Lesson Tuesday

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / Separation of Concerns in Text Analyzer: boldPassage() UI Function

Text Cheat sheet

We now have a very basic application where our business and user interface logic are completely separate. Our business logic is tested with pseudocode tests. Our user interface doesn't include any business logic. Instead, whenever the UI logic needs to interact with business logic, it calls one of the functions in our business logic.

However, it's not always easy to tell if logic should be UI or business logic. So let's muddy the water a bit and look at an example where separation of logic may not be entirely clear.

UI Logic, Business Logic, and Separation of Concerns with `boldPassage()`

We're going to add another function to our application called `boldPassage()`. This function will return a string with bolded word matches. For instance, let's say we want to find matches of the word "world" in the string "Hello world! It truly is a wonderful day — and a wonderful world!"

Our string will look like this in the UI:

"Hello **world**! It truly is a wonderful day — and a wonderful **world**!"

First off, is this business logic or user interface logic?

The answer should be pretty obvious. It returns a formatted string to the user, therefore it's user interface logic. But just because it's user interface logic doesn't mean we should just throw the new code into the event listener for the `window`'s load event. We can still extract user interface logic into functions which will make our code much cleaner. Also, just because a function should be user interface logic doesn't mean it needs to directly alter the DOM.

This is very important to reiterate because it would be really easy to just update our `numberOfOccurrencesInText()` to do this with the bare minimum amount of code. Here's what it might look like. **We do not want to actually do this. It is just an example.**

```
// Do not do this. It works but it does not separate logic!

function numberOfOccurrencesInText(word, text) {
  if (word.trim().length === 0) {
    return 0;
  }
  let para = document.createElement("p");
  const textArray = text.split(" ");
  let wordCount = 0;
  textArray.forEach(function(element) {
    if (element.toLowerCase().includes(word.toLowerCase())) {
      wordCount++;
      const bold = document.createElement("strong");
      bold.append(element);
      para.append(bold, " ");
    } else {
      para.append(element, " ");
    }
  });
  document.querySelector("div#bolded-passage").append(para);
  return wordCount;
}
```

In this example, we create a `para` variable that's set to a new paragraph element. Then when we loop through the `textArray`, we don't just count the number of occurrences of a specific word. We also create a new `<strong>` element and add the matching word as its text. On every iteration of our loop, we add to the paragraph, appending words (represented by `element`) and a space between each word. Finally, when the loop ends, we add the paragraph element to the end of the `<div>` that contains all of our HTML.

Again, it should be obvious why this is bad. This function is just supposed to do one thing — count the number of occurrences of a word in a text. Sure, it may seem convenient to use the loop and conditional we already have to construct formatted HTML elements for the DOM. And it may seem particularly tempting because we can potentially solve the problem with *less* code if we do this. But if we do so, we won't be able to reuse the function elsewhere. If the HTML changes, the function will break. It's also harder to read and reason about what this function does — and it's just plain ugly.

And no, you can't escape the problem by returning the HTML paragraph element (with the bolded matching words) with the word count, like this (returning an array with both values): `return [wordCount, para];`. While our function would no longer directly interact with the DOM, it would still be doing formatting work intended for the user interface.

In other words, we're combining two separate functionality goals (counting matching words and formatting text) into one function, which isn't great. This interferes with a very important programming design pattern: separation of concerns.

Separation of concerns dictates that each function should only focus on one thing and not know about anything else in the application. In this context, a **concern** is a responsibility. So when we apply separation of concerns to our code, we're separating the functionality of our webpage into multiple different functions, each of which has a single responsibility.

In the case of `numberOfOccurrencesInText()`, it should take two strings, determine how many times a specific substring occurs in that string, and then return the count. It shouldn't do anything else. This means that formatting text is not the 'concern' of `numberOfOccurrencesInText()`.

Adding the `boldPassage()` UI Function

So let's do this the right way. We'll create a function that handles the UI logic for bolding a passage. **We can even use TDD to test it, though keep in mind that TDD is generally used for business logic, *not* UI logic. (There are other ways to test UI logic such as end to end tests.) You will not be expected to write tests for your UI logic on your independent project.** However, we are going to walk through the process of using TDD for this function because it can still help us break down the problem into smaller parts and it will help us separate our code better.

