EN

EPUB/PDF

The JavaScript language → Code quality

16th January 2020

Debugging in Chrome

Before writing more complex code, let’s talk about debugging.

Debugging is the process of finding and fixing errors within a script. All modern browsers and most other environments support debugging tools – a special UI in developer tools that makes debugging much easier. It also allows to trace the code step by step to see what exactly is going on.

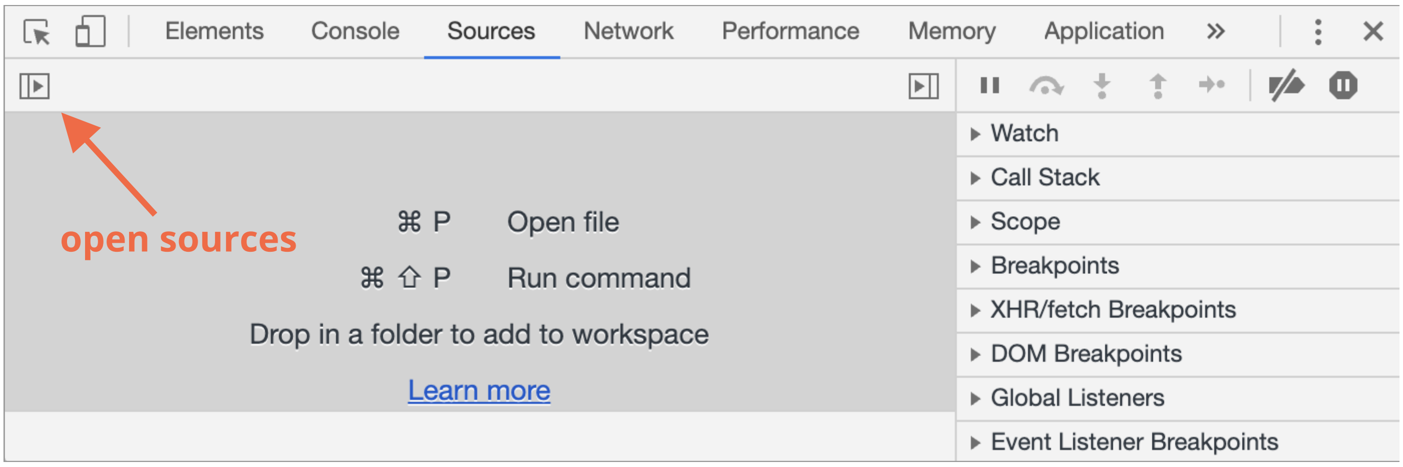
We’ll be using Chrome here, because it has enough features, most other browsers have a similar process.

The “Sources” panel

Your Chrome version may look a little bit different, but it still should be obvious what’s there.

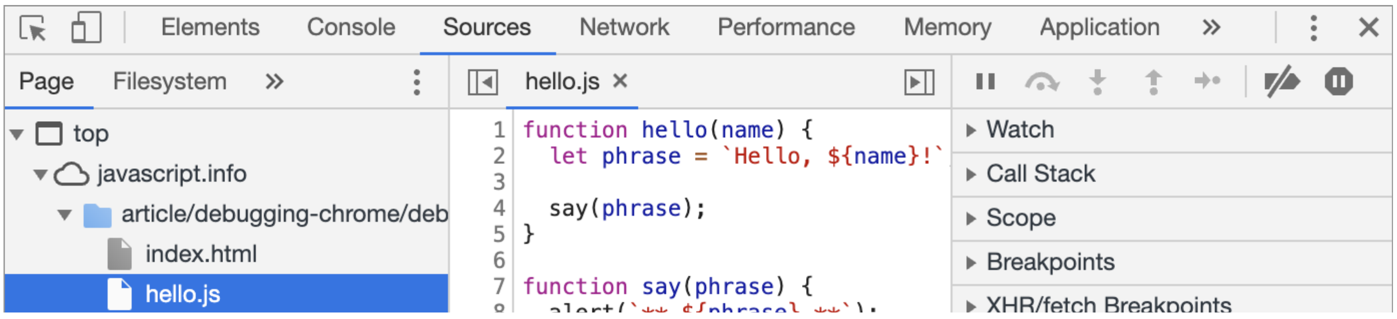
- Open the example page in Chrome.
- Turn on developer tools with `F12` (Mac: `Cmd+Opt+I`).
- Select the `Sources` panel.

