# Modern Debugging Tips and Tricks

*by* TIFFANY B. BROWN

With the rise of mobile devices, web development and debugging is more complex than ever. We have more browsers and platforms to support. We have more screen sizes and resolutions. And we're building in-browser applications instead of the flat, brochure-ware sites of yore.

Luckily, we also have better tools. The JavaScript console is a standard feature of most major browsers. Both JavaScript and the HTML DOM offer native error handling. We also have services and applications that help us remotely debug our sites.

In this article I'll cover error thowing and handling, code injection, and mobile debugging. For more on debugging, see Hallvord R.M. Steen and Chris Mills' 2009 article, Advanced Debugging With JavaScript.

## Throwing and catching errors

JavaScript lets you report and handle errors through a combination of the `throw` and `try…catch` statements, and the `error` object.

Error throwing is useful for catching runtime errors—say, a function that has incorrect arguments. In the example below, `add()` accepts two parameters. It will throw an error if the supplied arguments are null, or are neither a number nor a numeric string. (Line wraps marked » —*Ed.*)

```
function add(x,y){
if( isNaN(x) || isNaN(y) ){
throw new Error("Hey, I need two numbers to add!");
} else {
// ensure we're adding numbers not concatenating »
numeric strings.
return (x * 1) + (y * 1);
}
}
```

Let's try invoking `add()` using invalid arguments. We'll then catch the error thrown by using a `try...catch` block and output it to the console:

```
var a;

try{
a = add(9);
} catch(e) {
console.error( e.message );
}
```

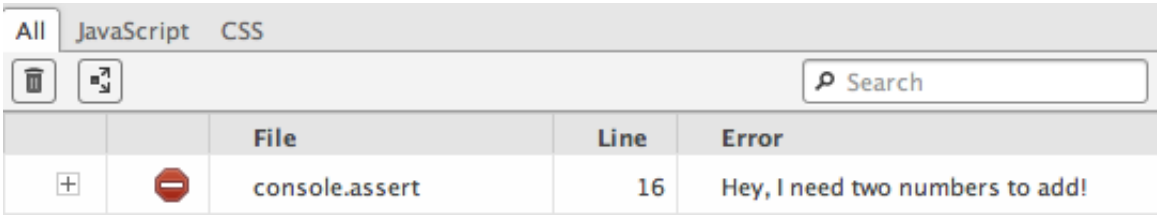


Fig 1: The Dragonfly console error

In Opera Dragonfly (above), we see the error message and its corresponding line number, relative to the script. Keep in mind in these examples we're embedding JavaScript within our HTML page.

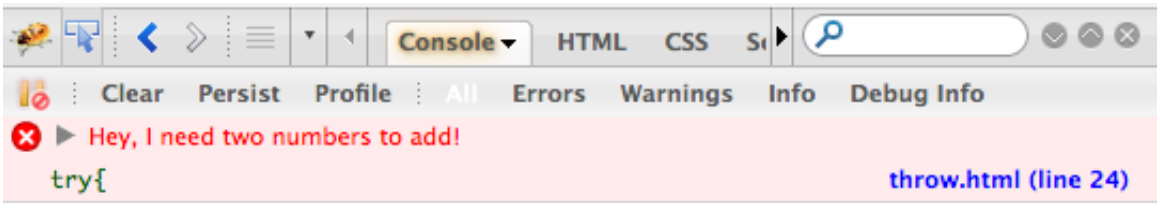


Fig. 2: The Firebug error console

Firebug also includes the thrown error message and line number, but relative to the document.

All error objects have three standard properties:

- `constructor`: returns a reference to the Error function that created an instance's prototype,

- `message`: the message thrown—the message you passed as an argument, and
- `name`: the type of error—usually `error`, unless you use a more specific type.

As of this writing, error objects in Firefox also include two non-standard properties: `fileName` and `lineNumber`. Internet Explorer includes two non-standard properties of its own: `description` (which works similarly to `message`) and `number` (which outputs the line number).

The `stack` property also isn't standard, but it is more or less supported by the latest versions of Chrome, Firefox, and Opera. It traces the order of function calls, with corresponding line numbers and arguments. Let's modify our example to alert the `stack` instead:

```
var a;

try{
a = add(9);
} catch(e) {
alert( e.stack );
}
```

