**GOOGLE CONSOLE DEBUGGING TOOLS**

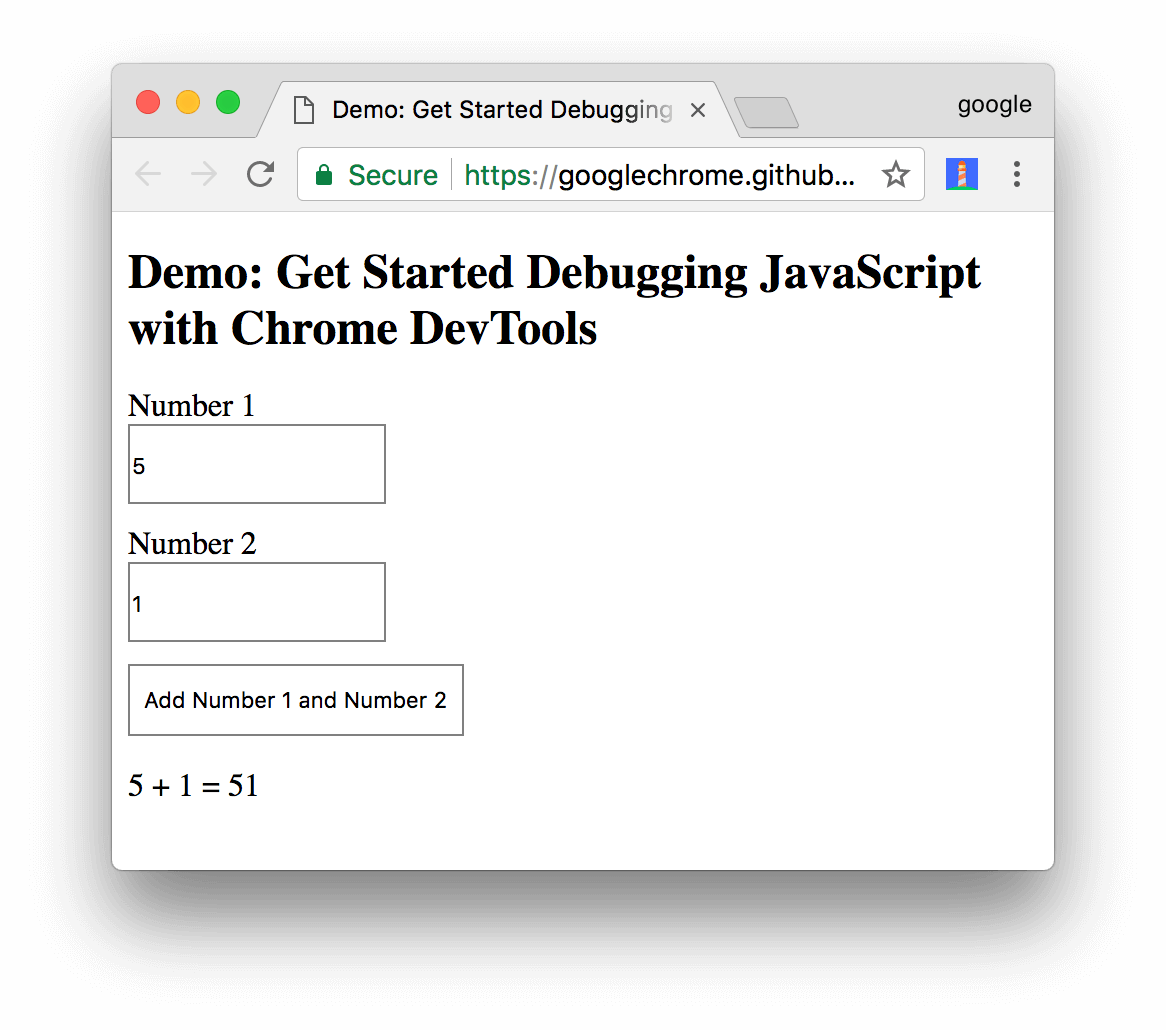
Step 1: Reproduce the bug

Finding a series of actions that consistently reproduces a bug is always the first step to debugging.

1. Click **Open Demo**. The demo opens in a new tab.

[OPEN DEMO](https://googlechrome.github.io/devtools-samples/debug-js/get-started)

1. Enter 5 in the **Number 1** text box.
2. Enter 1 in the **Number 2** text box.
3. Click **Add Number 1 and Number 2**. The label below the button says 5 + 1 = 51. The result should be 6. This is the bug you're going to fix.

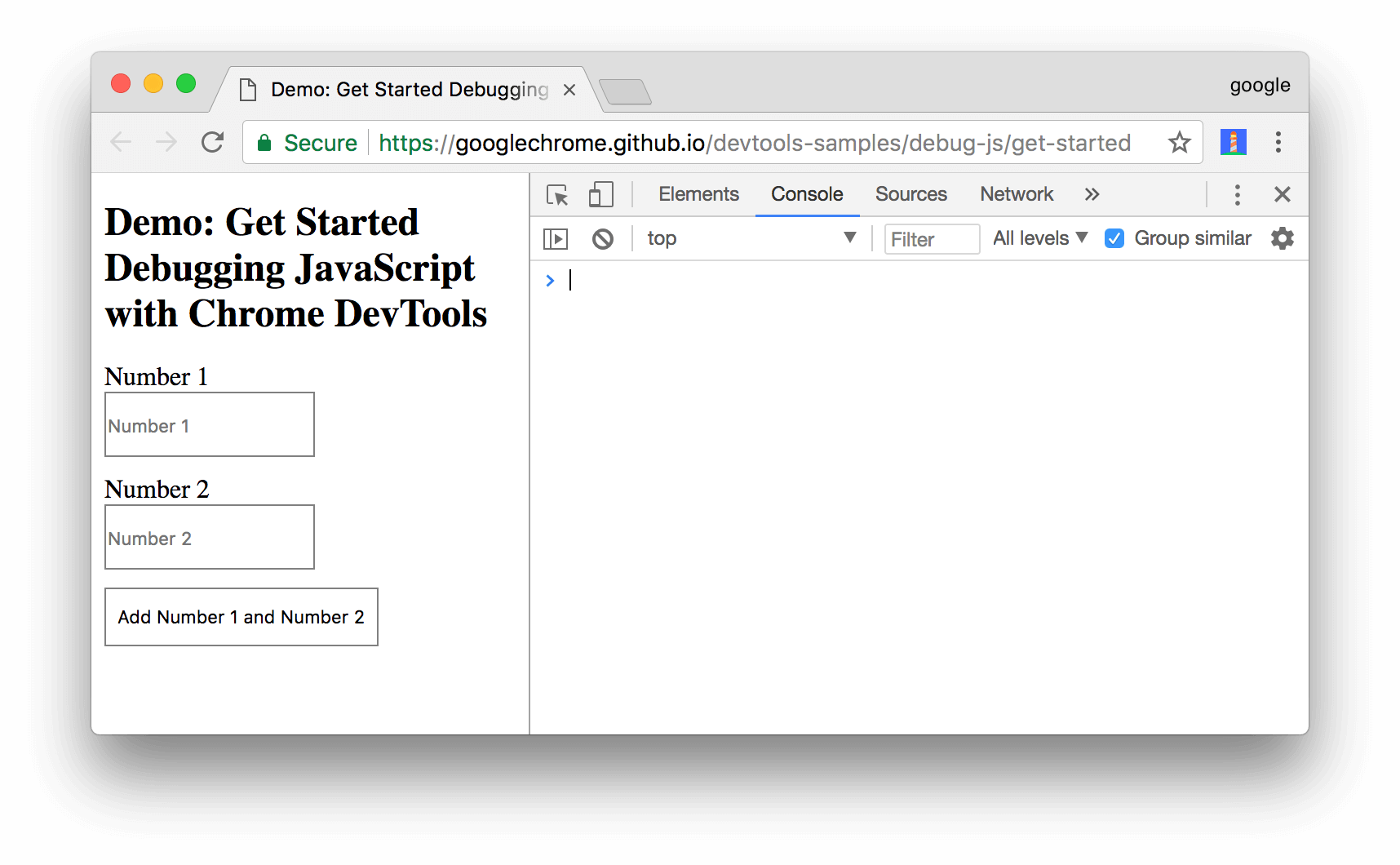


1. **Figure 1**. The result of 5 + 1 is 51. It should be 6.

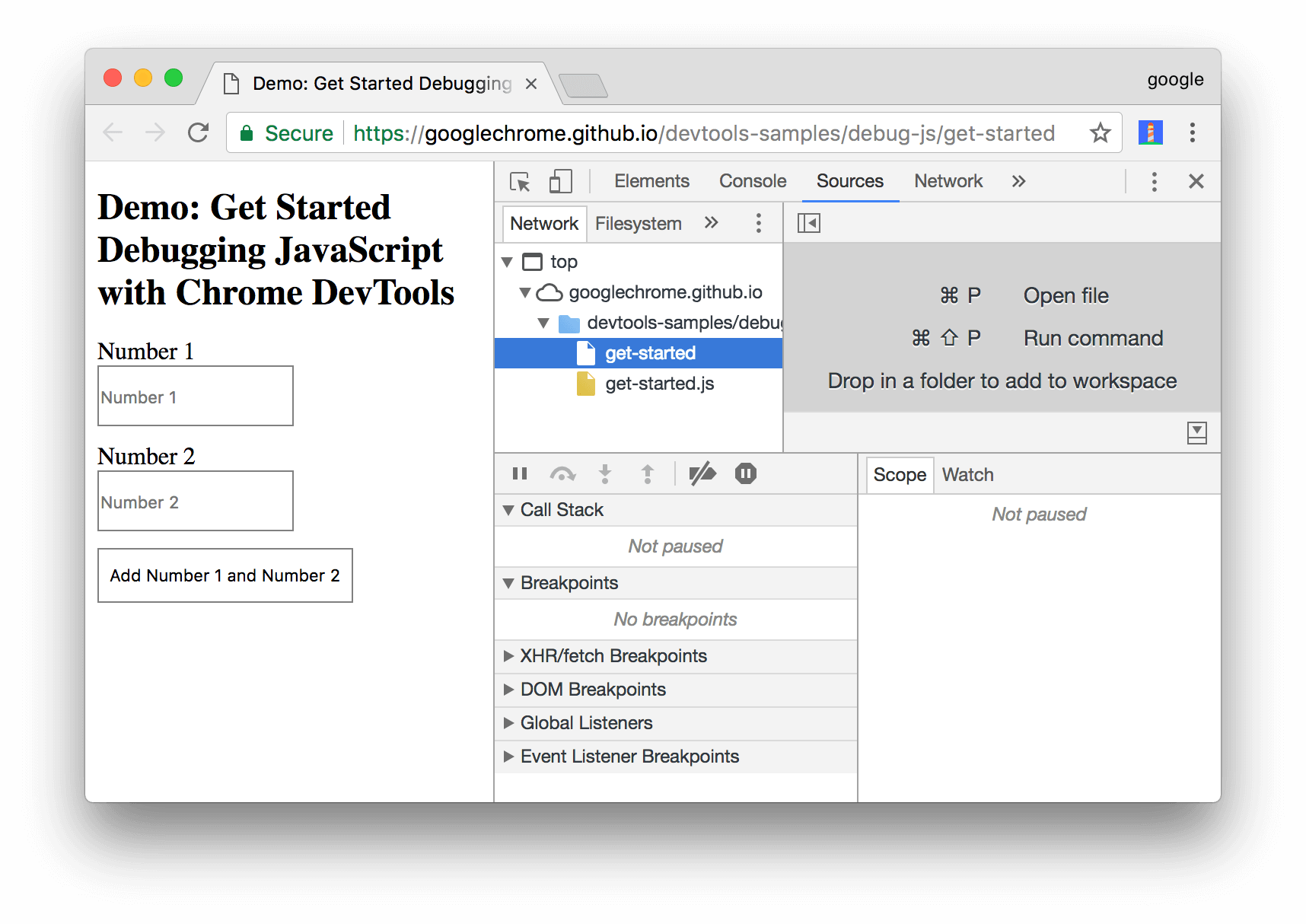
Step 2: Get familiar with the Sources panel UI

DevTools provides a lot of different tools for different tasks, such as changing CSS, profiling page load performance, and monitoring network requests. The **Sources** panel is where you debug JavaScript.

1. Open DevTools by pressing Command+Option+I (Mac) or Control+Shift+I (Windows, Linux). This shortcut opens the **Console** panel.

