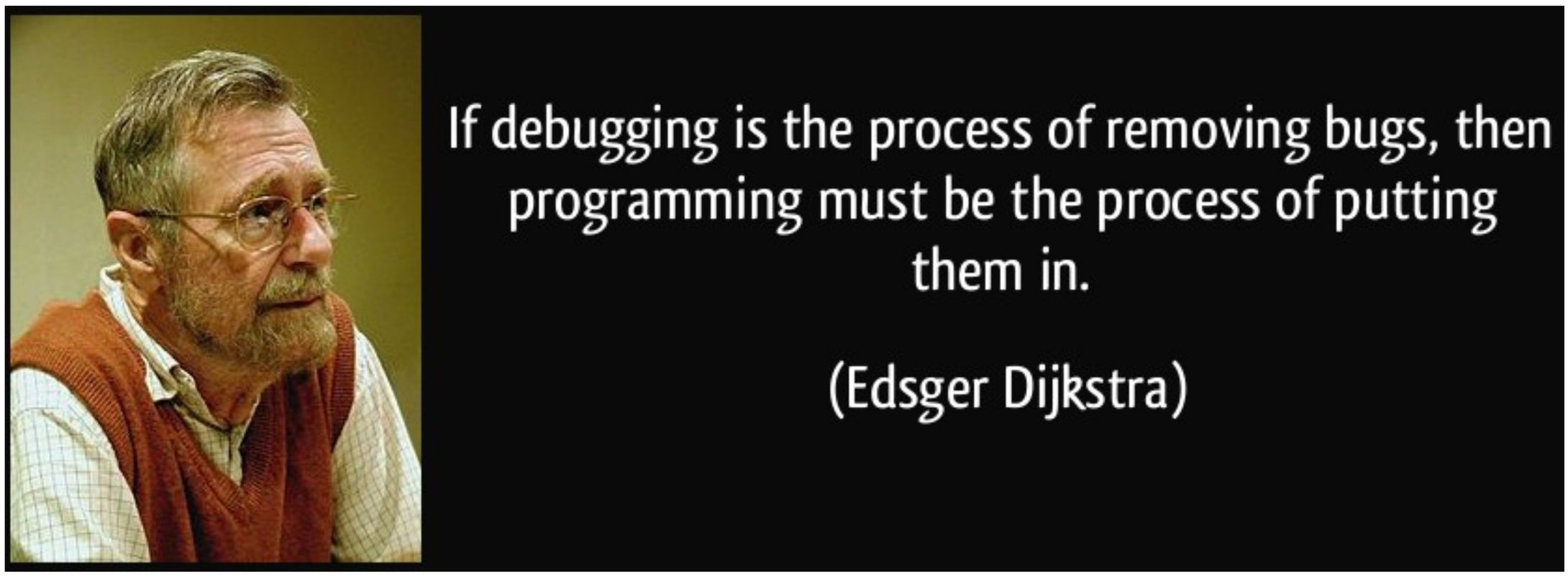
Back to Week 2

JS | Debugging, Error Handling and JS Hint

LESSON

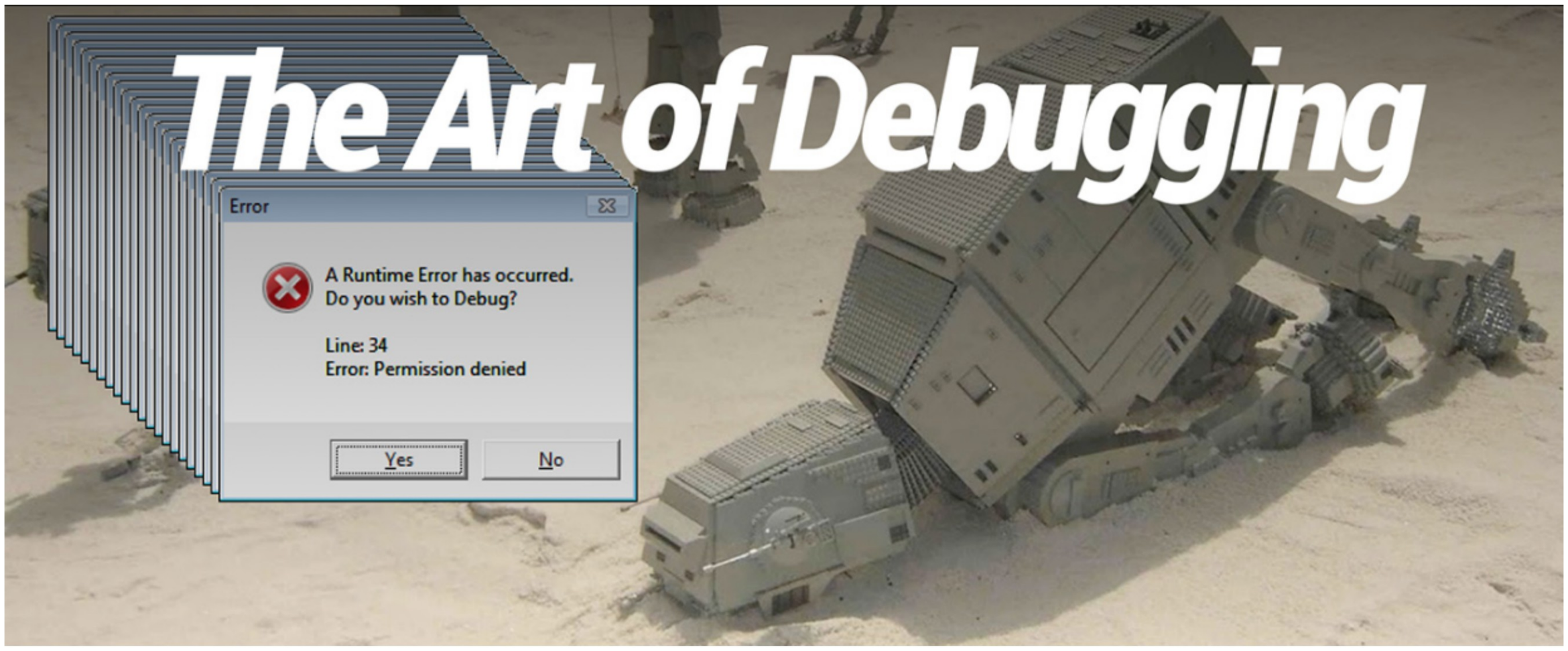


Learning Goals

After this lesson, you should be able to:

- Understand what *debugging* is
- Correctly use the `console` object
- Use a debugging tool to handle the errors
- Use breakpoints, `watch` and `call stack`
- Access variables and their behavior during execution time
- Use execution time controls
- Identify the most common errors in JavaScript
- Understand what **JS Hint** is and how it works

Debugging



> ***Debugging*** *is a multi-step process that involves identifying a problem, isolating the source of the problem, and then either correcting the problem or determining a way to work around it.*

The final step of debugging is to test the correction or workaround and make sure it works.

In this lesson, we are going to teach you how to face errors, identify them, and fix them.

