

# How to step through your code



By Dave Gash

Dave is a Tech Writer



By Paul Bakaus

Open Web Developer Advocate at Google • Tools, Performance, Animation, UX



By Kayce Basques

Technical Writer for Chrome DevTools

**Warning:** This page is deprecated. At the top of each section, there's a link to an up-to-date page where you can find similar information.

By executing code one line or one function at a time, you can observe changes in the data and in the page to understand exactly what is happening. You can also modify data values used by the script, and you can even modify the script itself.

*Why is this variable value 20 instead of 30? Why doesn't that line of code seem to have any effect? Why is this flag true when it should be false?* Every developer faces these questions, and steps through code to find out.

After setting breakpoints, return to the page and use it normally until a breakpoint is reached. This pauses all JavaScript on the page, focus shifts to the DevTools Sources panel, and the breakpoint is highlighted. You can now selectively execute code and examine its data, step by step.

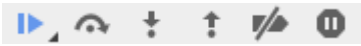
## TL;DR








- Step through code to observe issues before or while they happen and test out changes through live editing.
- Prefer stepping over console logging, as logged data is already stale the moment it arrives in the console.

- Enable the 'Async call stack' feature to gain greater visibility into the call stack of asynchronous functions.
- Blackbox scripts to hide third-party code from your call stacks.
- Use named functions rather than anonymous ones to improve call stack readability.

## Stepping in action

**Warning:** This page is deprecated. See [Step through code](#) for up-to-date information.

All step options are represented through clickable icons  in the sidebar, but can also be triggered via shortcut. Here's the rundown:

Icon/Button	Action	Description
	Resume	Resumes execution up to the next breakpoint. If no breakpoint is encountered, normal execution is resumed.
	Long Resume	Resumes execution with breakpoints disabled for 500ms. Convenient for momentarily skipping breakpoints that would otherwise continually pause the code, e.g., a breakpoint inside a loop. <b>Click and hold <i>Resume</i> until expands to show the action.</b>
	Step Over	Executes whatever happens on the next line and jumps to the next line.
	Step Into	If the next line contains a function call, <i>Step Into</i> 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.
	Pause on exceptions	Automatically pauses the code when an exception occurs.

Use **step into** as your typical "one line at a time" action, as it ensures that only one statement gets executed, no matter what functions you step in and out of.

Use Pause on exceptions when you suspect an uncaught exception is causing a problem, but you don't know where it is. When this option is enabled, you can refine it by clicking the

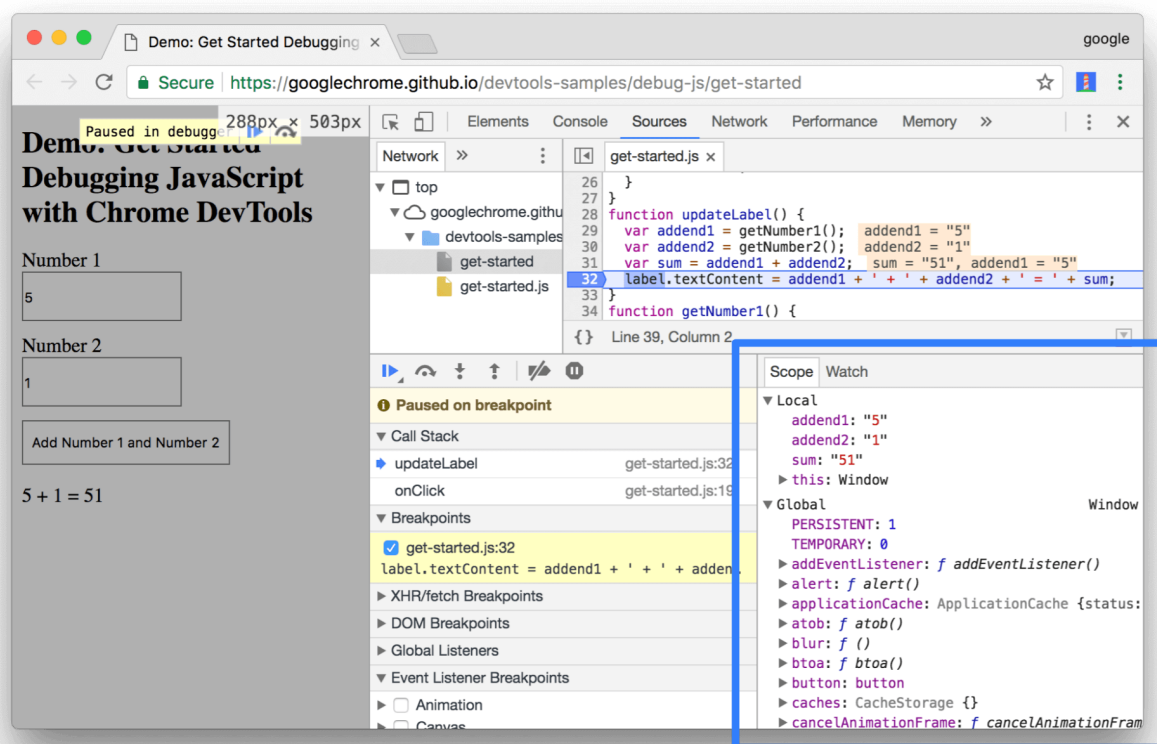
**Pause On Caught Exceptions** checkbox; in this case, execution is paused only when a specifically-handled exception occurs.

## View properties by scope

**Warning:** This page is deprecated. See [View and edit local, closure, and global properties](#) for up-to-date information.

When you pause a script, the **Scope** pane shows you all of the currently-defined properties at that moment in time.

The pane is highlighted in blue in the screenshot below.



The Scope pane is only populated when a script is paused. While your page is running, the Scope pane is empty.

The Scope pane shows you properties defined at the local, closure, and global levels.

If a property has a carat icon next to it, it means that it's an object. Click on the carat icon to expand the object and view its properties.

