Debugging Your JavaScript

One of the most important skills for programmers to develop is the ability to fix errors in their code. For web developers, not all errors are immediately obvious in the browser window. Luckily, there are a few strategies and tools that we have and can develop that allow us to, not only find and fix the bugs, but to develop a code sense that lets us guess where the error probably is based on how the application behaves (or doesn't).

Checking Your Work

The workflow in web development is an iterative process, where we build one piece of the machine, test it to see what happens, then make adjustments based on the results of that test. Waiting until the whole thing is finished before testing for the first time does not save time in the long run, it only serves to make debugging more difficult by hiding or compounding errors. The iterative programming philosophy has been adopted in our industry formally (Agile project management) and informally ("Fail fast!"), and from my perspective, it should be implemented on a personal basis as well.

At the individual coding level, an iterative programming approach can be implemented simply by hitting refresh after adding a block of markup or code. However, when handling variables, we can't always see their value on the screen as they change through our logic. There are several tools in our toolbox that will allow us to see these values and make sure our formulae are doing what they are supposed to - especially in JavaScript, where type coercion can turn an empty string into a Boolean *false* value if data types are not managed properly.

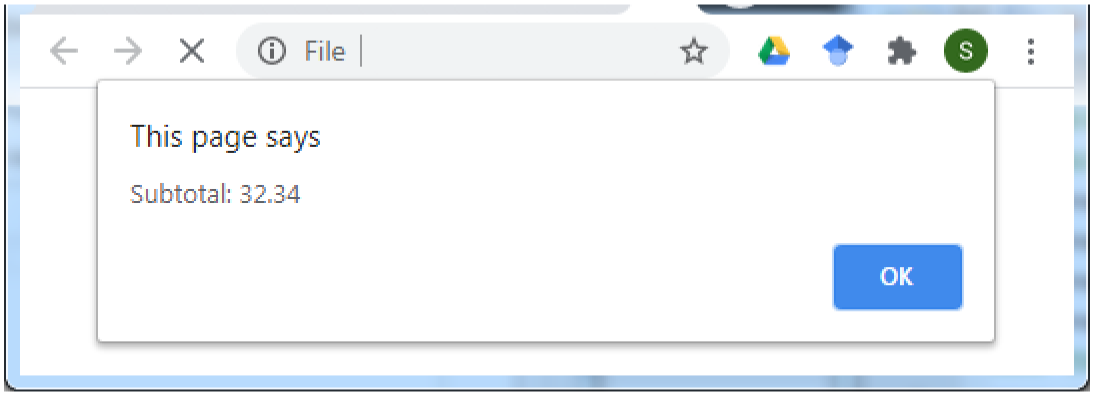
alert()

Alert boxes are popups that stop the execution of the code and can display text. By adding alerts to strategic points in our code, we can use them to stop the execution of our code at key points and check the value of variables at those points in the program. Remembering our "Cheque please!" lab from Week 2, we ran three original variables through several mathematical stages to arrive at a final alert displaying the total owed by each of our guests:

```
    subTotal = item1 + item2 + item3;//Add bill items.
    subAfterDiscount = subTotal - discount;//Subtract coupon.
    endTotal = subAfterDiscount * salesTax;//Multiply tax.
    eachOwes = endTotal / numOfDiners;//Get total for each diner.
```

We can use alerts after each of these lines of code to verify our results at each stage of the operations by outputting the variable value (and a helpful string of text) with an alert:

```
    subTotal = item1 + item2 + item3;//Add bill items.
alert("Subtotal: " + subTotal);
    subAfterDiscount = subTotal - discount;//Subtract coupon.
alert("After Discount: " + subAfterDiscount);
    endTotal = subAfterDiscount * salesTax;//Multiply tax.
alert("With tax: " + endTotal);
    eachOwes = endTotal / numOfDiners;//Get total for each diner.
alert("Each owes: " + eachOwes);
```



This page says

Subtotal: 32.34

Another technique using alerts in code is the "Marco Polo" method (okay, I think I made that up, but you'll see what I mean). Add various alerts in your code to see how far your code makes it. When a popup doesn't show up, you will know at which point that the problem occurred:

```
alert("Before the loop");
```

