

Lesson

Wednesday

# Introduction to Programming

## (/introduction-to-programming)

### / JavaScript and Web Browsers

## (/introduction-to-programming/javascript-and-web-browsers)

### / Debugging in JavaScript: Using `console.log()`

Text

Cheat sheet

This is the second of four lessons on debugging in JavaScript. Previously, we've covered working with your DevTools console open and keeping an eye out for console errors. In this lesson, we've intentionally added two errors to our Mad Libs project so that we can learn how to use `console.log()` statements to help us debug. In the process, we'll also cover a very common error for JavaScript beginners — and how to easily spot it.

This lesson includes instructions for coding along, but you are welcome to simply read through this lesson if you prefer.

## Using `console.log()`

`console.log()` is a very commonly used tool in the JavaScript developer's toolbox. This method does exactly what it sounds like: logs a message to the console. In order to read the logged messages, the DevTools console must be open.

The `console` object itself is a Web API (tools/structures that web browsers provide to developers) and it represents and provides access to the browser's DevTools console. We won't work with much beyond the `console.log` method in the program, but there are other properties to explore including other ways to print text to the console. Check out the reference page for `console` on MDN (<https://developer.mozilla.org/en-US/docs/Web/API/console>).

## Finding the First Error

Let's drop a few `console.log()` statements into our Mad Libs script file. Remember, **we are intentionally adding an error to our Mad Lib's scripts.js**. If you are coding along with this lesson, copy and paste the following JS into your `scripts.js` file of your Mad Libs project, and do not change anything in the following code.

```
js/scripts.js
```

```
window.onload = function() {  
    // new line below  
    console.log("Script executing!");  
  
    let form = document.querySelector("form");  
    form.onsubmit = function(event) {  
        // new line below  
        console.log("Submit form successfully reached.");  
        const person1Input = document.getElementById("person1Input").value;  
        const person2Input = document.getElementById("person2Input").value;  
        const animalInput = document.getElementById("animalInput").value;  
        const exclamationInput = document.getElementById("exclamationInput").value;  
        const verbInput = document.getElementById("verbInput").value;  
        const nounInput = document.getElementById("nounInput").value;  
  
        document.querySelector("span#person1a").innerText = person1Input;  
        document.querySelector("span#person1b").innerText = person1Input;  
        document.querySelector("span#person1c").innerText = person1Input;  
        document.querySelector("span#person2a").innerText = person2Input;  
        document.querySelector("span#person2b").innerText = person2Input;  
        document.querySelector("span#animal").innerText = animalInput;  
        document.querySelector("span#verb").innerText = verbInput;  
        document.querySelector("span#noun").innerText = nounInput;  
        document.querySelector("span#exclamation").innerText = exclamationInput;  
  
        document.querySelector("div#story").removeAttribute("cl
```

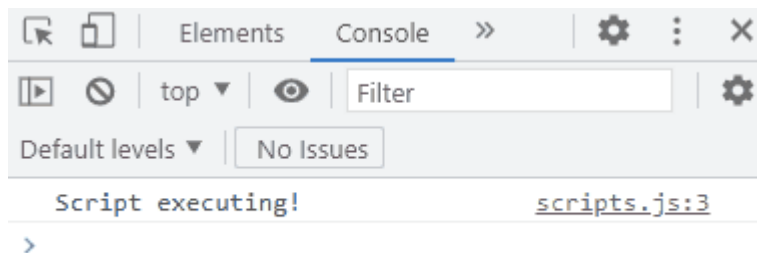
```
ass");  
    };  
};
```

Notice that we've added two new `console.log()` statements to the scripts:

- `console.log("Script executing!");` is now the second line of our script. We are using this `console.log()` to confirm our script is running as expected. (We already know our script is executing — but this is a good way to demonstrate what `console.log()` does.)
- Next, we put `console.log("Submit form successfully reached.");` inside our `onsubmit` event handler.

Save the script and refresh the page in the browser. Make sure the DevTools console pane is open. We'll see `Script executing!` as expected. Now let's try to submit the form — but pay very close attention to the console while you do.

You'll see that our second message very briefly logs to the console and then disappears. The gif below demonstrates this. So what exactly is happening?