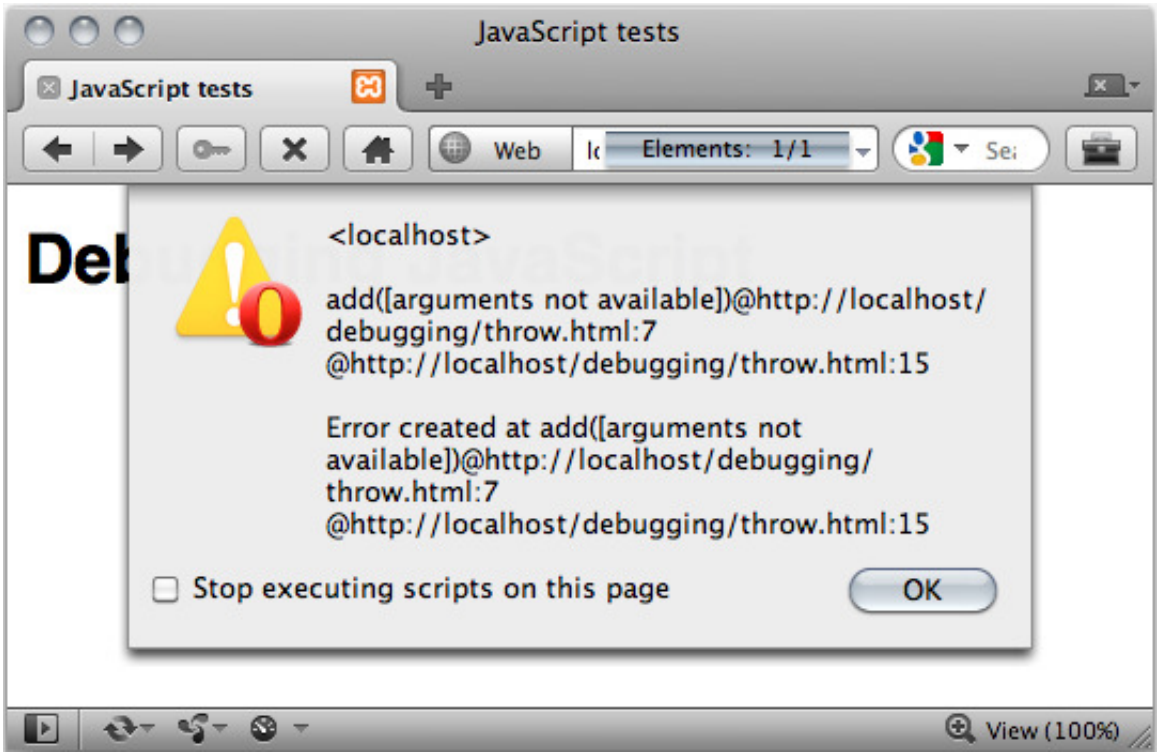


Fig 3: The stack property revealing the throw error in the code

The `stack` property reveals where in the code `throw Error` exists (in this case, line seven) and on which line the error was triggered (in this case, line 15).

You don't have to throw an error object. You could, for example, throw a message: `throw "The value of x or y is NaN."` Throwing an error, however, offers richer information in most

browsers.

Using `try...catch` can, however, have a negative effect on script minification and performance. While handy for debugging, your production-ready code should use `try...catch` sparingly, if at all.

## Handling errors using the `window.onerror` event

The Document Object Model also offers a mechanism for capturing errors: the `window.onerror` event. Unlike `try...catch`, you can set an event handler for `window.onerror` that captures errors you *don't* throw. This can happen if you try to invoke an undefined function or access an undefined variable.

When the `window.onerror` event is fired, the browser will check to see whether a handler function is available. If one isn't available, the browser will reveal the error to the user. If one is available, the handler function receives three arguments:

- the error message,
- the URL in which the error was raised, and
- the line number where the error occured.

You can access those arguments in one of two ways:

1. by using the `arguments` object that is native to and locally available to all JavaScript functions; or
2. by using named parameters.

In the example below, we will use `arguments`. For readability, though, you should use named parameters:

```
window.onerror = function(){
alert(arguments[0] +'\n'+arguments[1]+'\n'+arguments[2]);
}

init(); // undefined and triggers error event.
```

