DZone Refcardz

CONTENTS INCLUDE:

- Basics
- Common Problems
- Debugging JQuery
- Mobile Debugging
- Debugging Tools...and More!

Debugging JavaScript

By: Ashutosh Sharma

INTRODUCTION

JavaScript is the most popular programming language in the world and is used to specify logic on web pages as well as in HTML-based applications. It is also being used on the server side with Node.js [1].

With JavaScript supported by all popular browsers, all of them also ship with built-in developer tools. Firefox also has a popular web development add-on.

Browser	Developer Tool
Chrome	DevTools (built-in)
Firefox	Firebug (add-in)
Opera	Dragonfly (built-in)
Internet Explorer	IE Developer Tools (built-in)
Safari	Web Inspector

Each of these has its strengths and weaknesses, and it is useful to pick the right tool to debug the issue you are working on. For most of the discussion in this document, we will be using Chrome DevTools.

BASIC DEBUGGING

Errors can typically be classified into one of the following:

1. Syntax or interpretation errors
2. Runtime exceptions
3. Incorrect logic

Syntax Errors

These include mismatched or missing quotes, parentheses and braces, incorrect case, spelling mistakes, illegal characters, and more. These can usually be caught easily by *static code analyzers* such as JSHint [2] and JSLint [3]. These tools analyze your code before it is executed on a web page and point out syntax and other common errors before you make your code live on your web site or application.

Runtime Exceptions

These are the errors that occur when your JavaScript code executes. Such errors can be triggered by referring to an undefined variable, dividing by zero, by a failed "assert" statement, or by using a "throw" statement in your (or a JavaScript library's) code to manually specify that an exception has occurred.

When a runtime exception occurs, any JavaScript code after that line of code is not executed. Hence, if runtime exceptions are not caught and handled appropriately, they can leave your web page or application in an unexpected state.

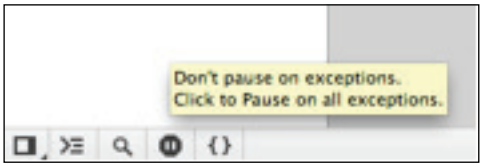
You can catch and handle runtime exceptions by wrapping your code (that may throw an exception) with "try {}" and specifying the handler with "catch(exception) {}":

```
try {
   var v = dummy; // causes a runtime exception
} catch(e) {
   console.log(e.message);  // "dummy is not defined"
}
```

In most browsers, you can also specify a global handler for all exceptions that have not been caught using a "try... catch" block, by defining *window.onerror*:

```
window.onerror = function(errorMessage, url, lineNumber) {
  // ...
```

In Google Chrome, you can have DevTools pause JavaScript execution on all runtime exceptions by clicking the "pause" button at the bottom of the Sources panel. (You can launch DevTools by pressing Cmd-Opt-I (Mac) or F12 (Windows), or via Menu > Tools > Developer Tools.) To pause JavaScript execution on only uncaught runtime exceptions, you can click the "pause" button again. Clicking it again will disable pausing on exceptions.



Incorrect Logic

Incorrect logic in your code does not show any errors (e.g. in a JavaScript error console) but causes your code to not do what you intend it to. Debugging such errors requires some practice using debugging tools that your browser provides.

Debugging logic errors typically involves the following:

1. Logs and Asserts

Most browsers allow you to add logging statements to your code, to dump useful information about the execution of your JavaScript:

```
console.log("Mouse coordinates: " + evt.pageX + "," + evt.
pageY);
```

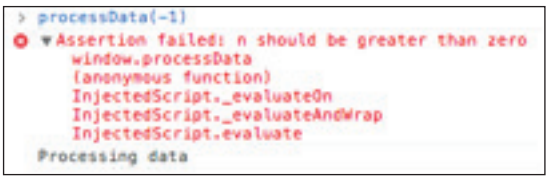
In Google Chrome, this output can be seen in the *Console* panel in DevTools. In addition to "log," you can also use "warn," "error," and "debug." These different categories can be filtered in the DevTools *Console*.