Here’s what you should see if you are doing it for the first time:

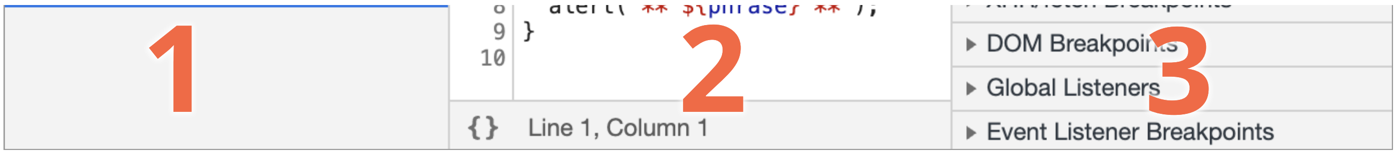


The toggler button opens the tab with files.

Let’s click it and select `hello.js` in the tree view. Here’s what should show up:



```
function hello(name) {
  let phrase = `Hello, ${name}!`

  say(phrase);
}

function say(phrase) {
```



The Sources panel has 3 parts:

1. The **File Navigator** pane lists HTML, JavaScript, CSS and other files, including images that are attached to the page. Chrome extensions may appear here too.
2. The **Code Editor** pane shows the source code.
3. The **JavaScript Debugging** pane is for debugging, we'll explore it soon.

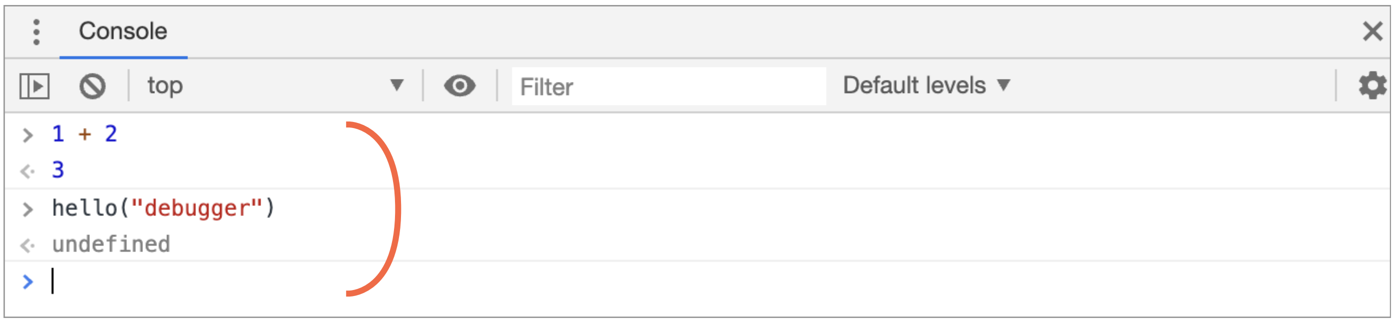
Now you could click the same toggler again to hide the resources list and give the code some space.

Console

If we press `Esc`, then a console opens below. We can type commands there and press `Enter` to execute.

After a statement is executed, its result is shown below.

For example, here `1+2` results in `3`, and `hello("debugger")` returns nothing, so the result is `undefined`:



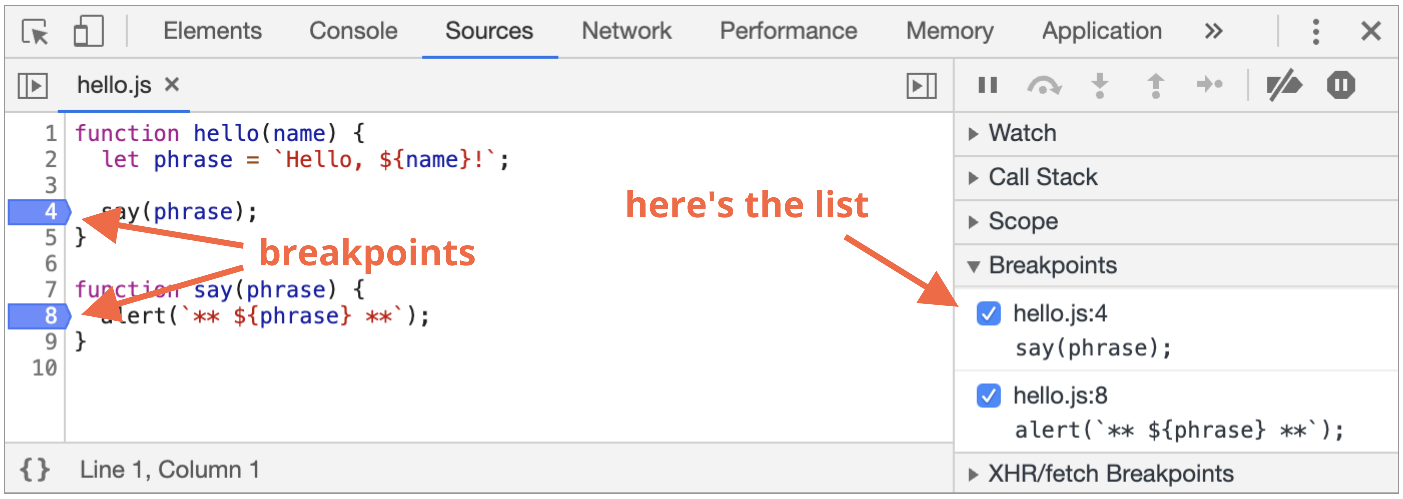
```
> 1 + 2
<· 3
> hello("debugger")
<· undefined
>
```

