

console

The console object provides access to the browser's debugging console (e.g. the Web Console in Firefox). The specifics of how it works varies from browser to browser, but there is a *de facto* set of features that are typically provided.

The **console** object provides access to the browser's debugging console (e.g. the [Web Console](#) in Firefox). The specifics of how it works varies from browser to browser, but there is a *de facto* set of features that are typically provided.

The **console** object can be accessed from any global object. **Window** on browsing scopes and **WorkerGlobalScope** as specific variants in workers via the property `console`. It's exposed as `Window.console`, and can be referenced as simply `console`. For example:

```
console.log("Failed to open the specified link")
```

This page documents the [Methods](#) available on the **console** object and gives a few [Usage](#) examples.

Note: This feature is available in [Web Workers](#).

Note: The actual **console** interface is defined as all lower case (i.e. not **Console**), for historical reasons.

Methods

`console.assert()`

Log a message and stack trace to console if the first argument is `false`.

`console.clear()`

Clear the console.

`console.count()`

Log the number of times this line has been called with the given label.

`console.countReset()`

Resets the value of the counter with the given label.

`console.debug()`

Outputs a message to the console with the log level `debug`.

`console.dir()`

Displays an interactive listing of the properties of a specified JavaScript object. This listing lets you use disclosure triangles to examine the contents of child objects.

`console.dirxml()`

Displays an XML/HTML Element representation of the specified object if possible or the JavaScript Object view if it is not possible.

`console.error()`

Outputs an error message. You may use [string substitution](#) and additional arguments with this method.

`console.exception()`

An alias for `error()`.

`console.group()`

Creates a new inline [group](#), indenting all following output by another level. To move back out a level, call `groupEnd()`.

`console.groupCollapsed()`

Creates a new inline [group](#), indenting all following output by another level. However, unlike `group()` this starts with the inline group collapsed requiring the use of a disclosure button to expand it. To move back out a level, call `groupEnd()`.

`console.groupEnd()`

Exits the current inline [group](#).

`console.info()`

Informative logging of information. You may use [string substitution](#) and additional arguments with this method.

`console.log()`

For general output of logging information. You may use [string substitution](#) and additional arguments with this method.

`console.profile()`

Starts the browser's built-in profiler (for example, the [Firefox performance tool](#)). You can specify an optional name for the profile.

`console.profileEnd()`

Stops the profiler. You can see the resulting profile in the browser's performance tool (for example, the [Firefox performance tool](#)).

`console.table()`

Displays tabular data as a table.

`console.time()`

Starts a [timer](#) with a name specified as an input parameter. Up to 10,000 simultaneous timers can run on a given page.

```
console.timeEnd()
```

Stops the specified [timer](#) and logs the elapsed time in seconds since it started.

```
console.timeLog()
```

Logs the value of the specified [timer](#) to the console.

```
console.timeStamp()
```

Adds a marker to the browser's [Timeline](#) or [Waterfall](#) tool.

```
console.trace()
```

Outputs a [stack trace](#).

```
console.warn()
```

Outputs a warning message. You may use [string substitution](#) and additional arguments with this method.

Examples

Outputting text to the console

The most frequently-used feature of the console is logging of text and other data. There are four categories of output you can generate, using the `console.log()`, `console.info()`, `console.warn()`, and `console.error()` methods respectively. Each of these results in output styled differently in the log, and you can use the filtering controls provided by your browser to only view the kinds of output that interest you.

There are two ways to use each of the output methods; you can simply pass in a list of objects whose string representations get concatenated into one string, then output to the console, or you can pass in a string containing zero or more substitution strings followed by a list of objects to replace them.

Outputting a single object

The simplest way to use the logging methods is to output a single object:

```
var someObject = { str: "Some text", id: 5 };  
console.log(someObject);
```

The output looks something like this:

```
[09:27:13.475] ({str:"Some text", id:5})
```

Outputting multiple objects

You can also output multiple objects by simply listing them when calling the logging method, like this:

```
var car = "Dodge Charger";  
var someObject = { str: "Some text", id: 5 };  
console.info("My first car was a", car, ". The object is:", someObject);
```

This output will look like this:

```
[09:28:22.711] My first car was a Dodge Charger . The object is: ({str:"Some  
text", id:5})
```

Using string substitutions

When passing a string to one of the `console` object's methods that accepts a string (such as `log()`), you may use these substitution strings:

`%o` or `%O`

Outputs a JavaScript object. Clicking the object name opens more information about it in the inspector.