You can also use "console.assert()" to ensure that certain conditions or assumptions are being met in your code:

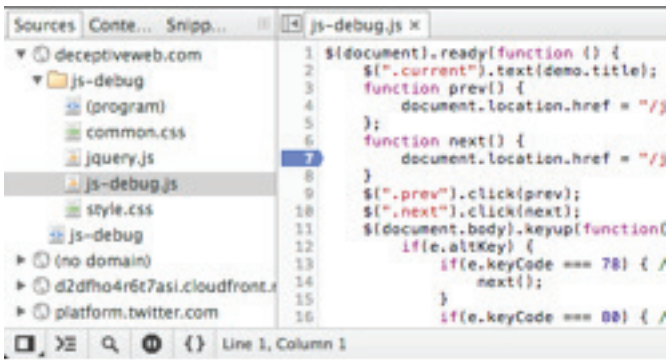
```
function processData(n, data) {
  console.assert(n > 0, "n should be greater than zero");
  console.log("Processing data");
}
```

When the specified condition is not true, the corresponding message will be printed to the *Console*.

Both "console.error" and "console.assert" also dump the execution call stack. Execution does continue after the assert.



2. Breakpoints
You can also specify breakpoints on any lines in your JavaScript code in the browser's debugger, e.g. in the *Sources* panel in Chrome DevTools. Similar functionality is available in all browsers. To specify a breakpoint on a line, you can click on its line number:



This will cause the debugger to pause the JavaScript when code execution reaches that line of code. This allows you to inspect the values of all variables, evaluate expressions, and inspect the call stack at that point in time. You can hover the mouse pointer over any variable name in the source text to see its value with the debugger paused.

You can see the call frames in the *Call Stack* side panel in the *Sources* panel. It shows the file names and line numbers for all the frames in the call stack. You can inspect the code and variable values at any of the call frames by moving to them and by clicking on them in the *Call Stack* side panel.

With the debugger paused, you can bring up the *Console* (by pressing the Escape key or moving to the *Console* panel in DevTools) and then evaluate any expression at that point in time. You have access to all variables in scope at the call frame selected in the *Call Stack* side panel (under *Sources*).

With the debugger paused on a line of code, you can step over the code line-by-line (e.g. by pressing F10 in DevTools) or step into the next function call (F11). You can also step out of the current function after executing the remainder of its code (Shift-F11) or continue execution by resuming the debugger (F8).

3. Watches
In addition to hovering the mouse pointer over variables to inspect their values, you can also set *watches* to allow you to monitor the values of variables and expressions.

Once added to the *Watches* side panel in the *Sources* panel, values of variables or expressions are kept up to date.

4. Conditional Breakpoints
If the breakpoint on a line gets hit to many times when you want it to break the execution only in specific cases, you can specify a condition on the breakpoint.

This can be done in DevTools by right-clicking on a line number and selecting "Add Conditional Breakpoint," or right-clicking on an existing breakpoint and selecting "Edit Breakpoint." You can then specify any expression as a condition for the breakpoint. The breakpoint will cause the JavaScript execution to pause only when the condition evaluates to true on that line of code.

DEBUGGING COMMON PROBLEMS

Let's look at some common issues we run into and how we can debug them.

Incorrect Comparisons

The equality operator "==" compares for equality after performing any necessary type conversions. The same is true for the inequality operator "!=". This results in the following results for comparisons:

Expressions	Result
0 == 0	true
0 ==	true
0 == false	true
undefined == null	true
" " == 0	true
false != 0	false

This results in the logic is incorrect when the code relies on the "==" and "!=" operators.

Unlike these operators, the identity operator "===" does not perform type conversion and evaluates a comparison to false if the types of the operands are different. The same is true for the "!==" operator.

To ensure that code logic does not rely on type coercions, you should always use the "===" and "!==" operators instead of "==" and "!=".

Value of "this"

"this" can get quite tricky, especially in event callbacks and setTimeout() or setInterval() handlers. For instance, if you extract a method from an object

```
var val = 1;
var obj = {
  val: 10,
  value: function() { return this.val; }
};
```

Also, when code is executed later via setTimeout() or setInterval(), "this" refers to the global "window" object.

