Exception and Error Handling

By Meggin Kearney

Meggin is a Tech Writer

By Flavio Copes

Flavio is a Full Stack Developer

Chrome DevTools provides tools to help you fix web pages throwing exceptions and debug errors in your JavaScript.

Page exceptions and JavaScript errors are actually quite useful - if you can get to the details behind them. When a page throws an exception or a script produces an error, the Console provides specific, reliable information to help you locate and correct the problem.

In the Console you can track exceptions and trace the execution path that led to them, explicitly or implicitly catch them (or ignore them), and even set error handlers to automatically collect and process exception data.

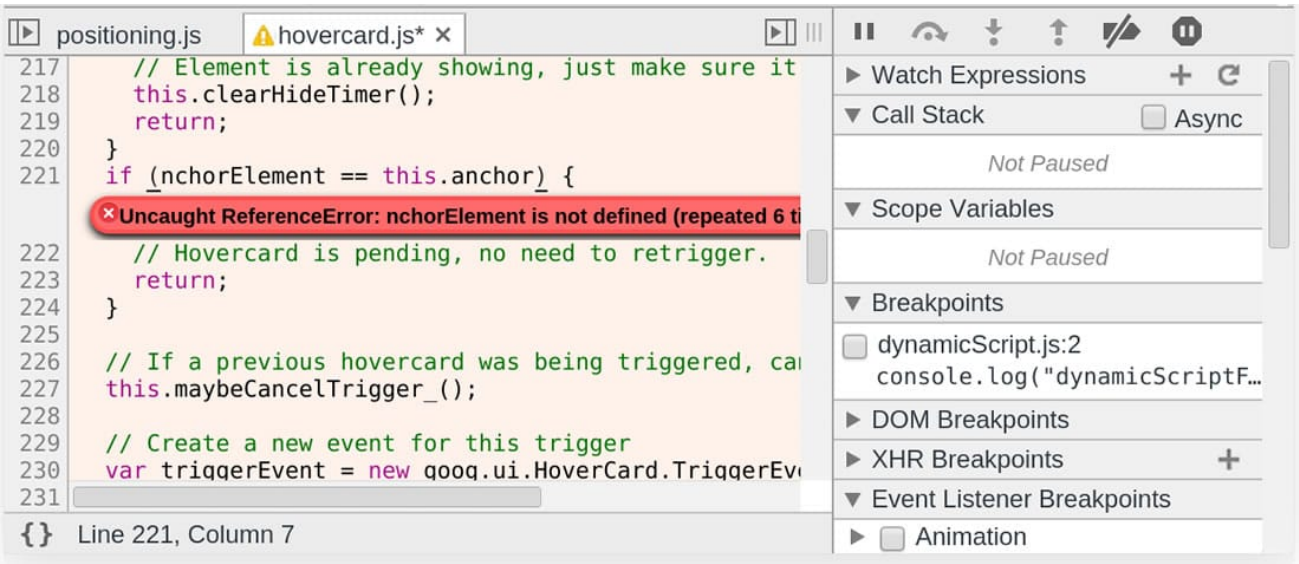
TL;DR

- Turn on Pause on Exceptions to debug the code context when the exception triggered.
- Print current JavaScript call stack using `console.trace`.
- Place assertions in your code and throw exceptions using `console.assert()`.
- Log errors happening in the browser using `window.onerror`.