Sometimes properties are dimmed down. For example, the property `constructor` is dimmer than the `confirm` property in the screenshot below.

```
► confirm: function confirm()  
► console: Object  
► constructor: function Window()  
► createImageBitmap: function createImageBitmap()
```

The darker properties are enumerable. The lighter, dimmed down properties are not. See the following Stack Overflow thread for more information: [What do the colors mean in Chrome Developer Tools Scope panel?](#)

## The call stack

**Warning:** This page is deprecated. See [View the current call stack](#) for up-to-date information.

Near the top of the sidebar is the **Call Stack** section. When the code is paused at a breakpoint, the call stack shows the execution path, in reverse chronological order, that brought the code to that breakpoint. This is helpful in understanding not just where the execution is *now*, but how it got there, an important factor in debugging.

## Example

▼ Call Stack	<input type="checkbox"/> Async
setall	dgjs.js:4
setone	dgjs.js:18
onclick	(index):50

An initial onclick event at line 50 in the `index.html` file called the `setone()` function at line 18 in the `dgjs.js` JavaScript file, which then called the `setall()` function at line 4 in the same file, where execution is paused at the current breakpoint.

## Enable the async call stack

Enable the async call stack feature to gain more visibility into the execution of your asynchronous function calls.

1. Open the **Sources** panel of DevTools.
2. On the **Call Stack** pane, enable the **Async** checkbox.

The video below contains a simple script to demonstrate the async call stack feature. In the script, a third-party library is used to select a DOM element. A function called `onClick` is registered as the `onClick` event handler for the element. Whenever `onClick` is called, it in turn calls a function named `f`, which just forces the script to pause via the `debugger` keyword.



In the video, a breakpoint is triggered, and the call stack is expanded. There is only one call in the stack: `f`. The async call stack feature is then enabled, the script resumes, the breakpoint is triggered again, and then the call stack is expanded a second time. This time, the call stack contains all of the calls leading up to `f`, including third-party library calls, and the call to `onClick`. The first time that the script was called, there was only one call in the call stack. The second time, there were four. In short, the async call stack feature provides increased visibility into the full call stack of asynchronous functions.

## Tip: name functions to improve call stack readability

Anonymous functions make the call stack difficult to read. Name your functions to improve readability.

The code snippets in the two screenshots below are functionally equivalent. The exact functioning of the code is not important, what is important is that the code in the first screenshot uses anonymous functions, while the second uses named functions.

In the call stack in the first screenshot, the top two functions are both just titled (**anonymous function**). In the second screenshot, the top two functions are named, which makes it easier to understand the program flow at a glance. When you are working with numerous script files, including third-party libraries and frameworks, and your call stack is five or ten calls deep, it is much easier to understand the call stack flow when functions are named.

Call stack with anonymous functions:

The screenshot shows the Chrome DevTools interface. The 'Sources' panel is active, displaying the file 'anon.html'. The code in the editor is as follows:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>anon functions --> harder to read</title>
5   </head>
6   <body>
7     <button id="button">button</button>
8     <script src="http://code.jquery.com/jquery-1.11.3.min.js">
9   </script>
10  $(document).ready(function() {
11    $('#button').on('click', function() {
12      setTimeout(function() {
13        debugger;
14      }, 500);
15    });
16  });
17  </script>
18
```

The call stack is visible at the bottom of the window. It shows the following frames:

- Call Stack (Async)
 (anonymous function) anon.html:13
 setTimeout (async)
 (anonymous function) anon.html:12
 m.event.dispatch jquery-1.11.3.min.js:4
 m.event.add.r.handle jquery-1.11.3.min.js:4
- Breakpoints
 DOM Breakpoints
 XHR Breakpoints
 Event Listener Breakpoints

The 'Watch' panel is empty, displaying 'No Watch Expressions'.

Call stack with named functions:

The screenshot shows the Chrome DevTools interface. The top pane displays the source code of 'named.html'. The code is as follows:

```

3 <head>
4 <title>named functions --> easier to read</title>
5 </head>
6 <body>
7 <button id="button">button</button>
8 <script src="http://code.jquery.com/jquery-1.11.3.min.js"
9 </script>
10 $(document).ready(function() {
11   var f = function() {
12     debugger;
13   };
14   var onClick = function() {
15     setTimeout(f, 500);
16   };
17   $('#button').on('click', onClick);
18 });
19 </script>
20

```

The bottom pane shows the Call Stack with the following entries:

Function	File
f	named.html:12
<i>setTimeout (async)</i>	
onClick	named.html:15
m.event.dispatch	jquery-1.11.3.min.js:4
m.event.add.r.handle	jquery-1.11.3.min.js:4

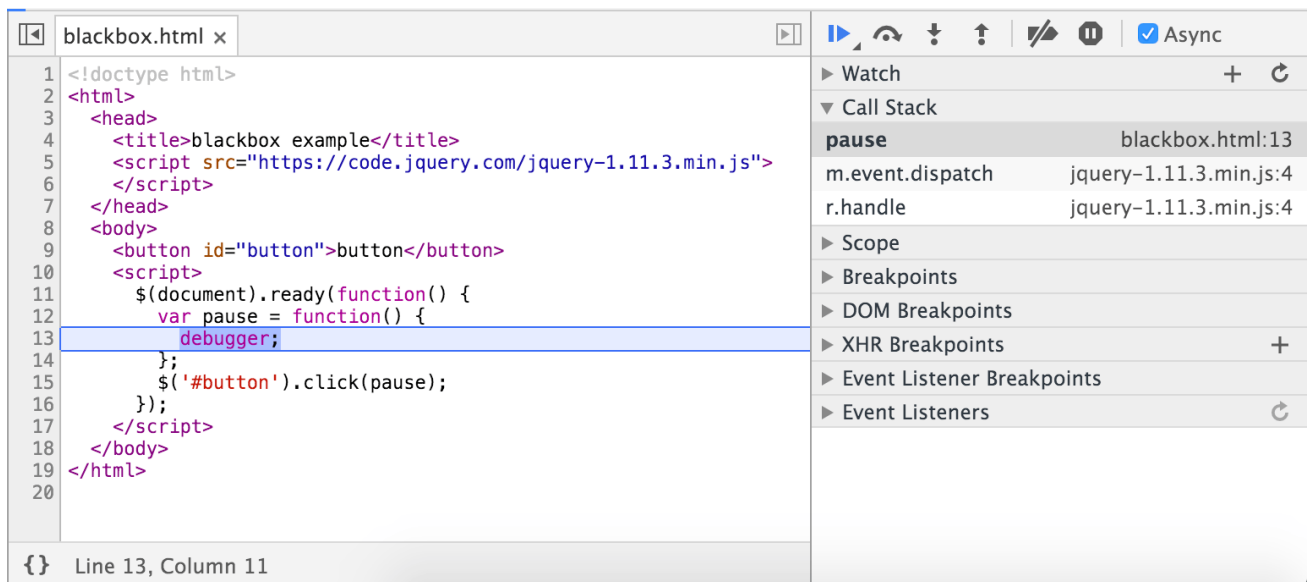
The 'Watch' pane is empty, showing 'No Watch Expressions'.

## Blackbox third-party code

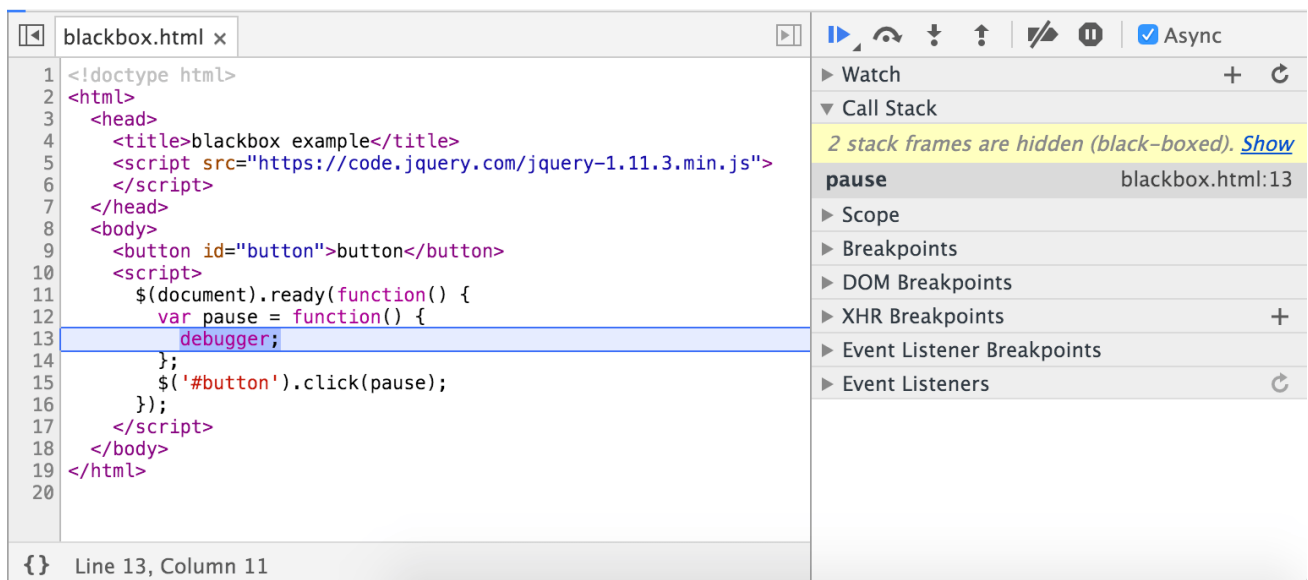
**Warning:** This page is deprecated. See [Ignore a script or pattern of scripts](#) for up-to-date information.

Blackbox script files to omit third-party files from your call stacks.

Before blackbox:



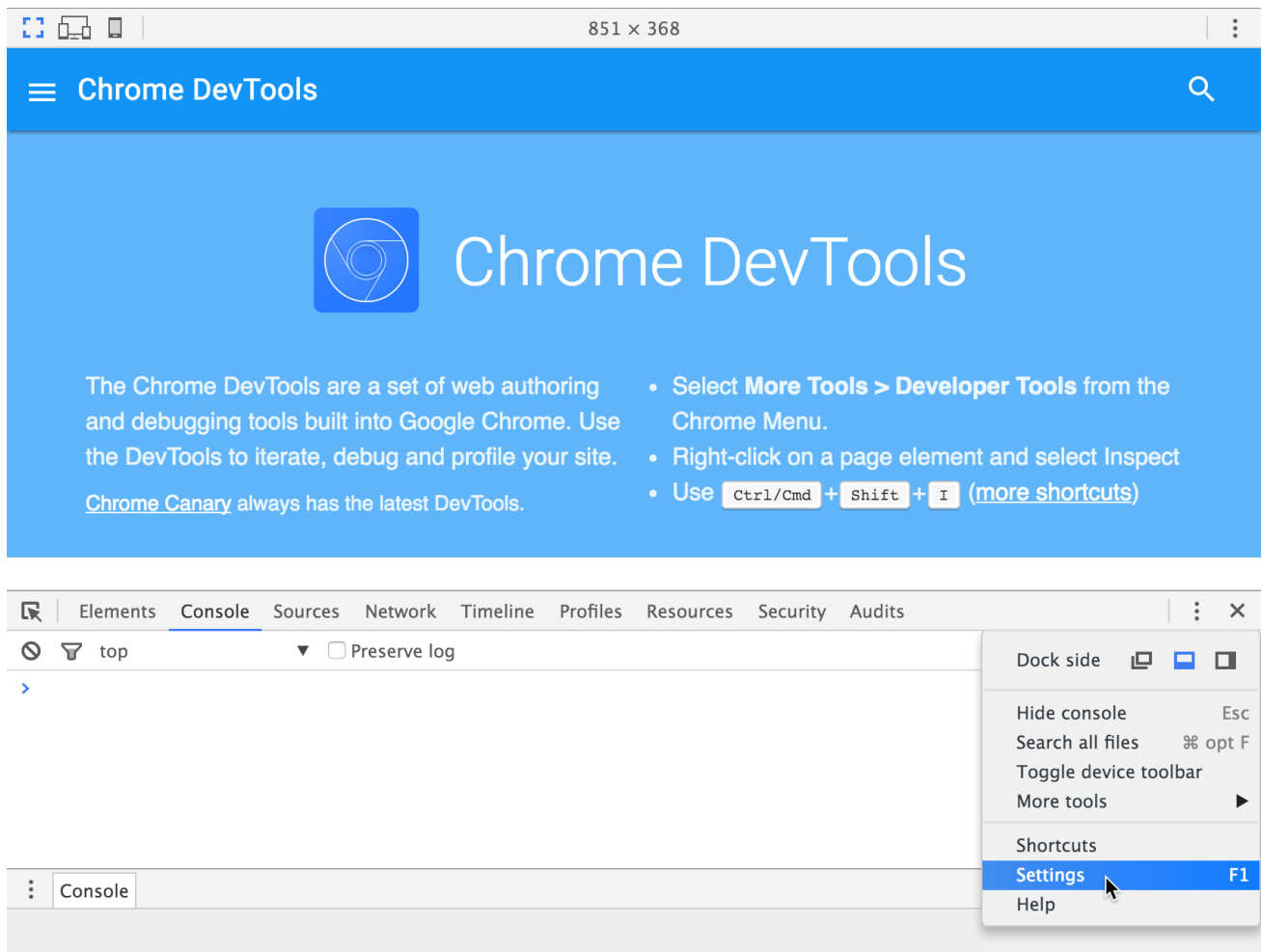
After blackbox:



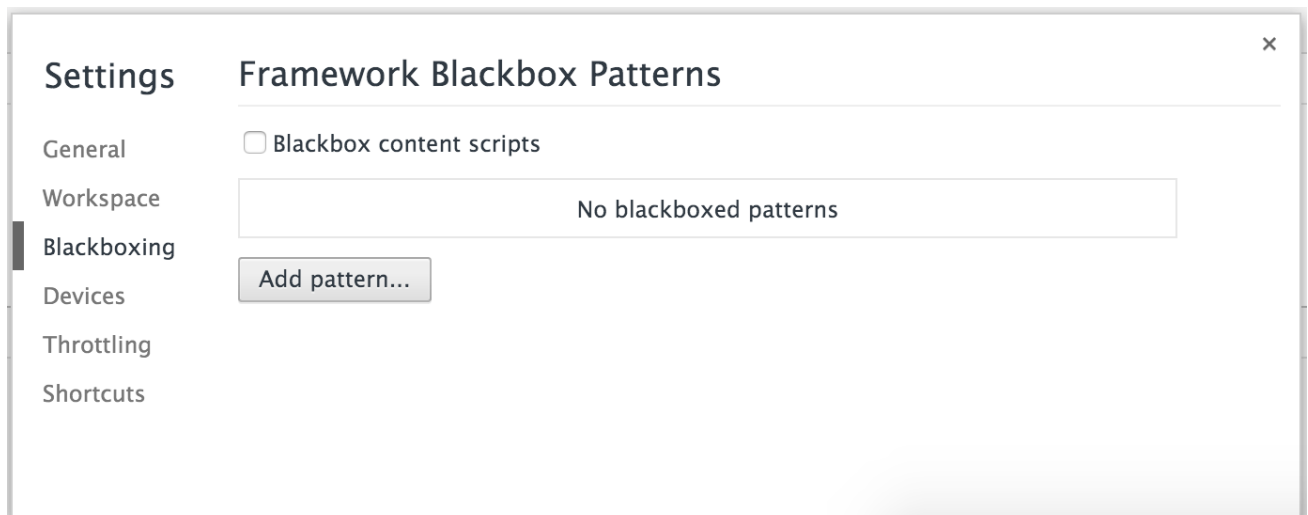
To blackbox a file:

1. Open DevTools Settings.



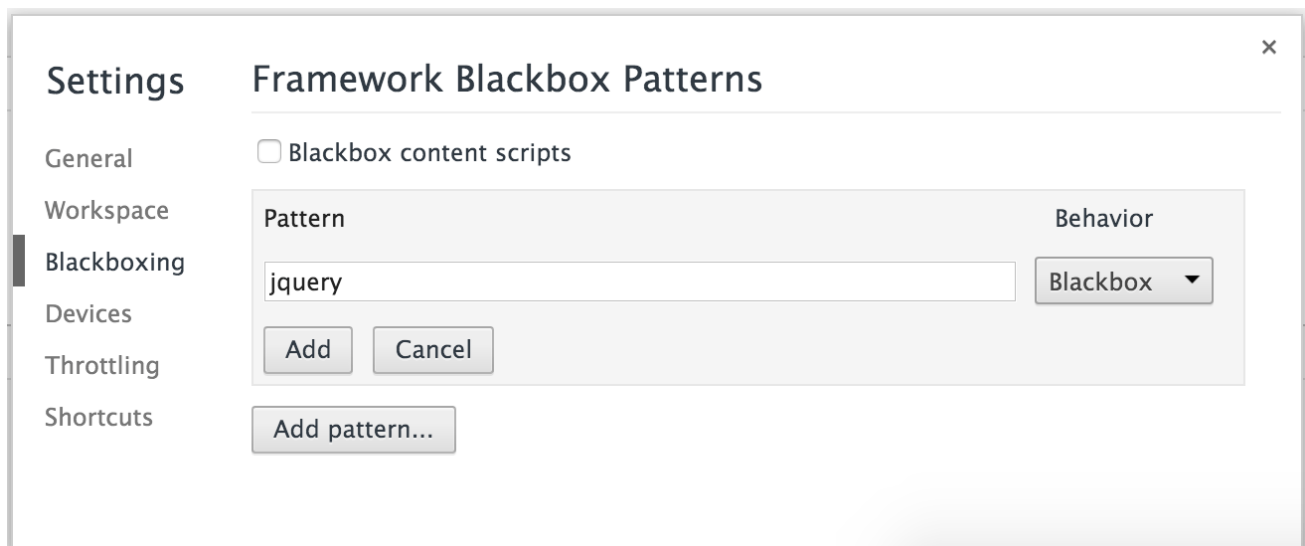


1. In the navigation menu on the left, click **Blackboxing**.



1. Click **Add pattern**.

2. In the **Pattern** textfield enter the filename pattern that you wish to exclude from your call stack. DevTools excludes any scripts that match the pattern.



1. In the dropdown menu to the right of the textfield, select **Blackbox** to execute the script files but exclude the calls from the call stack, or select **Disabled** to prevent the files from executing.
2. Click **Add** to save.

