Front End Debugging

How do I Debug my Program? This is a critical skill in programming, and you must learn how to do it well. It is largely an art; there is no single way to do it, and the best way to learn it is to do lots of it. Here are some ideas to get you started, based on our experience.

The Zen of Debugging

The biggest obstacle to overcome is your belief that your code is right and the computer is somehow wrong. Compiler and library bugs exist, but they are very rare compared to programming errors. So, 99.99999% of the time the computer is doing exactly what you told it to do, and you just told it wrong. Your task is to find the wrong thing you said - it is in there somewhere!

The overall best way to find a bug is to gather information about the conditions under which the bug happens and exactly what the bug does. There are a variety of ways to do this, such as trying different input to see if there is a pattern in the bug's behavior, or checking program flow and variable values.

As you come to understand the bug, you can begin to deduce what parts of your program are relevant, and start to zero-in on ever smaller parts of your program. Eventually you can ask "What does this line of code do compared to what it is supposed to do?" and you will find the exact wrong thing in your code.

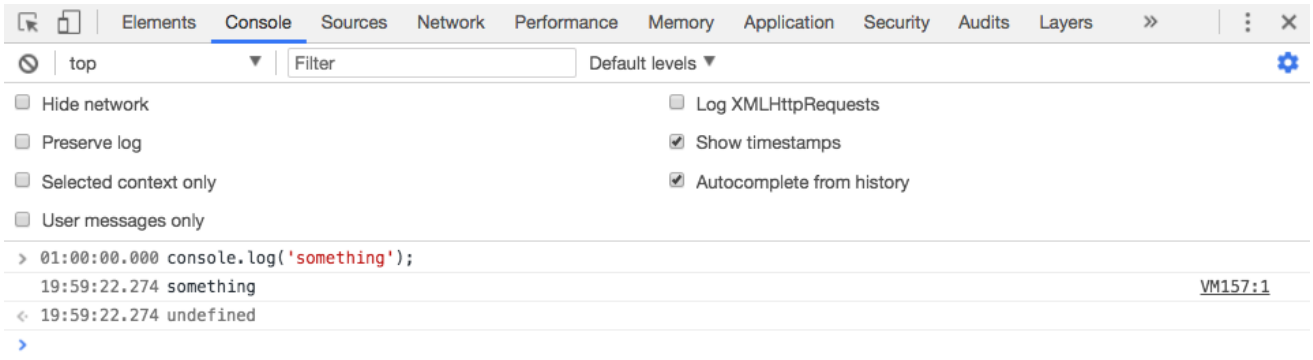
Most of these are "stupid" small errors or typos that passed the compiler, but expect many of them to be "profound" errors, especially if you are learning the programming language and the software concepts.

Console

To open Chrome DevTools:

- Select More Tools > Developer Tools from Chrome’s Main Menu.
- Right-click a page element and select Inspect.
- Press Command+Option+I (Mac) or Control+Shift+I (Windows, Linux).

Look at the Console tab and what is there.



Front End Debugging Help

There are a lot of strategies to debug the problem in your code. From my experience there is no a silver bullet to manage all the situations with only one strategy. Generally I found three main strategies ranged by common usage from the most to the least popularity.

1. **Brute Force method** This method is the most common and least efficient. Here developer just prints out all relevant memory stacks and observes the potential place of inconsistency with expected data.
2. **Back Tracking method** This approach is good in a quite small applications. The process starts from the place where a particular symptom gets detected. From there backward tracing is done across the entire source code. The steps here: observing who was a caller for a particular function, getting to its caller, checking if problem related to that function appeared there. If not, going to the caller of that function, and so on.
3. **Cause Elimination method** This approach is used quite rarely. It's also called induction and deduction. Data related to the error occurrence are organized to isolate potential causes. Developer create a hypothesis of a bug cause and compose a special data shape to pass into function which should prove or disprove that hypothesis.
4. **Experience method** It needs a good knowledge of the product and its potential weak places. In such case developer already knows what could be the reason of a reported bug based on knowledge that API could unexpectedly be changed and source code is sensitive to such change. Or data type was changed but source code doesn't have an appropriate fail check for it.

Let's overview some practical methods of debug process.

Simple debugging approach with logging output

In such approach developer just adds print outs into the code (console.log, alert, etc). This helps to understand some intermediate states related to the code. This approach stays somewhere in the middle between Brute Force and Back Tracking methods (closer to a Brute Force however). In the system where in development mode there are already tons of logging messages, it becomes quite

painful to add extra logs and match these logs with a concrete place in the code. And, of course, all this information should be kept in developers mind and there is high chance to shift focus from problem solving to remembering console.logs placement and getting your mental stack overflow at the end. And in such case developer should already rely on source-mapping tools to not deal with minified code.