Track exceptions

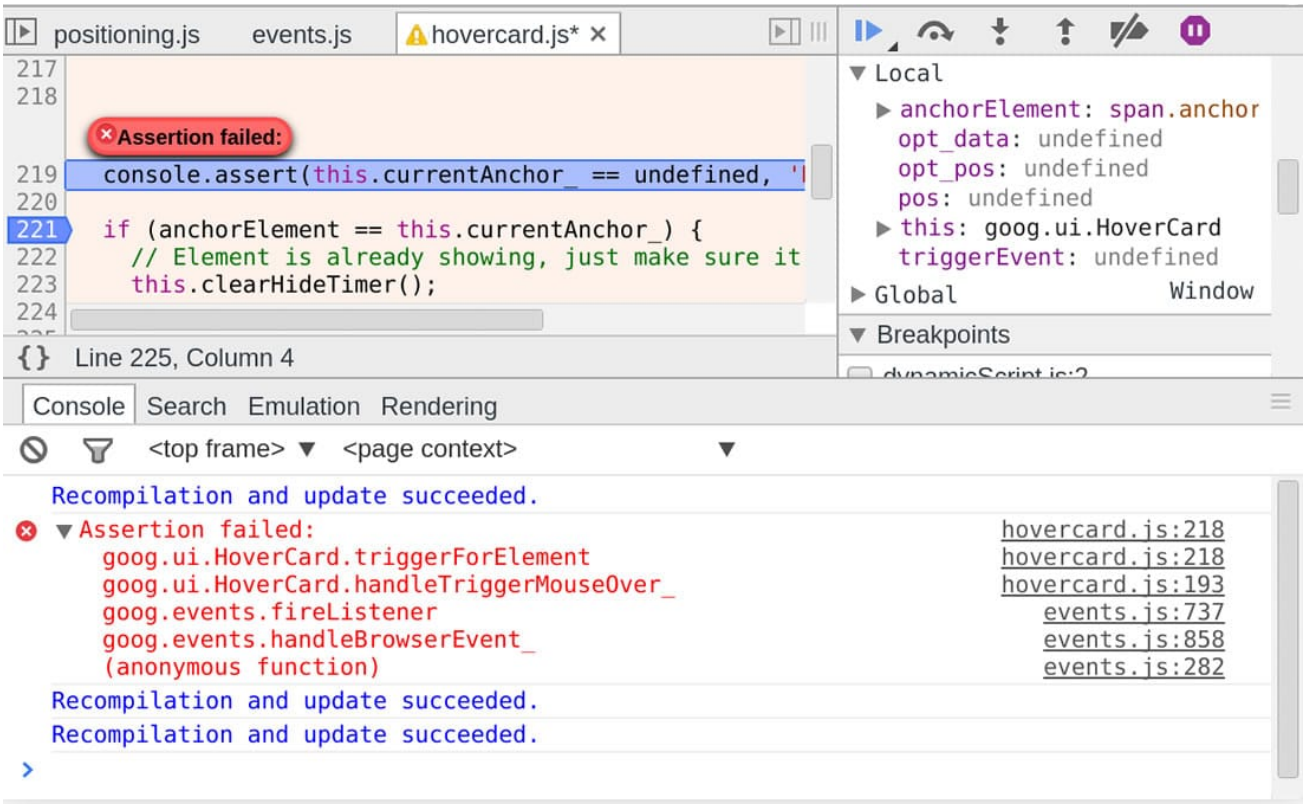
When something goes wrong, open the DevTools console (`Ctrl+Shift+J` / `Cmd+Option+J`) to view the JavaScript error messages. Each message has a link to the file name with the line number you can navigate to.

An example of an exception:



View exception stack trace

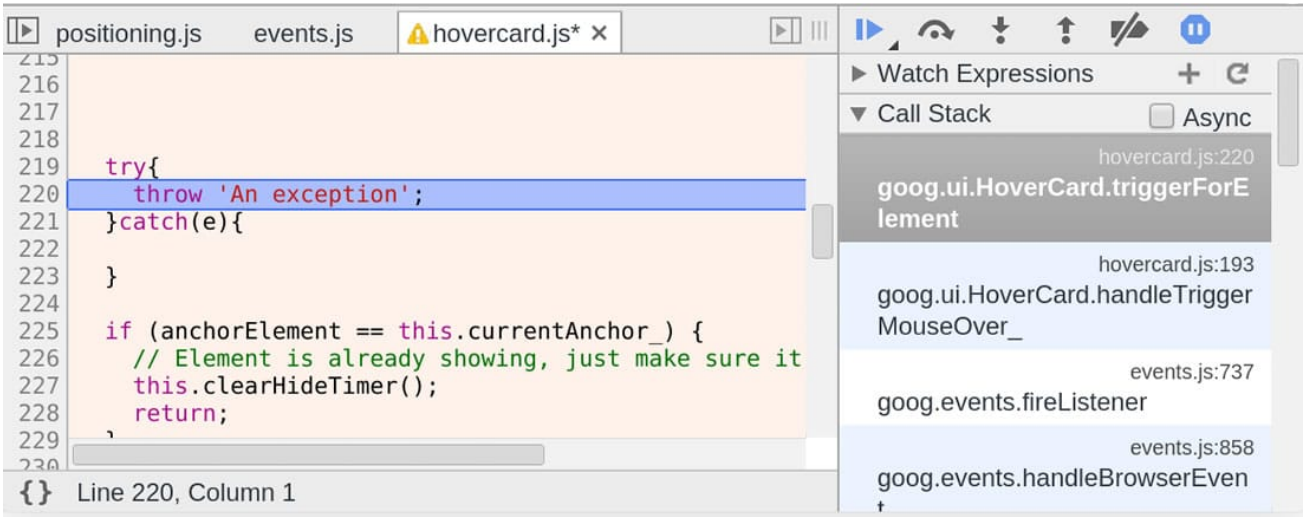
It's not always obvious which execution path lead to an error. Complete JavaScript call stacks accompany exceptions in the console. Expand these console messages to see the stack frames and navigate to the corresponding locations in the code:



Pause on JavaScript exceptions

The next time an exception is thrown, pause JavaScript execution and inspect its call stack, scope variables, and state of your app. A tri-state stop button at the bottom of the Scripts panel enables you to switch among different exception handling modes:

Choose to either pause on all exceptions or only on the uncaught ones or you can ignore exceptions altogether.