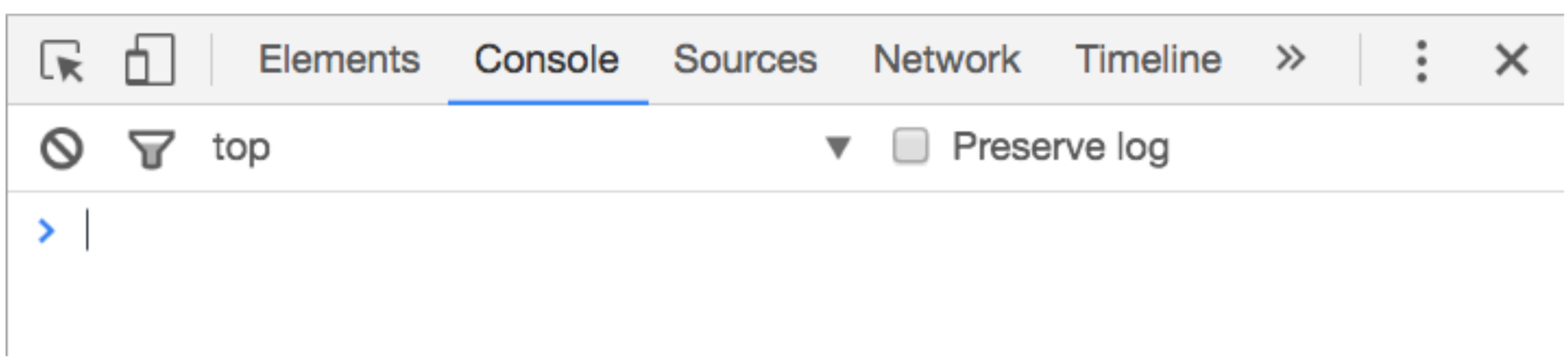
Also, at the end of the lesson, we will take a look at some of the most common errors in JavaScript, how to detect them and how to solve them.

The *JavaScript Console* tool

The **Chrome Dev Tools** offer us several tools to develop web applications, but one of the most useful features is the *JavaScript Console*.

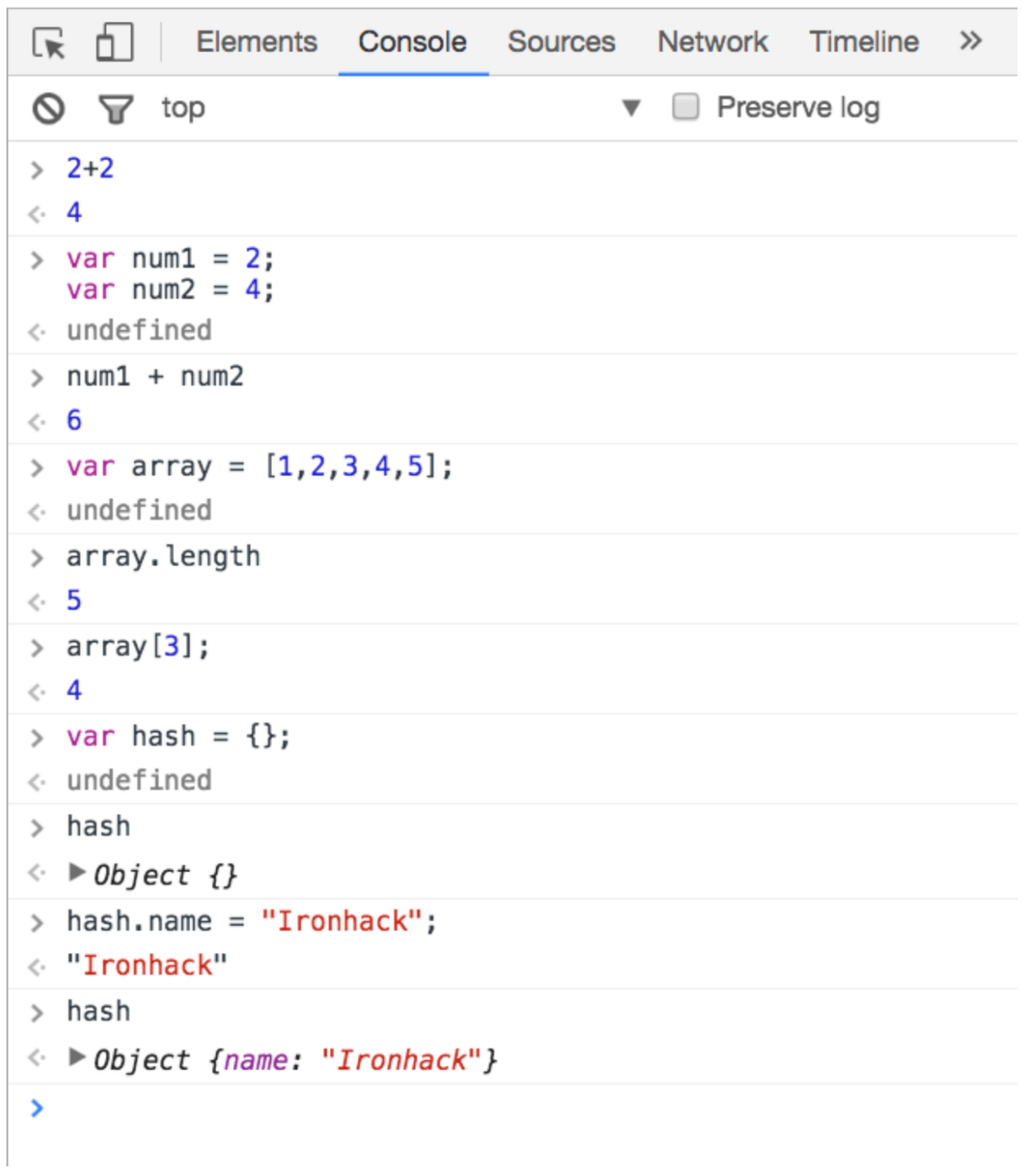
The *JavaScript Console* is a logger and a **command line interface**. It allows us to easily interact with our apps, run JavaScript commands and display messages for debugging help.

:open*file_folder: We can bring up the console by simply opening the Developers tools (*`Cmd + Alt + i` *on Mac and* `Ctrl + Shift + i` *on PC) inside Chrome and clicking on the _Console* tab.



Using the console

The console could be used as **REPL**: We can write and execute code immediately. Let's try it, write some code in the prompt and press the *Enter* (`return`) key. The code will be interpreted and the output will be shown.



