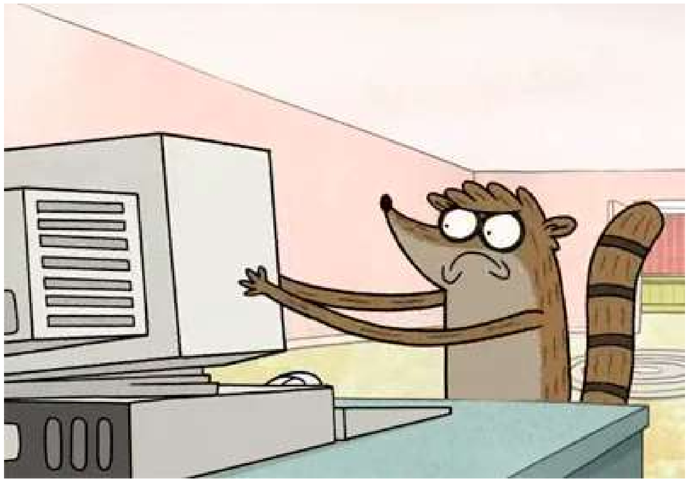
Module #7 - Where's the $@#*'ing Error? (Required)

A Fact of Life

A great coder once said, *"$@#$ my life!!! Why isn't this working!?!?"*



In fact, nearly every coder, at some point, has said something like this. You see, despite what many people think, coding is far from the effortless work of writing line after brilliant line of new features and visual effects.

Instead, it is often a slow and cumbersome process—one that will resist every effort to make it work as you expect. In fact, you should already brace yourself for the reality of spending hours (if not days) chasing down missing commas, misplaced parenthesis, and diffuse cases of capitalization.

And despite what many of you may be thinking, this reality is one that will follow you throughout your career. Whether you remain a green-eyed *n00b* forever or whether you become a seasoned system architect at Google, know that errors will be a fact of life. Thus, your job (and your professional career) will partly pivot around your ability to not only create ideas and to code them out, but it will also rely on your ability to fix things when they fail to work as expected.

Let us repeat this again *together*: **Errors will be a fact of life.**

Do not let them discourage you. Do not let them bring you down. Do not let them make you quit.

Instead, relish *every* instance in which you fix something broken, and celebrate the fact that you have entered the fray as a professional developer.

But I want to be the very best. To catch them is my test....

And for that you should be applauded!

Error-catching is an important part of every developer's role. The faster you catch bugs and resolve them, the sooner you can get back to creating applications and building out your clients' latest requests.

The process of fixing broken code is called *debugging*, and just as with all problems in life, there are right ways to deal with coding issues, and there are wrong ones. In this course you will develop a fleet of strategies for debugging issues, but in this pre-work module, we'll be giving you a small taste of what it's like to work through perplexing code errors.

How to Debug

Tip #1: Start Small

Let's start with the wrong way.

Often, new developers, in facing an instance of unworking code, immediately assume that "I must have done *everything* wrong. Let me start over (or give up)."

This is a great way to waste a lot of time, but it's not a great way to learn or to get code working.

Instead, when facing an issue, always try to isolate your attention to the lines of code closest to the feature that is broken. If clicking that button doesn't trigger a pop-up like you expected, look closely at the code associated with the button. Check your syntax carefully. Check how you've named things. Check your parenthesis. Check if your capitalization is off.

Start small and center your attention on what's broken. *Then* move outward to the farther stretches of the code.

Perhaps the button isn't broken after all, but the pop-up is. Perhaps you forgot to save. Perhaps you aren't even looking at the right file. Don't be so quick to assume there is a gigantic problem with your code. Sometimes it's the smallest thing keeping you from a working solution.

Tip #2: Reference Working Code

Many new developers have this tendency to think *I'm cheating if I look back at old code.*

This thought couldn't be further from the truth! Instead, as a developer, think of yourself as a curator of references. In working with many examples of code (as you will in this class), you are continually building a bank of *good code*. Whenever you face a task or an issue, you can always reference this *Bank of Good Code* to remember how to solve a previous problem.

This is how professional developers work as well. They constantly search through their own historic code, and those of others, to quickly tackle their issues at hand. Over time, certain bits of code become second nature... but this takes time and takes many hours of practice. Don't be hard on yourself if you aren't there yet. Memorizing will take time.

But let's get back to debugging. Say we have the following instance of a broken button:

Broken Code: Up Button

```
$(".upButton").on(click, function(){
    $(".captainplanet").animate({top:"-=200px"}, "normal");
});
```

The above code was written in the hopes of moving a Captain Planet image on button click. Yet, despite our best efforts, the code isn't moving Captain Planet like we expected. Instead, he just sits there.

Now we could hunt and peck through the code at random...

or we could pull an example of working code like the one we have here:

Working Code: Down Button

```
$(".downButton").on("click", function(){
    $(".captainplanet").animate({top: "+=200px"}, "normal");
});
```

Take a moment to compare the code between these two snippets. Can you spot the error? (Really try here!)

Hopefully, you will have noticed the missing quotation marks around the word "click". Now even if you've never been exposed to JavaScript before, it should be clear that having access to the working code can help you spot the issue.

This is the value of *good code*references.