Breakpoints

Let's examine what's going on within the code of the example page. In `hello.js`, click at line number `4`. Yes, right on the `4` digit, not on the code.

Congratulations! You've set a breakpoint. Please also click on the number for line `8`.

It should look like this (blue is where you should click):



```
1  function hello(name) {
2    let phrase = `Hello, ${name}!`;
3
4    say(phrase);
5  }
6
7  function say(phrase) {
8    alert(`** ${phrase} **`);
9  }
10
```

☑ hello.js:4
say(phrase);
☑ hello.js:8
alert(`** ${phrase} **`);

A *breakpoint* is a point of code where the debugger will automatically pause the JavaScript execution.

While the code is paused, we can examine current variables, execute commands in the console etc. In other words, we can debug it.

We can always find a list of breakpoints in the right panel. That's useful when we have many breakpoints in various files. It allows us to:

- Quickly jump to the breakpoint in the code (by clicking on it in the right panel).
- Temporarily disable the breakpoint by unchecking it.
- Remove the breakpoint by right-clicking and selecting Remove.
- …And so on.

Conditional breakpoints

Right click on the line number allows to create a *conditional* breakpoint. It only triggers when the given expression is truthy.

That's handy when we need to stop only for a certain variable value or for certain function parameters.

Debugger command

We can also pause the code by using the `debugger` command in it, like this:

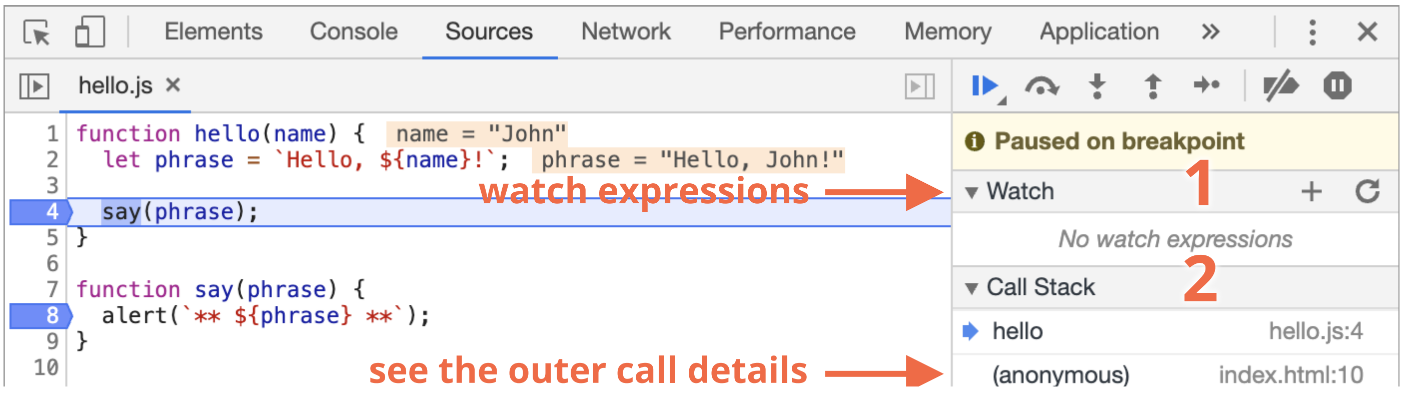
```
function hello(name) {
  let phrase = `Hello, ${name}!`;

  debugger;  // <-- the debugger stops here

  say(phrase);
}
```

That's very convenient when we are in a code editor and don't want to switch to the browser and look up the script in developer tools to set the breakpoint.

