a map, where the keys are words and the values are the number of times the words occurred.

> We'll use a JavaScript `Map` instead of an object because it's more explicit—we're mapping words to counts. And it'll be easier and cleaner when we want to iterate over our data.

Think about capitalized words. For example, look at these sentences:

```
"After beating the eggs, Dana read the next step:"
"Add milk and eggs, then add flour and sugar."
```

What do we want to do with "After", "Dana", and "add"? In this example, your final map should include *one* "Add" or "add" with a value of 2. Make *reasonable* (not necessarily *perfect*) decisions about cases like "After" and "Dana".

Assume the input will only contain words and standard punctuation.

> You could make a reasonable argument to use **regex** in your solution. We won't, mainly because performance is difficult to measure and can get pretty bad (http://blog.codinghorror.com/regex-performance/).

Gotchas

Are you sure your code handles hyphenated words and standard punctuation?

Are you sure your code **reasonably handles the same word with different capitalization?**

Try these sentences:

```
"We came, we saw, we conquered...then we ate Bill's (Mille-Feuille) cake."
"The bill came to five dollars."
```

We can do this in $O(n)$ runtime and space.

The final map we return should be the ***only*** data structure whose length is tied to n.

We should only iterate through our input string **once**.

Breakdown

We'll have to go through the entire input string, and we're returning a map with every unique word. In the worst case every word is different, so our runtime and space cost will both be at least $O(n)$.

This challenge has several parts. Let's break them down.

1. **Splitting the words** from the input string
2. **Populating the map** with each word
3. **Handling words that are both uppercase and lowercase** in the input string

How would you start the first part?

We could use a built-in `split()` function to separate our words, but if we just split on spaces we'd have to iterate over all the words before or after splitting to clean up the punctuation. And consider em dashes or ellipses, which *aren't* surrounded by spaces but nonetheless separate words. Instead, we'll make our *own* `split()` function, which will let us iterate over the input string only once.

How can we check if a character in our input string is a letter?

Two good options are to build a helper function or to use regular expressions. Either will work for this problem. We'll build our own helper function `isLetter()`:

```javascript
function isLetter(character) {
    return 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'.indexOf(character) >= 0;
}
```

Now how can we split each word? Let's assume, for now, that our helper function will return an array of words.

We'll iterate over all the characters in the input string. How can we identify when we've reached the end of a word?

We can store `currentWord` in a variable, and append its value to the array every time we hit a space.

Here's a simple example. It doesn't work perfectly yet—you'll need to add code to handle the end of the input string, hyphenated words, punctuation, and edge cases.

```javascript
function splitWords(inputString) {
    var words = [];
    var currentWord = '';
    for (var i = 0; i < inputString.length; i++) {
        var character = inputString[i];
        if (character === ' ') {
            words.push(currentWord);
            currentWord = '';
        } else if (isLetter(character)) {
            currentWord += character;
        }
    }
    return words;
}
```

Now we've solved the first part of the challenge, splitting the words. The next part is **populating our map with unique words.** What do we do with each word?

If the word is in the map, we'll increment its count. Otherwise, we'll add it to the map with a count of 1.

```javascript
var wordsToCounts = new Map();

function addWordToHashMap(word) {
    if (wordsToCounts.has(word)) {
        wordsToCounts.set(word, wordsToCounts.get(word) + 1);
    } else {
        wordsToCounts.set(word, 1);
    }
}
```

Alright, last part! **How should we handle words that are uppercase and lowercase?**

Consider these sentences:

```
"We came, we saw, we ate cake."
"Friends, Romans, countrymen! Let us eat cake."
"New tourists in New York often wait in long lines for cronuts."
```

Take some time to think of possible approaches. What are some other sentences you might run into. What are all your options?

When are words that *should be* lowercase *not*?

Here are a few options:

1. Only make a word uppercase in our map if it is *always* uppercase in the original string.
2. Make a word uppercase in our map if it is *ever* uppercase in the original string.
3. Make a word uppercase in our map if it is ever uppercase in the original string *in a position that is not the first word of a sentence.*
4. Use an API or other tool that identifies proper nouns.
5. Ignore case entirely and make every word lowercase.

What are the pros and cons for each one?

Pros and cons include:

1. **Only make a word uppercase in our map if it is *always* uppercase in the original string:** this will have reasonable accuracy in very long strings where words are more

likely to be included multiple times, but words that *only* ever occur as the first word in a sentence will always be included as uppercase.

2. **Make a word uppercase in our map if it is *ever* uppercase in the original string:** this will ensure proper nouns are *always* uppercase, but any words that are *ever* at the start of sentences will always be uppercase too.
3. **Make a word uppercase in our map if it is ever uppercase in the original string *in a position that is not the first word of a sentence:*** this addresses the problem with option (2), but proper nouns that are *only* ever at the start of sentences will be made lowercase.
4. **Use an API or other tool that identifies proper nouns:** this has a lot of potential to give us a high level of accuracy, but we'll give up control over decisions, we'll be relying on code we didn't write, and our practical runtime may be significantly increased.
5. **Ignore case entirely and make every word lowercase:** this will give us simplicity and consistency, but we'll lose all accuracy for words that should be uppercase.