This is a very common error, especially for beginners, and there are a couple of clues that show what's happening.

Whenever a result flashes across the screen — or the console like this — it suggests that the code is executing correctly but that the page is reloading.

Remember that the **default behavior** of a form submission event is to load a new page. That means that by default, the page will refresh unless we prevent that default with `event.preventDefault()`.

There is another subtle but important clue that let's us know when we've forgotten to add `event.preventDefault()` to our code. Check the URL after you submit the form and you'll see a `?` within the URL. The question mark `?` indicates that we've made a GET request with a form. A GET request means we're making a request to a server, and we'll learn more about this in the future. If we prevent the default behavior of the form, we won't make a GET request. We just stay on the page and no question mark `?` is added to the URL.

Let's fix the issue:

```
js/scripts.js
```

```
window.onload = function() {
  console.log("Script executing!");
  let form = document.querySelector("form");
  form.onsubmit = function(event) {
    // we've added event.preventDefault(); below
    event.preventDefault();
    console.log("Submit form successfully reached.");
    const person1Input = document.getElementById("person1Input").value;
    const person2Input = document.getElementById("person2Input").value;
    const animalInput = document.getElementById("animalInput").value;
    const exclamationInput = document.getElementById("exclamationInput").value;
    const verbInput = document.getElementById("verbInput").value;
    const nounInput = document.getElementById("nounInput").value;

    document.querySelector("span#person1a").innerText = person1Input;
    document.querySelector("span#person1b").innerText = person1Input;
    document.querySelector("span#person1c").innerText = person1Input;
    document.querySelector("span#person2a").innerText = person2Input;
    document.querySelector("span#person2b").innerText = person2Input;
    document.querySelector("span#animal").innerText = animalInput;
    document.querySelector("span#verb").innerText = verbInput;
    document.querySelector("span#noun").innerText = nounInput;
    document.querySelector("span#exclamation").innerText = exclamationInput;

    document.querySelector("div#story").removeAttribute("class");
  }
}
```

```
};  
};
```

We've made one change to our code to fix the issue: we've added `event.preventDefault()` right inside of the `onsubmit` event handler. If the `?` is still at the end of the URL, remove it and anything after it, and refresh the page. For example if you are serving the project with LiveServer, your URL might look like

`http://127.0.0.1:5501/mad-libs.html?person1Input=Mohammed`, and you'll want to reset it to `http://127.0.0.1:5501/mad-libs.html`. If you forget to do this, there shouldn't be an issue, but you should know that your URL won't automatically reset with a page refresh.

Now try filling out the form. The text from our fantastical adventure will appear — and both messages will be properly logged to the console. Finally, check out the URL — a `?` wasn't added.

If you ever find that a form submission isn't working correctly, checking the URL for a question mark is the first — and easiest — debugging step.

## Finding the Second Error

As noted in the example above, `console.log()` statements can be useful for seeing which code is reached. However, we can use `console.log()` for other bugs, too — including the next issue we are having with our Mad Libs story. Copy and paste the following scripts into your Mad Libs `scripts.js` file. **Remember that we're intentionally introducing errors into our scripts, so if you spot the new error, don't fix it!** The new error is a typo, which is an incredibly common issue in coding and can eat up *a lot* of a developer's time.

```
js/scripts.js
```

```
window.onload = function() {  
  let form = document.querySelector("form");  
  form.onsubmit = function(event) {  
    event.preventDefault();  
    const person1Input = document.getElementById("person1Input").value;  
    const person2Input = document.getElementById("person2Input").value;  
    const animalInput = document.getElementById("animalInput").value;  
    const exclamationInput = document.getElementById("exclamationInput").value;  
    const verbInput = document.getElementById("verbInput").value;  
    const nounInput = document.getElementById("nounInput").value;  
  
    document.querySelector("span#person1a").innerText = person1Input;  
    document.querySelector("span#person1b").innerText = person1Input;  
    document.querySelector("span#person1c").innerText = person1Input;  
    document.querySelector("span#person2a").innerText = person2Input;  
    document.querySelector("span#person2b").innerText = person2Input;  
    document.querySelector("span#animal").innerText = animalInput;  
    document.querySelector("span#verb").innerText = verbInput;  
    document.querySelector("span#noun").innerText = nounInput;  
    document.querySelector("span#exclamation").innerText = exclamationInput;  
  
    document.querySelector("div#story").removeAttribute("class");  
  };  
};
```