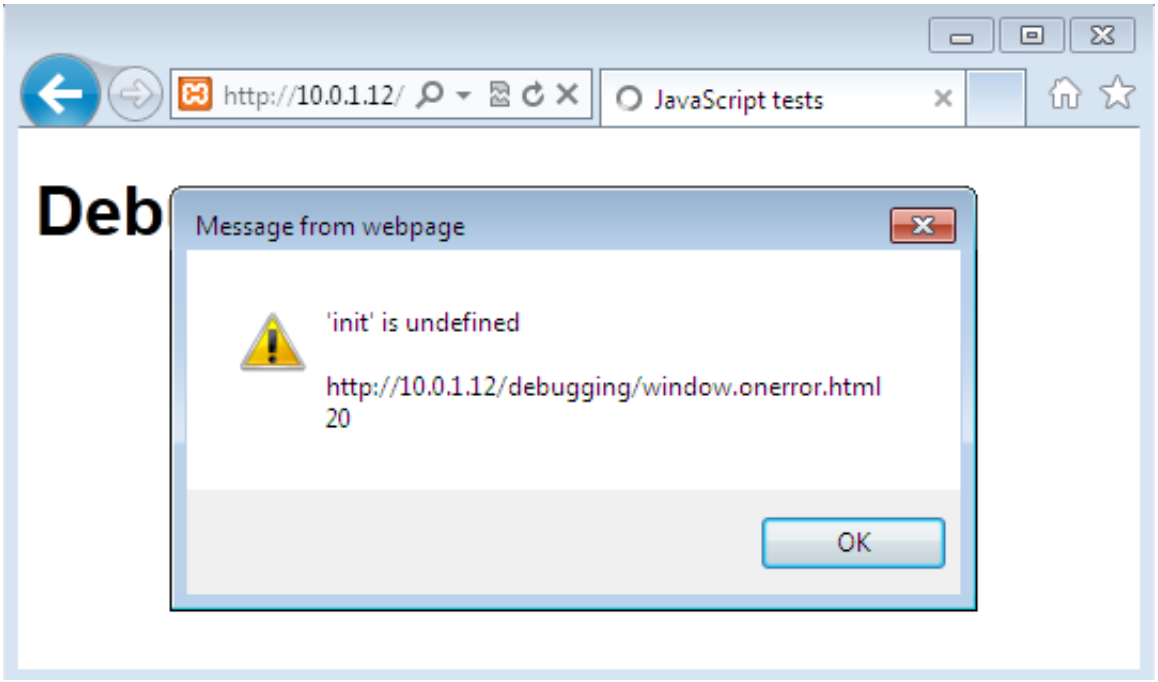


Fig 4: What our error looks like as an alert in Internet Explorer 9

Here `init()` has not yet been defined. As a result, the `onerror` event will be fired in supporting browsers.

Now the caveat: support for `window.onerror` is limited. Chrome 10+ and Firefox (including mobile) support it. Internet Explorer supports it, but truly helpful error messages are only available in version 9+. While the latest builds of WebKit support `window.onerror`, recent versions of Safari and slightly older versions of Android WebKit don't. Opera also lacks support. Expect that to change as the HTML5 specification evolves and browser vendors standardize their implementations.

## Modify JavaScript on the fly using the command line interface

One of the more powerful features available in today's debugging tools is the JavaScript console. It's almost a command line for JavaScript. With it, you can dump data or inject JavaScript to examine why your code has gone rogue.

## Launching the JavaScript Console

- In **Chrome**: View > Developer > JavaScript console
- In **Safari**: Develop > Show Web Inspector
- In **Internet Explorer 8 & 9**: Tools > Developer Tools (or use the F12 key)
- In **Opera**: Find Dragonfly under Tools > Advanced (Mac OS X) or Menu > Page > Developer Tools (Windows, Linux)

**Firefox** is a special case. For years, developers have used the Firebug extension. Firefox 4, however, added a native console (Tools > Web Console or Menu > Web Developer > Web Console).

Firebug fully supports the Console API, and has more robust CSS debugging features. I recommend installing it, though the Web Console is a capable tool for basic needs.

I'm using Opera's debugging tool Dragonfly in the examples below (yes, I work for Opera). These examples, however, work similarly in Chrome, Safari, Firefox, Firebug, and Internet Explorer.

Let's take another look at the code from our previous examples. We're going to add a new line—`var a = document.querySelector('#result');`—one that assumes an element with an id value of "result."

A quick note about the `querySelector()` method: it and `querySelectorAll()` are part of the DOM selectors API. `querySelector()` returns the first element matching the specified CSS selector. Both methods are supported by the latest versions of most browsers. You could also use `document.getElementById('result')`, but `document.querySelector()` is more efficient:

```
function add(x,y){
if( isNaN(x) || isNaN(y) ){
throw new Error("Hey, I need two numbers to add!");
} else {
// ensure we're adding numbers not concatenating numeric strings.
return (x * 1) + (y * 1);
}
}

var a = document.getElementById('result');

try{
a.innerHTML = add(9);
} catch(e) {
console.error(e.message);
}
```