The `console` object

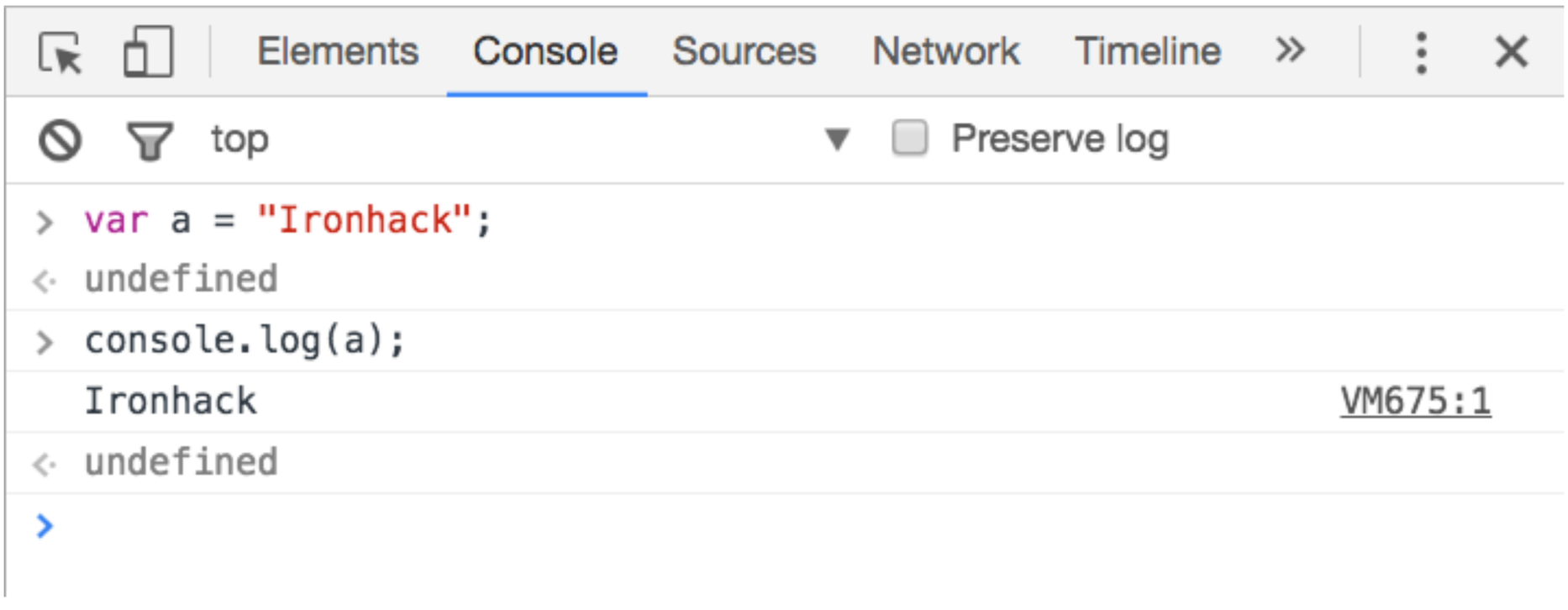
We can also interact with the console to debug our code. The quickest way to do that is by using the **`console` object**.

The `console` object enables us to use many different features, such as: printing values, clearing the console, throwing errors, and logging the current time.

How to use: `console.log`

We have been using the `console.log` method since the very beginning, so we are already familiar with it at this point. The `console.log` method allows us to print values into the JavaScript console. By printing values at different points in your code, you can inspect its behavior.

> ⚠️ Note that `console.log` **displays the parameters**, but **returns** `undefined`.



Debugging in the browser

To follow the rest of this lesson, **clone this repo**:

```
$ git clone https://github.com/ironhack-labs/lab-javascript-debugging-error-and-js-hint
```

Explain this code

The Chrome DevTools debugger

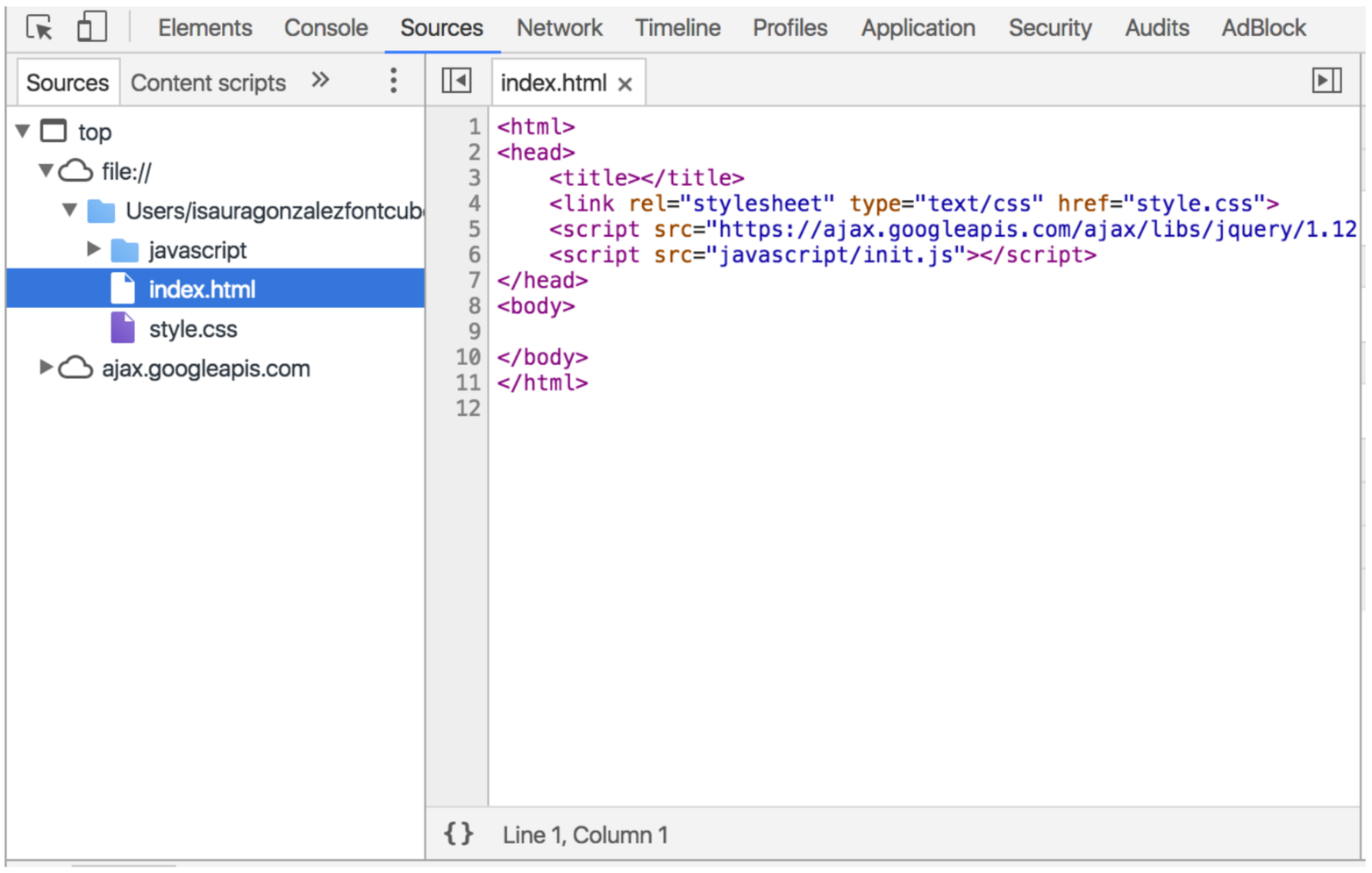
> Chrome DevTools is a set of web developer tools built directly into the Google Chrome browser. DevTools can help you edit pages on-the-fly and diagnose problems quickly, which ultimately helps you build better websites, faster.

The DevTools include gadgets organized in panels or tabs:

- **Elements**: HTML DOM & CSS Styles
- **Console**: JavaScript REPL
- **Network**: Inspect Network requests
- **Profiles**: Performance monitoring by inspecting the memory use of our application
- **Application**: Access to the browser services (LocalStorage, cookies, Cache Storage, Service Workers, etc.)
- **Timeline**: Records how our page loads in time
- **Sources**: All resources of a page (HTML, CSS, JavaScripts, images, etc.)

The Sources Tab

Another useful tool that Chrome DevTools offer us is the Sources Tab. In this section, you will be able to inspect the files loaded in your web app, either local or external and explore them to debug and optimize your code.