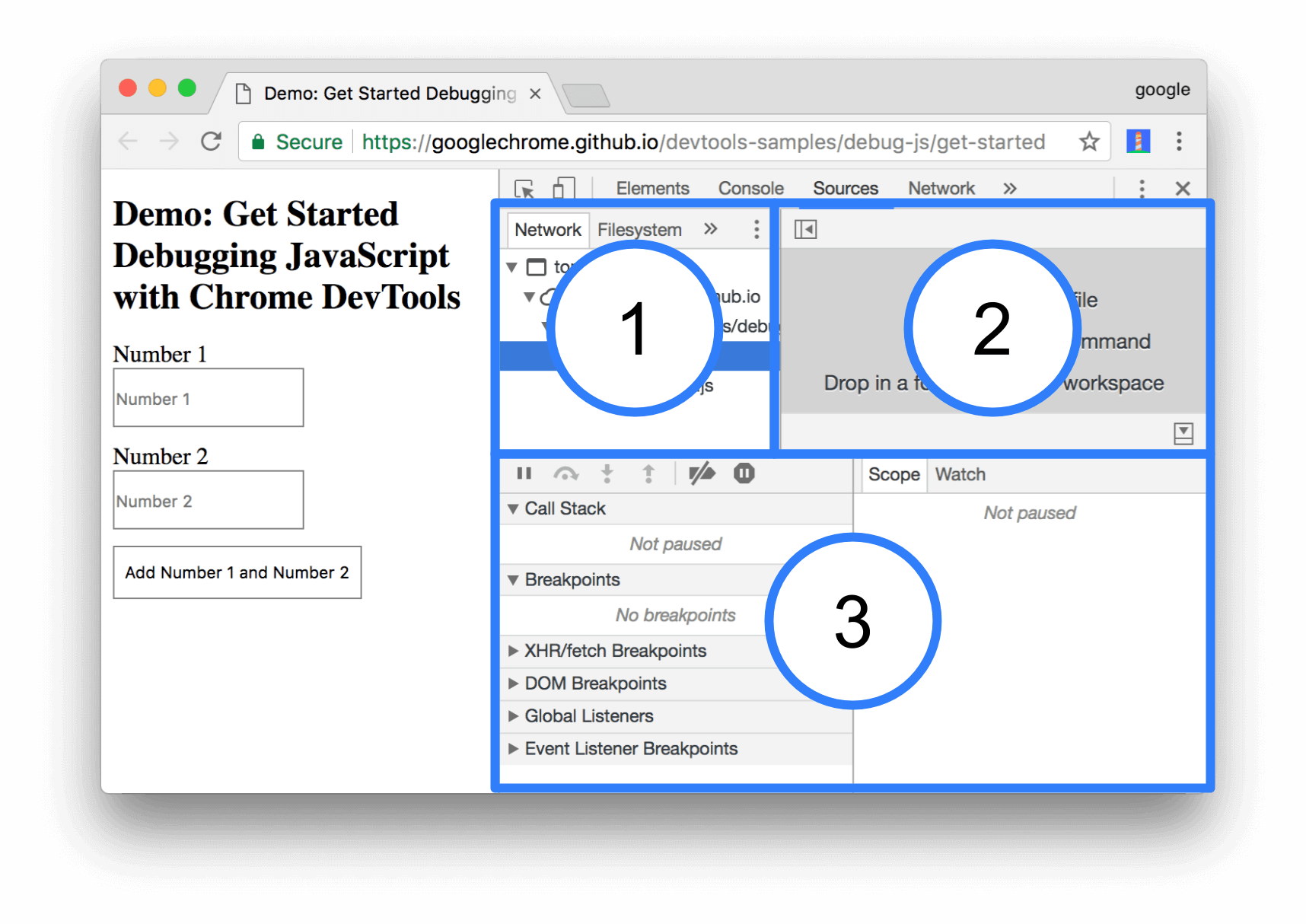


1. **Figure 2**. The **Console** panel
2. Click the **Sources** tab.



1. **Figure 3**. The **Sources** panel

The **Sources** panel UI has 3 parts:



**Figure 4**. The 3 parts of the **Sources** panel UI

1. The **File Navigator** pane. Every file that the page requests is listed here.
2. The **Code Editor** pane. After selecting a file in the **File Navigator** pane, the contents of that file are displayed here.
3. The **JavaScript Debugging** pane. Various tools for inspecting the page's JavaScript. If your DevTools window is wide, this pane is displayed to the right of the **Code Editor** pane.

Step 3: Pause the code with a breakpoint

A common method for debugging a problem like this is to insert a lot of console.log() statements into the code, in order to inspect values as the script executes. For example:

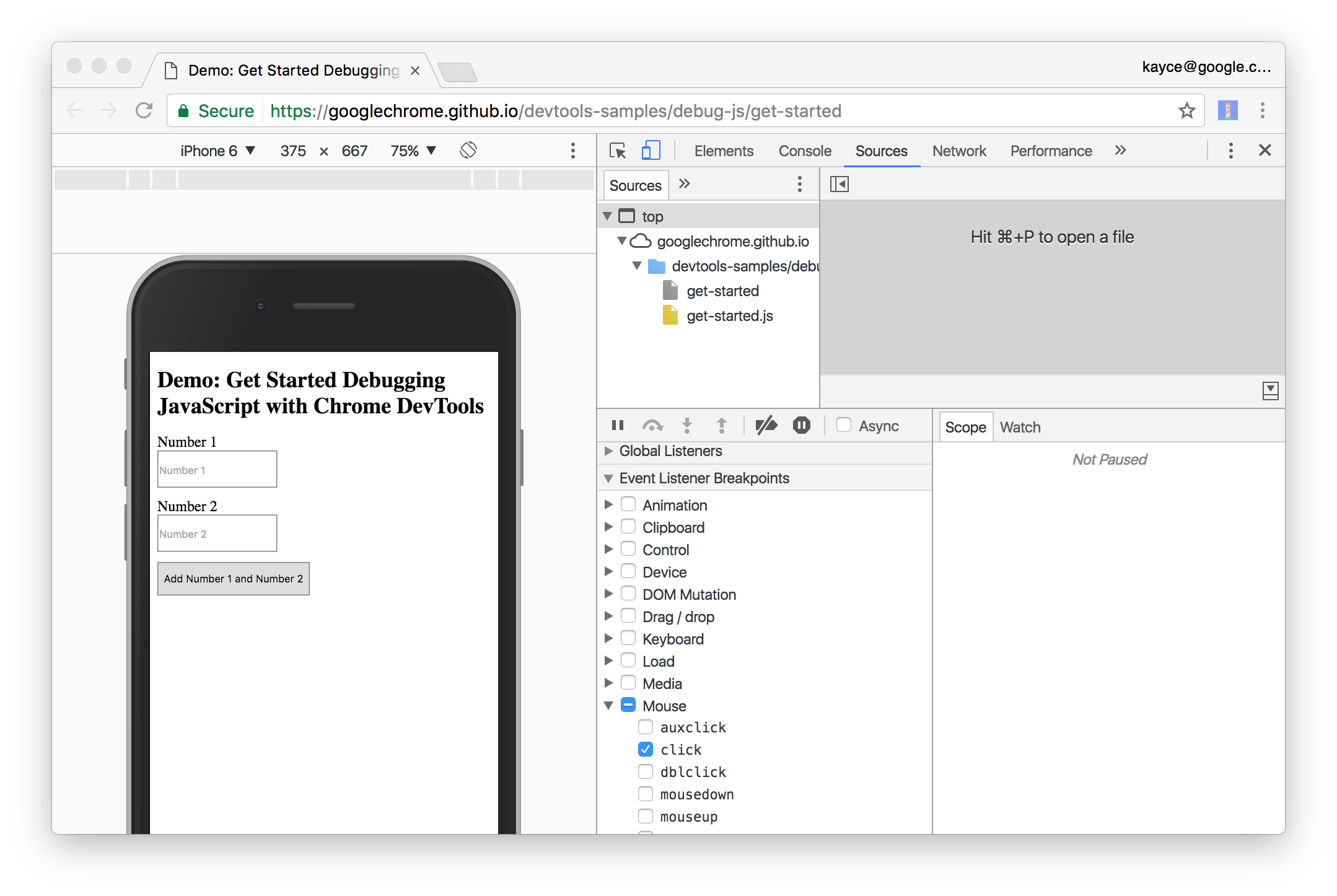
The console.log() method may get the job done, but **breakpoints** can get it done faster. A breakpoint lets you pause your code in the middle of its execution, and examine all values at that moment in time. Breakpoints have a few advantages over the console.log() method:

* With console.log(), you need to manually open the source code, find the relevant code, insert the console.log() statements, and then reload the page in order to see the messages in the Console. With breakpoints, you can pause on the relevant code without even knowing how the code is structured.
* In your console.log() statements you need to explicitly specify each value that you want to inspect. With breakpoints, DevTools shows you the values of all variables at that moment in time. Sometimes there are variables affecting your code that you're not even aware of.

In short, breakpoints can help you find and fix bugs faster than the console.log() method.

If you take a step back and think about how the app works, you can make an educated guess that the incorrect sum (5 + 1 = 51) gets computed in the click event listener that's associated to the **Add Number 1 and Number 2** button. Therefore, you probably want to pause the code around the time that the click listener executes. **Event Listener Breakpoints** let you do exactly that:

1. In the **JavaScript Debugging** pane, click **Event Listener Breakpoints** to expand the section. DevTools reveals a list of expandable event categories, such as **Animation** and **Clipboard**.
2. Next to the **Mouse** event category, click **Expand**. DevTools reveals a list of mouse events, such as **click** and **mousedown**. Each event has a checkbox next to it.
3. Check the **click** checkbox. DevTools is now set up to automatically pause when any click event listener executes.



1. **Figure 5**. The **click** checkbox is enabled
2. Back on the demo, click **Add Number 1 and Number 2** again. DevTools pauses the demo and highlights a line of code in the **Sources** panel. DevTools should be paused on this line of code:

Function onClick(){

1. If you're paused on a different line of code, press **Resume Script Execution** until you're paused on the correct line.

**Note**: If you paused on a different line, you have a browser extension that registers a **click** event listener on every page that you visit. You were paused in the extension's **click** listener. If you use Incognito Mode to [browse in private](https://support.google.com/chrome/answer/95464), which disables all extensions, you can see that you pause on the correct line of code every time.

**Event Listener Breakpoints** are just one of many types of breakpoints available in DevTools. It's worth memorizing all the different types, because each type ultimately helps you debug different scenarios as quickly as possible. See [Pause Your Code With Breakpoints](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints) to learn when and how to use each type.

Step 4: Step through the code

One common cause of bugs is when a script executes in the wrong order. Stepping through your code enables you to walk through your code's execution, one line at a time, and figure out exactly where it's executing in a different order than you expected. Try it now:

On the **Sources** panel of DevTools, click **Step into next function call**  to step through the execution of the onClick() function, one line at a time. DevTools highlights the following line of code:

If(inputsAreEmpty()){

1. Click Step over next function call . DevTools executes inputsAreEmpty() without stepping into it. Notice how DevTools skips a few lines of code. This is because inputsAreEmpty() evaluated to false, so the if statement's block of code didn't execute.

That's the basic idea of stepping through code. If you look at the code in get-started.js, you can see that the bug is probably somewhere in the updateLabel() function. Rather than stepping through every line of code, you can use another type of breakpoint to pause the code closer to the probable location of the bug.

## Step 5: Set a line-of-code breakpoint

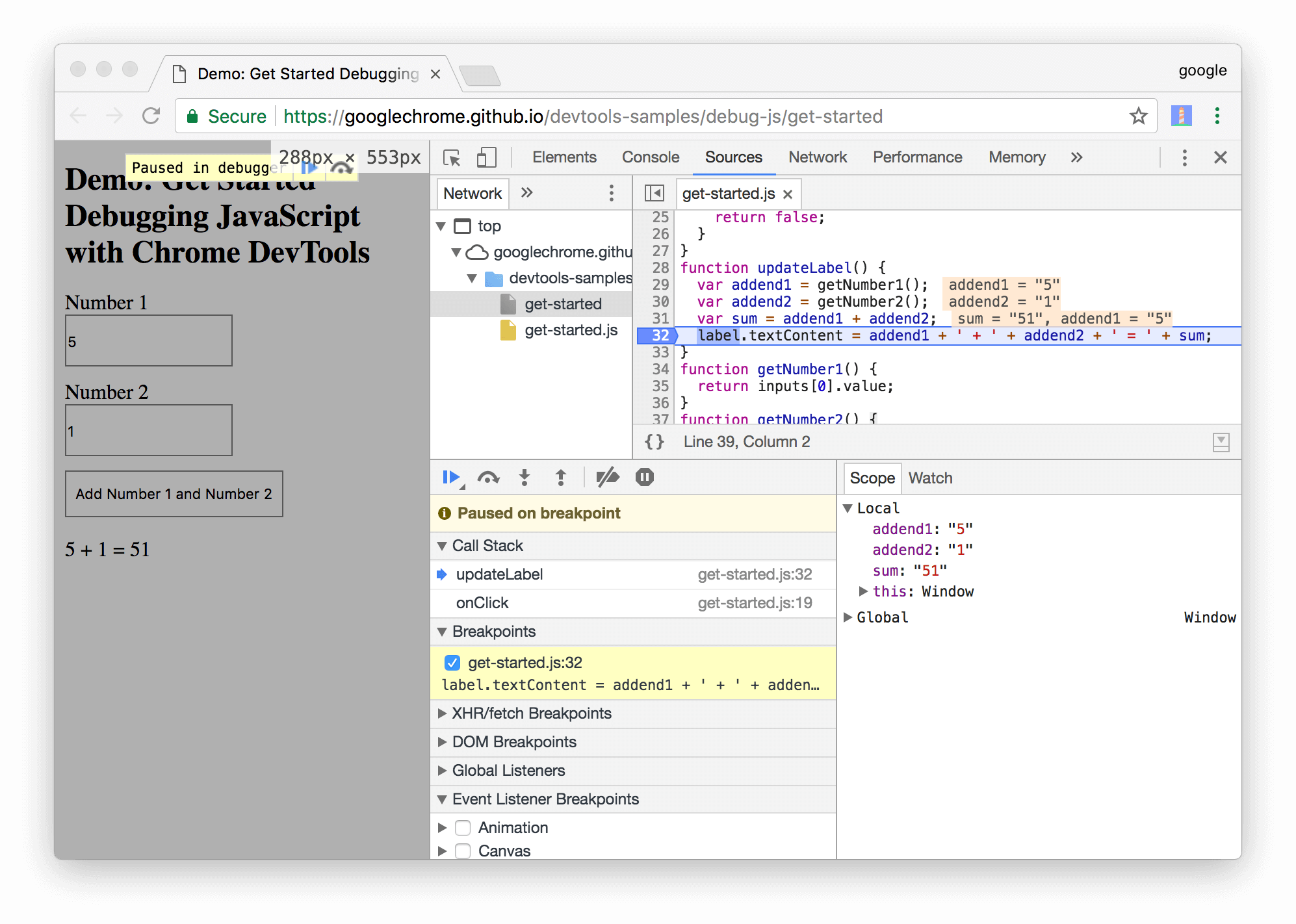
Line-of-code breakpoints are the most common type of breakpoint. When you've got a specific line of code that you want to pause on, use a line-of-code breakpoint:

1. Look at the last line of code in updateLabel():

Label.textContent=addend1+ ’+’ +addend2+ ‘+’ + ‘=’ +sum;

To the left of the code you can see the line number of this particular line of code, which is **32**. Click on **32**. DevTools puts a blue icon on top of **32**. This means that there is a line-of-code breakpoint on this line. DevTools now always pauses before this line of code is executed.

