2010-04-13

# Debugging Closures and Modules(/Debugging-Closures-and-Modules.html)

The most common complaint with using closures to keep private variables in JavaScript is that it makes debugging harder. This complaint definitely holds water, and the loss of easy debugging and inspection using the Module Pattern(http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth) is a serious concern. This is where one of JavaScript's non-standard but well-supported features comes in: the **debugger statement**. In this article, I'll introduce this statement, and show how I use it to solve this deficiency in closure-based code.

## Intro to `debugger`

The `debugger` statement is not standardized, but **is provided by all implementations** of JavaScript. It essentially acts like a programmatic breakpoint. When popular JavaScript debuggers like Firebug or the IE Developer Tools execute a `debugger;` statement, execution stops and the script debugger opens up, just like any breakpoint. Here's a simple example:

```
function foo() {
        var a = 1;
        debugger; // execution stops until manually resumed
        a = 2;
}
foo();
```

In the above, we'll be able to inspect the local variables when our debugger pauses execution, and we'll see that `a` is still `1`. After resuming execution, the next line will execute and `a` will be assigned the value `2`.

## Debugging Modules

The great thing is that execution stops at the `debugger` statement, complete with the local call stack from the point of execution. This provides a really nice ability to pair with the module pattern, or any other closure-based privacy pattern. Let's take a simple module, but add extra capabilities to it by exposing a public `.debug()` method.

```
var foo = (function () {
        var privateVar = 10;
        return {
                publicFunc: function () {/*...*/},
                debug: function () {
                        debugger;
                }
        };
}());
```

Now we can call `foo.debug()` from the Firebug console, and the closure of the anonymous constructor is opened up for inspection. By looking at the **call stack** in our debugger, we can inspect the **scope chain** for the private state, which will be in local variables. Checking the value of `privateVar` will be quite easy.

Notice that it's very important for the `.debug()` property to be created and assigned during the normal construction. Assigning it after-the-fact will not provide the same functionality, because the local call stack will not contain the anonymous constructor. This is an unfortunate limitation, but there isn't a way around it.

## Safety and Configuration

You might not want to make it so easy to stop program execution by triggering the `.debug()` function on one of your modules, or you might want to disable this functionality. One approach is to ship a version of your code without this property at all, but that's difficult to accomplish in some applications. An easier method would be to do some extra checks in the `.debug()` method, before firing the `debugger` statement.

```
return {
        debug: function () {
                if (DEBUG_MODE) {
                        debugger;
                }
        }
};
```

Or, perhaps check for a specific debugger first:

```
return {
        debug: function () {
                if (DEBUG_MODE && "firebug" in console) {
                        debugger;
                }
        }
};
```

___Note___: I'm assuming you've already configured a global flag named `DEBUG_MODE` in your application.

Methods like this will allow you to provide finer-grained control over when and where your `debugger` statement runs. If you're building in a lot of logic, it makes sense to write a global helper function for this, but you'll have to be careful to preserve the right call stack:

```
function DEBUG() {
        if (DEBUG_MODE && "firebug" in console) {
                debugger;
        }
}

var foo = (function () {
        return {
                debug: function () {
                        DEBUG();
                }
        };
}());
```

Here we'll keep our callstack in tact, except the one we care about will be two deep instead of one deep.

### Problems With Minifiers

I tried running code with such a `.debug()` method through the YUI Compressor, and got a disappointing result:

```
[ERROR] 4:11:identifier is a reserved word
```

It seems that the YUI Compressor doesn't know about the `debugger` statement, which is understandable since it's non-standard. However, I didn't want to let this defeat my attempts, so a co-worker and I came up with a workaround. Instead of using a raw `debugger` statement, we put it behind `eval`, like this:

```
function debug() {
        eval('debugger;');
}
```

Now YUI no longer complains, and the code works just great. However, there are a few caveats here:

1. Using `eval` is generally considered **dangerous** and **bad practice**. Be sure you understand the implications, and are willing to use this workaround anyways. I think this is a reasonable decision to make, for the intended purpose, but ensure you've at least put some thought into it.
2. JSLint will complain at you. This isn't such a big deal since it would be complaining about the raw `debugger` statement anyways.

## Conclusions

So that's my technique for effectively debugging inside closures, especially when using the Module Pattern. I'm not saying this is a great technique for all uses or users, but it works well for me, and nicely solves the most common complaint developers have about the Module Pattern. I'd love to hear alternatives or reasons why my technique is dangerous and should be avoided, so leave a comment!

filed under javascript(/tag/javascript), module pattern(/tag/module pattern), and debugging(/tag/debugging)

**26 Comments**   **Adequately Good**   1 **Login**

**Recommend** 5   **Share**   **Sort by Oldest**

Join the discussion…

**Herhor** • 7 years ago

I use plain "console.log()". It's supported by firebug and chrome console and it's completely enough for my needs :-)

1 ^ | ⌄ • Reply • Share ›

**Ben Cherry** Mod ➦ Herhor • 7 years ago

Yeah, console.log() is a great tool, but `debugger` serves a different purpose. This lets you get into your anonymous "private" closures and freely inspect internal state, without having to set up complex logging statements for the same.

