

# Theseus: Understanding Asynchronous Code

Tom Lieber [tom@alltom.com](mailto:tom@alltom.com)



## The Problem: asynchronous control flow is convoluted

JavaScript relies on [callback functions](#) for events (UI, network, etc.), blocking I/O operations, and control structures like loops. They are [error-prone](#) and many libraries exist to help structure them.

Callbacks turn questions about whether a point in the code has been executed into [reachability questions that are difficult to answer](#) with ordinary call graphs and most debugging tools.

## 0 clicks: reachability coloring + call counts

```
0 calls  mouseClicked({ button: 1 }, function () {
38   $('body').append('first click recorded!');
39 });
40
1 call   keyPressed({ key: 32 }, function () {
42   $('body').append('keystroke recorded!');
43 });
44
0 calls  mouseClicked({ button: 1 }, function () {
46   $('body').append('second click recorded!');
47 });
48 </script>
```

Code that has never been executed is colored gray. Call counts are shown next to every function definition.

Users tend to leave the editor open on one side of the screen and [watch code light up as they interact with their application](#).



Download Theseus today!

## My Approach: visualize program behavior within the editor

Show information about a program's run-time behavior within the code using [syntax highlighting and widgets in the margin](#). These provide [information](#) and can answer questions before the user thinks to ask them.

Update information in real-time, making the code [part of the program's interface](#).

[Extend the call graph](#) with links for asynchronous event chains. Reveal caller/callee relationships in the program's log.

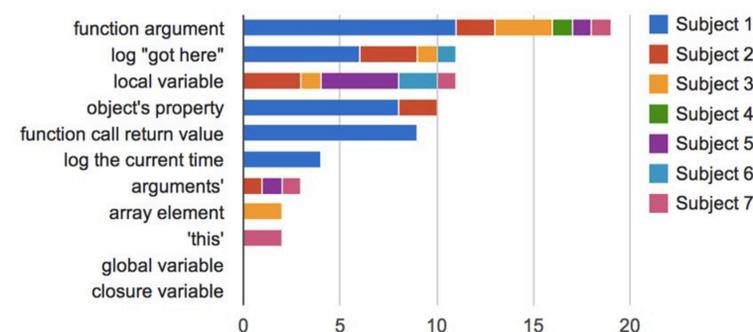
## 1 click: retroactive log

```
5 calls  function reverse(str) {
5     return str.split('').reverse().join('');
6 }
```

Log
reverse (index.html:4) 2:30:46.827 str = "Tom" return value = "moT" Backtrace →
reverse (index.html:4) 2:30:46.827 str = "Rob" return value = "boR" Backtrace →
reverse (index.html:4) 2:30:46.828 str = null exception = ▶ TypeError: Cannot read property 'split' of null
reverse (index.html:4) 2:30:46.828 str = "foo" return value = "oof" Backtrace →
reverse (index.html:4) 2:30:46.828 str = "bar" return value = "rab" Backtrace →

Click a function's call count to retroactively log the [arguments, return value, and any thrown exceptions](#) of every invocation of that function.

Statistics from a lab study about the types of values people print and inspect (without Theseus) suggest that inferring the values to print based on context may be sufficient in most cases:



## 2 clicks: asynchronous call tree

```
4 calls  function fetch(id, callback) {
37   var stream = new Stream(id);
38   var allData = '';
39
12 calls  stream.on('data', function (data) {
41     allData += data;
42   });
43
2 calls  stream.on('end', function () {
45     callback(null, allData);
46   });
47
2 calls  stream.on('error', function (err) {
49     callback(err);
50   });
51
52   return stream;
53 }
```

Log
fetch (index7.html:36) 2:03:44.794 id = 1 callback = ▶ Function return value = ▶ [object Object] Backtrace →
('error' handler) (index7.html:48) ASYNC 2:03:45.070 err = "stream error" this = ▶ [object Object] Backtrace →
fetch (index7.html:36) 2:03:44.796 id = 2 callback = ▶ Function return value = ▶ [object Object] Backtrace →
fetch (index7.html:36) 2:03:44.797 id = 3 callback = ▶ Function return value = ▶ [object Object] Backtrace →
('error' handler) (index7.html:48) ASYNC 2:03:45.094 err = "stream error" this = ▶ [object Object] Backtrace →
fetch (index7.html:36) 2:03:44.797 id = 4 callback = ▶ Function return value = ▶ [object Object] Backtrace →

Invocations with a caller/callee relationship are shown nested in a call tree. Event chains are shown the same way and flagged with 'async'.

This combination of console and call tree untangles the control flow of complex asynchronous code, visualizing control- and data-flow together.