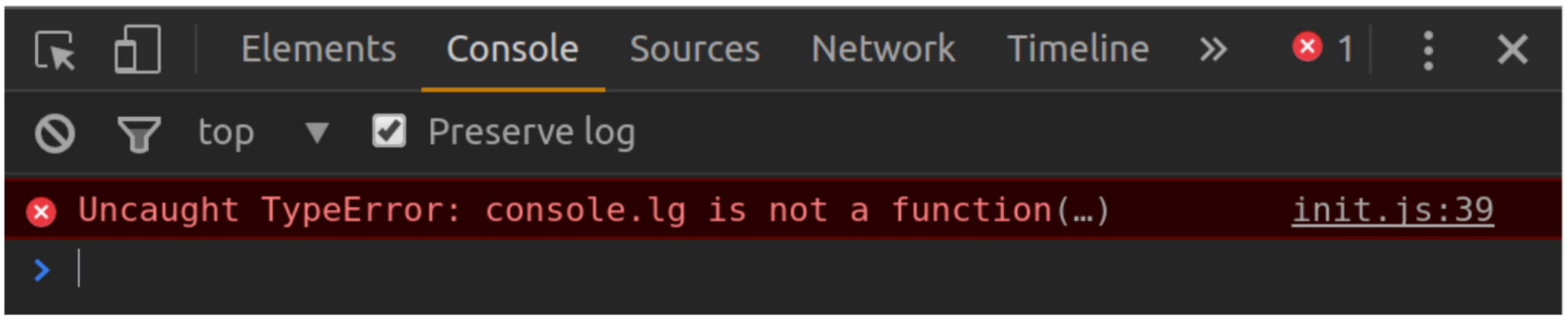


Load the files in your cloned project by opening the `index.html` in Google Chrome. Open the DevTools and inspect in the **Sources tab** the HTML, CSS and JavaScript files.

The Sources tab lets us inspect the files we are executing. DevTools have some functionality that will help us to identify and fix errors.

Errors displayed in `console`

When there's an error while our script is executing, the console will show us the error:



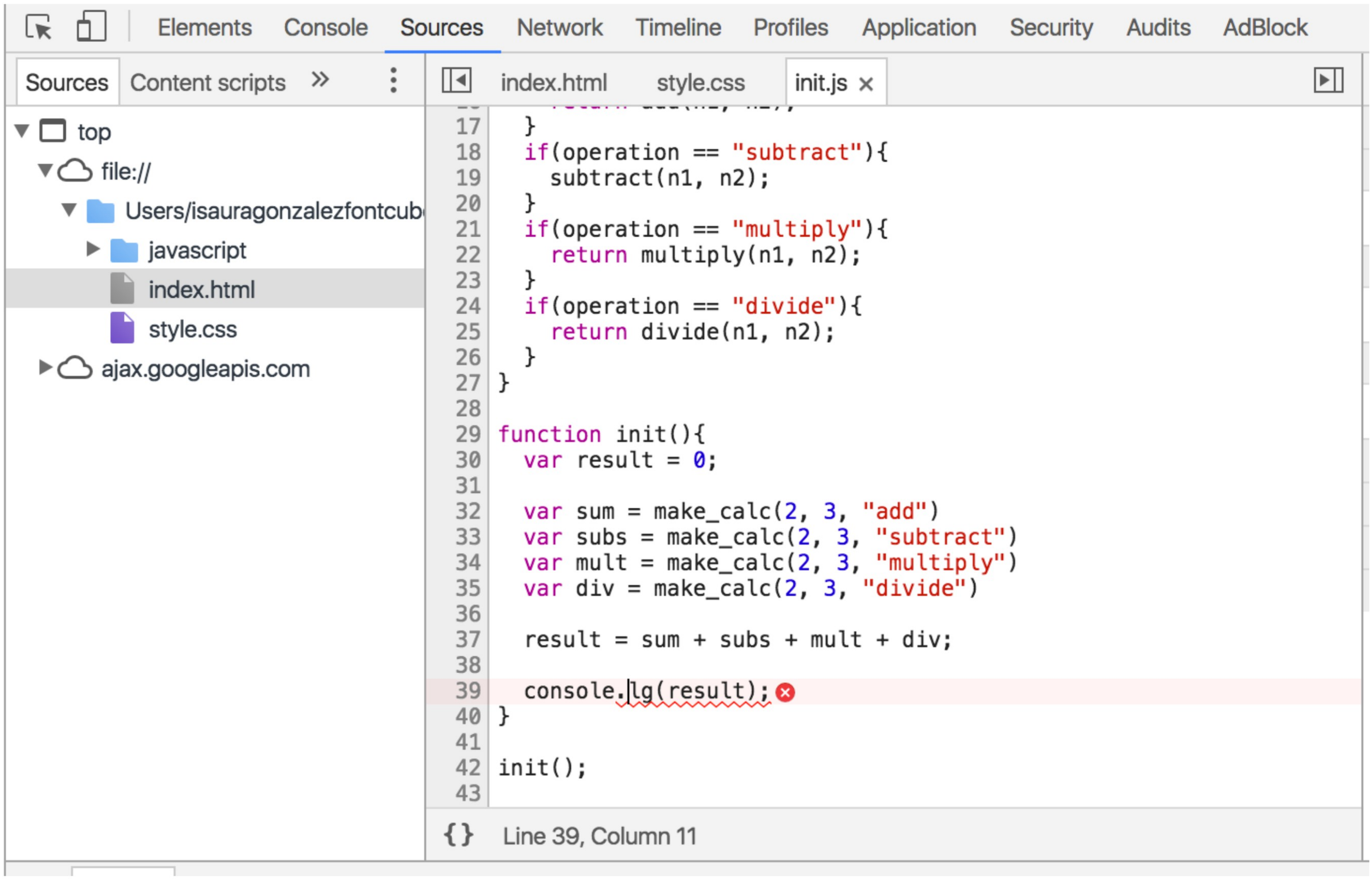
If we take a closer look, we can see at least 2 elements in every error:

- The **error exception:** `Uncaught TypeError: console.lg is not a function(…)` is **what** happens
- The location **where** it happened: `init.js:39`, the file and line number where the error exception was thrown.

💡 Detecting, reading and troubleshooting are all very important skills. Try to read errors carefully and we guarantee that in some errors you'll find at least a half of the solution. Knowing how to read the errors will boost your learning process greatly.

We can troubleshoot this error by clicking on the location. It will act as a link and redirect us to the *Sources Tab*, where the error will be shown at the exact location.