//Block of code here with the loop

```
alert("After the loop");
```

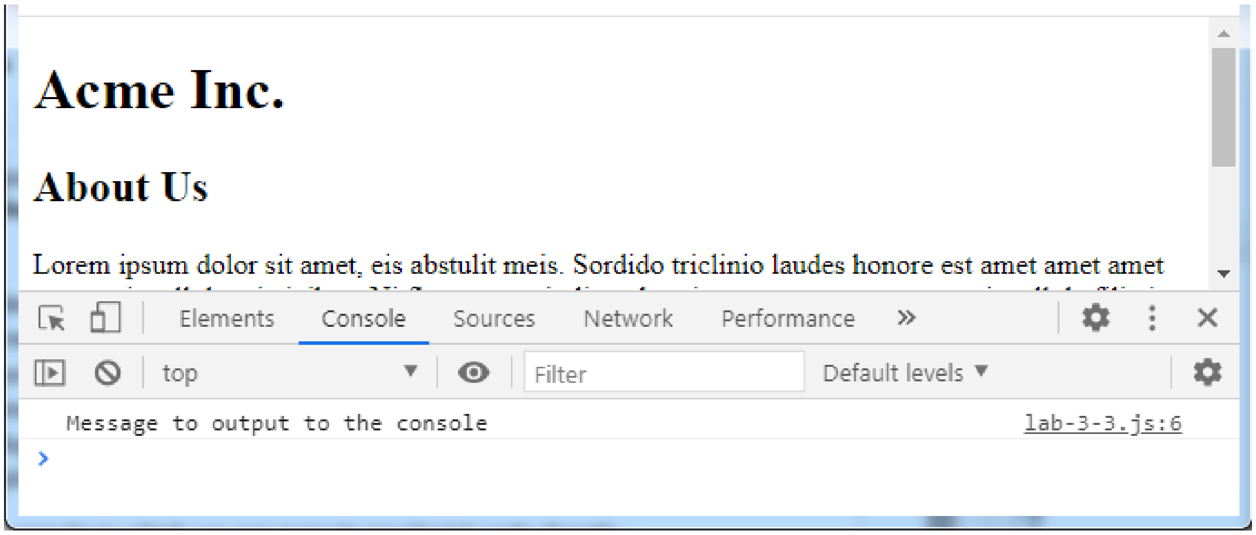
The benefit of alerts over the next technique that I will talk about is that **alerts will temporarily stop your program at that point in the code where the alert appears until you click, OK**. Be sure to remove your alerts once you have finished debugging.

The JavaScript Console

One of the most powerful tools in a web developer's toolkit is the set of developer tools built into most browsers. Chrome's DevTools provide a tremendous set of powerful tools that make our job much easier. You access them by right-clicking on the web page and selecting, Inspect, or by the keyboard shortcut: **Ctrl + shift +** j (Mac: **Option + Command + j**). At the top of the panel that appears, you will see numerous tabs. The first one, Elements displays the markup from the webpage (representing the DOM), and a second window that displays the styling for the elements.

The second tab, Console, is the JavaScript console in which we can execute JavaScript code directly, access the DOM directly, and receive messages from the JavaScript code. When a runtime error occurs (an error due to malformed JavaScript code), an error message will appear in this window that will provide information on the error, and the line number on which the error occurred. If you click on the line number on the far right, the console window will display the JavaScript code and focus on the line where it detected an error. Additionally, we can use this window for debugging, similar to the previously described way that we use alerts. The benefit of messaging through the console is that **using the console does not stop the code from executing**. To output a message, we simply add a line of code wherever we want to get feedback:

```
console.log("Message to output to the console");
```


Acme Inc.
About Us
Lorem ipsum dolor sit amet, eis abstulit meis. Sordido triclinio laudes honore est amet amet amet
Elements
Console
Sources
Network
Performance
top
Filter
Default levels

```
Message to output to the console                                   lab-3-3.js:6
```

You can see that the technique is practically identical to using an alert. Similarly, we can output variables in the same fashion:

```
    varSubtotal = varItem1 + varItem2 + varItem3;//Add bill items.
console.log("Subtotal:" + varSubtotal);
```

Because console.log doesn`t stop the execution of your program, it is easy to forget about them and leave them in your code, so be sure to do a search to remove your console.logs from completed code.

The Chrome DevTools are an extremely powerful toolkit (Firefox and Edge have their own sets of tools too), so take some time to see what else is there for you to use - in particular, check out how to use breakpoints for debugging - but for starting out, use console.log to check your values, and **always keep your console window open to see if any errors are occurring**. If you tell me that your code is not working, the first question I will ask you is, “What is the console showing you?”

Searching for Answers

No web developer can memorize everything in every language, therefore, the true talent of a developer is not to *have* all the answers, but to be able identify the true problem and *find* the answers. Some basic strategies for finding (“Googling”) the answers:

- use the language that you are working with in your search terms: “JavaScript limit to two decimal places”, “PHP loop through an associative array”
- use the error message as your search: “Failed to load resource: net::ERR_NAME_NOT_RESOLVED”
- Identify your most likely helpers in the search results. You will start to see that some resources are better than others, depending on the type of help that you are looking for. Is there something mysterious that JavaScript is doing? A help forum like stackoverflow.com will probably give you the answer that you are looking for, along with a conversation amongst other developers discussing best practices. Are you looking to see what methods or properties belong to an aspect of JavaScript? Official documentation (w3.org, Mozilla Developer Network) will likely provide the technical breakdown (plus examples) that you are looking for. Uncertain as to the browser support level for a new technique? The caniuse.com (“Can I use...”) website provides details of browser support for front-end techniques in HTML, CSS and JavaScript.

Training Yourself

Ultimately, your skills as a debugger will develop with each error that you encounter and solve. Look for patterns of behaviour in the browser and you will start to get a sense of where to start looking. Did the page fully load, but nothing happened? What does that suggest? Did the page not load at all? Where should you start looking? Did everything work, but the displayed values are incorrect? Where do you start your investigation? Does it work *sometimes*? When does it mean when code works *sometimes*???