Lesson | Tuesday

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / DRYing Code and Completing the Text Analyzer UI

Text | Cheat sheet

In the last lesson, we focused on the importance of keeping our business logic and UI logic separate. We could've updated our `numberOfOccurrencesInText()` function to do multiple things but this is a bad practice. A function should just do one thing if possible. We want to have **separation of concerns**, which means each function is concerned about just one thing and doesn't worry about anything else. That means `numberOfOccurrencesInText()` just cares about counting the number of occurrences of a substring in a string while `boldPassage()` should bold matches. Writing a function that did both things wouldn't be good even if it results in fewer lines of code.

In this lesson, we're going to discuss another very important programming concept known as **DRY**, which means **Don't Repeat Yourself**. There are a lot of good reasons not to repeat yourself:

- It results in unnecessary repeated code.
- It's harder to read and reason about because there's extra repeated code to deal with and read.
- If the code breaks or it needs to be updated, we have to change it in multiple places, not just one.

If you really practice separating concerns and keeping your code DRY, you are making a *huge* step towards writing amazing code that clearly communicates your intentions. These are some of the most important techniques you can learn as a coder.

So, let's start by reviewing how DRY our Text Analyzer application is. We'll also discuss tradeoffs and considerations when writing code that's both DRY and separated by concerns. Then, we'll end this lesson by completing our UI logic.

Finding a Balance between DRY Code and Code that Is Separated by Concern

If we review our Text Analyzer scripts, we can find a fair amount of repetition. For example, our two business logic functions and the `boldPassage()` UI function perform similar tasks:

- Checking whether the `word` or `text` parameters are empty.
- Splitting the text input into an array.
- Looping through each element of the text array to perform different actions on it.

As you might guess, sometimes separating our code makes it harder not to repeat ourselves. If we'd just put this all in one function, it would be DRY but have poor separation of logic. With good separation of logic, though, it's not as DRY. **Often, the best way to handle this is to *extract* any repeated code into its own function.**

So, let's DRY up our code in a small way: we'll extract the first conditional we use to check whether the `word` or `text` parameters are empty into its own function.

First, let's put these functions next to each other. Notice how the conditionals are similar:

```
// Business Logic

function wordCounter(text) {
  if (text.trim().length === 0) {
    return 0;
  }
  ...
  return wordCount;
}

function numberOfOccurrencesInText(word, text) {
  if (word.trim().length === 0) {
    return 0;
  }
  ...
  return wordCount;
}

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

There's two main differences: what each conditional returns and the number of parameters we check. But this won't be a problem — we can easily accommodate these differences.

We'll extract the functionality of the conditional into its own function, and we will put this at the top of our file and call it *Utility Logic*. You'll see the reason for the name later in this lesson.

scripts.js

```
// Utility Logic

function isEmpty(testString) {
  return (testString.trim().length === 0);
}
```

The `isEmpty()` function returns a boolean. If any inputted string is empty, it will return `true`. Otherwise, it will return `false`.

Next, we can plug it into our functions. Here's how we do it:

scripts.js

```
// Business Logic

function wordCounter(text) {
  if (isEmpty(text)) {
    return 0;
  }
  ...
  return wordCount;
}

function numberOfOccurrencesInText(word, text) {
  if (isEmpty(word)) {
    return 0;
  }
  ...
  return wordCount;
}

function boldPassage(word, text) {
  if (isEmpty(word) || isEmpty(text)) {
    return null;
  }
  ...
  return p;
}
```

As we can see, instead of checking `word.trim().length === 0`, our code now checks `isEmpty(word)`.

Hmm... doesn't seem like much of an improvement. Is it really worth it?

Well, imagine if we were using that same code in ten different functions and we realized that we also wanted to account for punctuation. For instance, if someone enters the following: `numberOfOccurrencesInText("!", ".");`, we want `isEmpty()` to return `false`, not `true`. Would you rather update that code in one place (the `isEmpty()` function) or in ten different functions? Also, what if in the process of updating the code in ten different places, you missed an eleventh place in the code that needed to be updated as well? These are the sort of scenarios that we need to think about when we're writing JavaScript.

In other situations you'll find that you're able to DRY up a much larger chunk of code. For instance, imagine that we have many different functions that all verify that every input is an actual English word. Imagine that this verification process involves a lot of different steps. Extracting this English-verifying functionality into a separate function that we can then call from any function that needs it makes a lot of sense.

So while refactoring our code to use the `isEmpty()` function is a very small example of DRYing our code, it illustrates the basic principle of extracting repeated code into its own function.

These kinds of functions are sometimes known as **helper** or **utility** functions. You should look for these kinds of opportunities to DRY up your code wherever possible. And again, while the example here is a very small one, it illustrates how we can keep our code DRY with helper functions while still keeping our business and user interface logic separate.

Completing our UI Logic

There's one more thing we need to do to get our application working. We've written our `boldPassage()` function but we aren't calling it yet. It needs to be called when the form is submitted, so we'll add it to the `handleFormSubmission()` UI function:

scripts.js

```
...

function handleFormSubmission() {
  event.preventDefault();
  const passage = document.getElementById("text-passage").v
alue;
  const word = document.getElementById("word").value;
  const wordCount = wordCounter(passage);
  const occurrencesOfWord = numberOfOccurrencesInText(word,
passage);
  document.getElementById("total-count").innerText = wordCo
unt;
  document.getElementById("selected-count").innerText = occ
urrencesOfWord;
  // new lines here!
  let boldedPassage = boldPassage(word, passage);
  if (boldedPassage) {
    document.querySelector("div#bolded-passage").append(bol
dedPassage);
  } else {
    document.querySelector("div#bolded-passage").innerText
= null;
  }
}

window.addEventListener("load", function() {
  document.querySelector("form#word-counter").addEventListe
ner("submit", handleFormSubmission);
});
```

Look how nice and clean that is! No logic cluttering up this section of the code at all. Instead, it's totally separated out. Even though `boldPassage()` is a function that deals with UI logic, it doesn't directly alter the DOM. It just returns a paragraph element. That makes it easy to test and easy to separate out. Then we can just call the function when we need it, and then use `Element.append()` to actually add it to the DOM.

Notice that we only update the DOM if the `boldedPassage` variable (the result from calling `boldPassage()`) is truthy. If the `boldedPassage` variable is `null` or any other falsey value, the code in the else block will run, setting the contents of the div to `null`, deleting anything inside of it including any HTML elements.

And with that, we've completely connected all of the logic for our Text Analyzer application. Hooray! Our code is nicely separated and we even added a little utility function to DRY things up a bit.

In a future lesson, we'll create a brand new function for our Text Analyzer application that uses a `for` loop. We'll also use a `for` loop when we explore JavaScript's arguments object (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments) to expand the capacity of the `isEmpty()` function.

Visit the cheat sheet to see the completed scripts and HTML for the Text Analyzer application.

Previous (/introduction-to-programming/arrays-and-looping/separation-of-concerns-in-text-analyzer-boldpassage-ui-function)
Next (/introduction-to-programming/arrays-and-looping/practice-using-tdd-with-text-analyzer)

(http://www.epicodus.com)