Any of these could be considered reasonable. Importantly, **none of them are perfect**. They all have tradeoffs, and it is very difficult to write a highly accurate algorithm. Consider "cliff" and "bill" in these sentences:

```
"Cliff finished his cake and paid the bill."
"Bill finished his cake at the edge of the cliff."
```

You can choose whichever of the options you'd like, or another option you thought of. For this breakdown, we're going to choose option (1).

Now, how do we update our `addWordToMap()` function to avoid duplicate words?

Think about the different possibilities:

1. The word is **uppercase or lowercase.**
2. The word is **already in the map** or not.
3. **A different case of the word is already in the map** or not.

Moving forward, we can either:

1. Check for words that are in the map in **both** cases *when we're done populating the map*. If we add "Vanilla" three times and "vanilla" eight times, we'll combine them into *one* "vanilla" at the end with a value 11.

2. *Avoid **ever** having a word in our map that's both uppercase and lowercase.* As we add "Vanilla"s and "vanilla"s, we'd *always only ever have one version* in our map.

We'll choose the second method since it will save us a walk through our map. How should we start?

If the word we're adding is already in the map in its current case, let's increment its count. What if it's *not* in the map?

There are three possibilities:

1. **A lowercase version is in the map** (in which case we *know* our input word is uppercase, because if it is lowercase and already in the map it would have passed our first check and we'd have just incremented its count)
2. **An uppercase version is in the map** (so we *know* our input word is lowercase)
3. **The word is not in the map** in any case

Let's start with the first possibility. What do we want to do?

Because we only include a word as uppercase if it is always uppercase, we simply increment the lowercase version's count.

JavaScript

```
// current map
// {"blue" => 3}

// adding
// "Blue"

// code
var lowerCaseCount = wordsToCounts.get(word.toLowerCase());
wordsToCounts.set(word.toLowerCase(), lowerCaseCount + 1);

// updated map
// {"blue" => 4}
```

What about the second possibility?

This is a little more complicated. We need to remove the uppercase version from our map if we encounter a lowercase version. **But we still need the uppercase version's count!**

```JavaScript
// current map
// {"Yellow" => 6}

// adding
// "yellow"

// code (we will write our "capitalize()" function later)
var capitalizedCount = wordsToCounts.get(capitalize(word));
wordsToCounts.set(word, capitalizedCount + 1);
wordsToCounts.delete(capitalize(word));

// updated map
// {"yellow" => 7}
```

Finally, what if the word is not in the map at all?

Easy—we add it and give it a count of 1.

```JavaScript
// current map
// {"purple" => 2}

// adding
// "indigo"

// code
wordsToCounts.set(word, 1);

// updated map
// {"purple" => 2, "indigo" => 1}
```

Now we have all our pieces! We can split words, add them to a map, and track the number of times each word occurs without having duplicate words of the same case. Can we improve our solution?

Let's look at our runtime and space cost. We iterate through every character in the input string once and then every word in our array once. That's a runtime of $O(n)$, which is the best we can achieve for this challenge (we *have* to look at the entire input string). The space we're using

includes an array for each word and a map for every unique word. Our worst case is that every word is different, so our space cost is also $O(n)$, which is also the best we can achieve for this challenge (we *have* to return a map of words).

But we can still make some optimizations!

How can we make our *space cost* even smaller?

We're storing all our split words in a separate array . That at least doubles the memory we use! How can we eliminate the need for that array ?

Right now, we store each word in our array *as we split them*. Instead, let's just immediately populate each word in our map!

Solution

In our solution, we made four decisions:

1. **We used a class.** This allowed us to tie our functions together, calling them on instances of our class instead of passing references.
2. For our method of avoiding duplicate words with different cases, **we chose to make a word uppercase in our map only if it is *always* uppercase in the original string**. While this is a reasonable approach, it is *imperfect* (consider proper nouns that are also lowercase words, like "Bill" and "bill").
3. **We built our own `split()` function** instead of using a built-in one. This allowed us to pass each word to our `addWordToMap()` function *as it was split*, and to split words and eliminate punctuation in *one* iteration.
4. **We made our own `isLetter()` function** instead of using regular expressions. Either approach would work for this challenge.

To split the words in the input string and populate a map of the unique words to the number of times they occurred, we:

1. **Split words** by spaces, em dashes, and ellipses—making sure to include hyphens surrounded by characters. We also include all apostrophes (which will handle contractions nicely but will break possessives into separate words).

2. **Populate the words in our map** as they are identified, checking if the word is already in our map in its current case or another case.