Lab Activity

Get Started with Debugging JavaScript in Chrome DevTools

Step 1: Reproduce the bug

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

Enter 5 in the Number 1 text box.

Enter 1 in the Number 2 text box.

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.

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

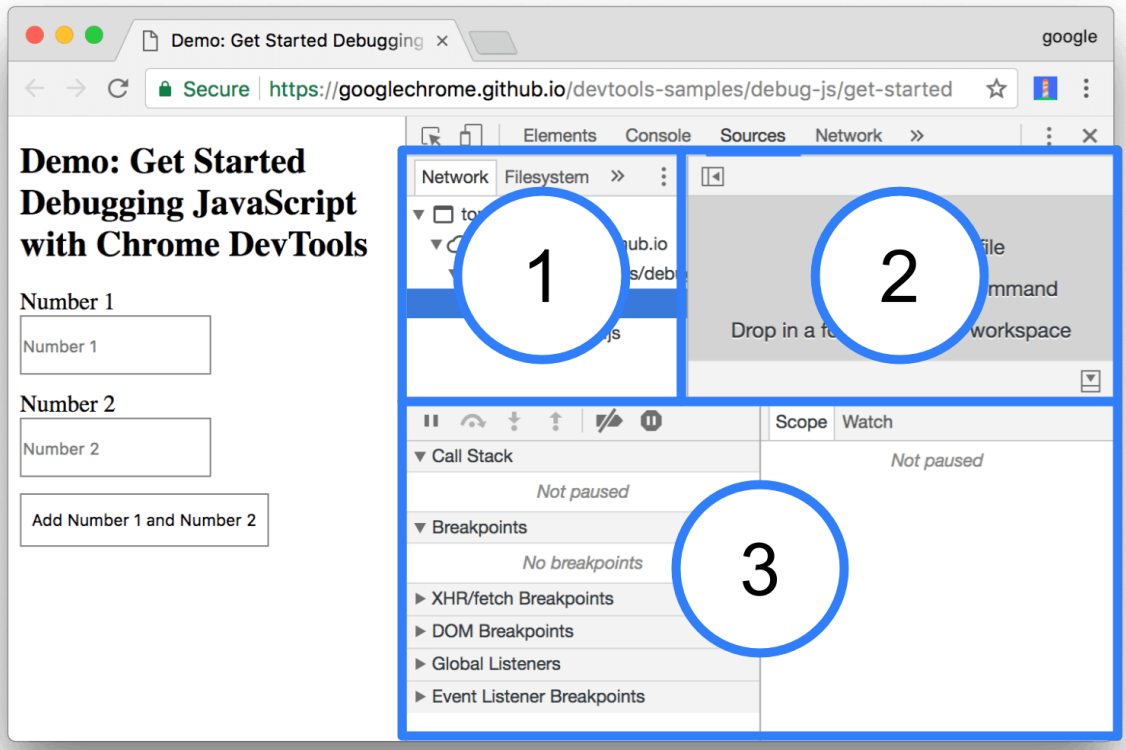
Step 2: Get familiar with the Sources panel UI

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.

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

2. Click the Sources tab.

The Sources panel UI has 3 parts:



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.

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:

```
function updateLabel() {
  var addend1 = getNumber1();
  console.log('addend1:', addend1);
  var addend2 = getNumber2();
  console.log('addend2:', addend2);
  var sum = addend1 + addend2;
  console.log('sum:', sum);
  label.textContent = addend1 + ' + ' + addend2 + ' = ' + sum;
}
```

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 Expand icon. 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.
4. 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() {}
```

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

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 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:

1. On the Sources panel of DevTools, click Step into next function call 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()) {}
```