The next time that you run the page and a breakpoint is triggered, DevTools hides any function calls from the blackboxed scripts from the call stack.

## Data manipulation

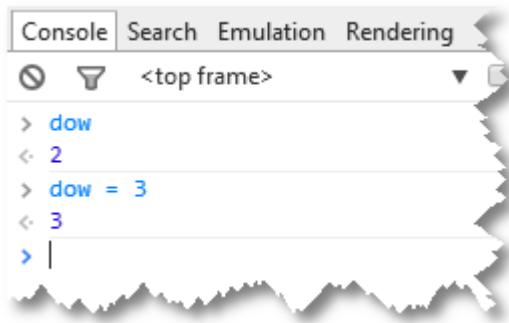
When code execution is paused, you can observe and modify the data it is processing. This is crucial when trying to track down a variable that seems to have the wrong value or a passed parameter that isn't received as expected.

Show the Console drawer by clicking **Show/Hide drawer**  or press ESC. With the console open while stepping, you can now:

- Type the name of a variable to see its current value in the scope of the current function
- Type a JavaScript assignment statement to change the value

Try modifying values, then continue execution to see how it changes the outcome of your code and whether it behaves as you expect.

## Example



We reveal that the value of the parameter `dow` is currently 2, but manually change it to 3 before resuming execution.

## Live editing

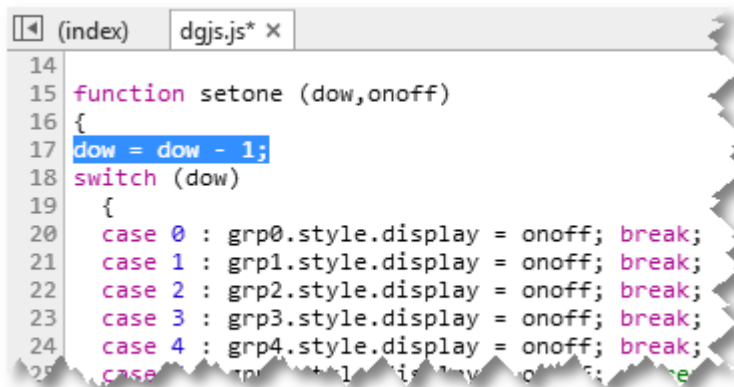
**Warning:** This page is deprecated. See [Edit a script](#) for up-to-date information.

Observing and pausing the executing code helps you locate errors, and live editing allows you to quickly preview changes without the need to reload.

To live edit a script, simply click into the editor part of the Sources panel while stepping. Make your changes as you would do in your editor, then commit the change with Ctrl + S (or Cmd + S on Mac). At this point, the entire JS file will be patched into the VM and all function definitions will be updated.

Now, you can resume execution; your modified script will execute in place of the original, and you can observe the effects of your changes.

## Example