The First Test

Let's start with a test:

```
Describe: boldPassage()

Test: "It should return null if no word or text is entered."
Code:
const text = "";
const word = "";
boldPassage(word, text);
Expected Output: null
```

We're starting out by handling the case when no word or text is inputted. We don't have to start here, but it's as good as any starting place because it's small and specific. When using TDD, it is common that you'll find multiple good starting places or next steps. So don't get hung up on choosing the right starting place or next step. Instead, make sure that whatever functionality you choose to create next, that it's small and specific.

So, with this test, we state that if either `word` or `text` is an empty string, then we'll simply return `null`. Why `null`? Well, we could choose any falsey value here. In our form submission event handler function, we'll use this return value to determine whether or not we should update the DOM. If there's no paragraph element, we don't want to update the DOM. We'll add code for this when we complete the UI logic in the next lesson.

Now let's get the test passing:

js/scripts.js

```
function boldPassage(word, text) {
  if ((text.trim().length === 0) || (word.trim().length === 0)) {
    return null;
  }
}
```

The Second Test

Next, let's write the second test:

```
Test: "It should return a non-matching word in a p tag."
Code:
const word = "hello";
const text = "yo";
boldPassage(word, text);
Expected Output: <p>yo</p>
```

Our function will just return a paragraph element with the text inside of it. No need to interact with the DOM at all! We'll keep it very simple. Both parameters are one word and the strings don't match.

Now let's get the test passing:

js/scripts.js

```
function boldPassage(word, text) {
  if ((text.trim().length === 0) || (word.trim().length === 0)) {
    return null;
  }
  const p = document.createElement("p");
  p.append(text);
  return p;
}
```

As we can see, if the word is not a match, it shouldn't be bolded. Therefore we should just return the paragraph element with the text appended inside. The test will pass.

The Third Test

Onto the third test!

```
Test: "It should return a matching word in a strong tag."
Code:
const word = "hello";
const text = "hello";
boldPassage(word, text);
Expected Output: <p><strong>hello</strong></p>
```

Now the one-word strings arguments match, which means they should be bolded. Here's the updated code:

js/scripts.js

```
function boldPassage(word, text) {
  if ((text.trim().length === 0) || (word.trim().length === 0)) {
    return null;
  }
  const p = document.createElement("p");
  if (word === text) {
    const bold = document.createElement("strong");
    bold.append(text);
    p.append(bold);
  } else {
    p.append(text);
  }
  return p;
}
```

This will get our test passing and will still be compliant with our first test.

The Fourth Test

Now let's move onto the fourth test:

```
Test: "It should wrap words that match in strong tags but not words
that don't."
Code:
const word = "hello";
const text = "hello there";
boldPassage(word, text);
Expected Output: <p><strong>hello</strong> there</p>
```

Now let's update our code to get the test passing. We are ready for a loop now. The function will change considerably but the conditional we've already written will still play an instrumental part in our code.

js/scripts.js

```
function boldPassage(word, text) {
  if ((text.trim().length === 0) || (word.trim().length === 0)) {
    return null;
  }
  const p = document.createElement("p");
  let textArray = text.split(" ");
  textArray.forEach(function(element) {
    if (word === element) {
      const bold = document.createElement("strong");
      bold.append(element);
      p.append(bold);
    } else {
      p.append(element);
    }
    p.append(" ");
  });
  return p;
}
```

There is still one more thing we need to fix to get this test passing, but first let's go over the changes we've made.

First, we create an empty paragraph element. We will append words to this string with our loop.

We take our split array and then loop through it. If the `word` and the `element` match, we should bold the word and append it to the paragraph element. That means we do the following:

```
if (word === element) {
  const bold = document.createElement("strong");
  bold.append(element);
  p.append(bold);
}
```

If the `word` and the `element` don't match, we just append the `element` without strong tags:

```
else {
  p.append(element);
}
```

Then, after we've appended the `element`, we add a space with `p.append(" ");` regardless of whether the element was a match. This is because when we split our string based on spaces, we also removed those spaces. So now we need to add them back in.

At the very end of the function and after the loop has completed, we return the `p` paragraph element.