2. Click Step over next function call 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.

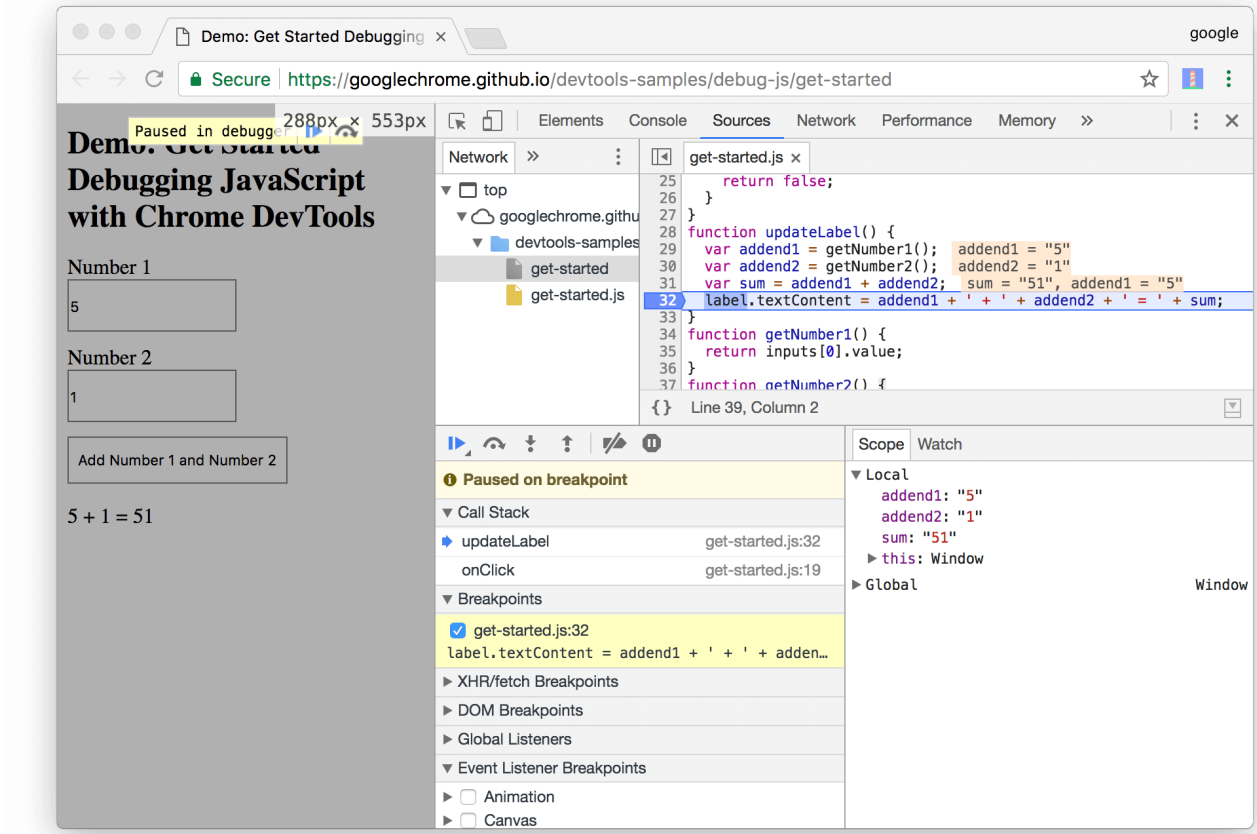
Step 5: Set a line-of-code breakpoint

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;
```

2. 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.
3. Click Resume script execution 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.

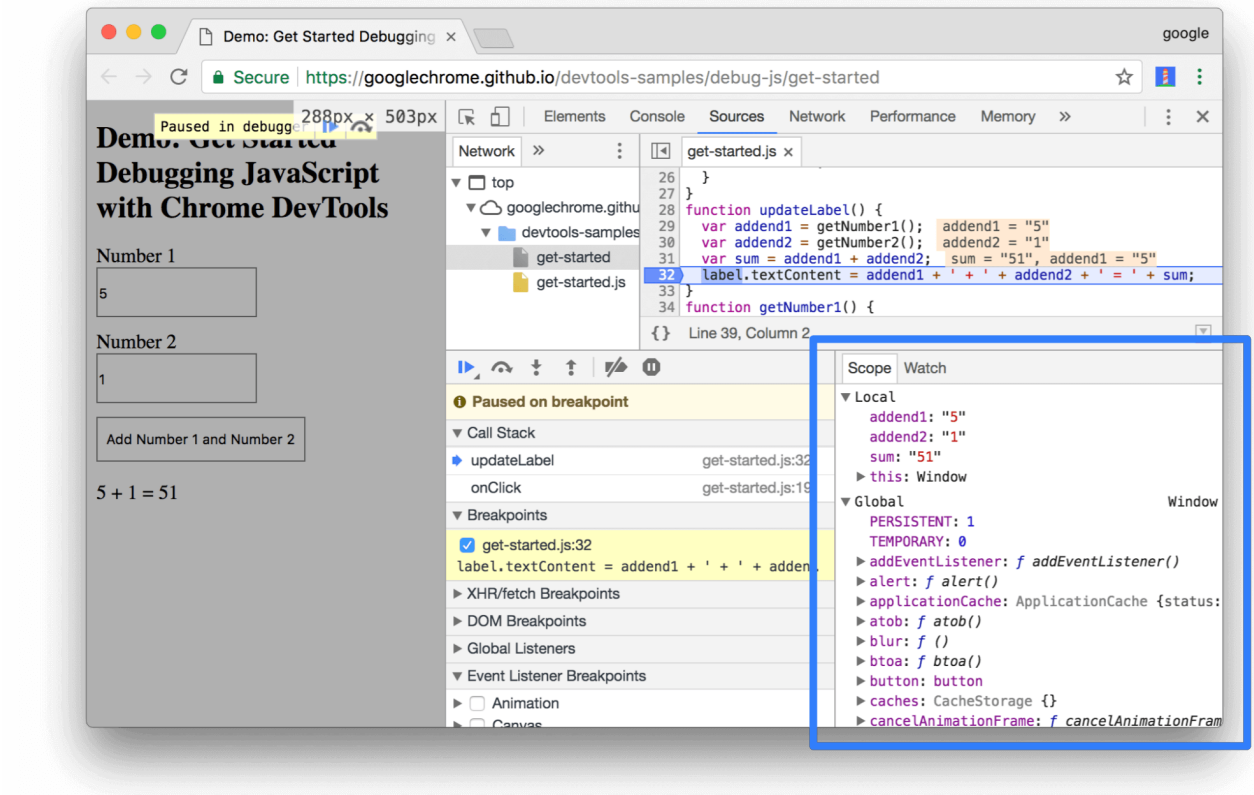


Step 6: Check variable values

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

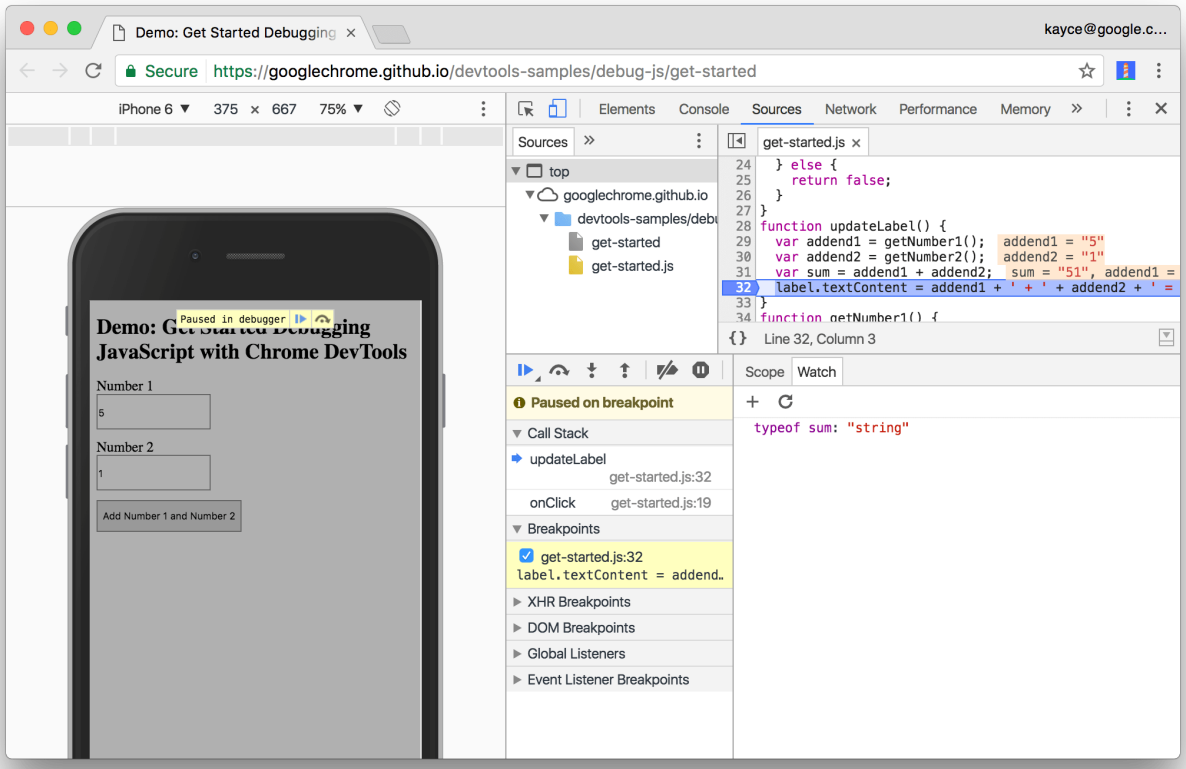