Print stack traces

Better understand how your web page behaves by printing log messages to the console. Make the log entries more informative by including associated stack traces. There are several ways of doing that.

Error.stack

Each Error object has a string property named stack that contains the stack trace:

```
events.js
983   // Caught a non-IE DOM event. 1 adc
984   var be = new goog.events.BrowserEve
985   try {
986     retval = goog.events.fireListener
987   } catch (e) {
988     console.log(e.stack);
989   } finally {
990     be.dispose();
991   }
992   return retval;
993 };
994
```

Watch Expressions
Call Stack
Not Paused
Scope Variables
Not Paused
Breakpoints
No Breakpoints
DOM Breakpoints
XHR Breakpoints

```
Error: An exception                                                    events.js:988
    at [object Object].triggerForElement (http://www.corp.google.com/~yurys/closure-
library/closure/goog/ui/hovercard.js:209:9)
    at [object Object].handleTriggerMouseOver_
(http://www.corp.google.com/~yurys/closure-library/closure/goog/ui/hovercard.js:186:10)
    at [object Object].handleEvent (http://www.corp.google.com/~yurys/closure-
library/closure/goog/events/listener.js:157:26)
    at Object.fireListener (http://www.corp.google.com/~yurys/closure-
library/closure/goog/events/events.js:753:21)
    at HTMLDocument.<anonymous> (http://www.corp.google.com/~yurys/closure-
library/closure/goog/events/events.js:986:26)
    at HTMLDocument.<anonymous> (http://www.corp.google.com/~yurys/closure-
library/closure/goog/events/pools.js:157:36)
>
```

console.trace()

Instrument your code with `console.trace()` calls that print current JavaScript call stacks:

positioning.js events.js hovercard.js* ×

```
214 goog.ui.HoverCard.prototype.triggerForElement = func
215
216   console.trace();
217   if (anchorElement == this.currentAnchor_) {
218     // Element is already showing, just make sure it
219     this.clearHideTimer();
220     return;
221   }
222
```

{} Line 214, Column 19

Watch Expressions
Call Stack Async
Not Paused
Scope Variables
Not Paused
Breakpoints
dynamicScript.js:2