```
14
15 function setone (dow,onoff)
16 {
17   dow = dow - 1;
18   switch (dow)
19   {
20     case 0 : grp0.style.display = onoff; break;
21     case 1 : grp1.style.display = onoff; break;
22     case 2 : grp2.style.display = onoff; break;
23     case 3 : grp3.style.display = onoff; break;
24     case 4 : grp4.style.display = onoff; break;
25     case 5 : grp5.style.display = onoff; break;
```

We suspect that the parameter `dow` is, in every case, off by +1 when it is passed to the function `setone()` – that is, the value of `dow`, as received, is 1 when it should be 0, 2 when it should be 1, etc. To quickly test whether decrementing the passed value confirms that this is the problem, we add line 17 at the beginning of the function, commit with `Ctrl + S` and resume.

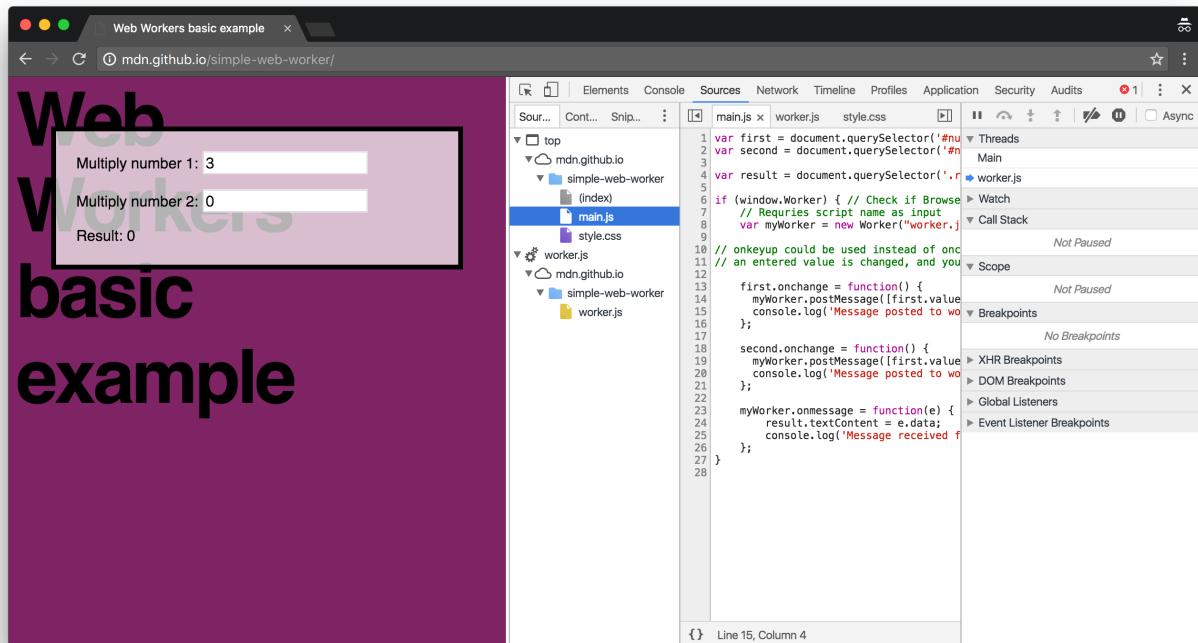
## Managing thread execution

**Warning:** This page is deprecated. See [Change thread context](#) for up-to-date information.

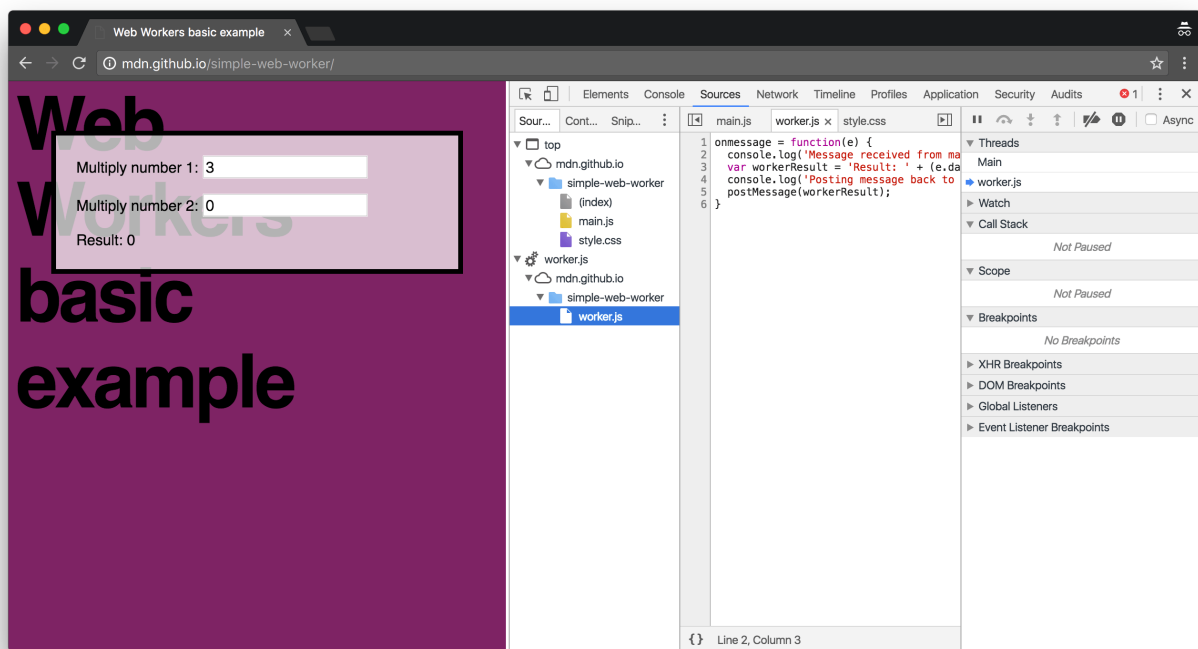
