Review: Debugging

[(https://github.com/learn-co-curriculum/phase-1-intro-to-debugging-readme)](https://github.com/learn-co-curriculum/phase-1-intro-to-debugging-readme) [(https://github.com/learn-co-curriculum/phase-1-intro-to-debugging-readme/issues/new)](https://github.com/learn-co-curriculum/phase-1-intro-to-debugging-readme/issues/new)

Learning Goals

- Define *tracing*
- Use the built-in `console` object for debugging
- Demonstrate `console.log()`
- Demonstrate `console.error()`
- Demonstrate `console.warn()`
- Demonstrate `console.table()`

Introduction

As developers, one of the things that we spend about half of our time on is debugging. Debugging is the process of figuring out where our code is either breaking, or giving us an unexpected result. Debugging is absolutely normal, and a crucial part of the development process. With a little practice, you'll become an expert! In this lesson, we'll look at a few different ways to print data to the JavaScript console as a means of debugging our code.

Define *tracing*

We've already used `console.log()` to print out data to the console, but we haven't really discussed why you'd want to do that. In short, it's one of the most basic, best tools in a JavaScript programmer's debugging toolkit.

> As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. We had to discover debugging. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs. — Maurice Wilkes, 1949

Tracing is using output statements (like `console.log()`) to provide feedback about "what the machine is thinking." Oftentimes we request our code to behave like a machine, or like a process...

...and sometimes that process doesn't *quite* give us what we want. Tracing allows us to check some assumptions. Taking Liz Lemon in the picture as an example, she'd want to know:

1. Did the person who places orders get my Mac N' Cheese order?
2. Did the person who receives orders get my Mac N' Cheese order?
3. Was my Mac N' Cheese order on the receipt? If not, problem happened *before* this point (investigate steps 1 and 2). If not, problem happened after.
4. (Judging by the fact that there are lunches on the table, clearly the order got here, was paid for, and was put on the table. No debugging needed)
5. Who opened up the box of lunch orders?
6. Did any of the writers see my order?
7. If someone saw it, and it's not there, someone took it. If no one saw it, call restaurant to make sure they fulfilled the receipt.

We can imagine that Liz could check these steps above with code:

- Check the `Object` of `lunchOrdersForTheWriters`
- Check the value for `lunchOrdersForTheWriters["liz"]`
- Check whether any `writers` `Array` element responds `truthy` to `sawOrder("Mac N Cheese")` ?

Debugging the order delivery process like this is "tracing the program."

Identify The Built-in `console` Object For Debugging

The **browser**, not **the JavaScript language** provides an object called `console`. When the first Developer tools were released, only Firefox had them. Firefox chose to call the console `console`, but other browsers didn't have tooling at all! Over time browsers followed Firefox and rolled in tooling and *have chosen* to call the console `console`, but they didn't have to.

This `console` object has specific methods that send text to the DevTools logging area, which pretty much everyone calls "the console."

Demonstrate `console.log()`

The `console` object's `log()` method logs general information to the console. It can take any number of arguments. If more than one argument is provided, the arguments will be printed out on the same line with a space in between:

```
console.log("Hello,", "world!");
// LOG: Hello, world!
```

Importantly, you can log not only simple things like `String`s or `Number`s but also objects, and use disclosure triangles to "expand out" the contained values.

> **Typographical Note**: When we use `console.log()` in code snippets, we'll preface the output statements with `LOG:`, such as in the above example. This is to differentiate messages logged out to the console from values `return`ed by an expression, which are represented with `=>`, e.g.:

```
function logReturner() {
  console.log(false);

  return true;
}

logReturner();
```

```
// LOG: false
// => true
```

As an example, here's some code. Where might we want to log information to debug this simple app?

```
const number = 10;

function addTwoNumbers(a, b) {
  a + b;
}

function multiplyByTwo(n) {
  number * 2;
}
```

Copy the code above into your console then run the following:

```
console.log(multiplyByTwo(addTwoNumbers(1, 3))); //=> undefined(?!)
```

Some questions to consider in debugging the code:

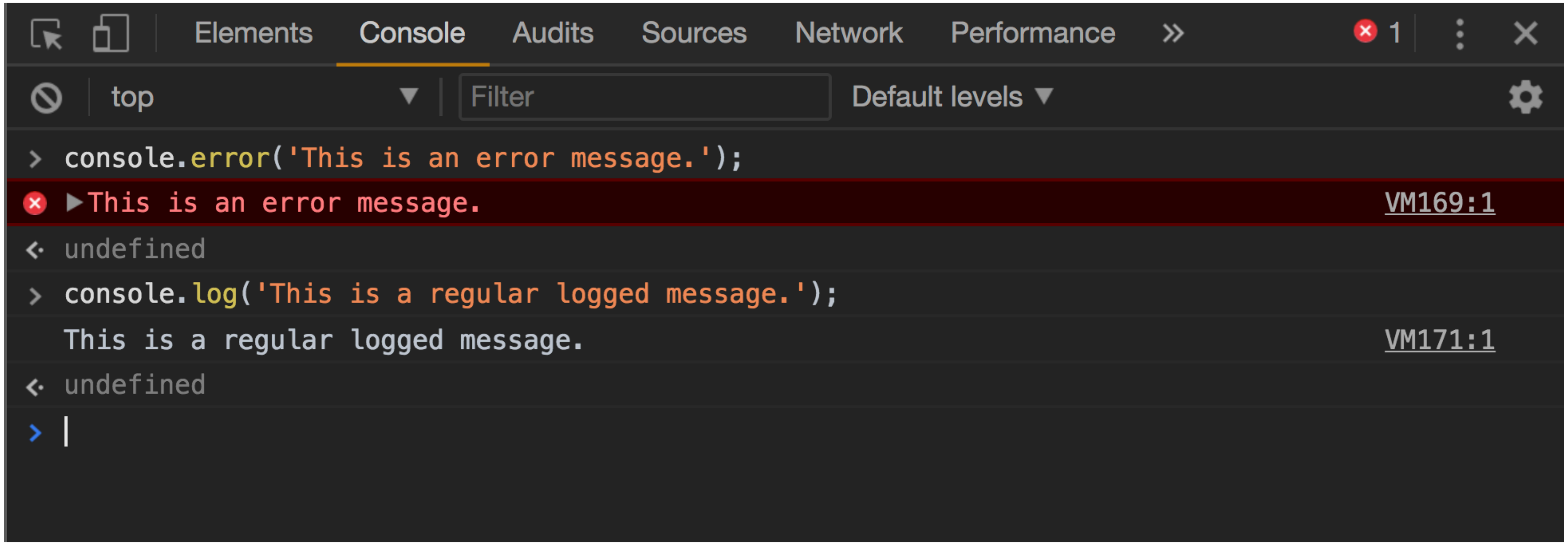
- Is what we passed in what the function got?
- Is the thing the function did what we expected it to do?
- Does the operator work like we thought it did?

Try adding some `console.log()` s to the code to answer these questions and figure out what the issue is.

To start, `console.log()` will be our main `console` debugging method. However, you'll also probably encounter the following two `console` methods, `error()` and `warn()` .

Demonstrate `console.error()`

The `console` object's `error()` method is for printing out an error to the console, and it can also take multiple arguments. Most browsers will style the error message differently from a regular message output with `log()` :



```
> console.error('This is an error message.');
⊗ ▶This is an error message.                                  VM169:1
<· undefined
> console.log('This is a regular logged message.');
  This is a regular logged message.                           VM171:1
<· undefined
>
```

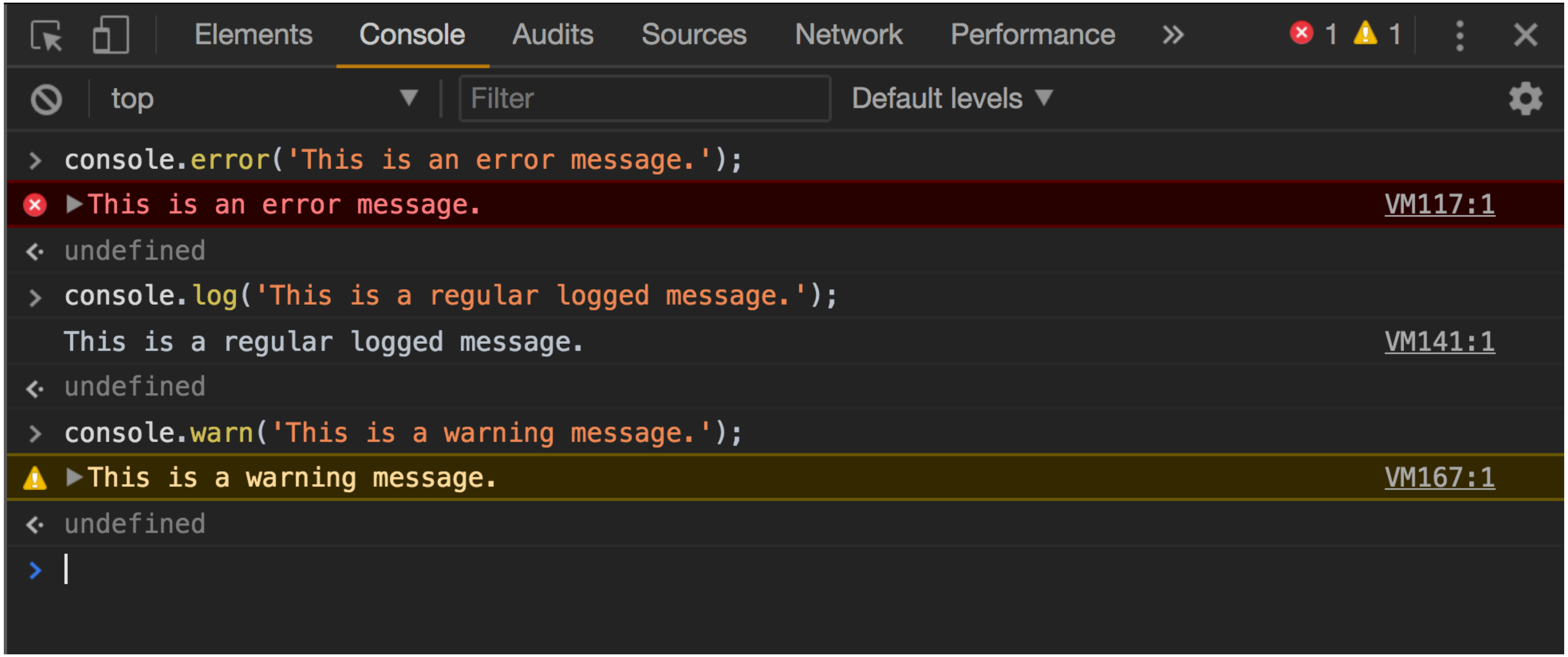
You might ask why we'd ever need to use this — isn't the goal of writing good code to **avoid** errors? Well, sure, but sometimes errors are out of our control: the network could go down, data could change, or a user could enter something invalid. In these cases, it's helpful to use the specialized `console.error()` method. That way, you're letting future engineers (including yourself) know that this message is more important than the average logged message.