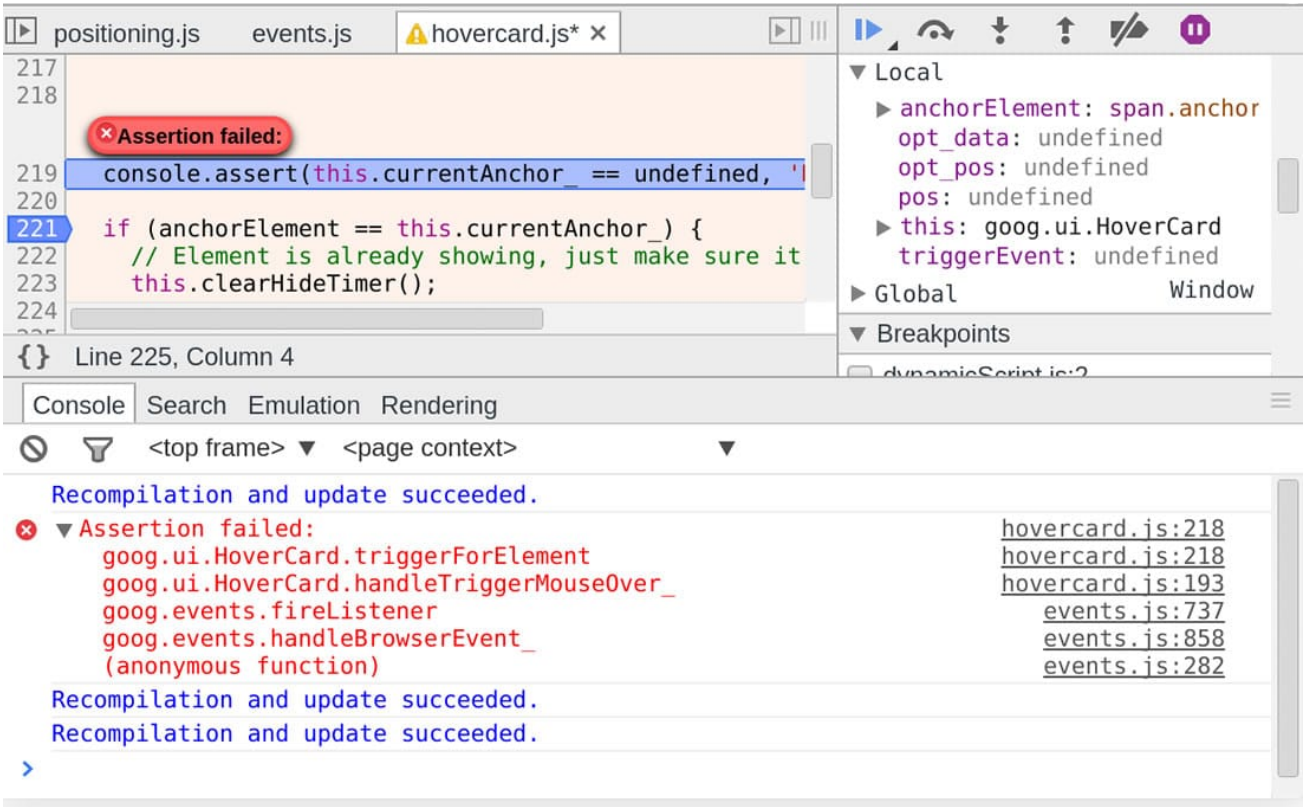
Console Search Emulation Rendering

<top frame> ▼ <page context> ▼

```
Recompilation and update succeeded.
Recompilation and update succeeded.
Recompilation and update succeeded.
4 ▼console.trace()                                          hovercard.js:216
    goog.ui.HoverCard.triggerForElement                     hovercard.js:216
    goog.ui.HoverCard.handleTriggerMouseOver_               hovercard.js:193
    goog.events.fireListener                                   events.js:737
    goog.events.handleBrowserEvent_                            events.js:858
    (anonymous function)                                       events.js:282
>
```

console.assert()

Place assertions in your JavaScript code by calling `console.assert()` with the error condition as the first parameter. When this expression evaluates to false, you will see a corresponding console record:



How to examine stack trace to find triggers

Let's see how to use the tools you've just learned about, and find the real cause of an error. Here's a simple HTML page that includes two scripts:

index.html

```html
<html>
<head>
  <script src="lib.js"></script>
  <script src="script.js"></script>

</head>
<body>

<p id="demo">Demo paragraph</p>
</body>
</html>
```

script.js

```js
document.addEventListener("click", function() {
  document.getElementById("demo").innerHTML = "Hello World";
  callLibMethod();
});
```

lib.js

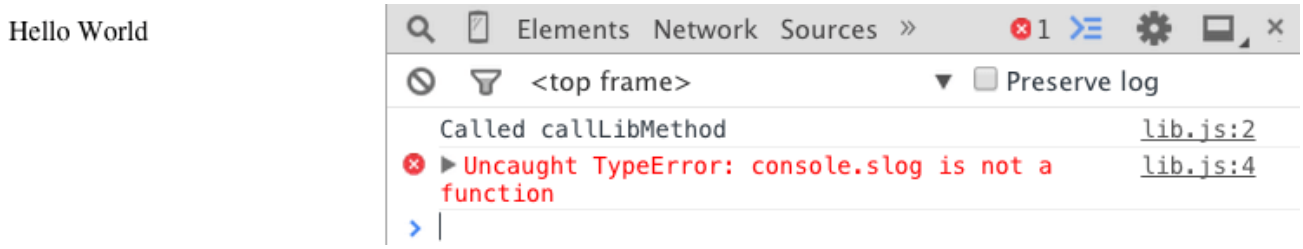
```js
var callLibMethod = function callLibMethod() {
  console.log('Called callLibMethod');

  console.slog('test');
};
```