3 ^ | ⌄ • Reply • Share ›

**olivvv** • 7 years ago

To catch the beginning of an action, without getting into a setinterval() / mousemove triggered function, I use:

onclick="debugger;"

1 ^ | ⌄ • Reply • Share ›

**zvona** • 7 years ago

Excellent post, Ben! And I have to admit I've never before heard about "debugger" keyword :blush:

What comes on checking whether debugging is enabled, I've been using following script:

```
var dbg = {
enable : function() {
if (!dbg.isEnabled()) {
window.name += "dbgEnabled";
}
},

disable : function() {
window.name = window.name.replace(/dbgEnabled/g, "");
},

isEnabled: function() {
return (!!~window.name.indexOf("dbgEnabled"));
}
}
```

This allows switching between debugging modes through console during session. And of course it can be extended to show only certain states in debugging, like "all" or "warnings" etc.

^ | ⌄ • Reply • Share ›

**Ben Cherry** Mod ➦ zvona • 7 years ago

Awesome, glad I covered something new :)

Your `dbg` configuration module looks good. I've been working on standardizing something for myself. I might take some ideas :)

Reply • Share ›

**johnjbarton** • 7 years ago

Why don't you just use breakpoints?

Reply • Share ›

**Ben Cherry** Mod johnjbarton • 7 years ago

You could use breakpoints, but it can be really bothersome to set breakpoints in large files, or in files that have been minified or obfuscated by tools like the YUI Compressor. Using `debugger` lets you set them programmatically in the source. I think the Chrome debugger has better support for setting breakpoints with code, since it's driven by the command line, but other tools like IE or Firebug don't have that. This helps correct that deficiency.

Reply • Share ›

**johnjbarton** Ben Cherry • 7 years ago

The 'debugger;' statement has great uses I agree. If you are in the unfortunate position of debugging compressed code without a decompressor, well good luck to you, use what ever you can.

Setting breakpoints in large files: what possible advantage can it be to edit the source, type in the statement then reload the browser just to halt the debugger vs clicking on the line in the Script panel?

What deficiency does a command line fix? A few people like them, but I don't know why.

Reply • Share ›

**Ben Cherry** Mod johnjbarton • 7 years ago

I think you're misunderstanding the intent of my technique. I agree that when you're breakpoint-debugging, you should be breakpoint-debugging. For some, the debugger statement might make that more convenient, but that's not the use I'm espousing here. I'm talking about attaching a permanent .debug() method to objects with private state, that allow you to type myObject.debug() in the console and crack that open, and inspect the private state when you need to. Setting a breakpoint inside the object for this purpose seems silly, when you can set up an on-demand method like this.

Reply • Share ›

**johnjbarton** Ben Cherry • 7 years ago

Ok yes that makes a lot more sense now thanks!

Reply • Share ›

**Christian C. Salvadó** • 7 years ago

Nice article Ben, just note that now the debugger statement is not anymore non-standard, it is now part of the ECMAScript 5th Edition Specification (Section 12.15).

of the ECMAScript 5th Edition Specification (Section 12.15).

Reply • Share ›

**Ben Cherry** Mod ➦ Christian C. Salvadó • 7 years ago

True, but I don't usually consider features from ES5 "standard" (even though they technically are!). As long as YUI Compressor chokes on it, we've still got time before we can start using ES5 entirely.

1 • Reply • Share ›

**AngusC** • 7 years ago

Its also useful for cross-browser and cross-session debugging (I often find myself having to reset firebug breakpoints for one reason or another - it gets annoying)

Never thought of wrapping it in a debug function though - nice!

1 • Reply • Share ›

**Ben Cherry** Mod ➦ AngusC • 7 years ago

Yeah, it's nice to get easy breakpoints in a lot of browsers, but annoying because you have to be sure to remove it. You could use a global function for this, and do code like this:

```
var DEBUG_LEVEL = 0; // set to control which breakpoints fire

function DEBUG(level) {
    if (DEBUG_LEVEL >= level) {
       debugger;
    }
}

function foo() {
  DEBUG(1);  // important breakpoint
  for (/*....*/) {
     DEBUG(2); // less important breakpoint
  }
}
```

Then you can set your DEBUG_LEVEL, and various breakpoints will fire (similar to using a debug level for logging statement detail). If DEBUG_LEVEL is 0, then nothing will fire (production code).