In both these cases, if you want "this" to refer to the original object (when calling class methods), you should explicitly bind the original object to the callback, using "bind" (Function.prototype.bind). For instance, in the example

```
var value = obj.value.bind(obj);
console.log(value()); // 10
```

above:

Unintentional Global Variables

In your JavaScript code, if you miss declaring a variable with the "var" keyword, the variable is defined as a global (i.e. on the "window" object). This can lead to bugs that can be hard to track down, since it is not a syntax

```
var val = 10;
function process() {
   //...
   val = 12; // This modifies the global variable "val"
}
process();
console.log(val); // 12
```

Similarly, if you misspell a variable's name in a function (e.g. incorrect case), it will create or overwrite a global variable of that name.

Network Data

You can inspect both the outgoing and incoming network data from your web page or application using Chrome DevTools, Firebug, and other tools. This can be very useful when you receive unexpected data on the network.

The Networks panel in DevTools shows you, for each network request, the "method" (e.g. GET, POST), the HTTP status code (e.g. 200, 403), the MIME-type (e.g. application/json), the content's size and whether the network request was fulfilled from the browser's cache. This information can come in useful when you do not see the effect of code recently modified on the server or on the client (because the browser was using the data from its cache).

Developer tools in other browsers (e.g. Firebug) provide similar functionality.

Using the Console API Effectively

The console.* API provides very useful functionality – we have already looked at log, warn, error, debug and assert.

In addition to these is "console.trace()" which prints out the call stack at the point where it is called. This can be very useful in debugging complex code that involves a number of conditions to decide the code execution flow.

In Chrome DevTools and Firebug, console.log() also accepts some printf-style format specifiers (such as "%s" and "%d").

console.dir() can be used to display all the properties of a JavaScript object.

console.time("name") and console.timeEnd("name") can be used to measure (and label) the time between two points in the code execution.

This helps you avoid creating Date objects to manually measure time intervals. Multiple console.time() and console.timeEnd() timers can also be used simultaneously.

It is also a good idea to keep the JavaScript Console panel opened when working on a web page or application. The Console shows any runtime errors that the browser wants to bring to your notice.

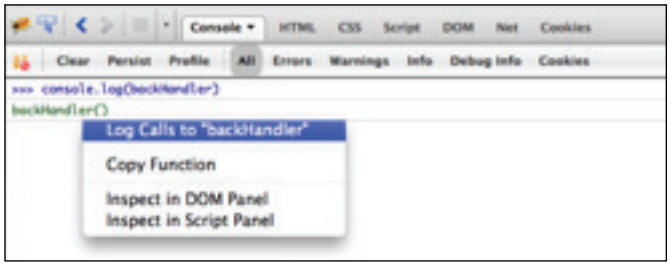
DEBUGGING DIFFICULT PROBLEMS

Let's now look at ways to debug problems that are hard to pin down.

Logging All Calls to a Function

Firebug has a unique functionality that allows you to log all invocations of a function. In addition, it also dumps the value of all arguments of the function with each invocation. This can be very helpful when one is deep within a workflow and in debugging third party code where one might not want to modify the existing code.

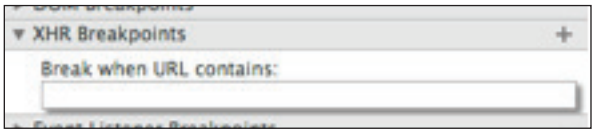
To log all calls to a function, one needs to console.log(functionName) in Firebug, and then right-click on the printed result and select "Log Calls to 'functionName'."



Finding Code that Initiated a Network Request

If you see a failed (or any) network request and you want to see the code that initiated it, Chrome DevTools provides this functionality. In the Networks panel, the "Initiator" column shows the line of code or HTML that was responsible for a network request. For AJAX requests initiated via JavaScript, it also shows the call stack that triggered the network request, which can be seen by hovering the mouse pointer over the initiator column:

You can also have DevTools pause JavaScript execution for all AJAX requests by setting an "XHR Breakpoint" in the Sources Panel. You can also specify a string that should be present in the URL for the breakpoint to be hit:



Passing Functions to setTimeout in a *for* Loop

If you pass an anonymous function as the first argument to setTimeout, and the anonymous function makes use of the for loop's index, each invocation of the callback uses the same value of the index, unlike what you might have

```
for(var i = 0; i < 5; i++) {
   setTimeout(function() {
      console.log(i);
   }, 100*i);
}
```

The above code prints out "5" five times to the *Console*. This is because the *for* loop completes before the first setTimeout callback executes and by then, the value of "i" (which is referenced by each of the five callbacks) is 5.