Make sure to save your scripts and refresh your browser opened to the Mad Libs project. Then, fill in the form and submit it. We should see our story display successfully, but with one input missing from the story! Here's how our story looks now (yours will look a little different depending on the words you input into your form):

"One day, Mohammed and Wei were walking through the woods, when suddenly a giant cat appeared. "woah", Mohammed cried. The two of them \_\_\_\_\_ as quickly as possible, and when they were safe, Mohammed and Wei gave each other a giant banana."

We are getting some input but not all of it. Specifically, the verb isn't showing up. We can see that the underscore \_\_\_\_\_ is left in place.

Now it's time to debug this issue! Let's think about where to start. Well, if the default value in the story isn't being updated for the verb, this clearly indicates that there is something wrong in the code that displays the verb value. So, how can we use `console.log()` to help us out? We can log all of the verb-related values to the console and see what shows up as an issue. Copy and paste the following code into your `scripts.js`. Note — we've removed the `console.log()` statements from the last error, and we've added one new `console.log()` statement.

**js/scripts.js**

```
window.onload = function() {
  let form = document.querySelector("form");
  form.onsubmit = function(event) {
    event.preventDefault();
    const person1Input = document.getElementById("person1Input").value;
    const person2Input = document.getElementById("person2Input").value;
    const animalInput = document.getElementById("animalInput").value;
    const exclamationInput = document.getElementById("exclamationInput").value;
    const verbInput = document.getElementById("verbInput").value;
    // Here's the first log we add.
    console.log("verbInput = " + verbInput);
    const nounInput = document.getElementById("nounInput").value;

    document.querySelector("span#person1a").innerText = person1Input;
    document.querySelector("span#person1b").innerText = person1Input;
    document.querySelector("span#person1c").innerText = person1Input;
    document.querySelector("span#person2a").innerText = person2Input;
    document.querySelector("span#person2b").innerText = person2Input;
    document.querySelector("span#animal").innerText = animalInput;
    document.querySelector("span#verb").innerText = verbInput;
    document.querySelector("span#noun").innerText = nounInput;
    document.querySelector("span#exclamation").innerText = exclamationInput;

    document.querySelector("div#story").removeAttribute("class");
  }
}
```

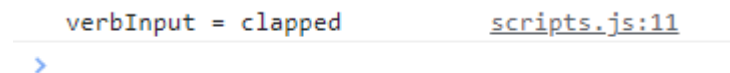
```
};  
};
```

With this new `console.log()` statement, we are checking the `verbInput` variable we create to make sure that we're successfully getting the input value from the form. Notice that we've added a variable *and* text as the argument to this `console.log()` method call:

```
console.log("verbInput = " + verbInput);
```

It's helpful to include a descriptive message that explains what the `console.log` is logging. We can combine text and variables using a plus sign `+` or a comma `,`.

Fill out and submit the form again and look in the DevTools console for the message. We should see something similar to the following:

A screenshot of a web browser's DevTools console. It shows a single log entry with the text "verbInput = clapped" in blue. To the right of the text, the source is listed as "scripts.js:11" in red. A blue arrow points to the log entry.

```
verbInput = clapped      scripts.js:11
```

Notice in the image above that there's `scripts.js:11` listed. This indicates the location of the `console.log()` in our code, including the file name and the line number.

What does this log tell us? The verb we put into the form is "clapped" and we've successfully grabbed it from the form. So, the bug is not here.

The next thing we can look at is the code that handles putting the form into the story:

```
document.querySelector("span#verb").inerText = verbInput;
```

As we look at this line of code, we can check it for typos: are we referencing the same `verbInput` variable? Yes, we are. Are we accessing the correct HTML element? It looks like it... How about the correct property for `innerHTML`? Hmm... that doesn't look right: there's a missing `n` in the `inerText` property. A typo! (Don't fix it just yet.)