Pause and look around

In our example, `hello()` is called during the page load, so the easiest way to activate the debugger (after we've set the breakpoints) is to reload the page. So let's press `F5` (Windows, Linux) or `Cmd+R` (Mac).

As the breakpoint is set, the execution pauses at the 4th line:



```
 1 function hello(name) {   name = "John"
 2   let phrase = `Hello, ${name}!`;   phrase = "Hello, John!"
 3
 4   say(phrase);
 5 }
 6
 7 function say(phrase) {
 8   alert(`** ${phrase} **`);
 9 }
10
```

Paused on breakpoint

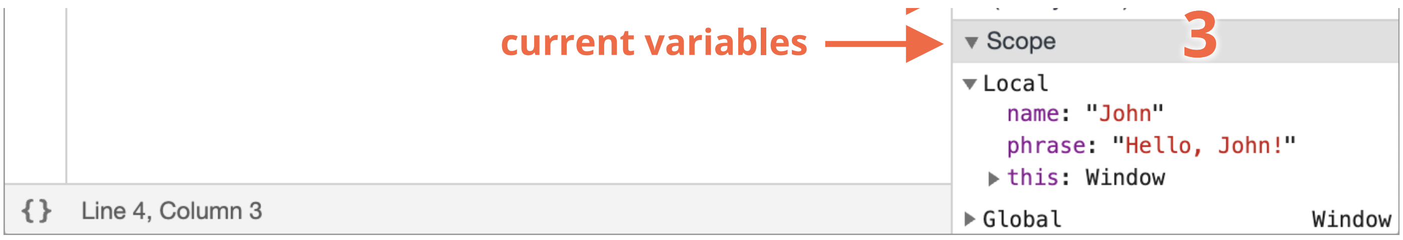
Watch

No watch expressions

Call Stack

hello hello.js:4

(anonymous) index.html:10



```
▼Local
   name: "John"
   phrase: "Hello, John!"
  ▶this: Window
▶Global                          Window
```

Please open the informational dropdowns to the right (labeled with arrows). They allow you to examine the current code state:

1. **`Watch` – shows current values for any expressions.**

   You can click the plus `+` and input an expression. The debugger will show its value at any moment, automatically recalculating it in the process of execution.

2. **`Call Stack` – shows the nested calls chain.**

   At the current moment the debugger is inside `hello()` call, called by a script in `index.html` (no function there, so it's called "anonymous").

   If you click on a stack item (e.g. "anonymous"), the debugger jumps to the corresponding code, and all its variables can be examined as well.

3. **`Scope` – current variables.**

   `Local` shows local function variables. You can also see their values highlighted right over the source.

   `Global` has global variables (out of any functions).

   There's also `this` keyword there that we didn't study yet, but we'll do that soon.

Tracing the execution

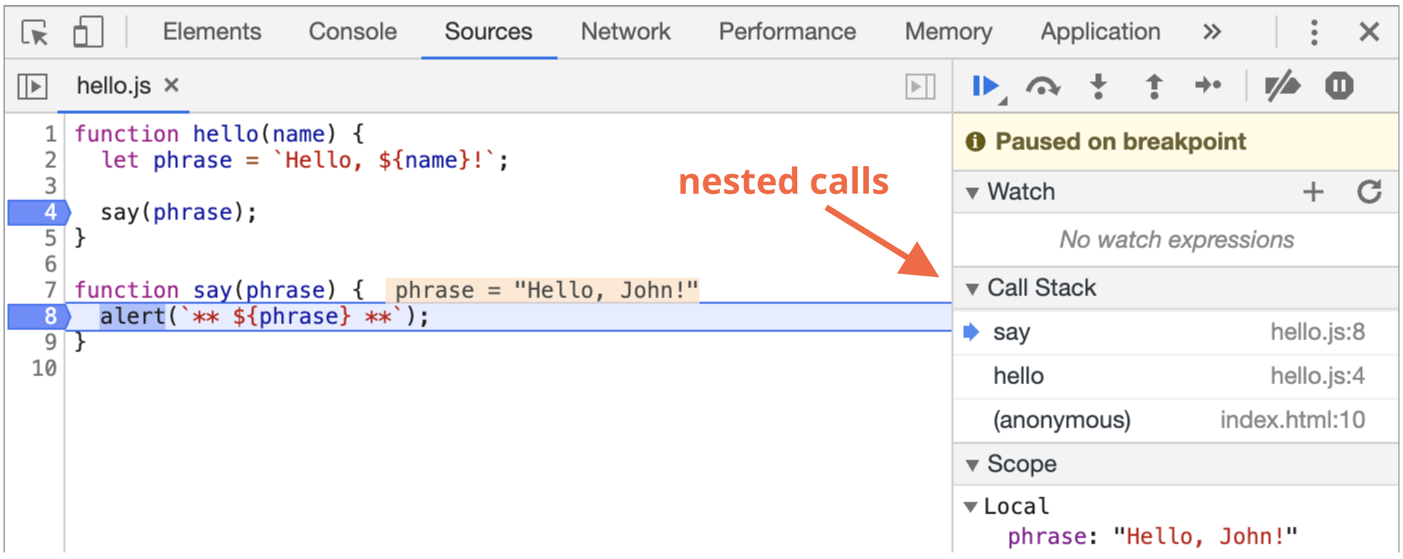
Now it's time to *trace* the script.