To avoid this problem, you need to ensure that each callback gets its own copy of "i". This can be done by either creating an anonymous closure for

```
for(var i = 0; i < 5; i++) {
   (function(index) {
      setTimeout(function() {
         console.log(index);
      }, 100*index);
   })(i);
}
```

Or

```
function setCallback(i) {
   setTimeout(function() {
      console.log(i);
   }, 100*i);
}

for(var i = 0; i < 5; i++) {
   setCallback(i);
}
```

Inspecting the Current Scope and Closure

When the debugger is paused (e.g. at a breakpoint), you can inspect the values of variables in the scope as well as the closure by using the *Scope Variables* side panel in the *Sources* panel in DevTools. This can be useful debugging code that involves JavaScript closures.

The Scripts tab in Firebug also shows the values of all variables in the scope or closure.

Browser Differences

Several functionalities are provided via different APIs by different browsers. For example, to initiate an AJAX request, one needs to instantiate different objects in different browsers – XMLHttpRequest or ActiveXObject("Microsoft.XMLHTTP"). Also, different versions of Internet Explorer require instantiating different ActiveXObjects.

Similarly, different browsers interpret JavaScript somewhat differently. For example, a trailing comma in a JavaScript array or object is ignored by

```
var arr = [ "a", "b", "c", ]; // ok for non-IE browsers
```

all browsers except Internet Explorer. In IE, a trailing comma can lead to unexpected or undefined behavior.

If your code works fine in one browser, it is not necessarily possible that it may break in another. It is always a good idea to test your web page or application in as many browsers as possible.

You can also use a library such as jQuery, which provides a cross-browser API that hides or works around individual browser issues (e.g. using jQuery.ajax for AJAX requests).

Loading Insecure Scripts

If a web page served over HTTPS loads a script over HTTP, most browsers do not execute the child script. This can cause your web page or application to not function correctly. On some browsers, the user can control whether insecure scripts should be loaded on secure web pages.

If your web page is served over HTTPS, you should ensure that all scripts that it loads (via its HTML or via JavaScript) should also be served over HTTPS.

Interference from Browser Extensions

If you have extensions (or add-ons) installed in your browser, they may interfere with your web page. While debugging, it might be useful to disable them to focus on debugging any issues in your JavaScript code.

A quick way of achieving this with Google Chrome is to use the browser's Incognito mode, which does not load any extensions by default.

Note that browser extensions may still interfere with your web page on the user's device, and you may want to handle that case.

DEBUGGING JQUERY

jQuery is perhaps the most popular JavaScript library that is used by web developers. Let's look at some tools that can help in debugging when you use jQuery.

Visual Event (Bookmarklet)

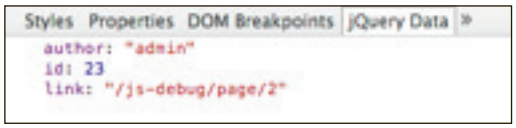
Visual Event [4] is a JavaScript bookmarklet that visually shows information about events that have been attached to DOM elements. It highlights the elements that have event listeners attached to them, the types of events, as well as the code for the event handlers.

In addition to several other libraries (e.g. YUI, MooTools, DOM 0 events) Visual Event also supports inspecting jQuery events. In the absence of such a tool, inspecting event listeners leads to code that lies in the jQuery library's source. With Visual Event, one can directly inspect the user code that handles jQuery events.

Browser Extensions

jQuery Debugger [5] is a browser extension for Google Chrome. It allows you to:

- Inspect jQuery selectors and watch their matching elements on the webpage
- See jQuery and HTML5 data of the selected element in a side panel in the Elements panel in DevTools
- See event handlers of the selected element



FireQuery [6] is an extension for Firebug (which itself is a browser extension for Firefox) that provides similar functionality.