If the input word is *uppercase* and *there's a lowercase version in the map*, we increment the lowercase version's count. If the input word is *lowercase* and *there's an uppercase version in the map*, we "demote" the uppercase version by adding the lowercase version and giving it the uppercase version's count.

JavaScript

```
function WordCloudData(inputString) {

    this.wordsToCounts = new Map();
    this.populateWordsToCounts(inputString);
}

WordCloudData.prototype.populateWordsToCounts = function(inputString) {
    // iterates over each character in the input string, splitting
    // words and passing them to this.addWordToMap()

    var currentWord = '';
    for (var i = 0; i < inputString.length; i++) {
        var character = inputString.charAt(i);

        // if we reached the end of the string we check if the last
        // character is a letter and add the last word to our map
        if (i === inputString.length - 1) {
            if (this.isLetter(character)) currentWord += character;
            if (currentWord.length) this.addWordToMap(currentWord);

        // if we reach a space or emdash we know we're at the end of a word
        // so we add it to our map and reset our current word
        } else if (character === ' ' || character === '\u2014') {
            if (currentWord.length) this.addWordToMap(currentWord);
            currentWord = '';

        // we want to make sure we split on ellipses so if we get two periods in
        // a row we add the current word to our map and reset our current word
        } else if (character === '.') {
            if (i < inputString.length - 1 && inputString.charAt(i + 1) === '.') {
                if (currentWord.length) this.addWordToMap(currentWord);
                currentWord = '';
            }

        // if the character is a letter or an apostrophe, we add it to our current word
        } else if (this.isLetter(character) || character === '\'') {
            currentWord += character;

        // if the character is a hyphen, we want to check if it's surrounded by letters
        // if it is, we add it to our current word
```

```
        } else if (character === '-') {
            if (i > 0 && this.isLetter(inputString.charAt(i-1)) &&
                    this.isLetter(inputString.charAt(i+1))) {
                currentWord += character;
            }
        }
    }
};

WordCloudData.prototype.addWordToMap = function(word) {

    var newCount;

    // if the word is already in the map we increment its count
    if (this.wordsToCounts.has(word)) {
        newCount = this.wordsToCounts.get(word) + 1;
        this.wordsToCounts.set(word, newCount);

    // if a lowercase version is in the map, we know our input word must be uppercase
    // but we only include uppercase words if they're always uppercase
    // so we just increment the lowercase version's count
    } else if (this.wordsToCounts.has(word.toLowerCase())) {
        newCount = this.wordsToCounts.get(word.toLowerCase()) + 1;
        this.wordsToCounts.set(word.toLowerCase(), newCount);

    // if an uppercase version is in the map, we know our input word must be lowercase.
    // since we only include uppercase words if they're always uppercase, we add the
    // lowercase version and give it the uppercase version's count
    } else if (this.wordsToCounts.has(this.capitalize(word))) {
        newCount = this.wordsToCounts.get(this.capitalize(word)) + 1;
        this.wordsToCounts.set(word, newCount);
        this.wordsToCounts.delete(this.capitalize(word));

    // otherwise, the word is not in the map at all, lowercase or uppercase
    // so we add it to the map
    } else {
        this.wordsToCounts.set(word, 1);
    }
};

WordCloudData.prototype.capitalize = function(word) {
```

```
    return word.charAt(0).toUpperCase() + word.slice(1);
};

WordCloudData.prototype.isLetter = function(character) {
    return 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'.indexOf(character) >= 0;
};
```

Complexity

Runtime and memory cost are both $O(n)$. This is the best we can do because we have to look at *every* character in the input string and we have to return a map of *every* unique word. We optimized to only make *one* pass over our input and have only *one* $O(n)$ data structure.

Bonus

1. We haven't explicitly talked about how to handle more complicated character sets. How would you make your solution work with more unicode characters? What changes need to be made to handle silly sentences like these:

   I'm singing ♬ on a   day.

      +   =   .

2. We limited our input to letters, hyphenated words and punctuation. How would you expand your functionality to include numbers, email addresses, twitter handles, etc.?
3. How would you add functionality to identify phrases or words that belong together but aren't hyphenated? ("Fire truck" or "Interview Cake")
4. How could you improve your capitalization algorithm?
5. How would you avoid having duplicate words that are just plural or singular possessives?

What We Learned

To handle capitalized words, there were lots of heuristics and approaches we could have used, each with their own strengths and weaknesses. Open-ended questions like this can really separate good engineers from great engineers.

Good engineers will come up with *a solution*, but great engineers will come up with *several solutions*, weigh them carefully, and choose the best solution for the given context. So as you're running practice questions, challenge yourself to keep thinking even after you have a first solution. See how many solutions you can come up with. This will grow your ability to quickly see multiple ways to solve a problem, so you can figure out the *best* solution. And use the hints and gotchas on each Interview Cake question—they're designed to help you cultivate this skill.