Tip #3: Keep Your Code Clean

The third key to error-free living is to keep your code *aesthetically* clean. The concept of clean code will make more sense as you progress through the class, but in the meantime, take a look at the below two blocks of code:

Sloppy Code

```
$("#randomButton").on("click", function()
{for (var i=0; i<3; i++){alert(Math.floor(Math.random() * 25) + 1);}})
```

Clean Code!

```
// When randomButton is clicked...
$("#randomButton").on("click", function(){

    // Loop through 3 times...
    for (var i=0; i<3; i++){

        // And create an alert with a new random number between 1 and 25.
         alert(Math.floor(Math.random() * 25) + 1);
    }

});
```

Believe it or not, both these blocks function *exactly* the same. However, to a developer needing to maintain or to improve the code, the latter is obviously preferable.

What you will find in this course is that you can often *get by* with creating sloppy code that is poorly indented, that is lazily organized, and that is lacking any comments. But know that YOU are the primary recipient of the pain to come. Disorganized code is harder to read and is harder to debug. Errors that are easy to spot when code is well-indented and is well-structured, become nightmares to find when it is not.

Take, for instance, the following *broken* (but very realistic) code:

Sloppy Broken Code

```
for (var i=0; i<=5; i++)
    {if(i==5)
{console.log("Yay! My favorite #5!!")}
else(){    console.log("I don't like this number very much.")
}
```

It's hard to spot, but the code was missing a `}` towards the end of the else statement.

Now take a look at the same code—with the same error—but structured with good indentation:

Clean, Broken Code

```
// Loop through a set of numbers
for (var i=0; i<=5; i++){

    // If the number is 2
    if(i==5){
        // Print it out.
        console.log("Yay! My favorite #5!!");
    }

    // If the number is not 2
    else{
        // Print it out
        console.log("I don't like this number very much.");

}
```

Assuming you knew that every `{` must end with a `}`, this format makes the error much easier to spot!

Unfortunately, the example of poorly formatted code was taken from the pages of previous students. Remember, you've been warned! Don't repeat their mistakes!

Tip #4: Read the Debug Error

As you proceed through the course, you will be introduced to browser-based debugging tools like Google Debugger. These tools will help flag troublesome lines of code that are difficult to spot.

Now, we know what you're thinking: *Omg. I am so relieved. Those last exercises were tough!*

And truthfully, you are right! These debugging tools are incredibly powerful. However, they are only as powerful as the developer who utilizes them.

Take, for instance, the following broken code and the resulting output provided by the Google Debugger.

Broken Code

```
var myName = "Sam";
var favFood = "Green Eggs and Ham";

```

```
alert(myName + " loves " + favfood);
```

Google Debugger Output

```
quick.html:3 Uncaught ReferenceError: favfood is not defined
```

For someone new to coding, the error message may be just as indecipherable as the original code, but let's try to dissect it together.

For one, we can see that it specifies a line number of 3, which means that it *thinks* the error is on line 3: `alert....` (We say *think* because sometimes it gets tripped up by complex code.)

Next, it tells us that the error has something to do with the entry `favfood` and with its not being "defined." We'll get into concepts like variables and definitions later, but for now, consider the difference in how the `myName` data and the `favFood`data look. If you look closely, you might notice that `favFood` is capitalized once as `favFood` and is not capitalized in another as `favfood`.

This minor oversight was picked up and was flagged by Google Debugger, and with a little bit of sleuthing, it could easily be resolved. Again, don't be afraid of error messages! They are fantastic clues to your solution.

Tip #5: Test Often

This is a highly effective *preventive*strategy for keeping bugs out of your code.

Often, when new developers are assigned a task, they attempt to write *all* of the code in one sitting. If, for instance, they are tasked with creating a game of tic-tac-toe, they attempt to create the layout, the logic, the button events, the rules for determining who won, and everything else.... all without testing a single piece of functional code.

Unfortunately, this is a recipe for a hopeless labyrinth of bugs. Instead, it's a better idea to take as minimalistic an approach as possible. As you code, get into the habit of making *modest* changes, of saving them, and of then immediately testing them in the browser. This way, you isolate the location and the number of bugs to *only* the block of code on which you are working. If you try to bite off too much, errors will creep into numerous places—each having a cascading effect.

Again, this will make more sense as you start coding, but keep this advice in the back of your mind!

Tip #6: Get Help

Despite all the stereotypes, coding is an actually very collaborative line of work. In professional settings, coders are constantly in communication with one another and with the online communities of other developers.

As you begin your journey into Web Development, always remember that it's OKAY to ask for help.

Put in the hard work. Put in the long hours. But there is no shame in asking for a second pair of eyes. Sometimes, a fresh perspective is all it takes to make a breakthrough!

Tip #7: Practice, Practice, Practice

Last—but definitely not least—is the best tip of all: *Always be coding!*

The single best thing you can do to become a faster debugger (and better coder) is to simply code a LOT. In the beginning, you can be certain that a LOT of those hours will go into mindless hunting, but don't disparage yourself by thinking those are *wasted hours*. Instead, consider every hour you spend debugging to be valuable knowledge gained. The more time you spend studying where errors exist and where they don't, the better equipped you'll be to solve any problem you come across.

Time to get Coding (and Debugging)!

Speaking of practice, it's time to work on an exercise!

Check out the assignment below to try your own hand at debugging some HTML and JavaScript.

P.S. We know you don't know HTML or JavaScript yet! That's the point :P

Assignment (Required):

- The Boo Boos in Boo's Website

Additional Reading:

- The Art of Debugging
- Do You Spend More Time Coding or Debugging

Supplemental Resources:

- Getting Started with Google Debugger (You will learn this in class, but it never hurts to learn early)
- Stack Overflow (You will be ALL over this site)

Copyright