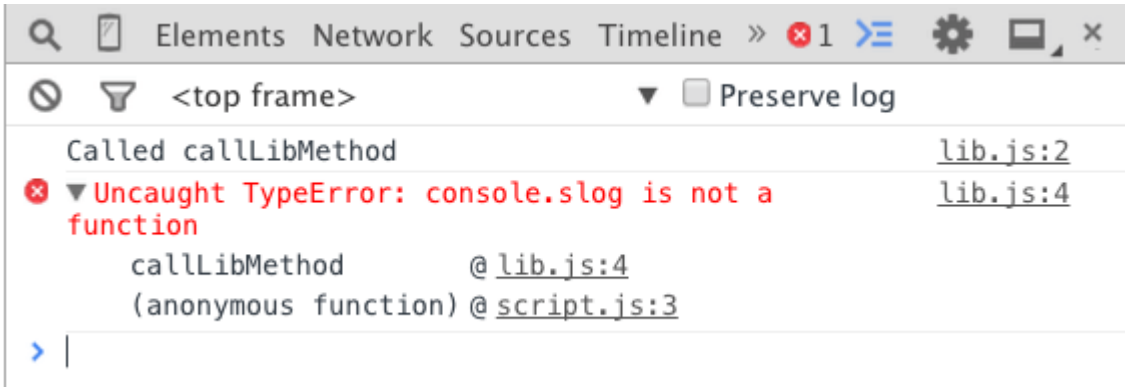
When the user clicks on the page, the paragraph changes its inner text, and the `callLibMethod()` function provided by `lib.js` is called.

This function prints a `console.log`, and then calls `console.slog`, a method not provided by the Console API. This should trigger an error.

When the page is run and you click on it, this error is triggered:



Click the arrow to can expand the error message:



The Console tells you the error was triggered in `lib.js`, line 4, which was called by `script.js` in the `addEventListener` callback, an anonymous function, in line 3.

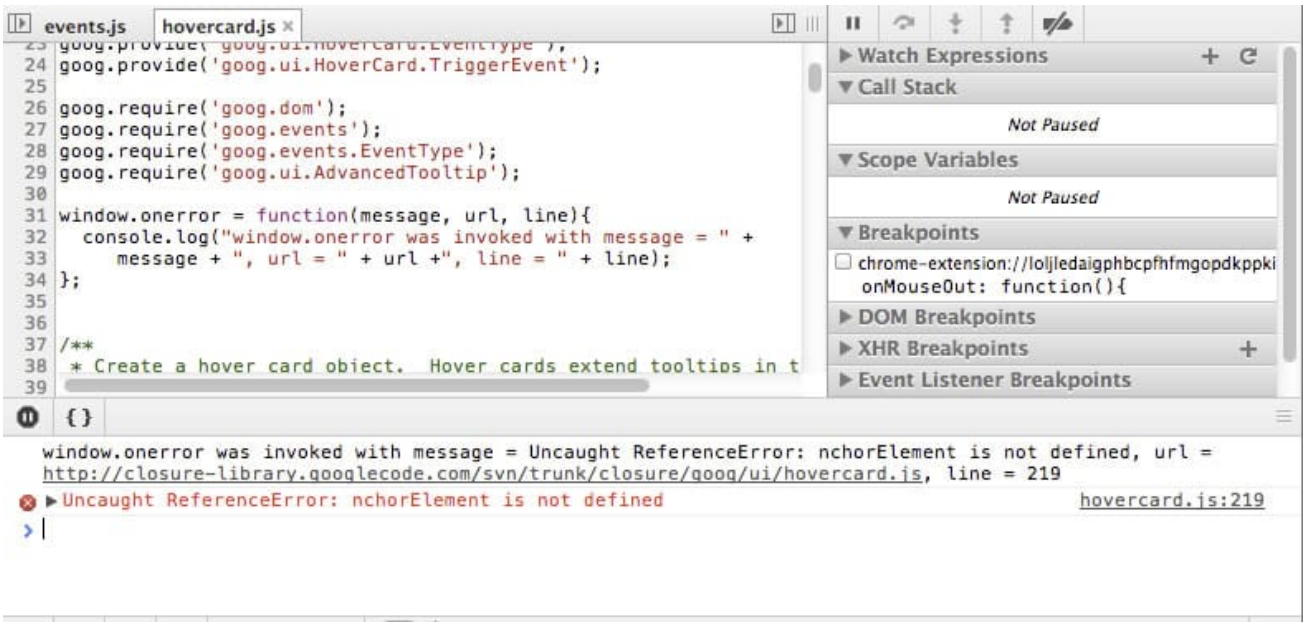
This is a very simple example, but even the most complicated log trace debugging follows the same process.

Handle runtime exceptions using window.onerror

Chrome exposes the `window.onerror` handler function, called whenever an error happens in the JavaScript code execution. Whenever a JavaScript exception is thrown in the window context and is not caught by a try/catch block, the function is invoked with the exception's message, the URL of the file where the exception was thrown, and the line number in that file, passed as three arguments in that order.

You may find it useful to set an error handler that would collect information about uncaught exceptions and report it back to your server using an AJAX POST call, for example. In this way, you can log all the errors happening in the user's browser, and be notified about them.

Example of using `window.onerror`:



Last updated July 2, 2018.