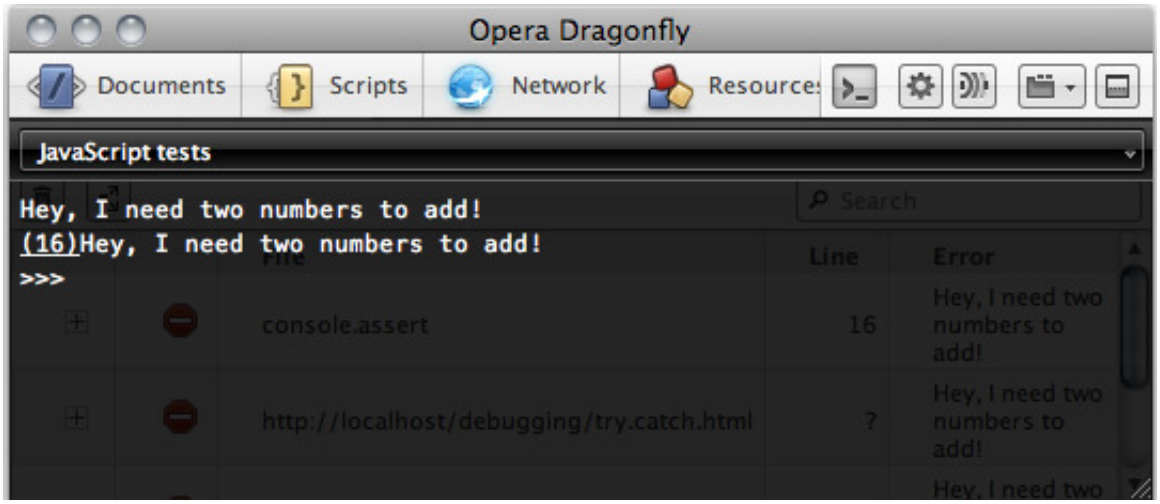


Fig 5: The Dragonfly console

Our thrown error is still written to the console. But let's inject some JavaScript that runs correctly. We'll enter `a.innerHTML = add(21.2, 40);` in our console:

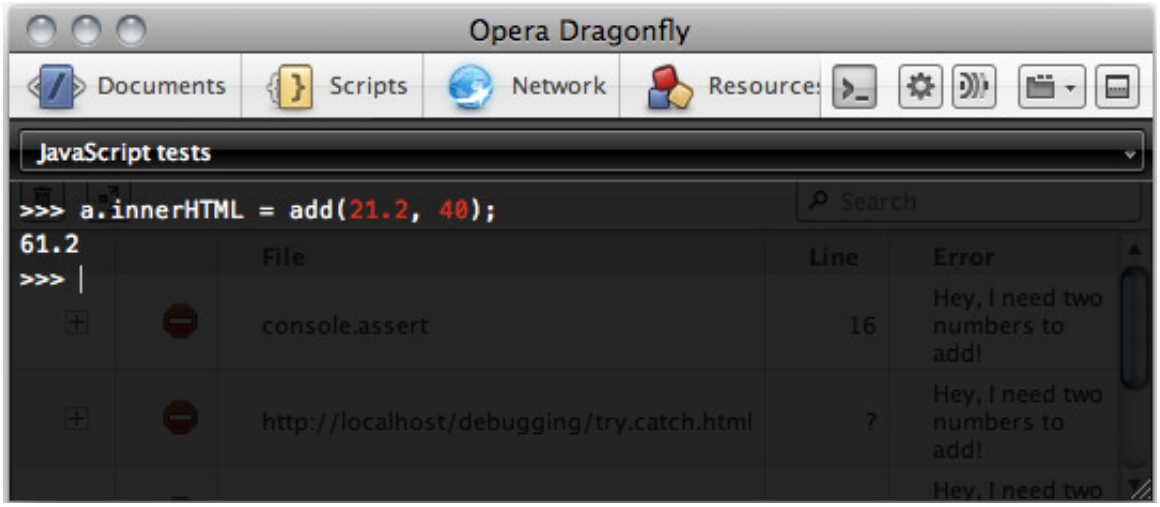


Fig 6: The Dragonfly console with injected code

As you can see, we have overwritten the `innerHTML` value of `a`:

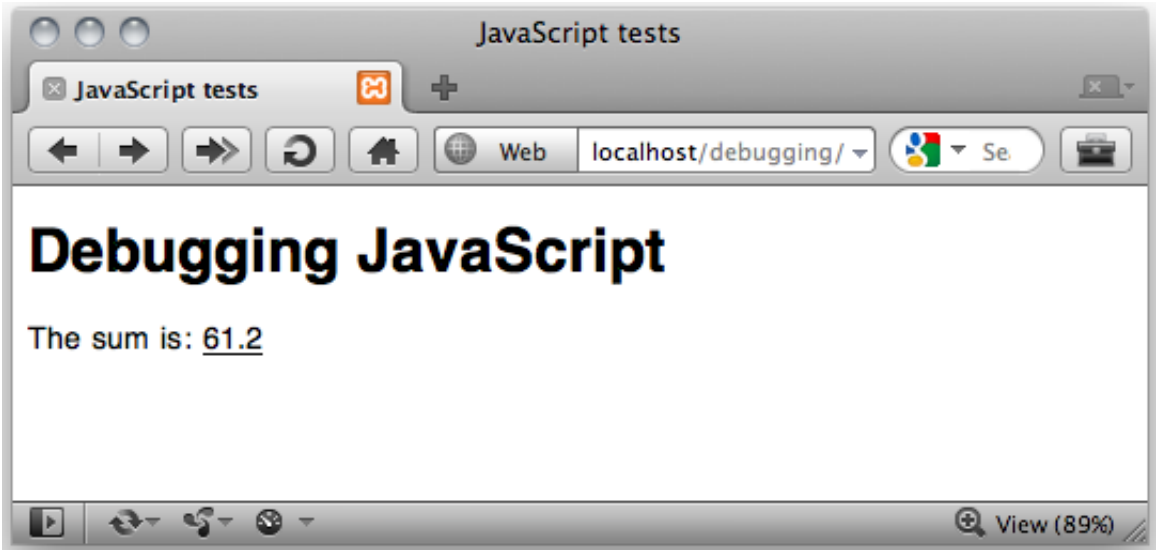


Fig 7: A page with injected code

Now let's change the value of `a` entirely. Enter `a = document.querySelector('h1'); a.innerHTML = add(45,2);` in the console:

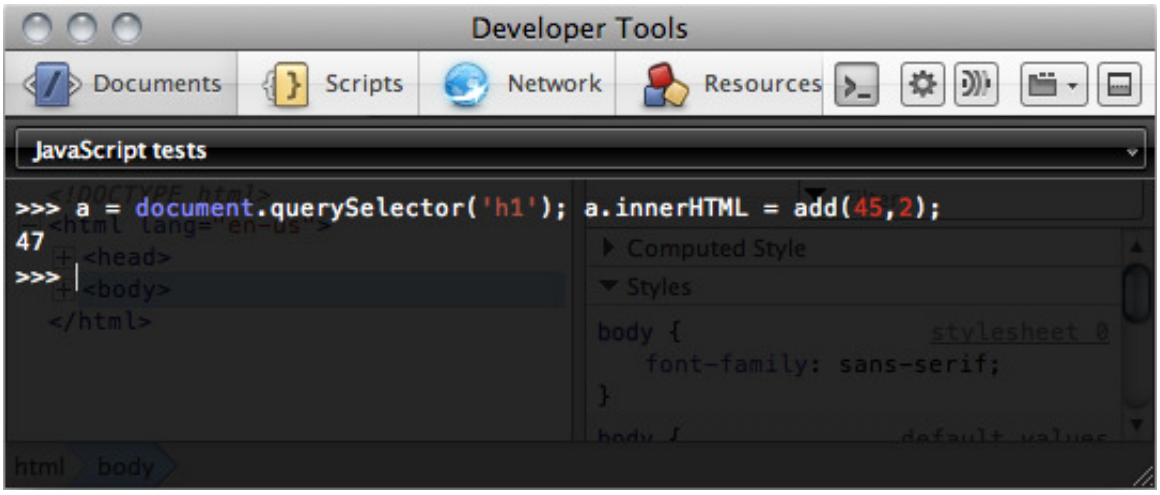


Fig 8: Changing code in the console

You'll see that 47 is written to the console, and it is also the new `innerHTML` of our `h1` element:

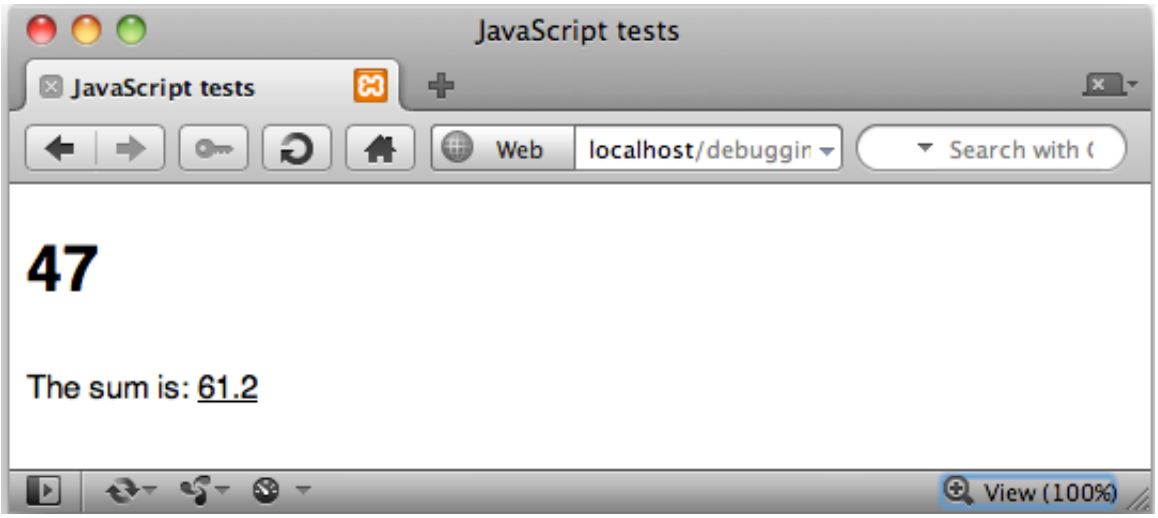


Fig 9: Modifying the DOM

Now, we can even redefine our `add()` function. Let's make `add()` return the product of two arguments and then update the `h1`. Enter `function add(){ return arguments[0] * arguments[1]; }` in the console, followed by `a.innerHTML = add(9,9);`:

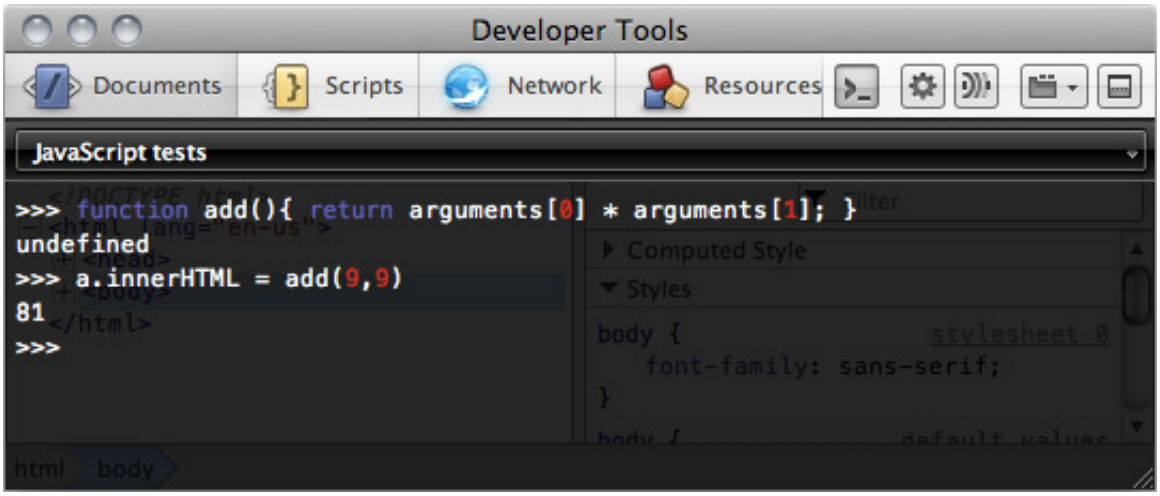


Fig 10: Overwriting a function using the JavaScript console

The new `innerHTML` for our `h1` element is now 81, the result of our redefined `add` function:

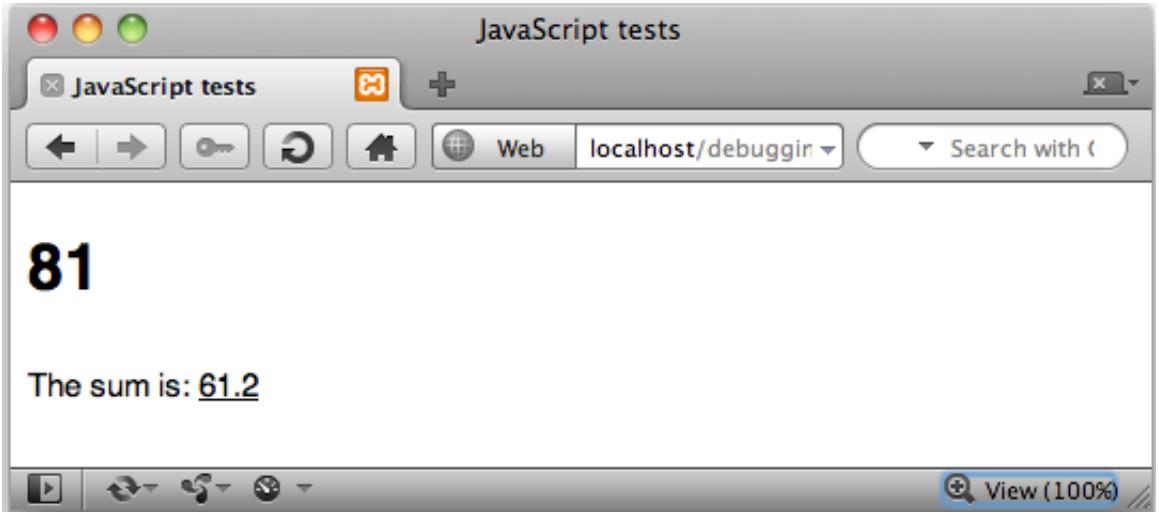


Fig 11: The results of overwriting a function

The JavaScript console offers a powerful tool for understanding how your code works. It's even more powerful when used with a mobile device.