Click **Resume script execution** . The script continues executing until it reaches line 32. On lines 29, 30, and 31, DevTools prints out the values of  addend1, addend2 and sum to the right of each line's semi-colon.



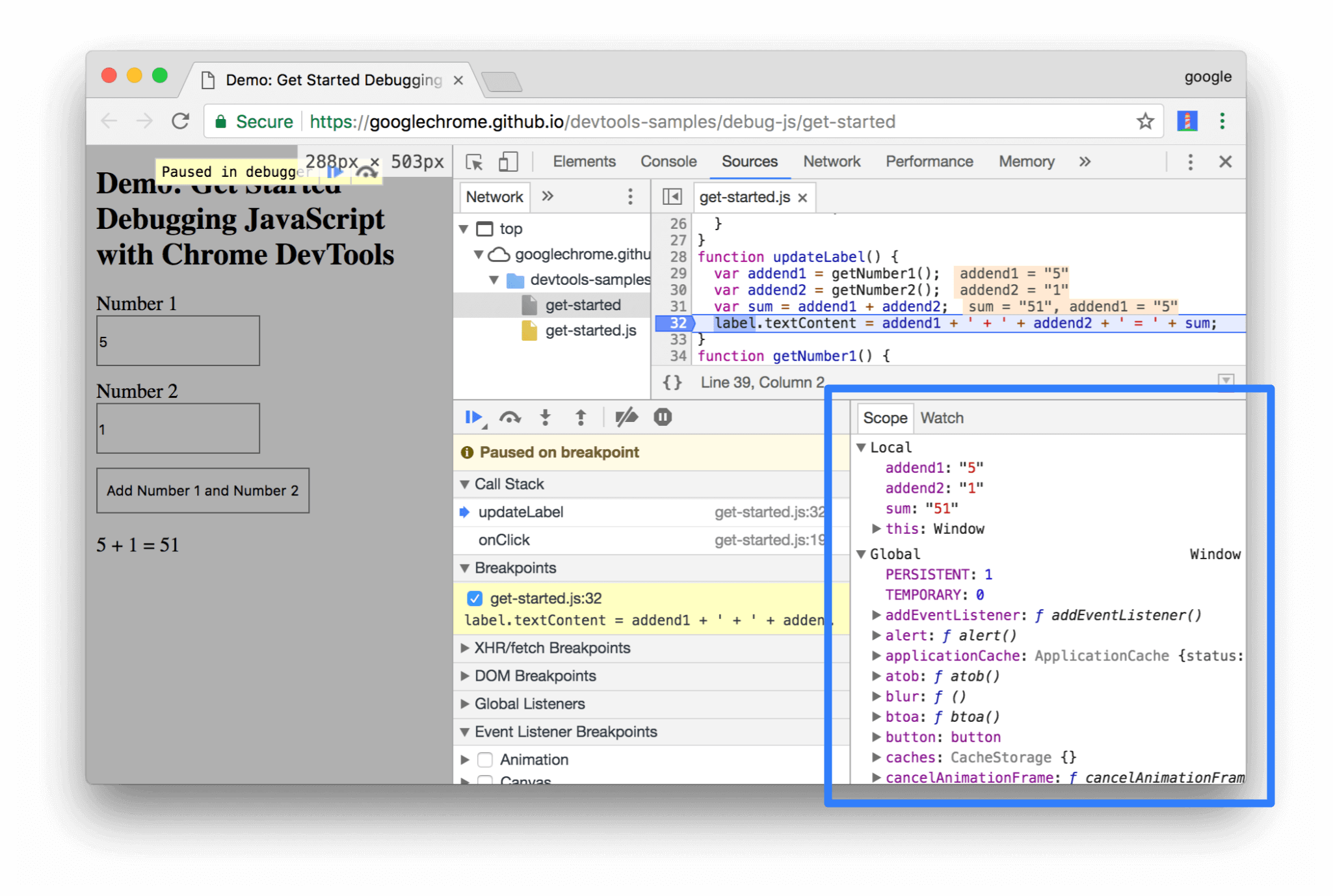
1. **Figure 6**. DevTools pauses on the line-of-code breakpoint on line 32

## Step 6: Check variable values

The values of addend1, addend2, and sum look suspicious. They're wrapped in quotes, which means that they're strings. This is a good hypothesis for the explaining the cause of the bug. Now it's time to gather more information. DevTools provides a lot of tools for examining variable values.

### **Method 1: The Scope pane**

When you're paused on a line of code, the **Scope** pane shows you what local and global variables are currently defined, along with the value of each variable. It also shows closure variables, when applicable. Double-click a variable value to edit it. When you're not paused on a line of code, the **Scope** pane is empty.

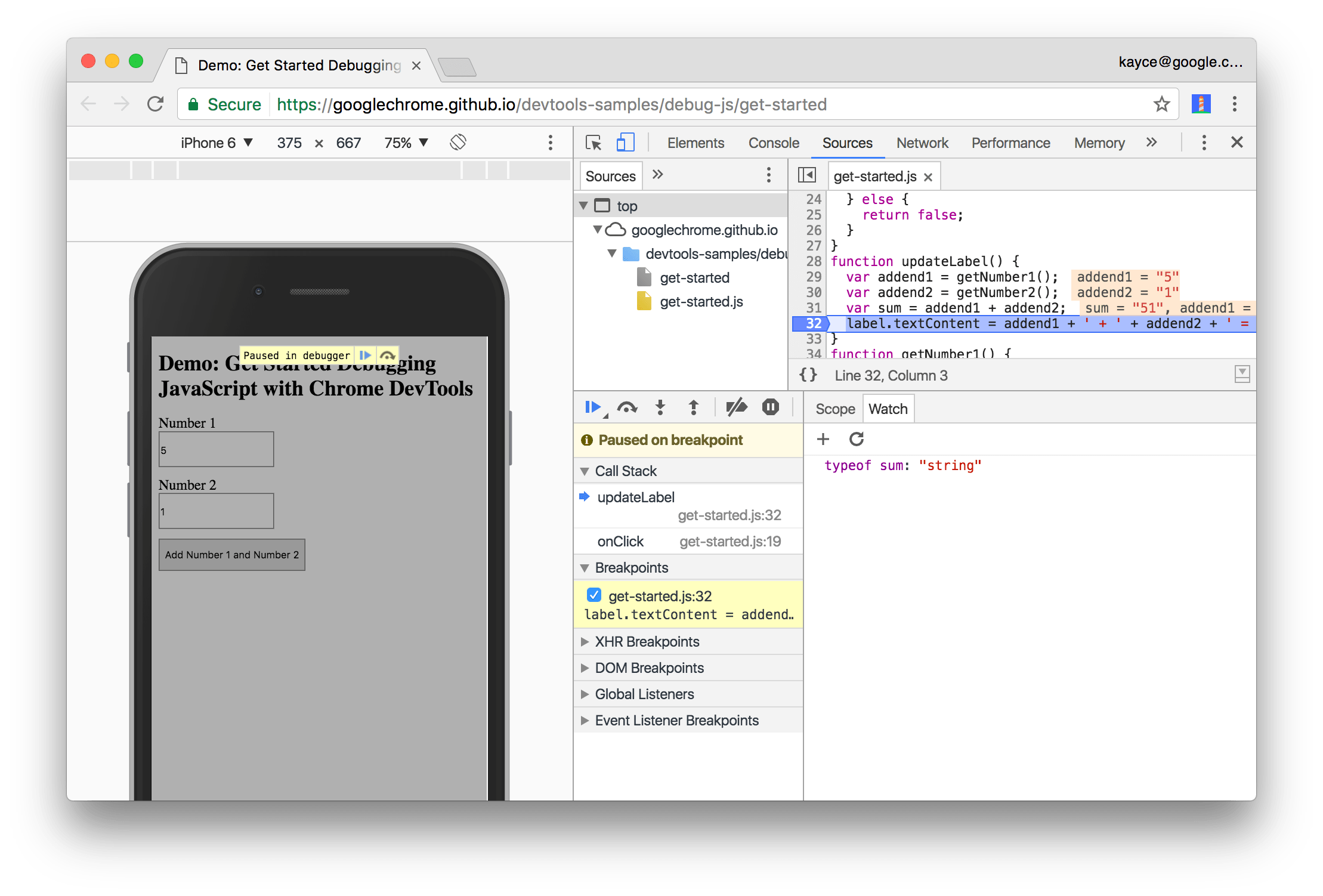


**Figure 7**. The **Scope** pane

### **Method 2: Watch Expressions**

The **Watch Expressions** tab lets you monitor the values of variables over time. As the name implies, Watch Expressions aren't just limited to variables. You can store any valid JavaScript expression in a Watch Expression. Try it now:

1. Click the **Watch** tab.
2. Click **Add Expression +.**
3. Type typeof sum.
4. Press Enter. DevTools shows typeof sum: "string". The value to the right of the colon is the result of your Watch Expression.



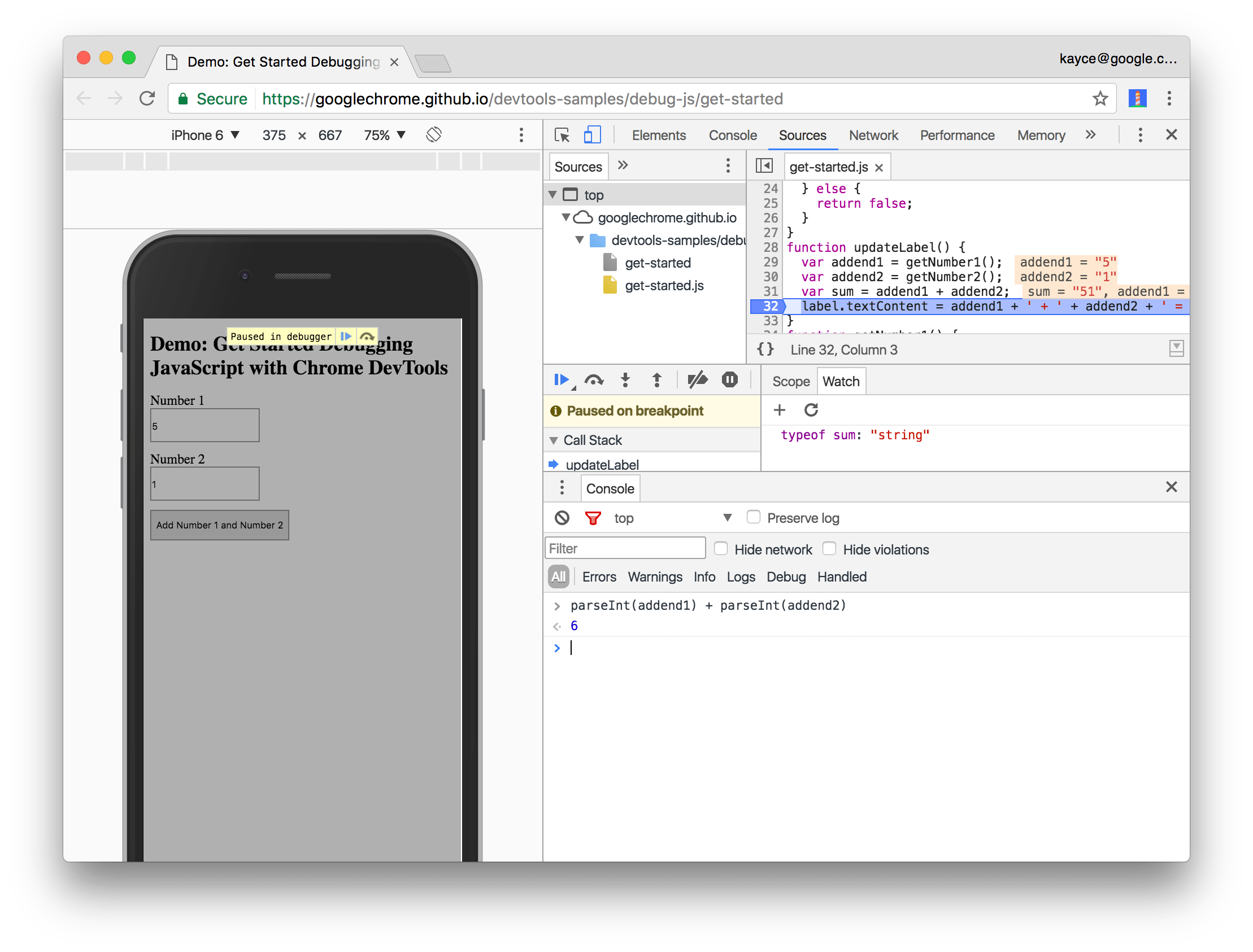
1. **Figure 8**. The Watch Expression pane (bottom-right), after creating the typeof sum Watch Expression. If your DevTools window is large, the Watch Expression pane is on the right, above the **Event Listener Breakpoints** pane.