When you're paused on a line of code, the Scope pane shows you what local and global variables are currently defined, along with the value of each variable. It also shows closure variables, when applicable. Double-click a variable value to edit it. When you're not paused on a line of code, the Scope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



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 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.



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.

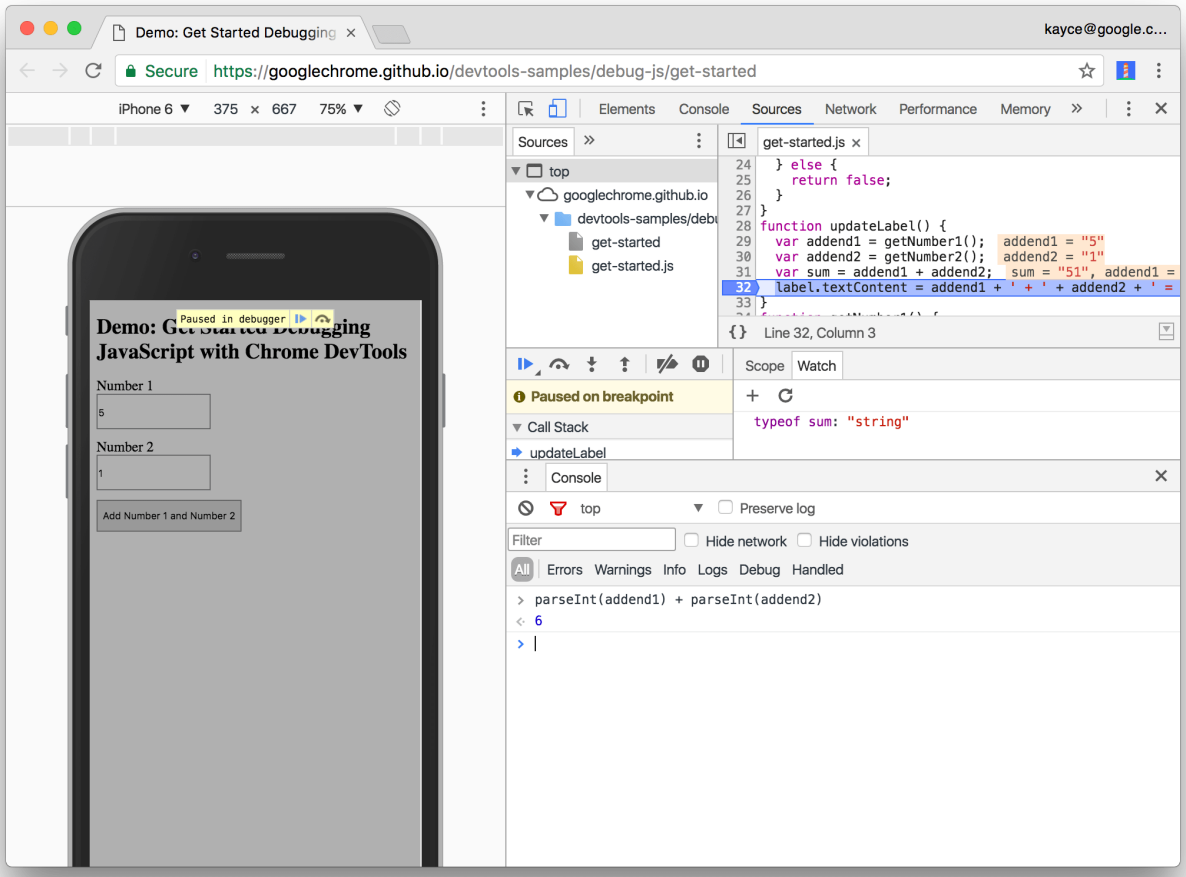
Method 3: The Console

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:

If you don't have the Console drawer open, press Escape to open it. It opens at the bottom of your DevTools window.

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.

Press `Enter`. DevTools evaluates the statement and prints out 6, which is the result you expect the demo to produce.



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 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 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.

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.

References:

1. https://developers.google.com/web/tools/chrome-devtools/javascript/