- **Click on the location to display the file**



Execution control: Breakpoints

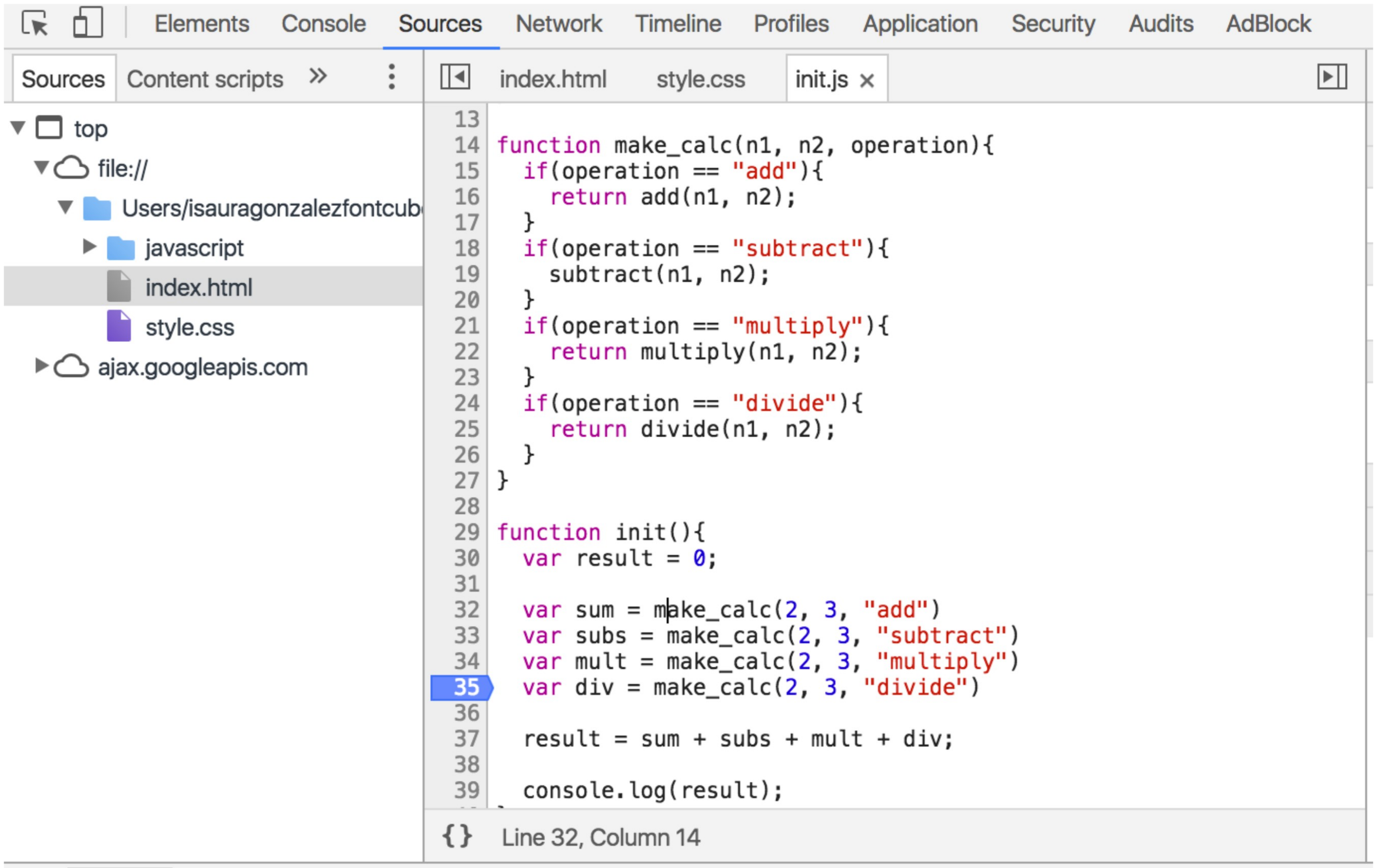
In the previous example, we can easily confirm this as a typo in a `console.log()` call. But what happens if the error is not so obvious? For example: **an error in a built-in JavaScript function**, or **a variable with an `undefined` value.**

In our example, our `console.log` of the `result` variable gives us `NaN` (Not a Number). If we wanted to figure out why we have to do some detective work.

Testing a variable's value by using a lot of `console.log` calls is not efficient. The Chrome DevTools provide **Breakpoints** to inspect the status of a variable in a particular time of the execution.

> 💡 **Breakpoints** are like flags placed in our JavaScript code that allow us to stop the execution once we are in a particular line.

Let's add a breakpoint after the multiplication by selecting the *Sources* tab and clicking on the 35th line. This will stop the execution right in that point:

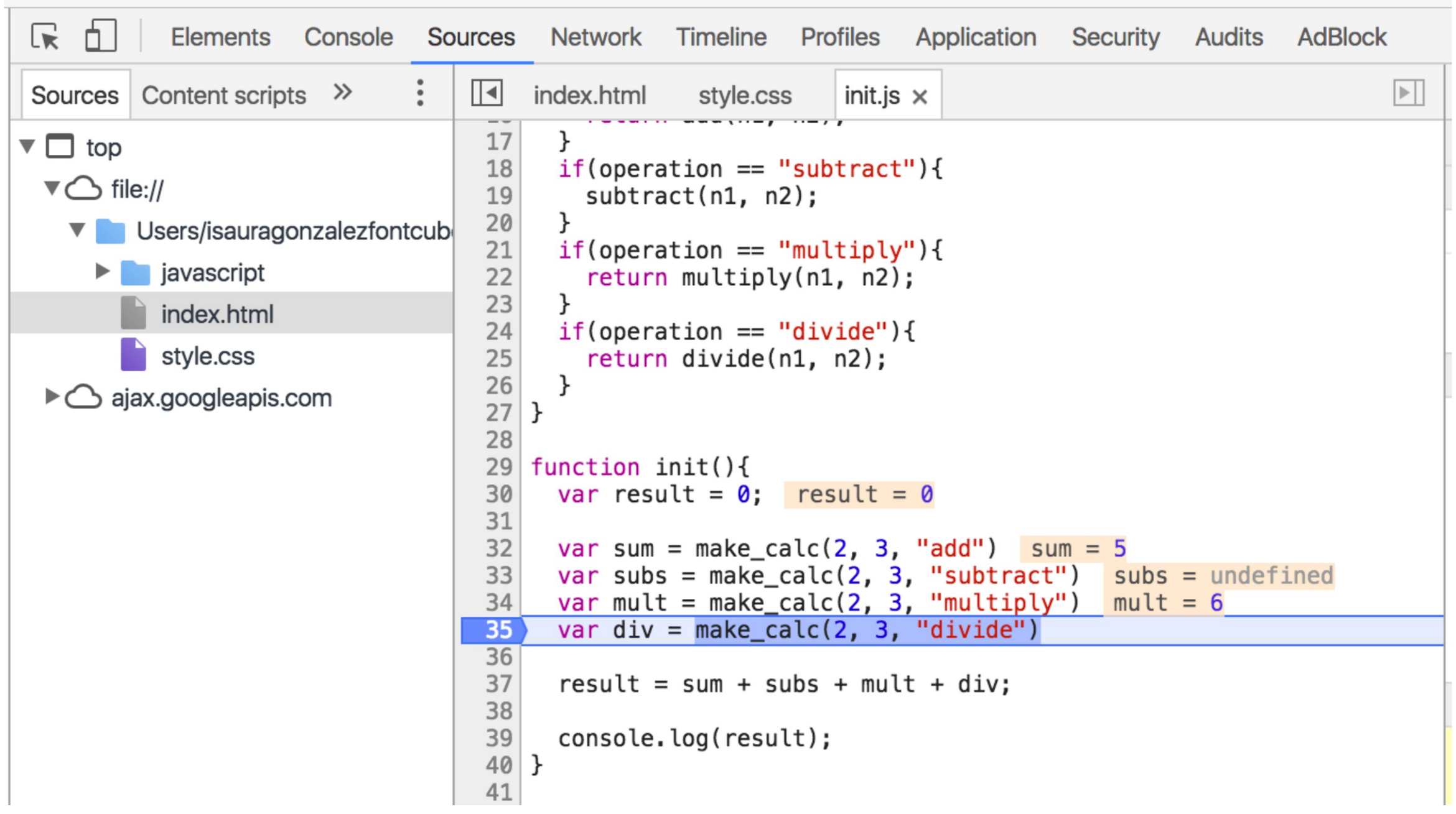


When the execution is stopped, use the console to get the value of `sum` and `result`. What happens?

```
console.log(sum);
console.log(result);
```

Explain this code

In the code panel you can see the value of the variables at that point (notice the `div` variable doesn't exist yet):



```
42 init();
43
```

{} Line 35, Column 13

💡 Execution is always stopped **right before** running the line where the breakpoint is.