TIPS AND TRICKS FOR DEBUGGING

Here are some tips and tricks to help you debug JavaScript more effectively:

debugger;

Instead of setting a breakpoint using the browser's debugger tool (e.g. Chrome DevTools), you can programmatically cause the JavaScript execution to pause at any point in your code by adding a "debugger;" statement at that point. If the debugger tool is not open, the "debugger;" statement is ignored. However, if it is open, the debugger will pause JavaScript execution and allow you to inspect the call stack and variables.

This can be more effective than adding a number of console.log() statements to print the values of variables at some point in your code.

Pretty-print

Production code is often minified (to reduce its download size), thereby making it very difficult to read. This is often the case with third party libraries (such as jQuery) and can also be true for your JavaScript code. You can improve the formatting of minified code in Chrome DevTools by clicking the "{}" button at the bottom of the Sources panel:

This not only makes the source code more readable, but also allows you to set breakpoints at points in code where you couldn't earlier (since they were in the middle of long lines of code in the minified file).

console.log in Conditional Breakpoints

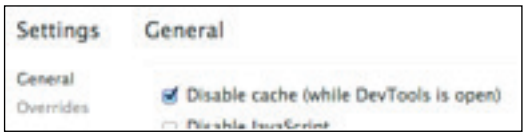
In addition to regular conditions that you can attach to breakpoints, you can also use console.log(...) as a condition. This allows you to print debugging information without modifying the code, e.g. when working with third-party code, or when you're deep within a workflow.

You can also use console.trace() as a breakpoint condition at a line of code to dump the call stack each time that line is executed.

By combining the two, "console.trace(); console.log(arguments)" can be used as a breakpoint condition at the first line of a function's body to dump the call stack and the value of arguments each time the function is invoked.

Disable the Browser Cache

To ensure that your web page is using the latest version of your code, you can disable the browser cache (while Chrome DevTools is open) via a DevTools setting (accessible via the cog wheel icon in the bottom right corner of DevTools). Without this, it is possible that the web page uses an old version of your code from the browser cache, while you keep making tweaks to it on the server.



Revisions of Your JavaScript Files

To force the users of your web page or application to use the latest version of your code, you can add a "?ver=number" suffix to the URL of your JavaScript file in your HTML source. Each time you push a new version of your code, you can increment the number in the URL.

This technique can also be used for your .css files.

monitorEvents()

In Chrome DevTools' Console, you can use the monitorEvents(element) command-line API obtain a log of all events for a specified element.

monitorEvents(document.body);

The above command will log all events on the body element. You can also pass an optional second argument to monitorEvents() to specify the events to log. You can specify individual events (e.g. "scroll," "resize") or a category of events (e.g. "mouse," "key," "touch," "control").

To specify the element to monitor (the first argument), you can use document.querySelector(selector) or use an existing reference to a DOM node. You can also select an element in the Elements panel, by clicking on it, and then using $0 as the first argument to monitorEvents(). $0 is a an alias for the element that is currently selected in the Elements panel.

To stop all monitoring events on an element, you need to call unmonitorEvents(element). You can optionally specify events or categories of events to stop monitoring by passing a second argument.

monitorEvents() and $0 are also supported by Firebug.

Clone JavaScript Objects for Logging

If you log a JavaScript object to the Console (with console.log) in a function that gets invoked a large number of times, and inspect the logs later, all of them would reflect the state of the object at the point in time when you later expand them (in the Console).

This can be a problem when the object is deep or has a large number of properties. (Using console.log() to log an object with only a few properties prints a string that shows the values of all its properties at the time it was logged.)

To capture the state of an object when it was logged, you can obtain a JavaScript Object Notation (JSON) for the object and use that to create a copy of the object:

```
console.log(JSON.parse(JSON.stringify(obj)));
```

instead of:

```
console.log(obj);
```

DEBUGGING MOBILE WEB PAGES

Let's look at some ways in which one can debug mobile web pages and applications, from a desktop machine.