1 • Reply • Share ›

**wavded** • 7 years ago

why have I never heard of this? (rhetorical question)
anyway, where did you learn about it?

Reply • Share ›

**Ben Cherry** Mod ➦ wavded • 7 years ago

No idea where I heard about `debugger;`, it just showed up in my toolkit one day :)

This technique for putting the statement in a method with privileged access to a private closure was my own discovery, though.

1 ⌃ | ⌄ • Reply • Share ›

**Andrew Hedges** • 7 years ago

I'm surprised eval('debugger;'); gives you the proper scope. Doesn't eval create a new evaluation context?

⌃ | ⌄ • Reply • Share ›

**Ben Cherry** Mod ➦ Andrew Hedges • 7 years ago

No, `(new Function("....."))()` does. `eval` always executes in the current scope like a normal line of code.

⌃ | ⌄ • Reply • Share ›

**Jonas** • 6 years ago

The eval("debugger"); hack to get it to jslint isnt very nice. Especially since eval is forbidden when enabling "The good parts".

Add this to the top of your .js file instead:
/*global debugger:true */

and the yui compressor will be happy

2 ⌃ | ⌄ • Reply • Share ›

**Jonas** ➦ Jonas • 6 years ago

Sorry, should be:
/*jslint debug: true */

Ps. openid login does not work, using google openid url: https://www.google.com/account...

2 ⌃ | ⌄ • Reply • Share ›

**Markbleichert** • 6 years ago

I like your post but what is clear to me yet, is 'how' the module pattern makes debugging harder. Do you have a concrete example ?

⌃ | ⌄ • Reply • Share ›

**Ben Cherry** Mod ➦ Markbleichert • 6 years ago

If you're using closures to keep private state in JavaScript (the Module Pattern), JavaScript debuggers don't not provide an adequate way to inspect private values. Other languages have powerful debuggers for private state, but not JS. For example:

```
var module = (function() {
var a = 1;
return {
```

```
setA: function(v) {
a = v;
}
};
}());
```

In this example, we can't inspect the private value `a`, even with a debugger.

^ | v • Reply • Share ›

**Duarte Cunha Leão** • 6 years ago

Very good technique, thank you! I've been trying to figure out how to inspect closure's internal state and this is great. The great thing about this technique is that you don't have to "wait" for the closure to be really called, as you would have to if a breakpoint were used. You can put a breakpoint anywhere, and as along as you have a reference to the Module you can inspect what's inside.
A related subject, that I don't know a good solution for, is the testability of private Module functions...

^ | v • Reply • Share ›

**Ben Cherry** Mod ➦ Duarte Cunha Leão • 6 years ago

Thanks, I'm glad it was useful!

I think you could also add a testing method to your module that returns all of the private functions on an object. You could also have a global flag in your app for whether you are running in production, development, or test, and have this method (and the debug method) throw an exception when called in production.

^ | v • Reply • Share ›

**Henry Q. Dineen** • 6 years ago

Great idea!

^ | v • Reply • Share ›

**Marghoob Suleman** • 6 years ago

Great! I never heard. :) but is there there any command to move in next line without using firebug. We have continue button in firebug.

^ | v • Reply • Share ›

## the author

Ben is a 25 year-old software engineer. He lives and works in San Francisco. Many people think he invented the term "hoisting" in JavaScript, but this is untrue.

- **Twitter(http://twitter.com/bcherry)** - @**bcherry(http://twitter.com/bcherry)**
- **GitHub(http://github.com/bcherry)** - My Code
- **LinkedIn(http://www.linkedin.com/in/bcherryprogrammer)** - Professional Profile
- **Facebook(http://www.facebook.com/bcherry)** - That Other Social Network
- **Presentations(http://www.bcherry.net/talks/)** - Slides From My Talks

Follow @bcherry 8,084 followers

## categories

- **javascript(/tag/javascript)** (21)
- **social gaming(/tag/social%20gaming)** (1)
- **css(/tag/css)** (1)
- **jquery(/tag/jquery)** (2)
- **performance(/tag/performance)** (5)
- **tools(/tag/tools)** (2)
- **html5(/tag/html5)** (3)
- **adequatelygood(/tag/adequatelygood)** (1)
- **timers(/tag/timers)** (2)
- **module pattern(/tag/module%20pattern)** (3)
- **talks(/tag/talks)** (1)
- **slide(/tag/slide)** (1)
- **python(/tag/python)** (1)
- **debugging(/tag/debugging)** (1)
- **testing(/tag/testing)** (2)
- **hashbang(/tag/hashbang)** (1)