The execution control panel

The DevTools debugger has a control panel that allows us to control how the execution continues. We can continue the execution after the breakpoint, skip a little ahead after the breakpoint or step into a function call.

- **RESUME**: Resumes execution up to the next breakpoint. If no breakpoint is encountered, normal execution is resumed.

- **STEP OVER:** Executes whatever happens on the current line and jumps to the next line.

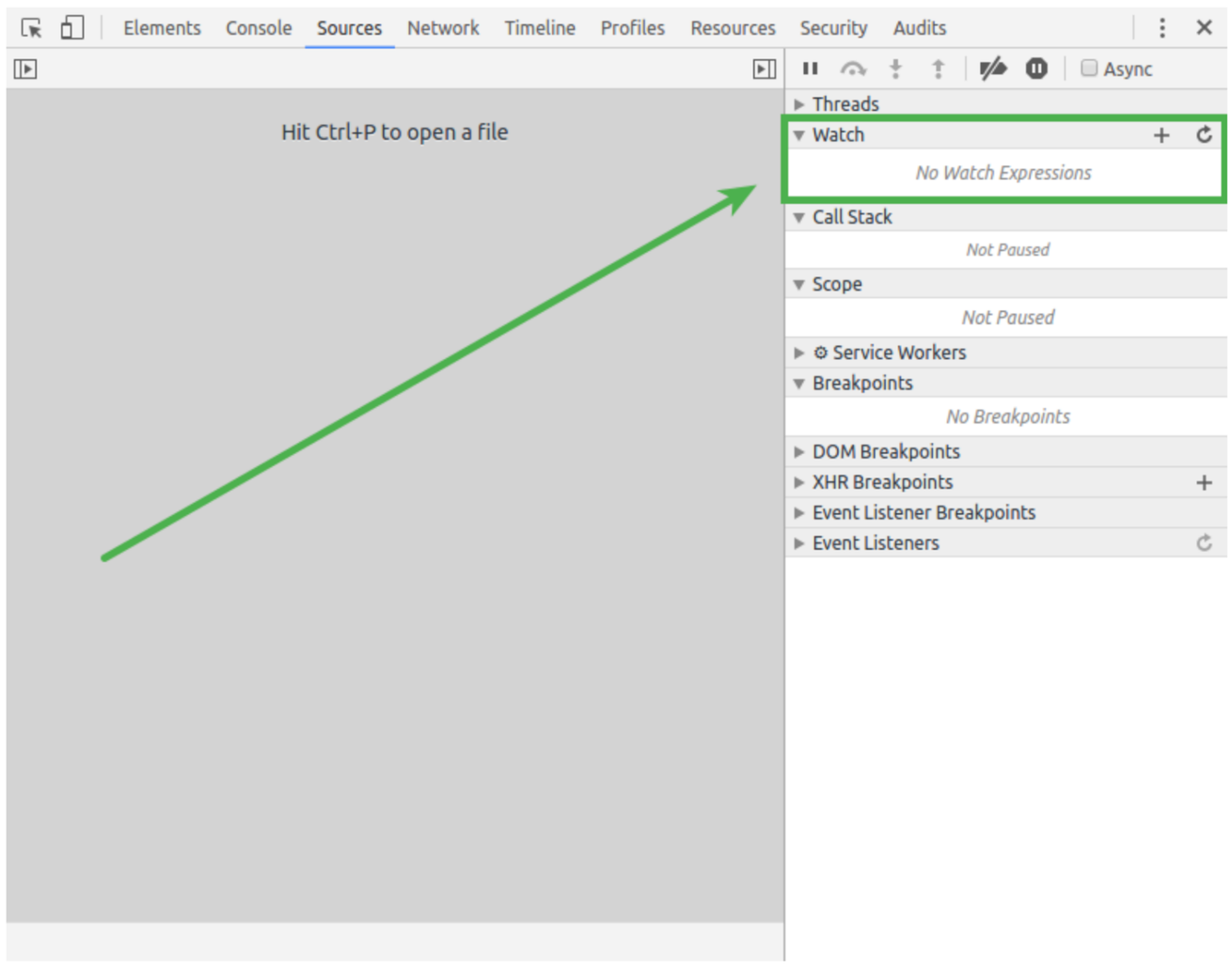
- **STEP INTO:** If the next line contains a function call, *Step Into* will jump to and pause that function at its first line.

- **STEP OUT:** Executes the remainder of the current function and then pauses at the next statement after the function call.

- **DEACTIVATE BREAKPOINTS**: Temporarily disables all breakpoints. Use to resume full execution without actually removing your breakpoints. Click it again to reactivate the breakpoints.

Execution control: Watch

Chrome DevTools allow you to easily see multiple variables throughout your application. The *Sources* tab allows you to **watch variables** within your application.