> **TYPOGRAPHICAL NOTE**: When we use `console.error()` in code snippets, we'll preface the output statements with `ERROR:` to differentiate them from other logged messages:

```
console.error("Uh oh, you done goofed.");
// ERROR: Uh oh, you done goofed.
```

Demonstrate `console.warn()`

A step down in severity from `console.error()` is `console.warn()` . It provides a step between a regular `log()` message and a more dire `error()` message.



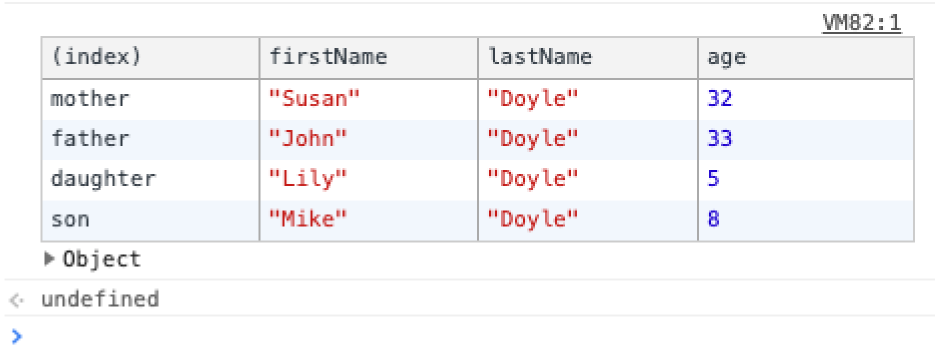
```
> console.error('This is an error message.');
✖ ▶This is an error message.                    VM117:1
<· undefined
> console.log('This is a regular logged message.');
  This is a regular logged message.             VM141:1
<· undefined
> console.warn('This is a warning message.');
⚠ ▶This is a warning message.                   VM167:1
<· undefined
>
```

Demonstrate `console.table()`

A very handy method to help work with `Object` s and `Array` s is `console.table()` . Given the following object:

```
const family = {
  mother: {
    firstName: "Susan",
    lastName: "Doyle",
    age: 32,
  },
  father: {
    firstName: "John",
    lastName: "Doyle",
    age: 33,
  },
  daughter: {
    firstName: "Lily",
    lastName: "Doyle",
    age: 5,
  },
  son: {
    firstName: "Mike",
    lastName: "Doyle",
    age: 8,
  },
};
```

If you call `console.table(family)` , it prints a tables of entries:



VM82:1

(index)	firstName	lastName	age
mother	"Susan"	"Doyle"	32
father	"John"	"Doyle"	33
daughter	"Lily"	"Doyle"	5
son	"Mike"	"Doyle"	8

```
▶ Object
<· undefined
>
```

Conclusion

Over the course of your programming career, you'll probably spend **significantly** more time debugging than actually writing new code. Just as your coding skills will improve with practice, so too will your debugging skills.

Debugging can sometimes make you feel sad. You'll fix one bug and ten new ones appear:

If it's any `console`-ation, we **all** make mistakes. Treat debugging as a learning opportunity. Often, looking at your code critically and trying to figure out why something isn't working will afford you a much deeper understanding of how some feature of the language actually works.

Also, sometimes difficulty debugging might hint at a program that needs some help from a mentor, a pair, or a friend. Some of our best code edits have started by talking to a friend and saying "This seems...really complicated and I can't debug it easily!"

We'll continue to use the `console` object and other tools throughout this course. By the end, you'll be on your way to being a debugging expert!

Resources

- MDN — Console (https://developer.mozilla.org/en-US/docs/Web/API/console)
- Wikipedia — Tracing (software) (https://en.wikipedia.org/wiki/Tracing_(software))

- **Nick Parlante (Stanford CS) — Debugging Zen (https://curriculum-content.s3.amazonaws.com/web-development/js/basics/intro-to-debugging-readme/nick_parlante_debugging_zen_1996.pdf)**
- **Google Chrome — Console Overview (https://developers.google.com/web/tools/chrome-devtools/console/)**