As suspected, sum is being evaluated as a string, when it should be a number. You've now confirmed that this is the cause of the bug.

### **Method 3: The Console**

In addition to viewing console.log() messages, you can also use the Console to evaluate arbitrary JavaScript statements. In terms of debugging, you can use the Console to test out potential fixes for bugs. Try it now:

1. If you don't have the Console drawer open, press Escape to open it. It opens at the bottom of your DevTools window.
2. In the Console, type parseInt(addend1) + parseInt(addend2) . This statement works because you are paused on a line of code where addend1 and addend2 are in scope.
3. Press Enter. DevTools evaluates the statement and prints out 6, which is the result you expect the demo to produce.



1. **Figure 9**. The Console drawer, after evaluating parseInt(addend1) + parseInt(addend2).

Step 7: Apply a fix

You've found a fix for the bug. All that's left is to try out your fix by editing the code and re-running the demo. You don't need to leave DevTools to apply the fix. You can edit JavaScript code directly within the DevTools UI. Try it now:

1. Click **Resume script execution ||>.**
2. In the **Code Editor**, replace line 31, var sum = addend1 + addend2, with var sum= parseInt(addend1) + parseInt(addend2).
3. Press Command+S (Mac) or Control+S (Windows, Linux) to save your change.
4. Click **Deactivate breakpoints** . It changes blue to indicate that it's active. While this is set, DevTools ignores any breakpoints you've set.
5. Try out the demo with different values. The demo now calculates correctly.

**Caution:** This workflow only applies a fix to the code that is running in your browser. It won't fix the code for all users that visit your page. To do that, you need to fix the code that's on your servers.

Next steps

Congratulations! You now know how to make the most of Chrome DevTools when debugging JavaScript. The tools and methods you learned in this tutorial can save you countless hours.

This tutorial only showed you two ways to set breakpoints. DevTools offers many other ways, including:

* Conditional breakpoints that are only triggered when the condition that you provide is true.
* Breakpoints on caught or uncaught exceptions.
* XHR breakpoints that are triggered when the requested URL matches a substring that you provide.

# **Pause Your Code With Breakpoints**

Use breakpoints to pause your JavaScript code. This guide explains each type of breakpoint that's available in DevTools, as well as when to use and how to set each type. For a hands-on tutorial of the debugging process, see [Get Started with Debugging JavaScript in Chrome DevTools](https://developers.google.com/web/tools/chrome-devtools/javascript/).

## Overview of when to use each breakpoint type

The most well-known type of breakpoint is line-of-code. But line-of-code breakpoints can be inefficient to set, especially if you don't know exactly where to look, or if you are working with a large codebase. You can save yourself time when debugging by knowing how and when to use the other types of breakpoints.

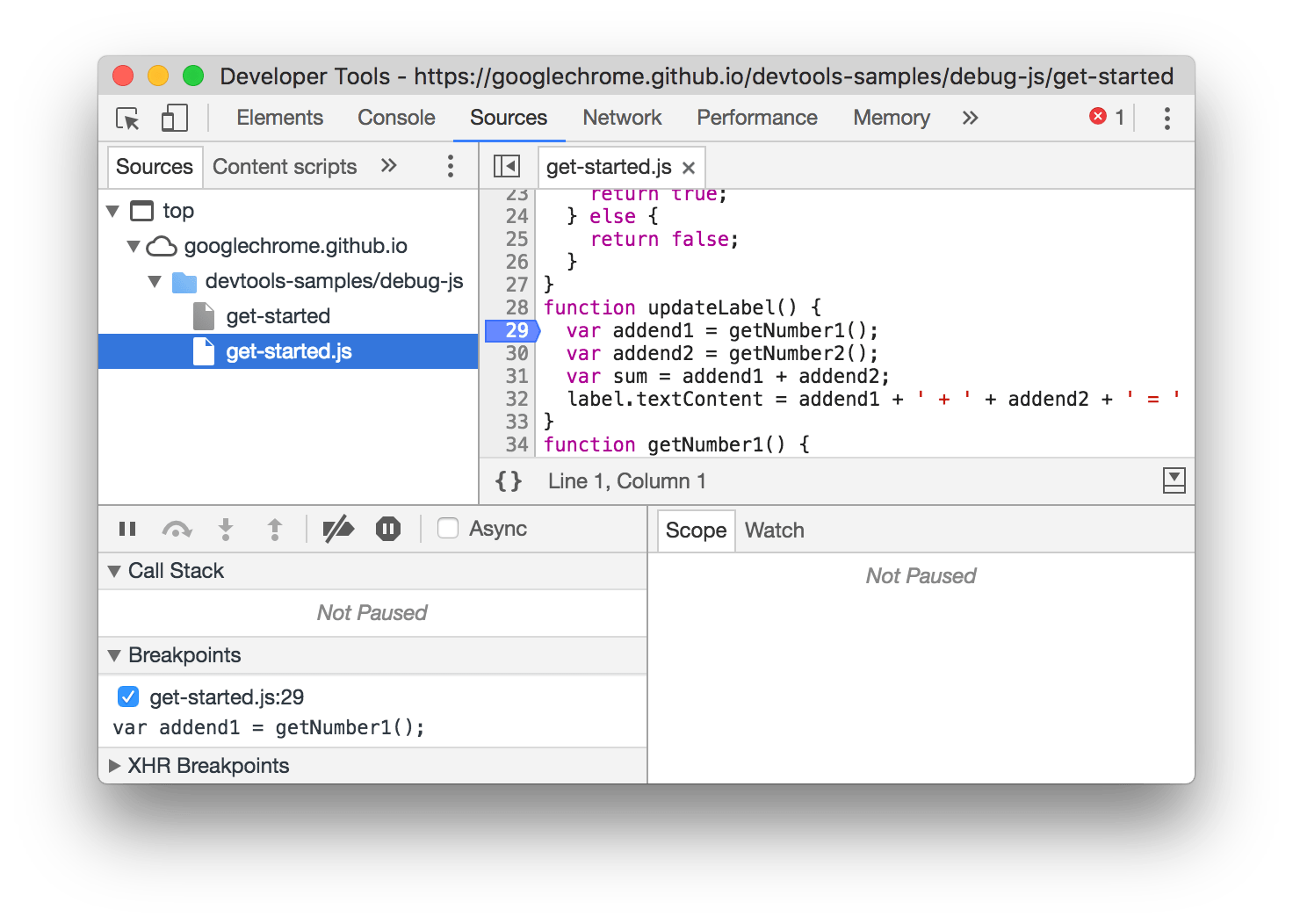
|  |  |
| --- | --- |
| Breakpoint Type | Use This When You Want To Pause... |
| [Line-of-code](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints#loc) | On an exact region of code. |
| [Conditional line-of-code](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints#conditional-loc) | On an exact region of code, but only when some other condition is true. |
| [DOM](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints#dom) | On the code that changes or removes a specific DOM node, or its children. |
| [XHR](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints#xhr) | When an XHR URL contains a string pattern. |
| [Event listener](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints#event-listeners) | On the code that runs after an event, such as click, is fired. |
| [Exception](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints#exceptions) | On the line of code that is throwing a caught or uncaught exception. |
| [Function](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints#function) | Whenever a specific function is called. |

## Line-of-code breakpoints

Use a line-of-code breakpoint when you know the exact region of code that you need to investigate. DevTools alwayspauses before this line of code is executed.

To set a line-of-code breakpoint in DevTools:

1. Click the **Sources** tab.
2. Open the file containing the line of code you want to break on.
3. Go the line of code.
4. To the left of the line of code is the line number column. Click on it. A blue icon appears on top of the line number column.



**Figure 1**: A line-of-code breakpoint set on line **29**

### **Line-of-code breakpoints in your code**

Call debugger from your code to pause on that line. This is equivalent to a [line-of-code breakpoint](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints#loc), except that the breakpoint is set in your code, not in the DevTools UI.

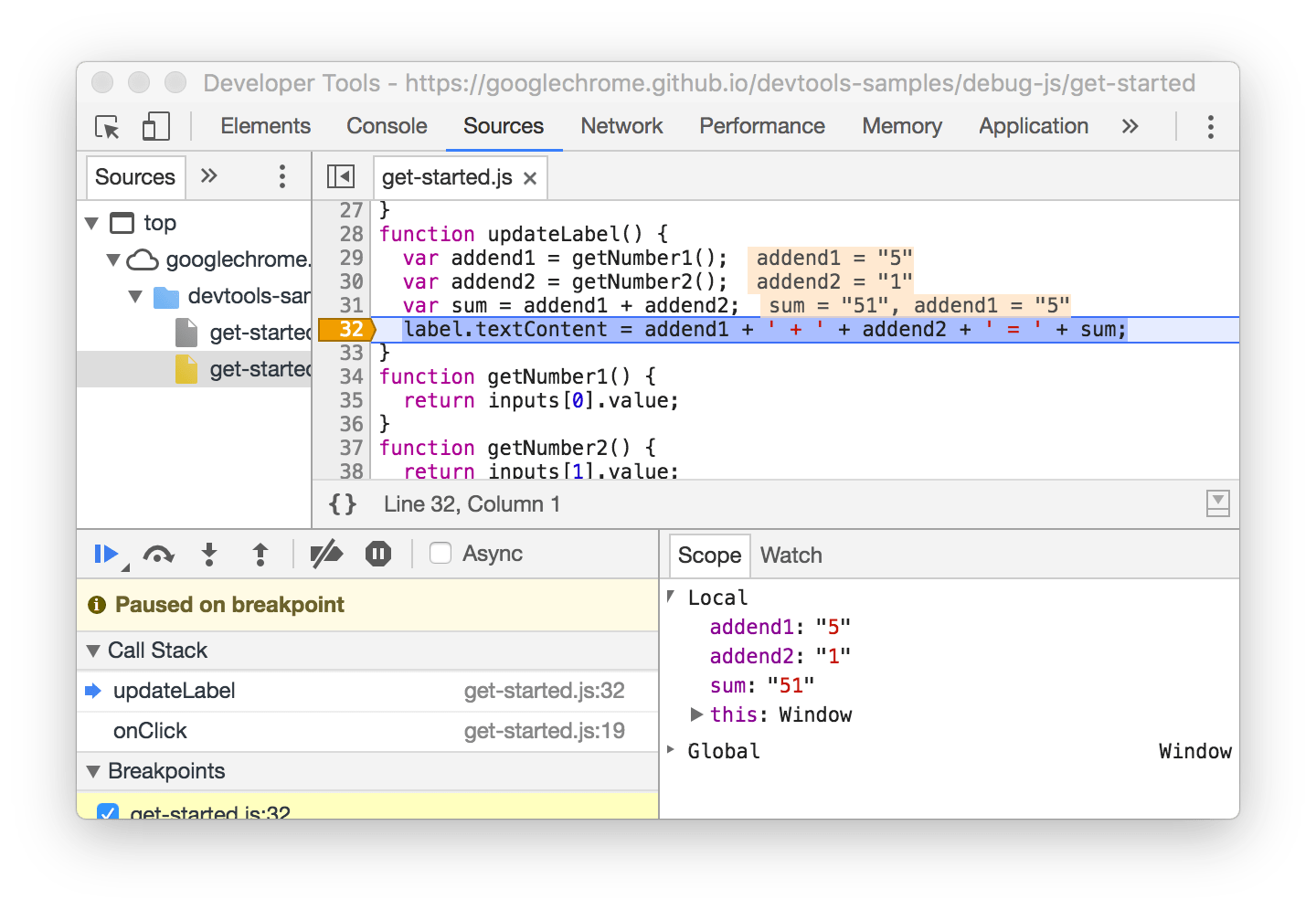
console.log('a');  
console.log('b');  
debugger;  
console.log('c');

### **Conditional line-of-code breakpoints**

Use a conditional line-of-code breakpoint when you know the exact region of code that you need to investigate, but you want to pause only when some other condition is true.