With this visual review of the code, we may be able to spot the bug. However, if the visual review of the code doesn't definitively locate a bug, we can use `console.log()` statements to further verify! Specifically, we can add `console.log()` statements to check all of the questions we ask in the last paragraph. Let's do so now. Copy and paste the following code into your `scripts.js`.

```
js/scripts.js
```

```
window.onload = function() {
  let form = document.querySelector("form");
  form.onsubmit = function(event) {
    event.preventDefault();
    const person1Input = document.getElementById("person1Input").value;
    const person2Input = document.getElementById("person2Input").value;
    const animalInput = document.getElementById("animalInput").value;
    const exclamationInput = document.getElementById("exclamationInput").value;
    const verbInput = document.getElementById("verbInput").value;

    // Here's the first log we added.
    console.log("verbInput = " + verbInput);
    const nounInput = document.getElementById("nounInput").value;

    document.querySelector("span#person1a").innerText = person1Input;
    document.querySelector("span#person1b").innerText = person1Input;
    document.querySelector("span#person1c").innerText = person1Input;
    document.querySelector("span#person2a").innerText = person2Input;
    document.querySelector("span#person2b").innerText = person2Input;
    document.querySelector("span#animal").innerText = animalInput;

    // Here are the 3 new logs!
    console.log("Correctly targeting <span>? = ", document.querySelector("span#verb"));
    console.log("Correctly targeting innerText? = ", document.querySelector("span#verb").innerText);
    document.querySelector("span#verb").innerText = verbInput;

    console.log("Correctly referencing verbInput and assigning value of innerText? = ", document.querySelector("span#verb").innerText);
  }
}
```

```
// Above are the 3 new logs!  
document.querySelector("span#noun").innerText = nounInput;  
  
document.querySelector("span#exclamation").innerText =  
exclamationInput;  
  
document.querySelector("div#story").removeAttribute("class");  
};  
};
```

We've added 3 new `console.log()` statements, each with a different piece of code that's been copy and pasted from the line of code that we're debugging. From top to bottom, this is the purpose of each new log:

- When we log `document.querySelector("span#verb")`, we're checking that we're accessing the right element in the DOM. We should get the actual HTML for the span element.
- When we log `document.querySelector("span#verb").innerText`, we're checking the value of the `innerText` property, which lets us know that we're accessing the right property of the span element we want to target. We should get \_\_\_\_\_ for this value. Note that we do this before the line of code we think the bug is in, because it allows us to see the value of the property **before** we assign it a value.
- Finally, we log `document.querySelector("span#verb").innerText` below the line of code in question, because we want to check the value of the `innerText` property **after** we've assigned a new value to it. This will help confirm that we're referencing the correct property as well as the correct `verbInput` variable. We should get the verb value that we're putting into the story, like `clapped` in our previous example.

Go ahead and save your scripts, refresh your browser, fill out the form, and submit it. In the DevTools console, we should see four logs (note that your values may look different depending on the words you input into your form):

```
verbInput = clapped                                scripts.js:11  
Are we correctly targeting the span in scripts.js:21  
story? =    <span id="verb">_____</span>  
Are we correctly targeting the span's scripts.js:22  
innerText property in story? = undefined  
Are we referencing the correct scripts.js:24  
verbInput variable and correctly assigning the value  
of innerText? = clapped  
>
```

So what can we understand from the above logs? A few things:

- The first log we're already familiar with: we are correctly getting the verb input from the form. This is what we expect to happen.
- The second log confirms that we are correctly accessing the verb's span element in the Mad Libs story output. This is what we expect to happen.
- The third log returns `undefined` which indicates that there's something wrong with how we're accessing the `innerText` property. This is not what we expect to happen. We expect to see `_____` returned to us.
- The fourth log returns `clapped` which means that we're at least assigning the `innerText` property correctly. This is what we expect to happen.

Because the third log returns `undefined` instead of `_____` as we expect, this unequivocally means there's something wrong with how we're accessing the `innerText` property. As we figured out earlier, we have a typo: there's a missing `n` in `inerText`.