## Remote debugging for mobile

Debugging code on a mobile device is still one of our biggest pain points. But, again: now we have tools. Opera Dragonfly and its remote debug feature provides developers a way to debug mobile sites from their desktop. WebKit recently added remote debugging to its core and Google Chrome has already folded it into its developer tools.

Independent developers offer similar products for other browsers. These include Bugaboo, an iOS app for Safari-based debugging; JS Console which is available on the web or as an iOS app; and Weinre for WebKit-based browsers.

Let's look at two: **Dragonfly remote debug** and **JSConsole**.

## Remote debugging with Opera Dragonfly

Dragonfly's strong suit is that you can debug CSS or headers (see the Network tab) in addition to JavaScript. But it does require installing Opera on your desktop and Opera Mobile on your device.

Both devices should be connected to the same local network. You will also need the IP address of the machine running Dragonfly. Then complete the following steps:

1. open Dragonfly from the Tools > Advanced (Mac OS X) or Page > Developer Tools (Windows, Linux) menus,
2. click the Remote Debug button ,
3. adjust the port number if you'd like, or use the default and click "Apply, "

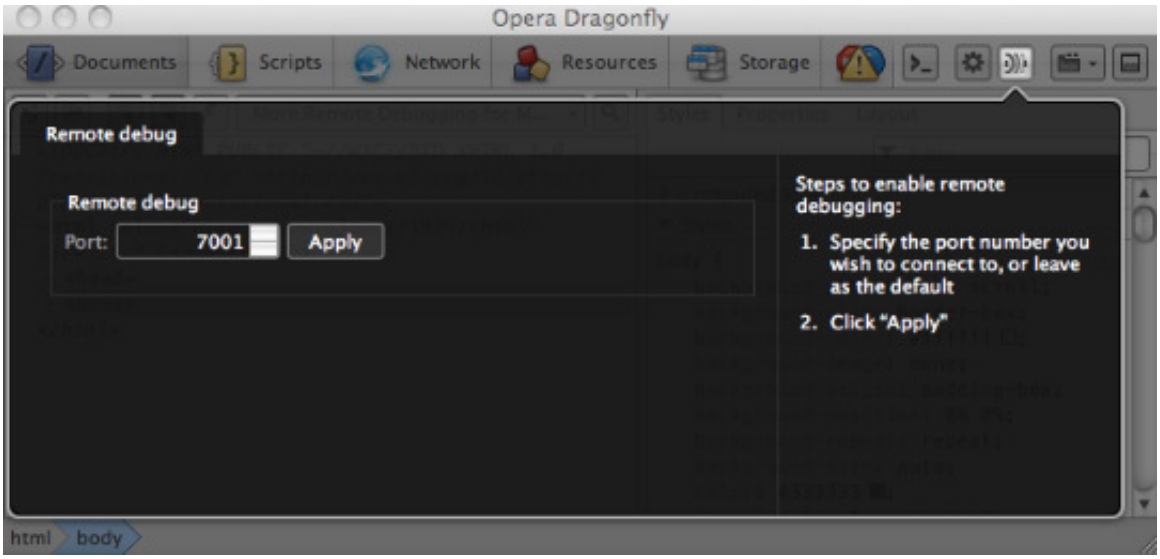


Fig 12: The remote debugging panel in Dragonfly

4. open Opera Mobile on your target device, and enter `opera:debug` in the address bar, and