To set a conditional line-of-code breakpoint:

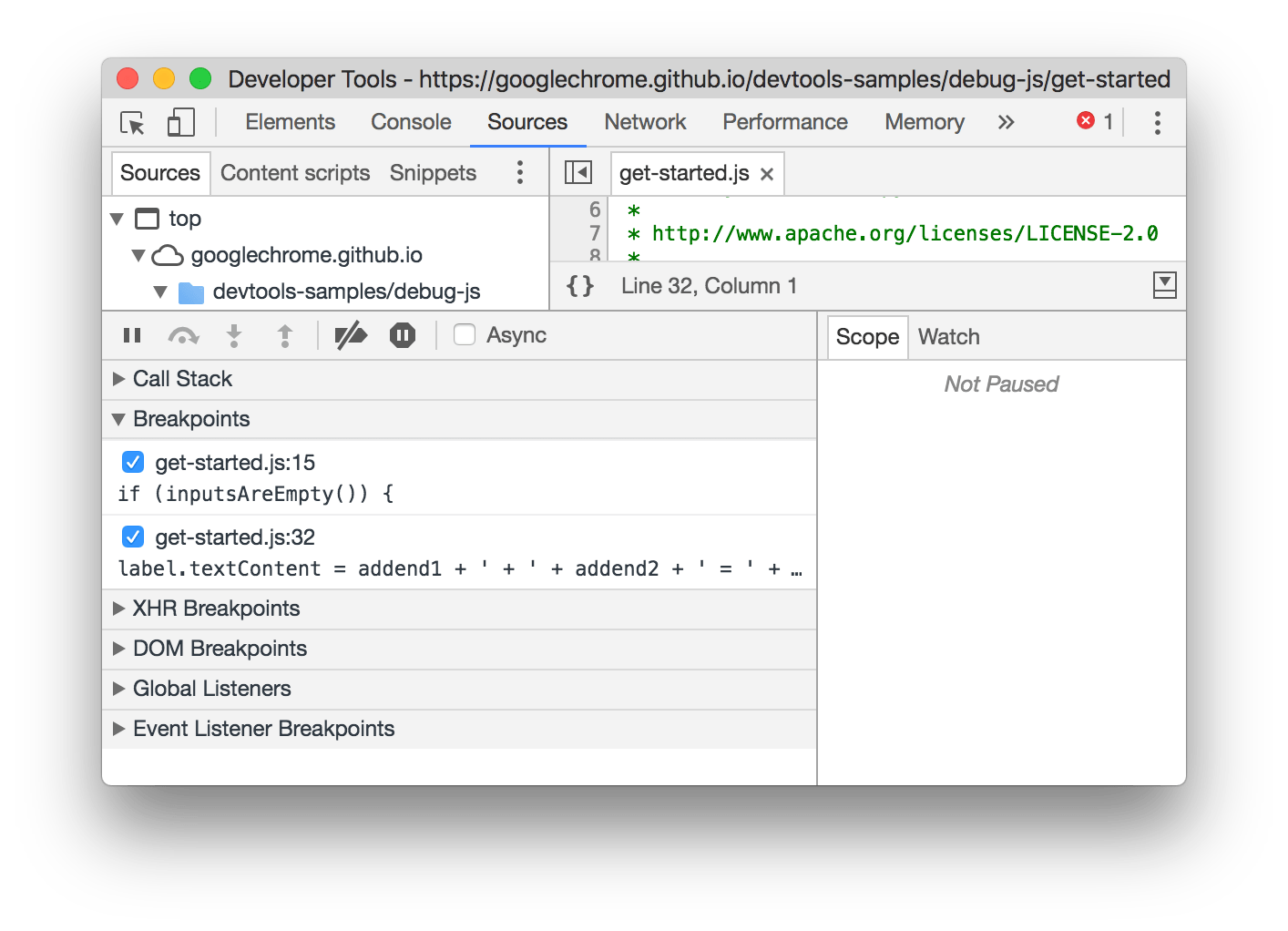
1. Click the **Sources** tab.
2. Open the file containing the line of code you want to break on.
3. Go the line of code.
4. To the left of the line of code is the line number column. Right-click it.
5. Select **Add conditional breakpoint**. A dialog displays underneath the line of code.
6. Enter your condition in the dialog.
7. Press Enter to activate the breakpoint. An orange icon appears on top of the line number column.



**Figure 2**: A conditional line-of-code breakpoint set on line **32**

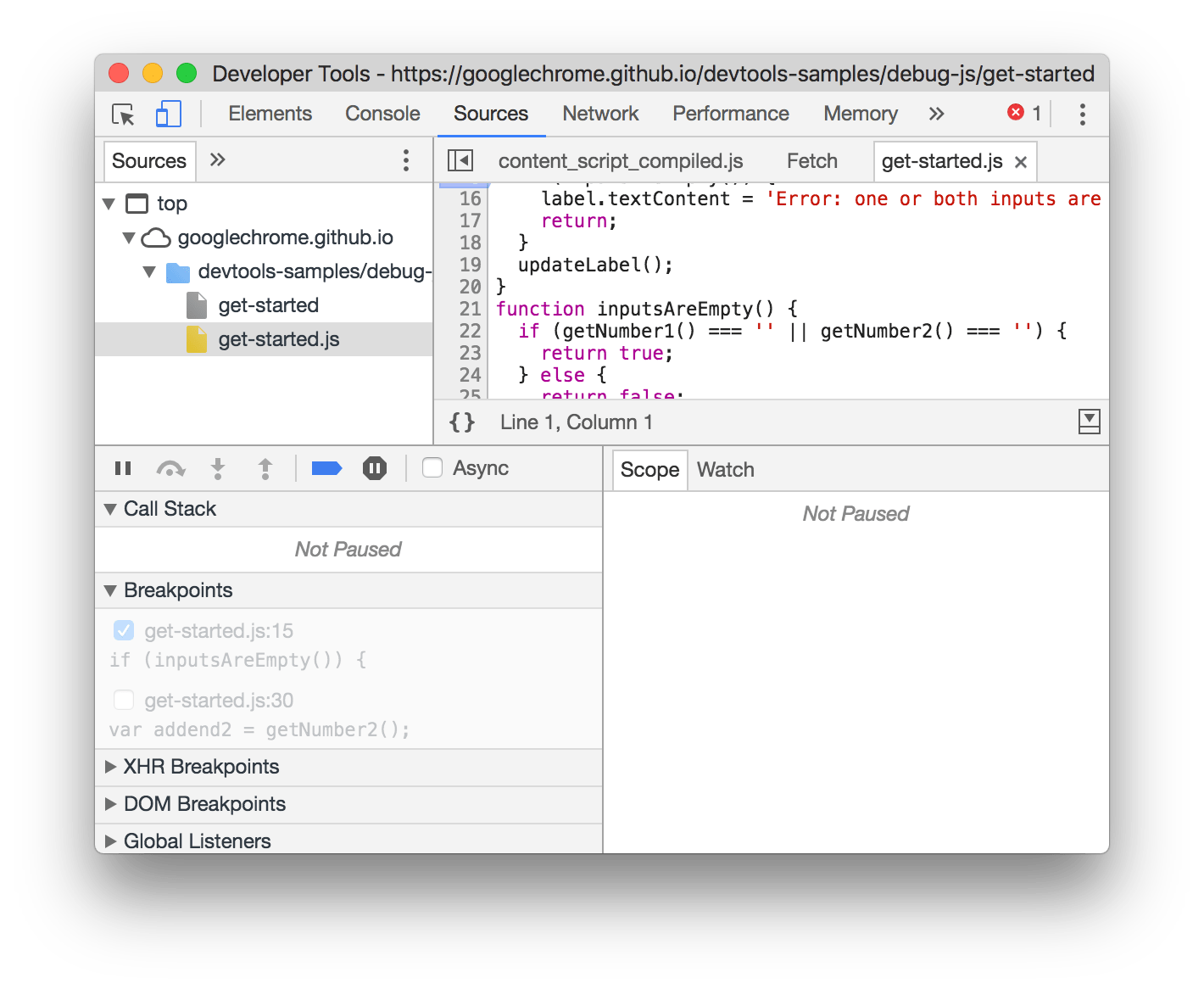
### **Manage line-of-code breakpoints**

Use the **Breakpoints** pane to disable or remove line-of-code breakpoints from a single location.



**Figure 3**: The **Breakpoints** pane showing two line-of-code breakpoints: one on line 15 of get-started.js, another on line 32

* Check the checkbox next to an entry to disable that breakpoint.
* Right-click an entry to remove that breakpoint.
* Right-click anywhere in the **Breakpoints** pane to deactivate all breakpoints, disable all breakpoints, or remove all breakpoints. Disabling all breakpoints is equivalent to unchecking each one. Deactivating all breakpoints instructs DevTools to ignore all line-of-code breakpoints, but to also maintain preserve their enabled state so that they are in the same state as before when you reactivate them.



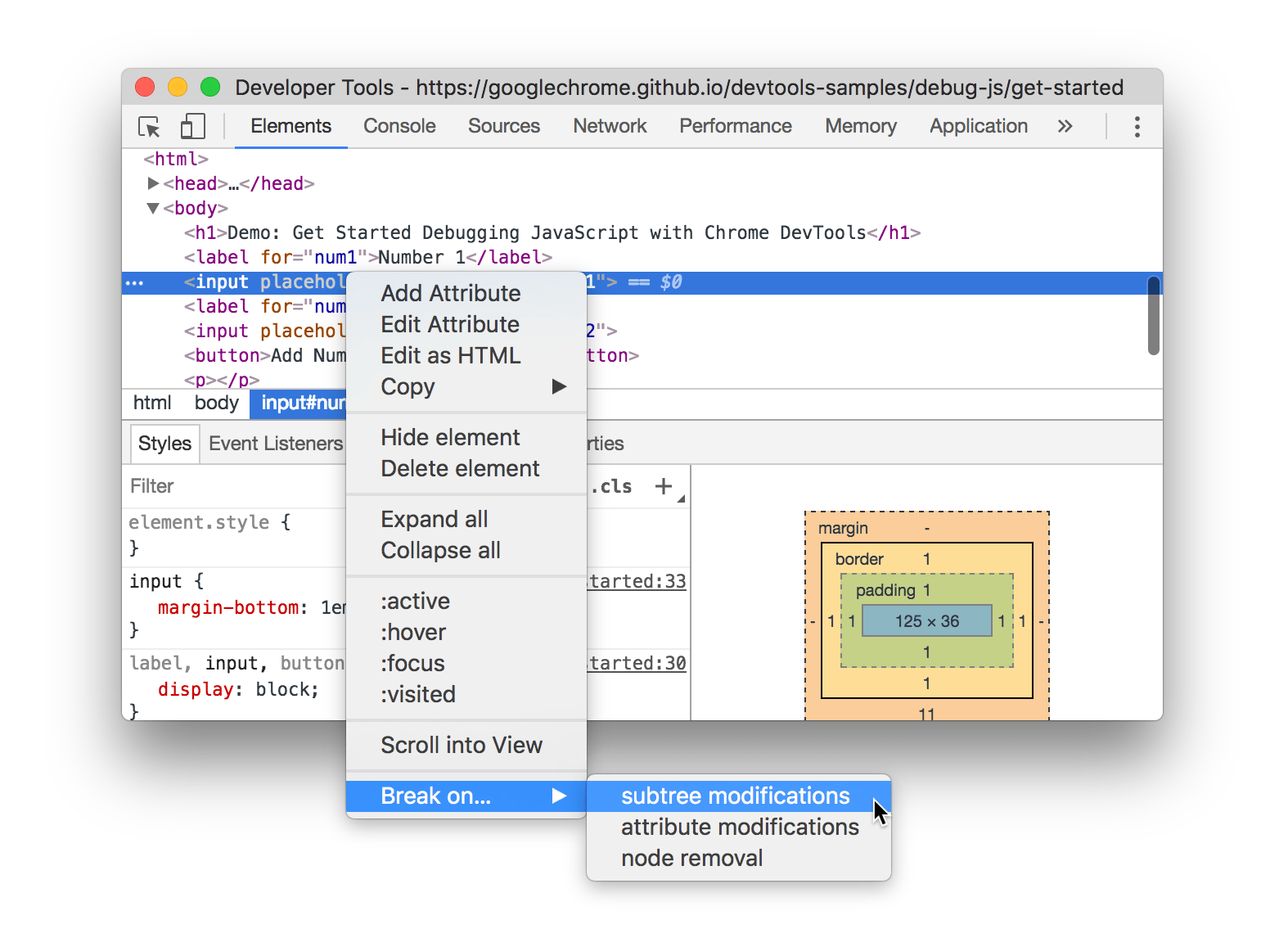
**Figure 4**: Deactivated breakpoints in the **Breakpoints** pane are disabled and transparent

DOM change breakpoints

Use a DOM change breakpoint when you want to pause on the code that changes a DOM node or its children.

To set a DOM change breakpoint:

1. Click the **Elements** tab.
2. Go the element that you want to set the breakpoint on.
3. Right-click the element.
4. Hover over **Break on** then select **Subtree modifications**, **Attribute modifications**, or **Node removal**.



**Figure 5**: The context menu for creating a DOM change breakpoint

### **Types of DOM change breakpoints**

* **Subtree modifications**. Triggered when a child of the currently-selected node is removed or added, or the contents of a child are changed. Not triggered on child node attribute changes, or on any changes to the currently-selected node.
* **Attributes modifications**: Triggered when an attribute is added or removed on the currently-selected node, or when an attribute value changes.
* **Node Removal**: Triggered when the currently-selected node is removed.