JavaScript Console Remoting

JSConsole [7] is a useful tool that allows you to see the output of console.log() statements running on a mobile device, on your desktop browser. After including a specific <script> tag in your web page, any calls to console.log() will show their output in the JSConsole session in the desktop browser.

Mobile Emulation

Chrome DevTools allows you to emulate a mobile device to some extent, in your desktop browser. You can access this functionality in the Overrides tab under DevTools' settings (accessible via the cog wheel icon in the bottom right corner of DevTools).

You can specify a mobile user agent, device metrics (e.g. screen resolution), mocked geolocation, device orientation as well as emulate touch events.

Remote Debugging

Remote debugging for mobile devices is supported by Chrome [8], Safari [9] and Firefox [10]. This allows you to debug web pages and applications running on a mobile device, using your desktop browser's debugging interface.

Some additional remote debugging tools are also available:

- Weinre, which uses the WebKit remote debugging protocol and allows you to debug web pages on mobile devices.
- Adobe Edge Inspect, which helps you preview & inspect web pages on multiple devices.

REFERENCES

[1] Node – Event-driven server-side JavaScript environment
http://nodejs.org/

[2] JSHint - Detect errors and potential problems in JavaScript code
http://jslint.com/

[3] JSLint - The JavaScript Code Quality Tool
http://jshint.com/

[4] Visual Event – Inspect events on DOM nodes
http://www.sprymedia.co.uk/article/visual+event+2

[5] jQuery Debugger – Browser Extension for Google Chrome
https://chrome.google.com/webstore/detail/jquery-debugger/dbhhnnnpaeobfddmlalhnehgclcmjimi

[6] FireQuery – Extension for Firebug
https://addons.mozilla.org/en-us/firefox/addon/firequery/

[7] JSConsole – JavaScript Console Remoting
http://jsconsole.com/

[8] Remote Debugging with Chrome
https://developers.google.com/chrome-developer-tools/docs/remote-debugging

[9] Remote Debugging with Safari
http://webdesign.tutsplus.com/tutorials/workflow-tutorials/quick-tip-using-web-inspector-to-debug-mobile-safari/

[10] Remote Debugging with Firefox
https://hacks.mozilla.org/2012/08/remote-debugging-on-firefox-for-android/

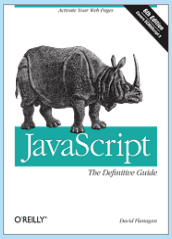
[11] Weinre
http://people.apache.org/~pmuellr/weinre/

[12] Adobe Edge Inspect
http://html.adobe.com/edge/inspect/

ABOUT THE AUTHOR

Ashutosh Sharma is a Senior Computer Scientist at Adobe. He has extensive experience as an architect and developer on major projects like Adobe AIR and Adobe Reader, and often writes and speaks on advanced web development and debugging. Ashutosh blogs at http://deceptiveweb.com/blog/ .

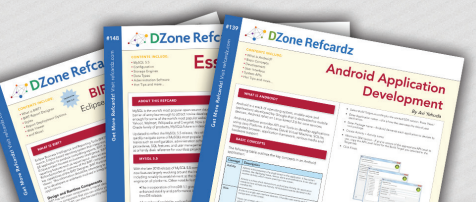
RECOMMENDED BOOK



Since 1996, JavaScript: The Definitive Guide has been the bible for JavaScript programmers—a programmer's guide and comprehensive reference to the core language and to the client-side JavaScript APIs defined by web browsers.

The 6th edition covers HTML5 and ECMAScript 5. Many chapters have been completely rewritten to bring them in line with today's best web development practices. New chapters in this edition document jQuery and server side JavaScript. It's recommended for experienced programmers who want to learn the programming language of the Web, and for current JavaScript programmers who want to master it.

BUY NOW!



Browse our collection of over 150 Free Cheat Sheets

Free PDF

Upcoming Refcardz

Java EE7
Open Layers
Wordpress

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **"DZone is a developer's dream",** says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-83-7
ISBN-10: 1-936502-83-6
50795
9 781936 502837
$7.95

Version 1.0