There are buttons for it at the top of the right panel. Let's engage them.

– "Resume": continue the execution, hotkey `F8`.

Resumes the execution. If there are no additional breakpoints, then the execution just continues and the debugger loses control.

Here's what we can see after a click on it:



```
function hello(name) {
  let phrase = `Hello, ${name}!`;

  say(phrase);
}

function say(phrase) {  phrase = "Hello, John!"
  alert(`** ${phrase} **`);
}

```

```
say            hello.js:8
hello          hello.js:4
(anonymous)    index.html:10
```

```
▼Local
   phrase: "Hello, John!"
```

```
{}  Line 8, Column 3
```

```
▶ this: Window
▶ Global                Window
```

The execution has resumed, reached another breakpoint inside `say()` and paused there. Take a look at the "Call Stack" at the right. It has increased by one more call. We're inside `say()` now.

– "Step": run the next command, hotkey `F9`.

Run the next statement. If we click it now, `alert` will be shown.

Clicking this again and again will step through all script statements one by one.

– "Step over": run the next command, but *don't go into a function*, hotkey `F10`.

Similar to the previous the "Step" command, but behaves differently if the next statement is a function call. That is: not a built-in, like `alert`, but a function of our own.

The "Step" command goes into it and pauses the execution at its first line, while "Step over" executes the nested function call invisibly, skipping the function internals.

The execution is then paused immediately after that function.

That's good if we're not interested to see what happens inside the function call.

– "Step into", hotkey `F11`.

That's similar to "Step", but behaves differently in case of asynchronous function calls. If you're only starting to learn JavaScript, then you can ignore the difference, as we don't have asynchronous calls yet.

For the future, just note that "Step" command ignores async actions, such as `setTimeout` (scheduled function call), that execute later. The "Step into" goes into their code, waiting for them if necessary. See DevTools manual for more details.

– "Step out": continue the execution till the end of the current function, hotkey `Shift+F11`.

Continue the execution and stop it at the very last line of the current function. That's handy when we accidentally entered a nested call using , but it does not interest us, and we want to continue to its end as soon as possible.

– enable/disable all breakpoints.

That button does not move the execution. Just a mass on/off for breakpoints.

– enable/disable automatic pause in case of an error.

When enabled, and the developer tools is open, a script error automatically pauses the execution. Then we can analyze variables to see what went wrong. So if our script dies with an error, we can open debugger, enable this option and reload the page to see where it dies and what's the context at that moment.

Continue to here

Right click on a line of code opens the context menu with a great option called "Continue to here".

That's handy when we want to move multiple steps forward to the line, but we're too lazy to set a breakpoint.

Logging

To output something to console from our code, there's `console.log` function.

For instance, this outputs values from `0` to `4` to console:

```
// open console to see
for (let i = 0; i < 5; i++) {
  console.log("value,", i);
}
```

Regular users don't see that output, it is in the console. To see it, either open the Console panel of developer tools or press `Esc` while in another panel: that opens the console at the bottom.

If we have enough logging in our code, then we can see what's going on from the records, without the debugger.

Summary

As we can see, there are three main ways to pause a script:

1. A breakpoint.
2. The `debugger` statements.
3. An error (if dev tools are open and the button is "on").

When paused, we can debug – examine variables and trace the code to see where the execution goes wrong.

There are many more options in developer tools than covered here. The full manual is at https://developers.google.com/web/tools/chrome-devtools.

The information from this chapter is enough to begin debugging, but later, especially if you do a lot of browser stuff, please go there and look through more advanced capabilities of developer tools.

Oh, and also you can click at various places of dev tools and just see what's showing up. That's probably the fastest route to learn dev tools. Don't forget about the right click and context menus!

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)