## XHR/Fetch breakpoints

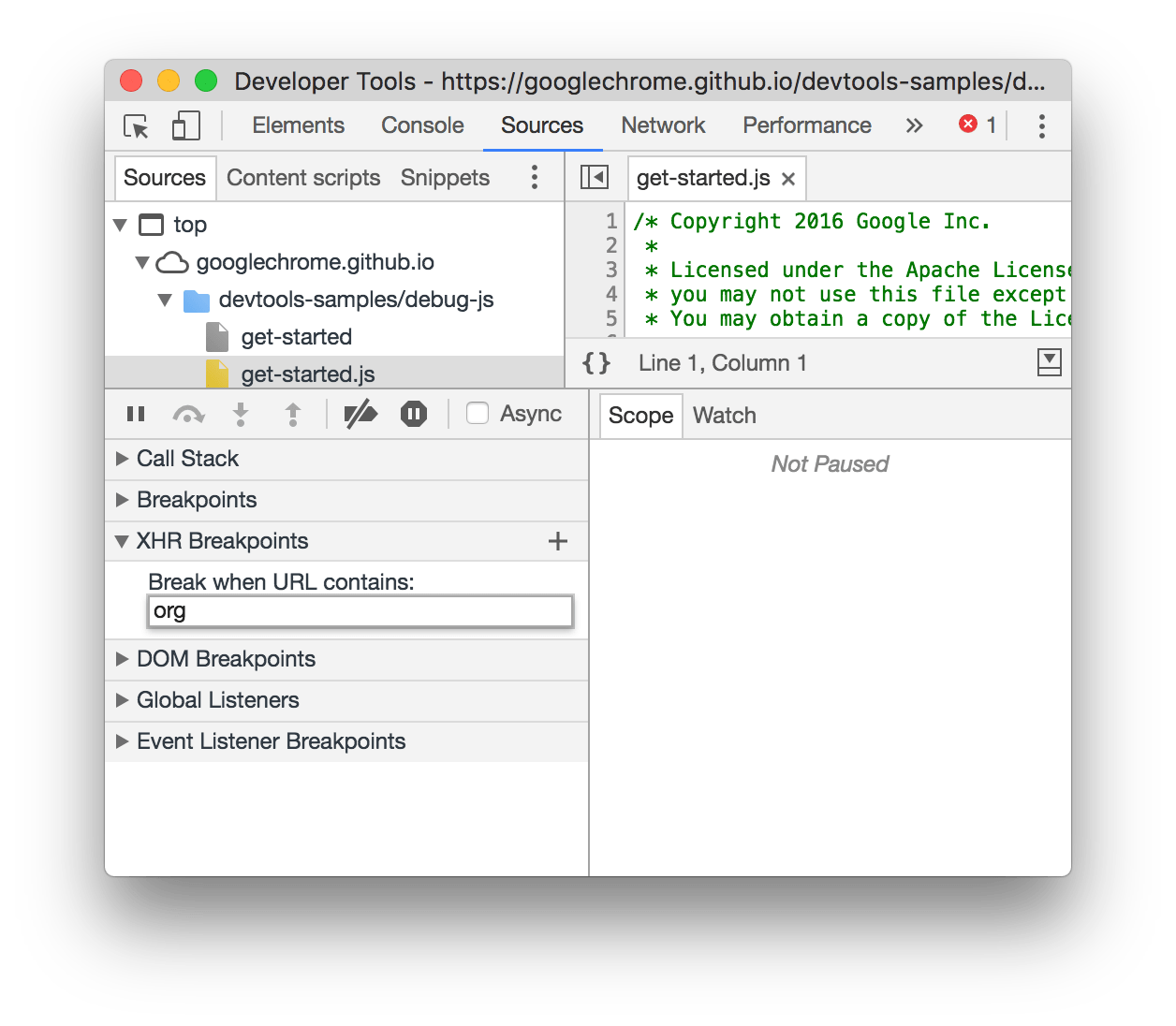
Use an XHR breakpoint when you want to break when the request URL of an XHR contains a specified string. DevTools pauses on the line of code where the XHR calls send().

**Note:** This feature also works with [Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) requests.

One example of when this is helpful is when you see that your page is requesting an incorrect URL, and you want to quickly find the AJAX or Fetch source code that is causing the incorrect request.

To set an XHR breakpoint:

1. Click the **Sources** tab.
2. Expand the **XHR Breakpoints** pane.
3. Click **Add breakpoint**.
4. Enter the string which you want to break on. DevTools pauses when this string is present anywhere in an XHR's request URL.
5. Press Enter to confirm.

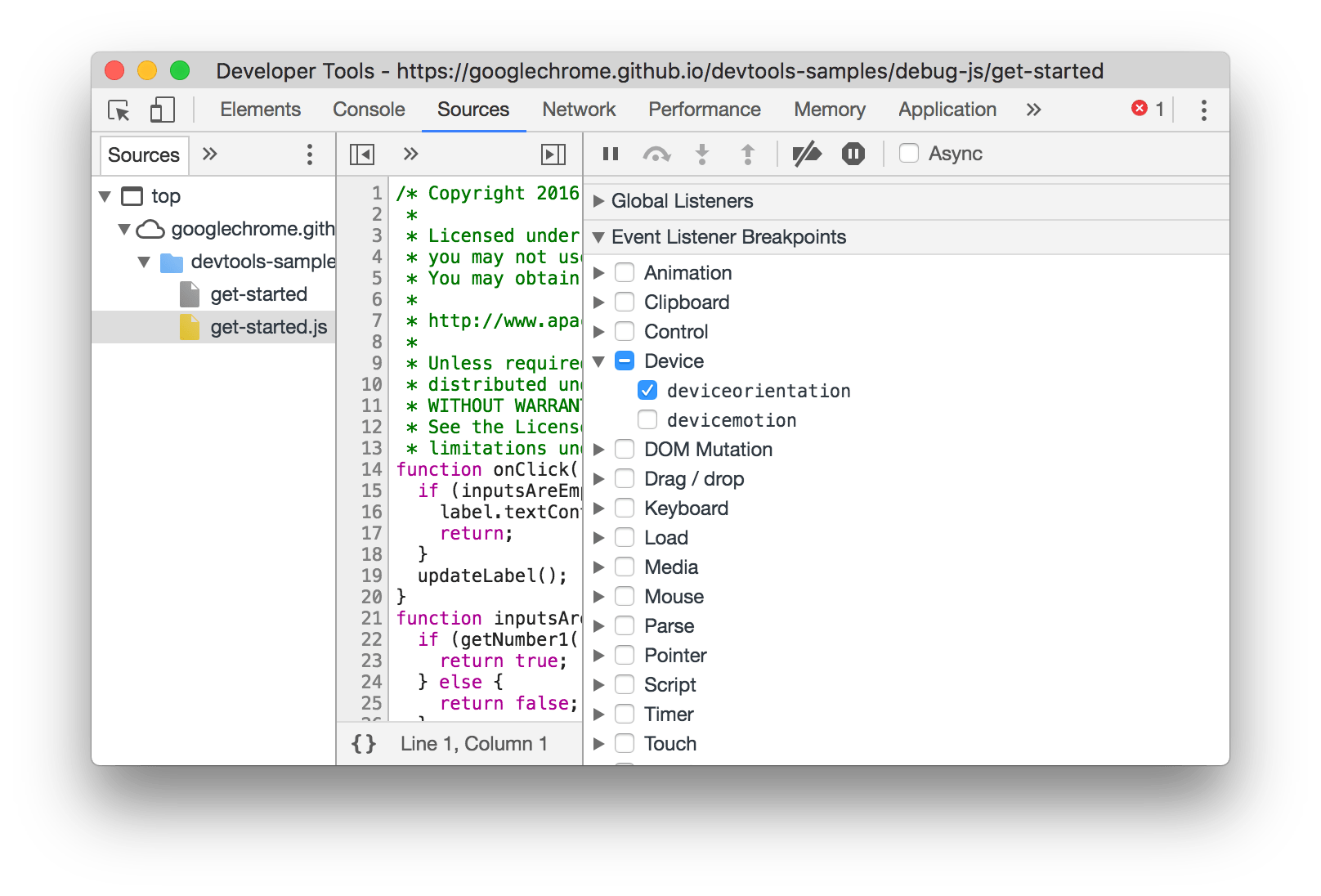


**Figure 6**: Creating an XHR breakpoint in the **XHR Breakpoints** for any request that contains org in the URL

Event listener breakpoints

Use event listener breakpoints when you want to pause on the event listener code that runs after an event is fired. You can select specific events, such as click, or categories of events, such as all mouse events.

1. Click the **Sources** tab.
2. Expand the **Event Listener Breakpoints** pane. DevTools shows a list of event categories, such as **Animation**.
3. Check one of these categories to pause whenever any event from that category is fired, or expand the category and check a specific event.

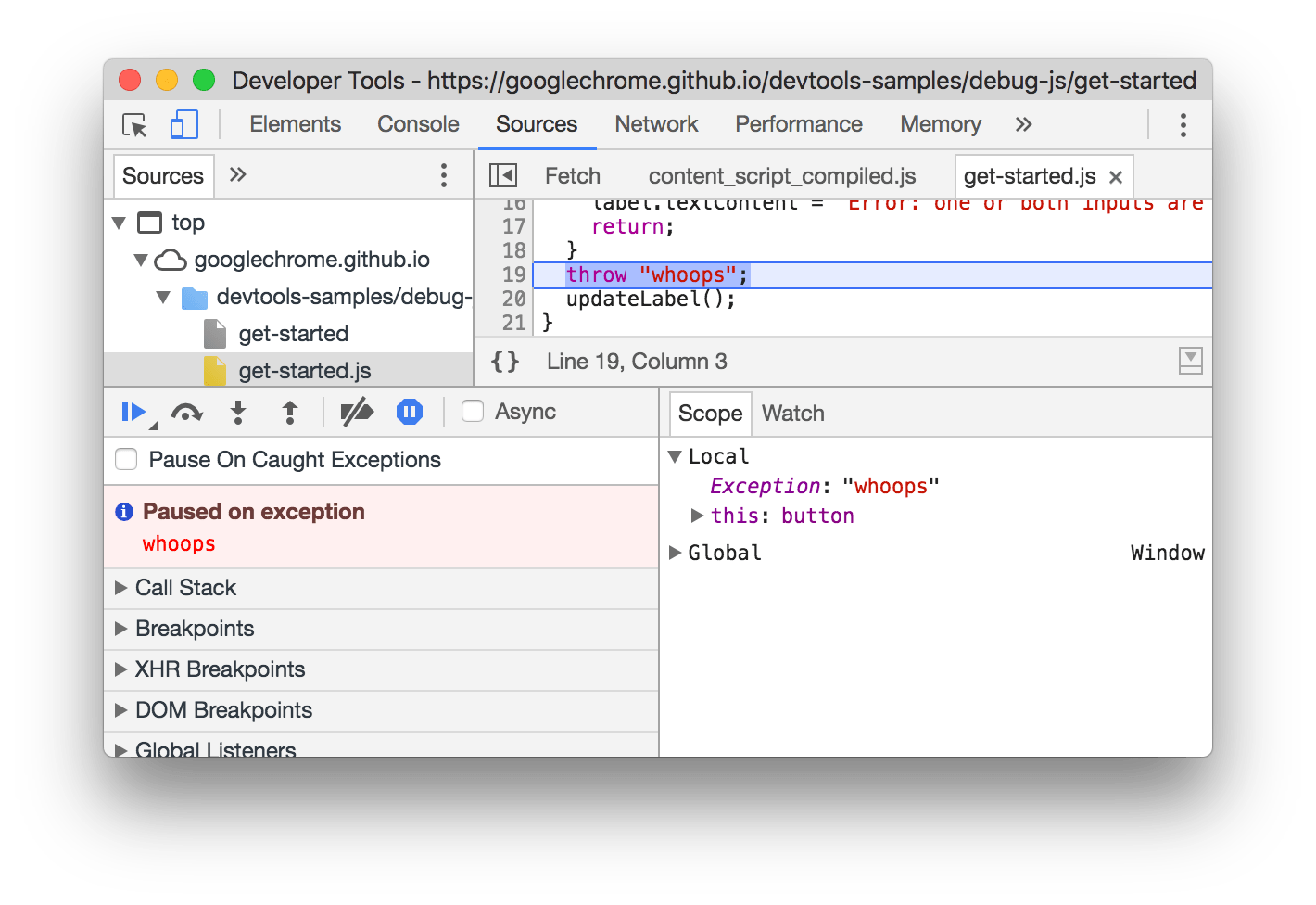


**Figure 7**: Creating an event listener breakpoint for deviceorientation

Exception breakpoints

Use exception breakpoints when you want to pause on the line of code that's throwing a caught or uncaught exception.

1. Click the **Sources** tab.
2. Click **Pause on exceptions**. It turns blue when enabled.
3. (Optional) Check the **Pause On Caught Exceptions** checkbox if you also want to pause on caught exceptions, in addition to uncaught ones.



**Figure 7**: Paused on an uncaught exception

## Function breakpoints

Call debug(functionName), where functionName is the function you want to debug, when you want to pause whenever a specific function is called. You can insert debug() into your code (like a console.log() statement) or call it from the DevTools Console. debug() is equivalent to setting a [line-of-code breakpoint](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints#loc) on the first line of the function.

function sum(a, b) {  
  let result = a + b; // DevTools pauses on this line.  
  return result;  
}  
debug(sum); // Pass the function object, not a string.  
sum();

### **Make sure the target function is in scope**

DevTools throws a ReferenceError if the function you want to debug is not in scope.

(function () {  
  function hey() {  
    console.log('hey');  
  }  
  function yo() {  
    console.log('yo');  
  }  
  debug(yo); // This works.  
  yo();  
})();  
debug(hey); // This doesn't work. hey() is out of scope.

Ensuring the target function is in scope can be tricky if you're calling debug() from the DevTools Console. Here's one strategy:

1. Set a [line-of-code breakpoint](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints#loc) somewhere where the function is scope.
2. Trigger the breakpoint.
3. Call Debug()in the DevTools Console while the code is still paused on your line-of-code breakpoint.