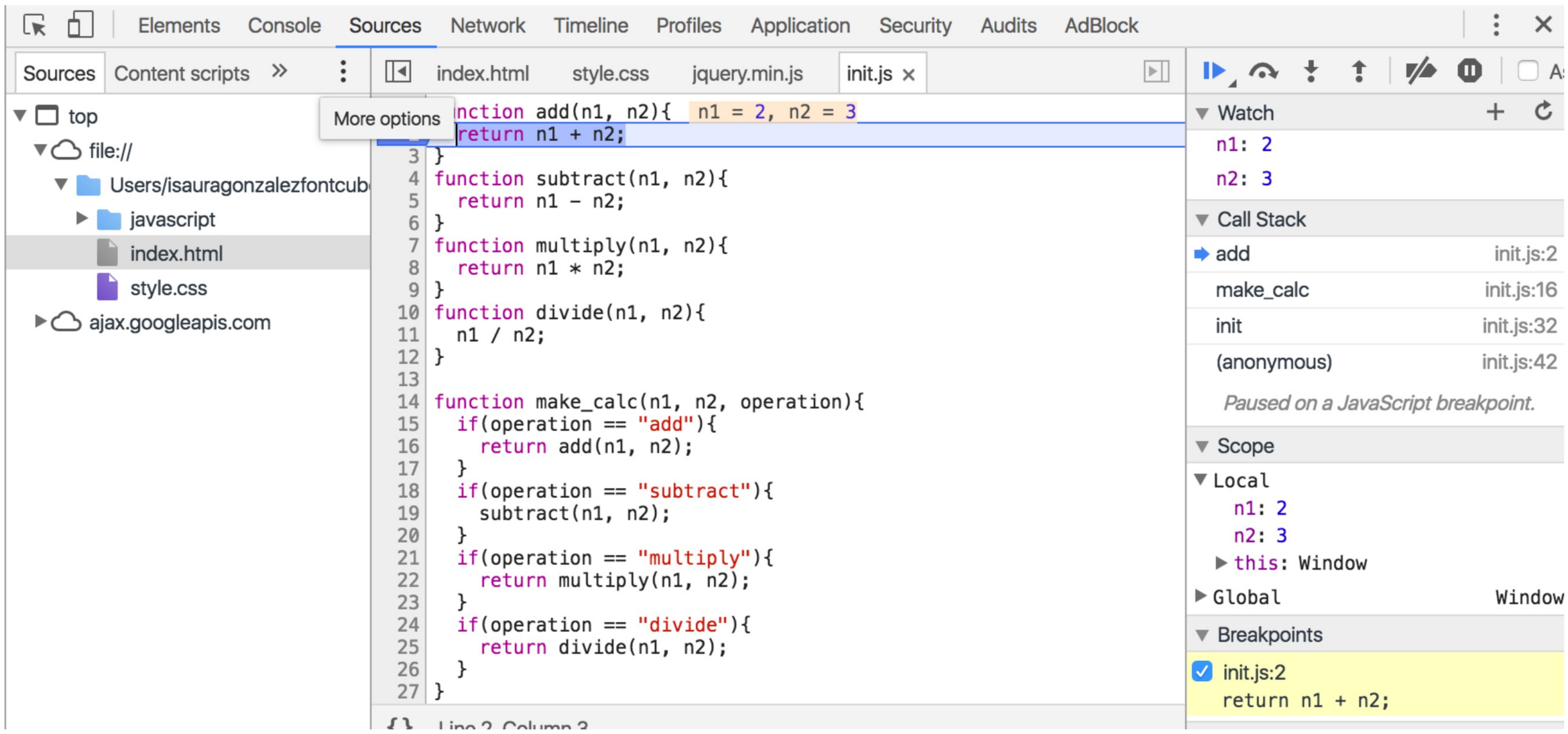


By taking advantage of this functionality we can remove all of our `console.log` calls.

Adding variables

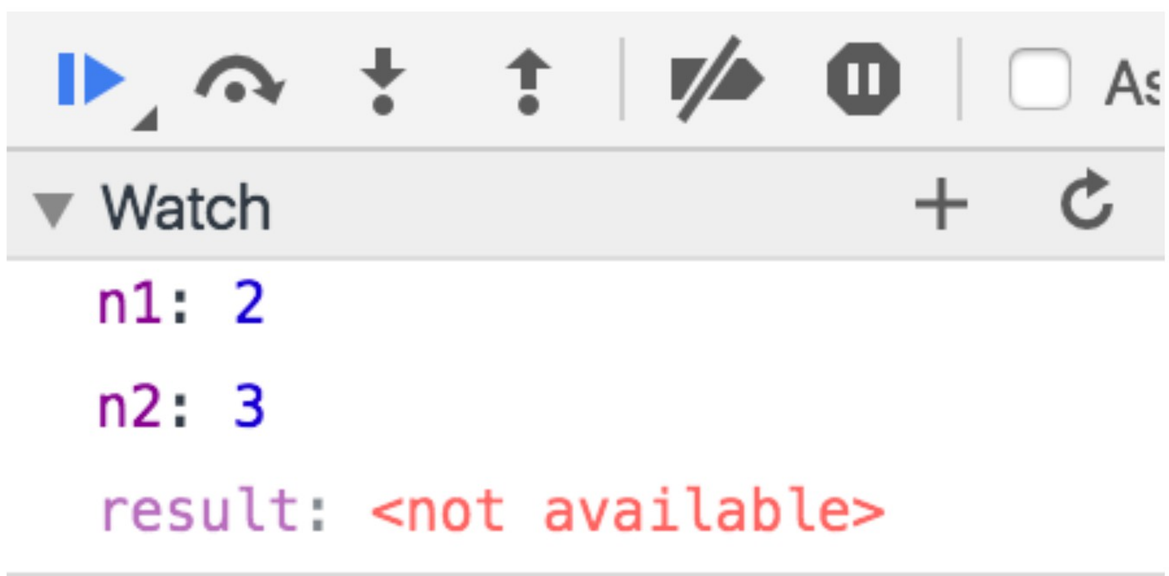
To add a variable to the watch list, use the **+** icon to the right of the section heading. This will open an inline input where you should provide the variable name you want to watch.

The watcher will show you the current value of the variable as it is added.



👍 If you remove all the breakpoints we had before, add a break point to any of the functions and add the variables `n1` and `n2` to **Watch** you will see the values of variables. You can inspect the objects and arrays as if you were in the console.

ℹ️ If the variable is not set or can't be found during the execution it will show `Not Available` as value.



Time to practice

Add `sum` and `subs` to your *Watch* list and make a breakpoint in the correct place to see their values.

Execution control: Call Stack

> The **`Call Stack`** is a *Chrome Dev Tools* functionality that allows us to trace the history of a function call.

In JavaScript, `function a` can call `function b`, and `function b` can call `function c`. When `function c` returns, it will return to the context of `function b` and so on.

Sometimes when we are debugging and we place a breakpoint in our code, it's useful to know which function called the current function we are executing.

> 💡 The **Call Stack** is an internal data structure that keeps track of the active functions in a program.

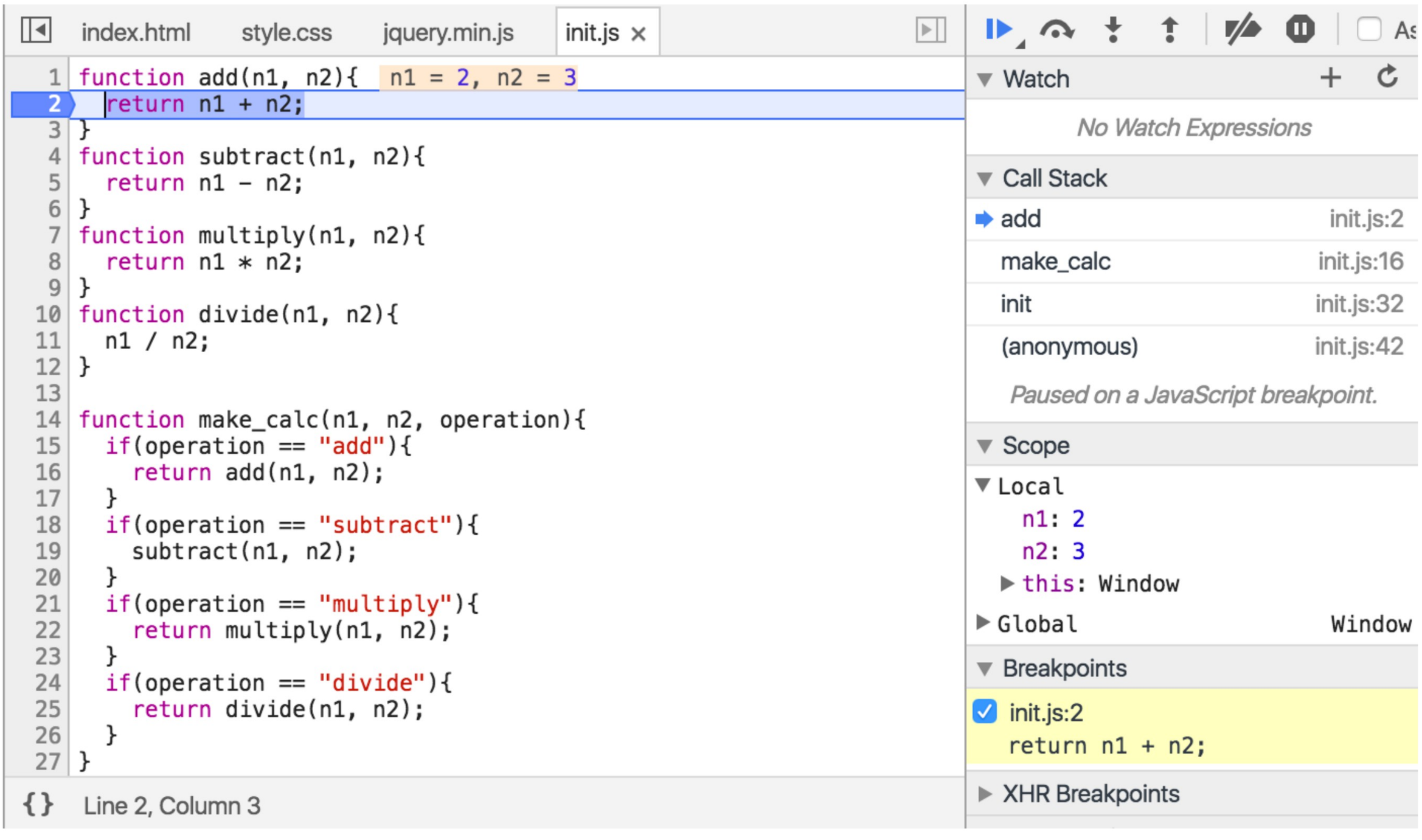
As an example, let's place a `break point` inside the `function add` and execute the code.