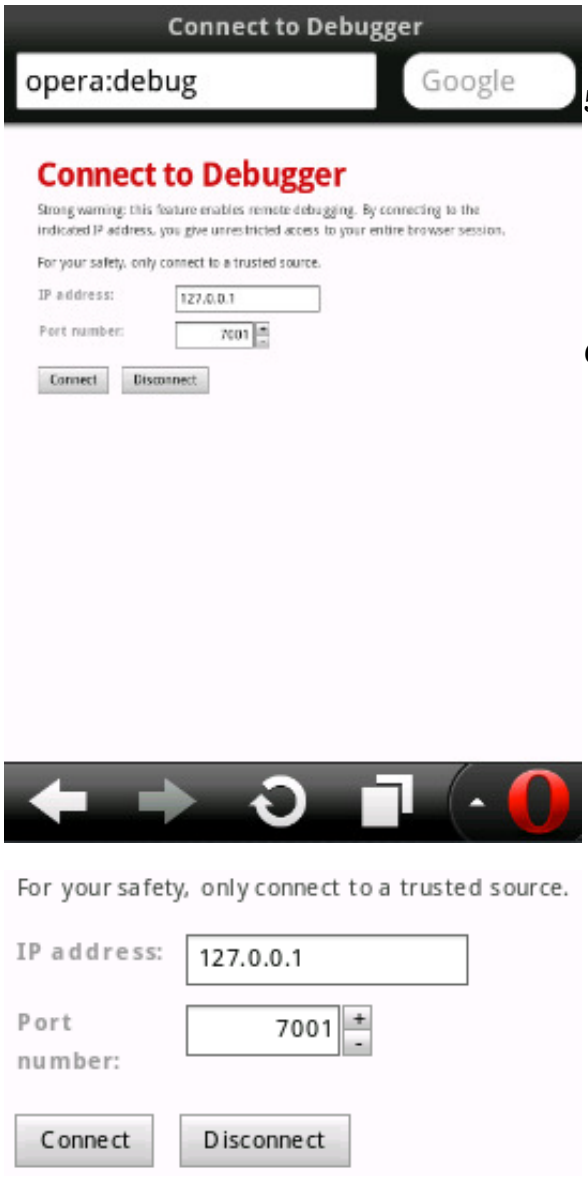


Fig 13: Opera Mobile debug console

5. enter the IP address and port number of the host machine and click "Connec,t" and

Fig 14: The IP and Port fields of the opera:debug console

6. navigate to the URL of the HTML page you wish to debug on your device.

Fig 15: An alert on Opera Mobile

Dragonfly on the host machine will load the remote page. You can then interact with the page as though it was on your desktop. You will see the results on the device. For example, if you enter `alert( add(8,923) )` in the host console, the alert appears on the mobile device screen.

## Remote debugging with JSConsole

JSConsole is a web-based, browser-independent service. Unlike Bugaboo, Weinre, and Dragonfly, your computer and device don't have to be connected to the same local network.

To use JSConsole:

To use JSConsole:


- visit the site and enter `:listen` at the prompt,
- add the returned script tag to the document you wish to debug, and
- open the document on your mobile device.

Remote console statements will appear in the JSConsole window (you do need to use `console.log()` rather than `console.error()` or `console.warn()`). You can also send code from the JSConsole window to your device. In this case, `alert( add(6,3) );`.


Fig 16: Sending a command using JSConsole.com

## Remote error logging

In the examples above, we're logging to the console, or launching an alert box. What if you logged your errors to a server-side script instead?

Consider the following code that uses `XMLHttpRequest()`. In it, we are :

```
function sendError(){
var o, xhr, data, msg = {}, argtype = typeof( arguments[0] );

// if it is an error object, just use it.
if( argtype === 'object' ){
msg = arguments[0];
}

// if it is a string, check whether we have 3 arguments...
else if( argtype === 'string') {
// if we have 3 arguments, assume this is an onerror event.
if( arguments.length == 3 ){
msg.message = arguments[0];
msg.fileName = arguments[1];
msg.lineNumber = arguments[2];
}
// otherwise, post the first argument
else {
msg.message = arguments[0];
}
}

// include the user agent
msg.userAgent = navigator.userAgent;

// convert to JSON string
data = 'error='+JSON.stringify(msg);

// build the XHR request
xhr = new XMLHttpRequest();
xhr.open("POST",'./logger/');
xhr.setRequestHeader("Content-type", "application/x-www- »
form-urlencoded");
xhr.send( data );


// hide error message from user in supporting browsers
return true;
}
```

Here we're posting our error messages to a script that logs them in a flat file using PHP:

```
<?php

// decode the JSON object.
$error = json_decode( $_POST['error'], true );
$file = fopen('log.txt','a');
fwrite($file, print_r( $error, true) );
fclose($file);

?>
```

Now the disclaimer: please for the love of tequila, *don't let this script write to a world-readable directory*. The potential for code injection due to spoofed headers or variables is not worth the risk. Logging scripts like this should only be used during development, and never on production servers.

### Conclusion

As the web has evolved, so have our tools. Code injection, error throwing and catching, and remote debugging services are all helping us ship better, less buggy apps.

---

## Original URL:

http://www.alistapart.com/articles/modern-debugging-tips-and-tricks/