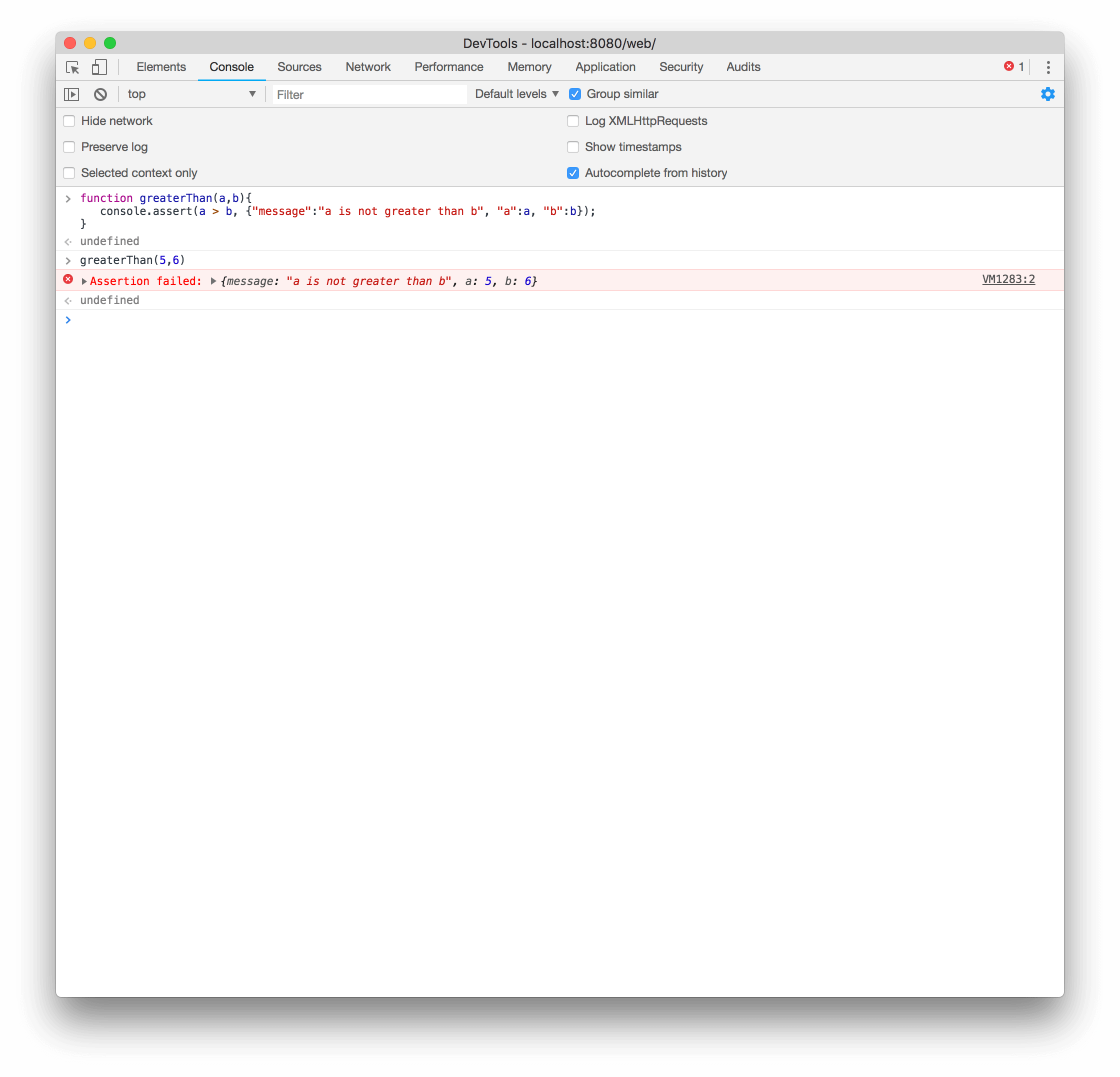
# **Console API Reference**

Use the Console API to write information to the console, create JavaScript profiles, and start a debugging session.

## console.assert(expression, object)

Writes an [error](https://developers.google.com/web/tools/chrome-devtools/console/console-reference#error) to the console when the evaluated expression is false.

function greaterThan(a,b) {  
  console.assert(a > b, {"message":"a is not greater than b","a":a,"b":b});  
}  
greaterThan(5,6);



## console.clear()

Clears the console.

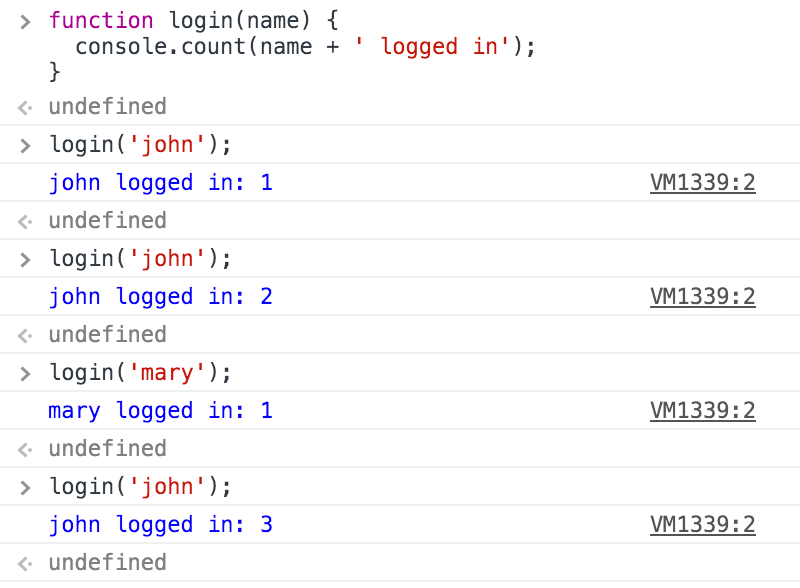
Console.clear();

If the [**Preserve log**](https://developers.google.com/web/tools/chrome-devtools/console/index#preserve-log) checkbox is enabled, console.clear() is disabled. However, pressing the **clear console** button () or typing the shortcut Ctrl+L while the Console is in focus still works.

## console.count(label)

Writes the number of times that count() has been invoked at the same line and with the same label.

function login(name) {  
  console.count(name + ' logged in');  
}



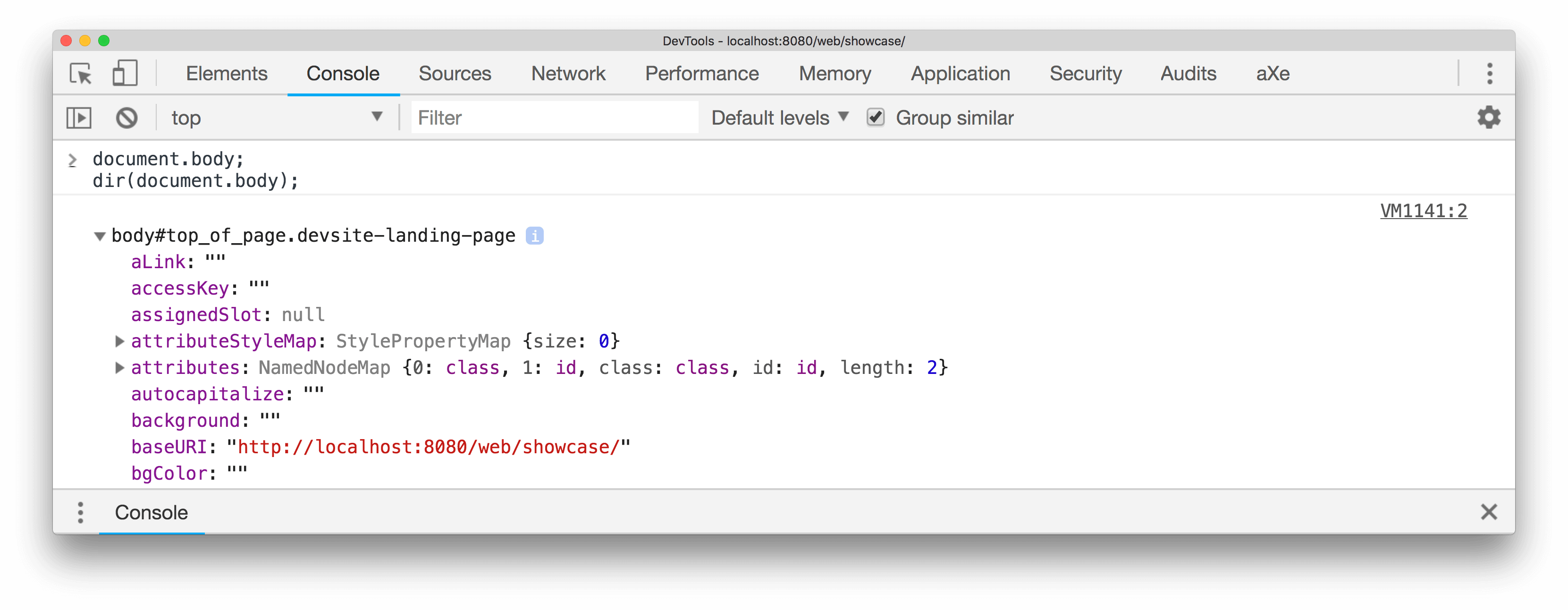
## console.debug(object [, object, ...])

Identical to console.log();

## console.dir(object)

Prints a JavaScript representation of the specified object. If the object being logged is an HTML element, then the properties of its DOM representation are printed, as shown below:

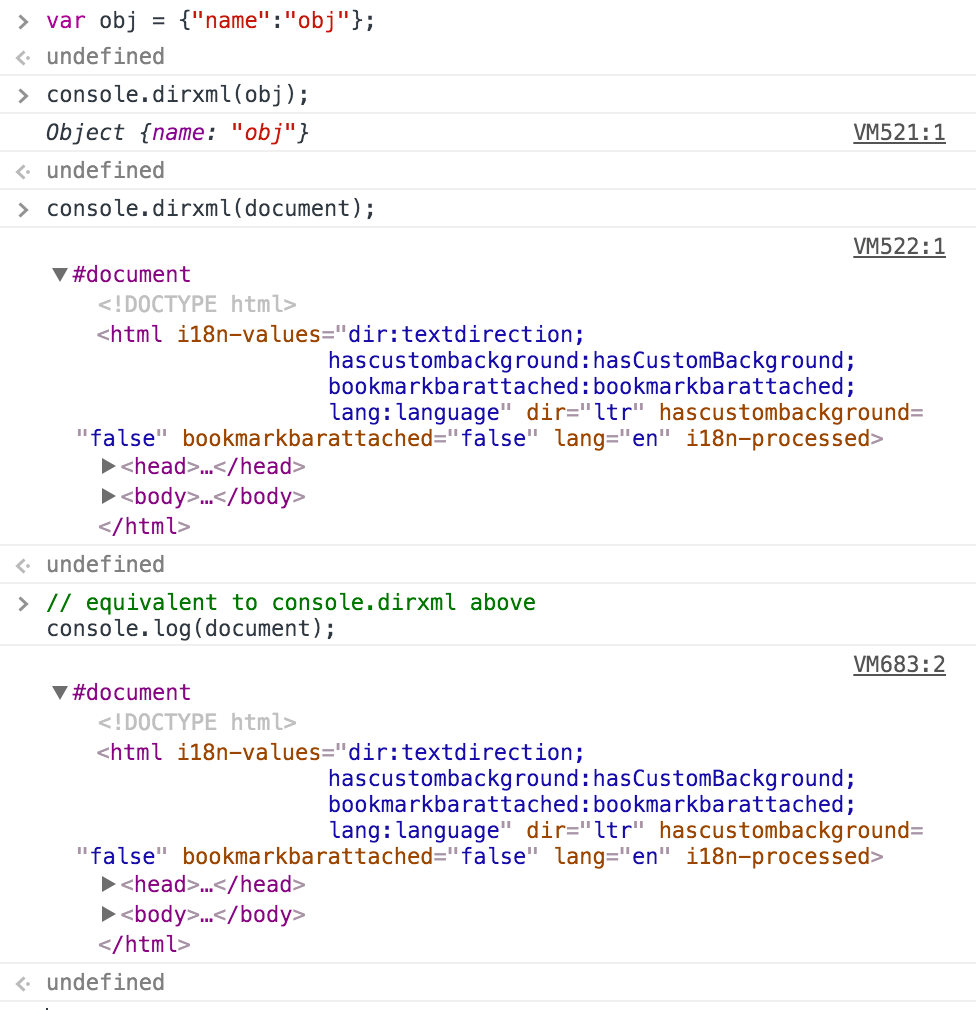
Console.dir(document.body);



## console.dirxml(object)

Prints an XML representation of the descendant elements of object if possible, or the JavaScript representation if not. Calling console.dirxml() on HTML and XML elements is equivalent to calling console.log().

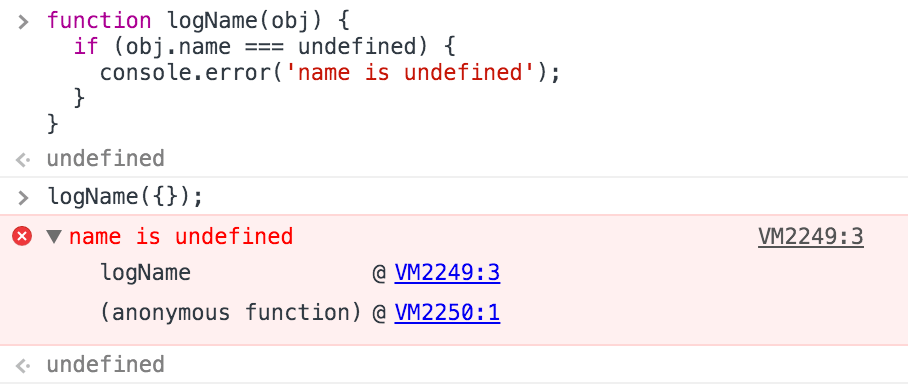
Console.dirxml(document);



## console.error(object [, object, ...])

Prints a message similar to console.log();, styles the message like an error, and includes a stack trace from where the method was called.