The confusing thing here is that the fourth log returns the correct value we're assigning to the `inerText` property: `clapped`. This is happening because with the following code:

```
document.querySelector("span#verb").inerText = verbInput;  
t;
```

We're actually creating a new property for the span element called `innerText` and we're assigning it the value of the verb input, `clapped`. In other words, we can use **dot notation** to access a property of an object, like `window.innerHeight`, as well as to create a brand new property, like `window.myCustomProperty`. And when we have a typo in a property that we're trying to access or assign a value to, like running `window.inerHight` (with two typos), we end up creating a brand new property.

To cement this concept, try creating a new property on a Web API. Enter the following code into the DevTools console:

```
> window.myCustomProperty = "hello world";  
> window.myCustomProperty;  
"hello world"
```

With the above code we've created a property called `myCustomProperty` for the `window` object. If we refresh the page, this property will be erased, because we can't actually add permanent properties to any Web API. Also note that we don't want to go around creating new properties for Web APIs. Instead we want to use their built-in properties and methods.

At this point, we've learned the basics of using `console.log()` to debug our code. Go ahead and fix the typo that in the Mad Libs scripts, save the scripts file, refresh the website, and resubmit the form. Our Mad Libs website should be working correctly again.

**Once we are done with our debugging, we always need to remove all `console.log()` statements from our code base.** Why? Anyone can open the DevTools console and see messages that are logged there and it makes our webpage look unfinished. That's why it's considered bad practice to leave behind `console.log()` statements from your debugging.



## Summary

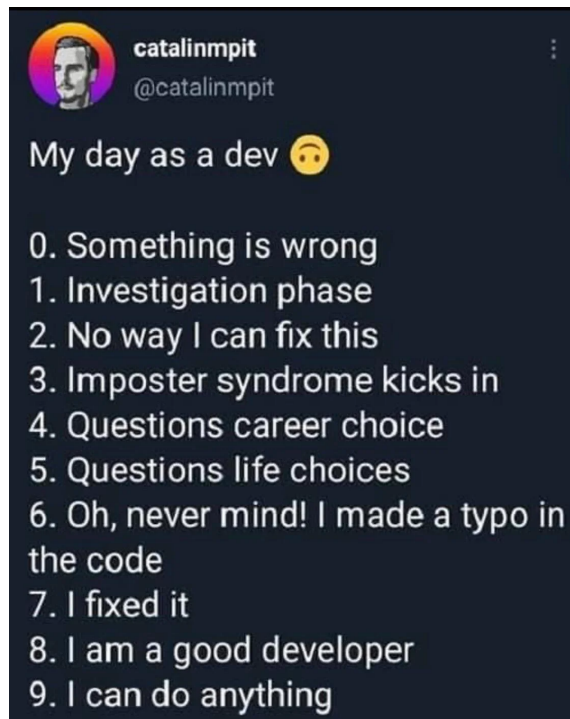
`console.log` can be a very useful tool, especially for beginning developers. We can use effectively in two cases:

- To quickly check if code is being reached.
- To see the value of a variable, just like we did above.

However, there are two things to consider about this tool. First, it's not as efficient as other debugging tools. It's definitely good to have in our toolbox, but usually it won't be the first tool we reach for. We'll learn about more sophisticated debugging techniques in upcoming lessons.

Second, once you are done debugging, you need to go back and remove `console.log()` statements from your code. They won't break anything but they make your finished code look sloppy. They are clear evidence of the debugging process — and should never be in portfolio-ready or production code. You will always be expected to remove any `console.log()` statements from your independent projects before they are submitted.

Lastly, keep in mind that typos are very common and pesky bugs! Don't let them ruin your sense of your capability for computer programming. Instead, do your best to debug, and then reach out for help after you've exhausted your debugging steps.



[Previous \(/introduction-to-programming/javascript-and-web-browsers/other-ways-to-organize-ui-logic\)](#)

[Next \(/introduction-to-programming/javascript-and-web-browsers/practice-forms\)](#)

Lesson 56 of 75

Last updated March 24, 2023

disable dark mode



Epicodus

(<http://www.epicodus.com>)

© 2023 Epicodus (<http://www.epicodus.com/>), Inc.