Once this `break point` stops the execution, we can see in the `Call Stack` section the line where the function was called and the function who called it, along with the line where that happened.

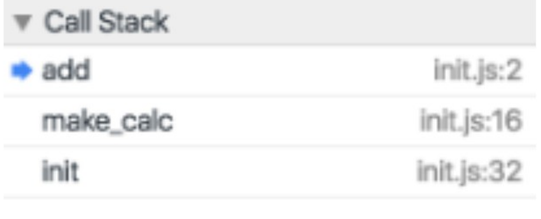
```
function add(n1, n2) {
  return n1 + n2;
}

function make_calc(n1, n2, operation) {
  if (operation == 'add') {
    return add(n1, n2);
  }
}

function init() {
  var result = make_calc(2, 3, 'add');
  console.log(result);
}

init();
```

Explain this code

When we try to execute this piece of code, the result is going to be printed once the operation is set and the calculation is made and returned. Let's see the trace of the `add` function:



We can see in the `Call Stack` how the trace is made from the last function (`add`) until the very first (`init`).



(anonymous) init.js:42

Try... Catch Statement

The **try ... catch** statement executes a piece of JavaScript code to `try`.

If any of the specified errors are thrown, the `catch` block created for that error would be executed instead.

```
try {
  myroutine();
} catch (error) {
  // ...
  logMyErrors(error);
} finally {
  // ...
}
```

Explain this code

- **Try:** the statement to be executed
- **Catch:** the statement that is executed if an exception is thrown in the try block.
- **Exception_var (`error`):** an identifier to hold an exception object for the associated catch clause.
- **Finally_statements (OPTIONAL):** statements that are executed after the try statement completes. These statements execute regardless of whether or not an exception was thrown or caught.

Trying to execute this code, we will see the generated output in the console:

```
function powerOf3(num) {
  return number ** 3;
}

try {
  powerOfThree(9);
} catch (error) {
  console.log("Gets executed if there's an error.");
  console.log(error);
} finally {
  console.log('Gets executed at the end no matter what.');
}
```

Explain this code

You can see how there's a typo with the `num` variable on **line 2**. This causes a `ReferenceError`. Since we are `catching` the errors, we can choose how to handle the error instead of displaying the predefined error that stops the execution of the app.

A `catch` clause contains statements that specify what to do if an exception is thrown in the try block. That is, you want the try block to succeed, and if it doesn't succeed, you want the control to pass to the catch block.

If any statement within the try block (or in a function called from within the try block) throws an exception, control immediately shifts to the catch clause. If no exception is thrown in the try block, the catch clause is skipped.

The `finally` clause executes after the try block and catch clause(s) execute but before the statements following the try statement. It always executes, regardless of whether or not an exception was thrown or caught.

Exceptions types

These are the primary exceptions for JavaScript:

- **RangeError:** Raised when a numeric variable exceeds its allowed range.

```
var goodArray = new Array(3);
console.log(goodArray);
// No error => [undefined × 3]

var badArray = new Array(-1);
// Error => Uncaught RangeError: Invalid array length(…)
```

Explain this code

- **ReferenceError:** Raised when an invalid reference is used.

```
var name = 'Ted';
console.log(nme);
// Uncaught ReferenceError: nme is not defined
```

Explain this code

- **SyntaxError:** Raised when a syntax error occurs while parsing JavaScript code.

```
function doSomething(){
  console.log("I'm doing something!);

// Uncaught SyntaxError: Unexpected end of input

```

Explain this code

- **TypeError:** Raised when the type of a variable is not as expected.

```
var name = 'Bob';
// (toFixed is a method for numbers)
name.toFixed();
// Uncaught TypeError: name.toFixed is not a function
```

Explain this code

- **URIError:** Raised when the encodeURI() or decodeURI() functions are used in an incorrect manner.

```
decodeURI('http:/google.com%');
// Uncaught URIError: URI malformed(…)
```

Explain this code

- **EvalError:** Raised when the eval() function is used in an incorrect manner.

Throw errors

When an error occurs, JavaScript will normally stop and generate an error message. The technical term for this is: **JavaScript will throw an exception (throw an error)**. The `throw` statement allows you to create a custom error.

> JavaScript will actually create an Error object with two properties: name and message.

We recommend using `throw` together with try and catch, so you can control program flow and generate custom error messages.

```
function getUserName(user) {
  if (!user.name) {
    throw new SyntaxError('Incomplete data: no name');
  }
  return user.name;
}

try {
  getUserName({ name: 'Raul', lastName: 'Lopez' });
} catch (error) {
  console.log("Gets executed if there's an error.");
  console.log(error);
} finally {
  console.log('Gets executed at the end no matter what.');
}
```

Explain this code

JS Hint

JSHint is a tool used for analyzing JavaScript source code and warns about quality problems.

To use it, we suggest just simply visiting **https://jshint.com/** and paste your code there.
Also you should add *jshint* extension in your VS Code editor.
Later, in the second module, you can install the **node-jshint** npm package - we will talk about that later on.

As the main takeaway, we encourage you to check your code as often as possible on the official *jshint* website since you will save yourself a lot of time when debugging.

If you would like to know more about JSHint, check out this **page**.

Debug Examples

Your turn

As you might notice, the *calculatorApp* you cloned before has a lot of errors.
Remember to use all the tools that **Chrome Dev Tools** provide, and check if everything is working once you've fixed the error.
The app should display the sum of the results of the four operations.

Summary

In this lesson, we learned how the **Chrome Dev Tools** work. Its features are:

- **The `console`** - `console.log` - Error stack trace
- **Sources** - `Break Points` - To stop the execution of your app at any certain line. - `Watch` - For variables to have their value at any moment - `Call Stack` - For functions to trace their execution path
- **Try... catch**
- **Throw**

- **JS Hint**

Debugging is one of the most important skills you will gain out of this course. Quick debugging and following errors is *vital* to your efficiency as a developer.

These are just tools. But these processes will take significant time and practice... and as a junior developer you will get a *ton* of debugging experience :)

> 👍 **Remember to read the information contained in the errors!**

Extra Resources

- Chrome Dev Tools - Code School
- learn-jshint

Mark as completed

PREVIOUS
JS | Variable scope, hoisting and shadowing

NEXT
JS | Async & Callbacks →