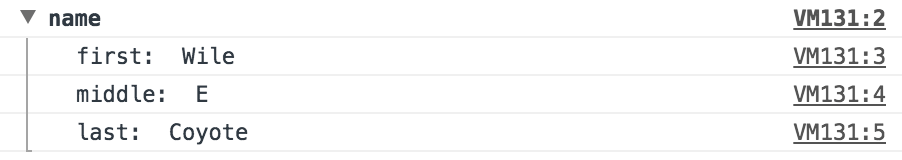
console.error('error: name is undefined');



## console.group(object[, object, ...])

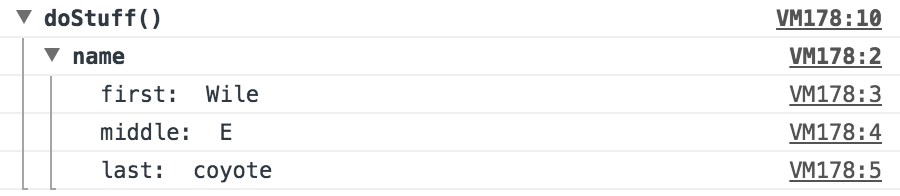
Starts a new logging group with an optional title. All console output that occurs after console.group() and before console.groupEnd() is visually grouped together.

function name(obj) {  
  console.group('name');  
  console.log('first: ', obj.first);  
  console.log('middle: ', obj.middle);  
  console.log('last: ', obj.last);  
  console.groupEnd();  
}  
  
name({"first":"Wile","middle":"E","last":"Coyote"});



You can also nest groups:

function name(obj) {  
  console.group('name');  
  console.log('first: ', obj.first);  
  console.log('middle: ', obj.middle);  
  console.log('last: ', obj.last);  
  console.groupEnd();  
}  
  
function doStuff() {  
  console.group('doStuff()');  
  name({"first":"Wile","middle":"E","last":"coyote"});  
  console.groupEnd();  
}  
  
doStuff();



## console.groupCollapsed(object[, object, ...])

Creates a new logging group that is initially collapsed instead of open.

console.groupCollapsed('status');  
console.log("peekaboo, you can't see me");  
console.groupEnd();

## console.groupEnd()

Closes a logging group.

## console.info(object [, object, ...])

Prints a message like console.log(),but also shows an icon (blue circle with white "i") next to the output.

## console.log(object [, object, ...])

Displays a message in the console. Pass one or more objects to this method. Each object is evaluated and concatenated into a space-delimited string.

Console.log(‘ Hello, Logs!’);

### **Format specifiers**

The first object you pass can contain one or more **format specifiers**. A format specifier is composed of the percent sign (%) followed by a letter that indicates the formatting to apply.

Related Guides:

* [Organizing Console Output](https://developers.google.com/web/tools/chrome-devtools/console/console-write)

## console.profile([label])

Starts a JavaScript CPU profile with an optional label. To complete the profile, call console.profileEnd();. Each profile is added to the **Profiles** panel.

function processPixels() {  
  console.profile("processPixels()");  
  // later, after processing pixels  
  console.profileEnd();  
}

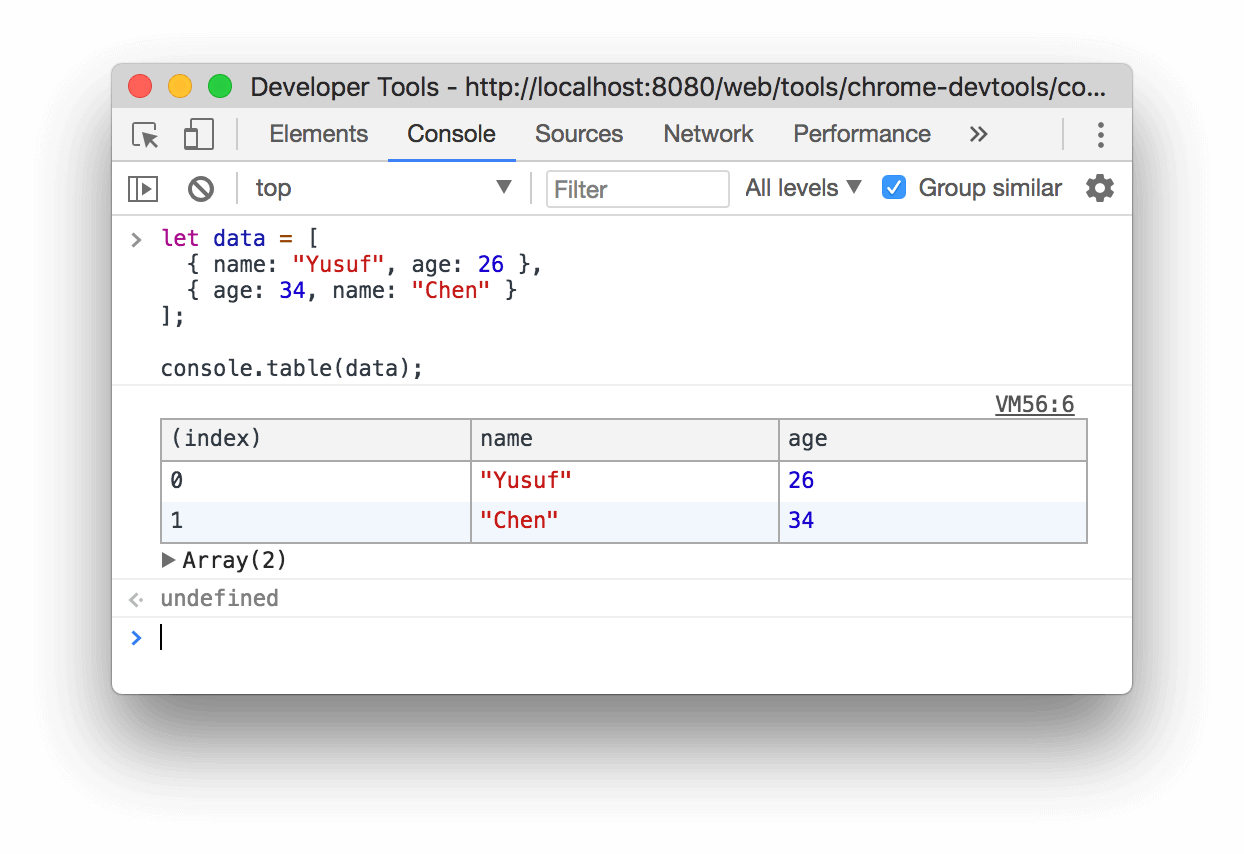
## console.profileEnd()

Stops the current JavaScript CPU profiling session if one is in progress and prints the report to the **Profiles** panel.

## console.table(array)

Logs an array of objects as a table.

let data = [  
  { name: "Yusuf", age: 26 },  
  { age: 34, name: "Chen" }  
];  
  
console.table(data);



## console.time([label])

Starts a new timer. Call  console.timeEnd() to stop the timer and print the elapsed time to the Console.

console.time();  
var arr = new Array(10000);  
for (var i = 0; i < arr.length; i++) {  
  arr[i] = new Object();  
}  
console.timeEnd();  
// default: 3.696044921875ms

Pass an optional label to change the output text that precedes the elapsed time. Call console.timeEnd() with the same label to end the timer.

console.time('total');  
var arr = new Array(10000);  
for (var i = 0; i < arr.length; i++) {  
  arr[i] = new Object();  
}  
console.timeEnd('total');  
// total: 3.696044921875ms

Use labels to run multiple timers at the same time.

console.time('total');  
console.time('init arr');  
var arr = new Array(10000);  
console.timeEnd('init arr');  
for (var i = 0; i < arr.length; i++) {  
  arr[i] = new Object();  
}  
console.timeEnd('total');  
// init arr: 0.0546875ms  
// total: 2.5419921875ms

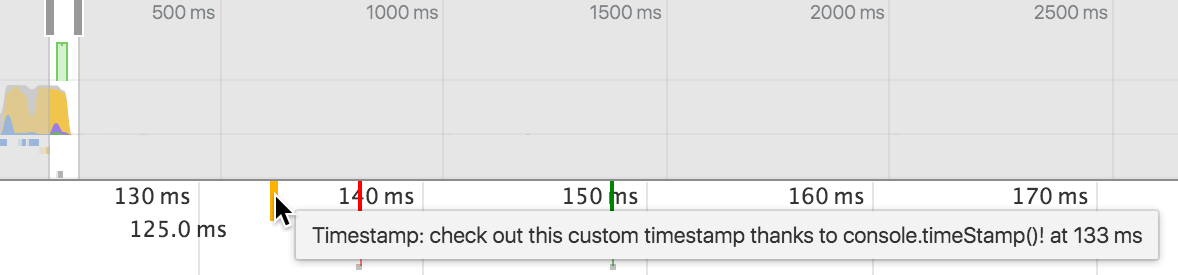
## console.timeEnd([label])

Stops a timer.

## console.timeStamp([label])

Adds an event to the **Timeline** during a recording session.

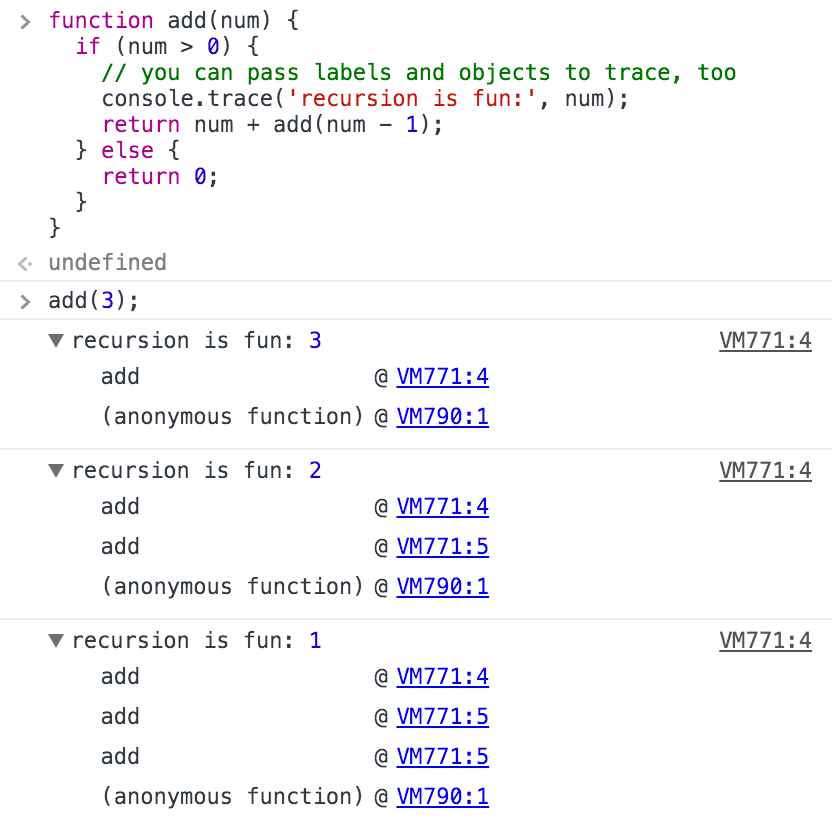
console.timeStamp('check out this custom timestamp thanks to console.timeStamp()!');



## console.trace(object)

Prints a stack trace from the point where the method was called.

Console.trace();



## console.warn(object [, object, ...])

Prints a message like console.log(), but also displays a yellow warning icon next to the logged message.

console.warn('user limit reached!');



Filtering by severity level is equivalent to the following:

|  |  |
| --- | --- |
| Option & Shows | |
| All | Shows all console output |
| Errors | Only show output from [console.error()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference" \l "consoleerrorobject--object-). |
| Warnings | Only show output from [console.warn()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference" \l "consolewarnobject--object-). |
| Info | Only show output from [console.info()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference#consoleinfoobject--object-). |
| Logs | Only show output from [console.log()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference#consolelogobject--object-). |
| Debug | Only show output from [console.timeEnd()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference" \l "consoletimeendlabel) and [console.debug()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference" \l "consoledebugobject--object-). |

|  |  |
| --- | --- |
| Setting & Description | |
| Hide network messages | By default, the console reports network issues. Turning this on instructs the console to not show logs for these errors. For example, 404 and 500 series errors will not be logged. |
| Log XMLHttpRequests | Determines if the console logs each XMLHttpRequest. |
| Preserve log upon navigation | Persists the console history during page refreshes or navigation. |
| Show timestamps | Prepends a timestamp to each console message showing when the call was made. Useful for debugging when a certain event occurred. This will disable message stacking. |
| Enable custom formatters | Control the [formatting](https://docs.google.com/document/d/1FTascZXT9cxfetuPRT2eXPQKXui4nWFivUnS_335T3U/preview) of JavaScript objects. |

* Use [console.log()](https://developers.google.com/web/tools/chrome-devtools/console/console-reference#log) for basic logging
* Use [console.error()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference" \l "error) and [console.warn()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference" \l "warn) for eye-catching stuff
* Use [console.group()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference" \l "group) and [console.groupEnd()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference" \l "groupend) to group related messages and avoid clutter
* Use [console.assert()](https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference" \l "assert) to show conditional error messages

Used in console. Log(), .warn(), .info(), ….

|  |  |
| --- | --- |
| Specifier | Output |
| %s | Formats the value as a string |
| %i or %d | Formats the value as an integer |
| %f | Formats the value as a floating point value |
| %o | Formats the value as an expandable DOM element. As seen in the Elements panel |
| %O | Formats the value as an expandable JavaScript object |
| %c | Applies CSS style rules to the output string as specified by the second parameter |