Use the **Threads** pane on the Sources panel to pause, step into, and inspect other threads, such as service worker or web worker threads.

To demonstrate the Threads pane, this section uses the following demo: [Web Workers basic example](#).

If you open DevTools on the app, you can see that the main script is located in `main.js`:

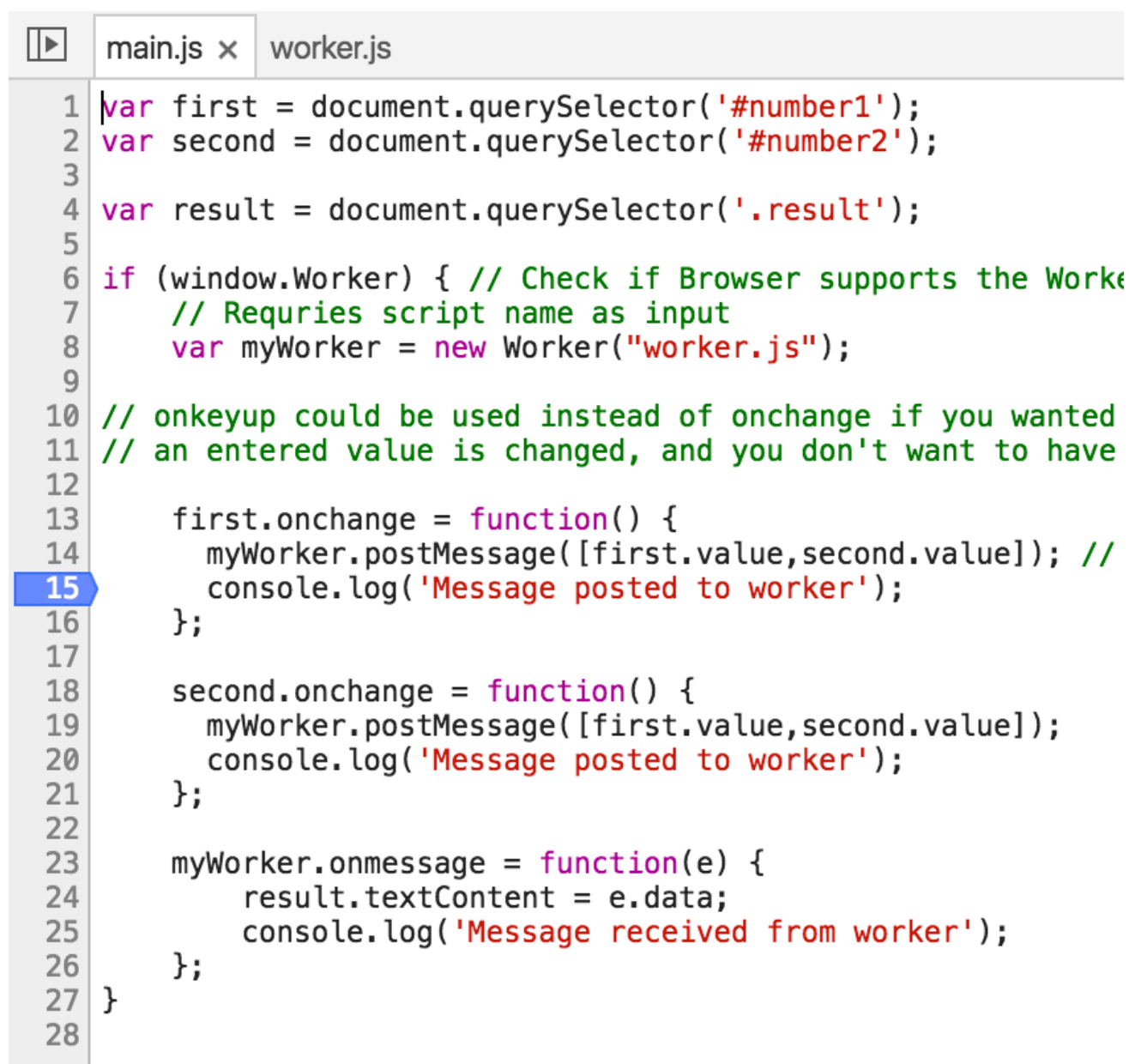


And the web worker script is located in `worker.js`:



The main script listens to changes to the **Multiply number 1** or **Multiply number 2** input fields. Upon change the main script sends a message to the web worker with the values of the two numbers to multiply. The web worker does the multiplication and then passes the result back to the main script.

Suppose that you set a breakpoint in `main.js` that's triggered when the first number is changed:

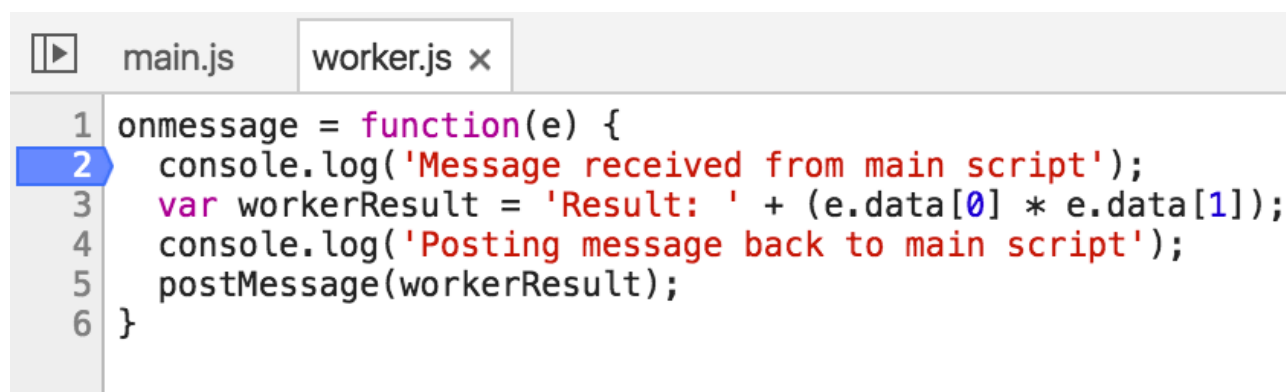


The screenshot shows a code editor with two tabs: `main.js` and `worker.js`. The `main.js` tab is active. The code in `main.js` is as follows:

```
1 var first = document.querySelector('#number1');
2 var second = document.querySelector('#number2');
3
4 var result = document.querySelector('.result');
5
6 if (window.Worker) { // Check if Browser supports the Worker
7     // Requires script name as input
8     var myWorker = new Worker("worker.js");
9
10    // onkeyup could be used instead of onchange if you wanted
11    // an entered value is changed, and you don't want to have
12
13    first.onchange = function() {
14        myWorker.postMessage([first.value,second.value]); //
15        console.log('Message posted to worker');
16    };
17
18    second.onchange = function() {
19        myWorker.postMessage([first.value,second.value]);
20        console.log('Message posted to worker');
21    };
22
23    myWorker.onmessage = function(e) {
24        result.textContent = e.data;
25        console.log('Message received from worker');
26    };
27 }
28
```

A blue highlight is on line 15, indicating a breakpoint.

And you also set a breakpoint in `worker.js` when the worker receives a message:

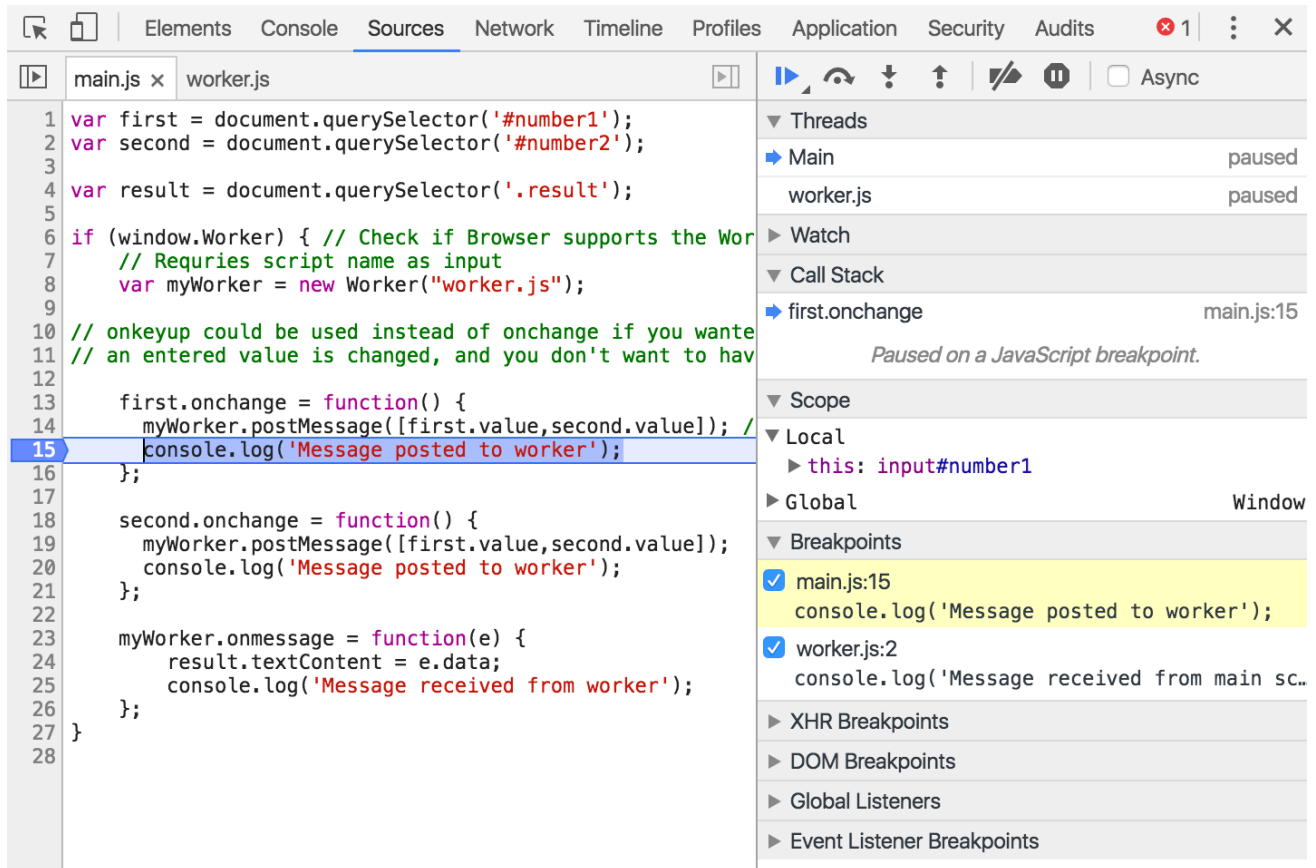


The screenshot shows a code editor with two tabs: `main.js` and `worker.js`. The `worker.js` tab is active. The code in `worker.js` is as follows:

```
1 onmessage = function(e) {
2     console.log('Message received from main script');
3     var workerResult = 'Result: ' + (e.data[0] * e.data[1]);
4     console.log('Posting message back to main script');
5     postMessage(workerResult);
6 }
```

A blue highlight is on line 2, indicating a breakpoint.

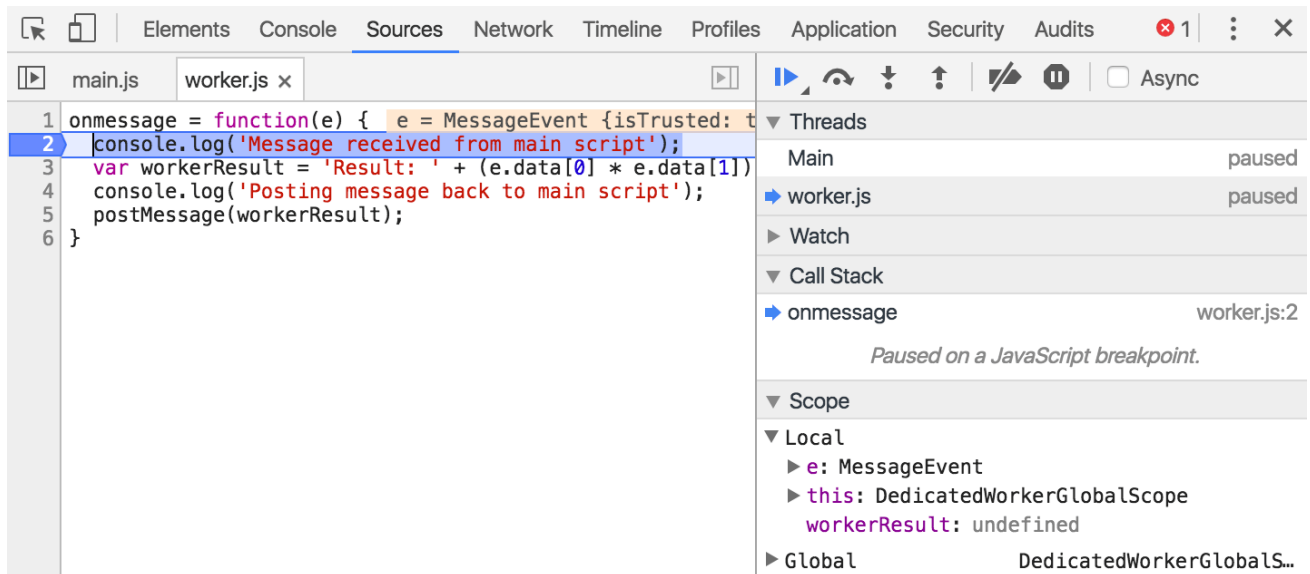
Modifying the first number on the app's UI triggers both of the breakpoints.



In the Threads pane the blue arrow indicates which thread is currently selected. For example, in the screenshot above the **Main** thread is selected.

All of the DevTools controls for stepping through code (resume or pause script execution, step over next function call, step into next function call, etc.) pertain to that thread. In other words, if you pressed the **Resume script execution** button while your DevTools looked like the screenshot above, the Main thread would resume executing, but the web worker thread would still be paused. The **Call Stack** and **Scope** sections are only displaying information for the Main thread, too.

When you want to step through the code for the web worker thread, or see its scope and call stack information, just click on its label in the Threads pane, so that the blue arrow is next to it. The screenshot below shows how the call stack and scope information changes after selecting the worker thread. Again, if you were to press any of the stepping through code buttons (resume script execution, step over next function call, etc.), that action would only pertain to the worker thread. The Main thread is not affected.



Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.