Fixing a Small Bug by Using the `index` Parameter in `Array.prototype.forEach()` Callback

If we try this test out in the DevTools console, we will see that something is not quite right:

```
> function boldPassage(word, text) {
  if ((text.trim().length === 0) || (word.trim().length === 0)) {
    return null;
  }
  const p = document.createElement("p");
  let textArray = text.split(" ");
  textArray.forEach(function(element) {
    if (word === element) {
      const bold = document.createElement("strong");
      bold.append(element);
      p.append(bold);
    } else {
      p.append(element);
    }
    p.append(" ");
  });
  return p;
}
> const p = boldPassage("hello", "hello there");
> p.innerText;
"hello there "
```

Our method adds an extra space at the end. What's the problem with this whitespace? You may or may not remember, but HTML actually ignores the whitespace in our HTML. So this whitespace at the end of `"hello there "` doesn't affect our HTML at all.

However, the DOM — the object model of our HTML — does track this whitespace as a separate node in the DOM's tree. Let's do a quick review. A tree is a hierarchical data structure in computer programming, and that's what the DOM is — a tree. Trees are made up of nodes that are organized hierarchically. There's a parent node that has children nodes, which may have sibling nodes, and so forth. This is very similar to a family tree.

All that aside, the main issue is that the DOM tracks this whitespace in order to make sure that it is presenting our code as it is actually written. If you want to dive a bit deeper into this whitespace issue, check out this article on MDN that explains it (https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Whitespace#html_largely_ignores_whitespace (We'll also be reviewing the DOM when we learn about the `Node` object to explore DOM nodes.)

So, ideally, we'll make sure that we're only including necessary whitespace in the HTML we add to the DOM. How can we fix this?

Well, this is a great opportunity to cover an important feature of `Array.prototype.forEach()` that we haven't discussed in detail yet: we can add an optional parameter to the callback we pass into the forEach loop, like this:

```
> const string = "I like cats!";
> string.split(" ").forEach(function(element, index) {
  console.log(element, index);
});
```

Try the above code in the DevTools console and the following will be logged:

```
I 0
like 1
cats! 2
```

So we can get both the element *and* the current index position of the element in the array. This is a very useful piece of information! As you keep working with loops, you'll find that you often need the index to solve a problem. Let's use this additional info to fix our function now:

js/scripts.js

```
function boldPassage(word, text) {
  if ((text.trim().length === 0) || (word.trim().length === 0)) {
    return null;
  }
  const p = document.createElement("p");
  let textArray = text.split(" ");
  textArray.forEach(function(element, index) {
    if (word === element) {
      const bold = document.createElement("strong");
      bold.append(element);
      p.append(bold);
    } else {
      p.append(element);
    }
    if (index !== (textArray.length - 1)) {
      p.append(" ");
    }
  });
  return p;
}
```

We simply add `index` as the second parameter in our `Array.prototype.forEach()` loop. **Note that the index *must* be the second parameter. The first parameter is *always* the current element we are looping through.** We don't have to call the parameters `element` and `index` — but at least for `index`, that's generally going to be the best name for this parameter because it describes exactly what it is.

Then we add a new conditional at the end of the loop that uses the new `index` parameter:

```
if (index !== (textArray.length - 1)) {
  p.append(" ");
}
```

If the current `index` does not match the index location of the end of the array (represented by `textArray.length - 1`), then we append a space. Otherwise, we don't add one.

So now our `boldPassage()` user interface function is working correctly. All done, right? Well, no. This function has the same problem that our `numberOfOccurrencesInText()` function initially had. We need to account for differences in case, punctuation, and so on.

So we should just write some more tests and complete that functionality, don't you think? Not so fast! In the next lesson, we are going to discuss another essential programming concept called DRY, which is an acronym for *Don't Repeat Yourself*. We'll also complete our UI logic by calling the `boldPassage()` function and printing the results in our webpage.

Previous (/introduction-to-programming/arrays-and-looping/separation-of-logic-fixing-a-bug-in-text-analyzer)
Next (/introduction-to-programming/arrays-and-looping/drying-code-and-completing-the-text-analyzer-ui)

Lesson 30 of 50
Last updated February 28, 2023