`%d` or `%i`

Outputs an integer. Number formatting is supported, for example `console.log("Foo %.2d", 1.1)` will output the number as two significant figures with a leading 0: `Foo 01`

`%s`

Outputs a string.

`%f`

Outputs a floating-point value. Formatting is supported, for example `console.log("Foo %.2f", 1.1)` will output the number to 2 decimal places: `Foo 1.10`

Note: Precision formatting doesn't work in Chrome

Each of these pulls the next argument after the format string off the parameter list. For example:

```
for (var i=0; i<5; i++) {  
  console.log("Hello, %s. You've called me %d times.", "Bob", i+1);  
}
```

The output looks like this:

```
[13:14:13.481] Hello, Bob. You've called me 1 times.  
[13:14:13.483] Hello, Bob. You've called me 2 times.  
[13:14:13.485] Hello, Bob. You've called me 3 times.
```

```
[13:14:13.487] Hello, Bob. You've called me 4 times.  
[13:14:13.488] Hello, Bob. You've called me 5 times.
```

The `console` interface's `trace()` method outputs a stack trace to the [Web Console](#).

Note: This feature is available in [Web Workers](#).

See [Stack traces](#) in the `console` documentation for details and examples.

Syntax

```
console.trace( [...any, ...data ] );
```

Parameters

...any, *...data* Optional

Zero or more objects to be output to console along with the trace. These are assembled and formatted the same way they would be if passed to the `console.log()` method.

Example

```
function foo() {  
  function bar() {  
    console.trace();  
  }  
  bar();  
}  
  
foo();  
<span>
```

In the console, the following trace will be displayed:

```
bar  
foo  
<anonymous>  
;
```

Specifications

Specification	Status	Comment
---------------	--------	---------

Specification	Status	Comment
Console API The definition of 'console.trace()' in that specification.	Living Standard	Initial definition

Styling console output

You can use the `%c` directive to apply a CSS style to console output:

```
console.log("This is %cMy stylish message", "color: yellow; font-style: italic; background-color: blue;padding: 2px");
```

The text before the directive will not be affected, but the text after the directive will be styled using the CSS declarations in the parameter.

This is **My stylish message**

The properties usable along with the `%c` syntax are as follows (at least, in Firefox — they may differ in other browsers):

- `background` and its longhand equivalents.
- `border` and its longhand equivalents
- `border-radius`
- `box-decoration-break`
- `box-shadow`
- `clear` and `float`
- `color`
- `cursor`
- `display`
- `font` and its longhand equivalents
- `line-height`
- `margin`
- `outline` and its longhand equivalents
- `padding`
- `text-*` properties such as `text-transform`
- `white-space`
- `word-spacing` and `word-break`
- `writing-mode`

Note: The console message behaves like an inline element by default. To see the effects of `padding`, `margin`, etc. you should set it to for example `display: inline-block`.

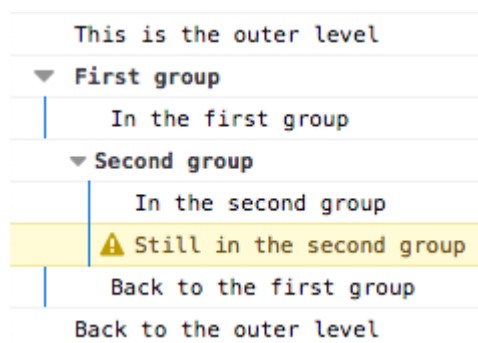
Using groups in the console

You can use nested groups to help organize your output by visually combining related material. To create a new nested block, call `console.group()`. The `console.groupCollapsed()` method is similar but creates the new block collapsed, requiring the use of a disclosure button to open it for reading.

To exit the current group, simply call `console.groupEnd()`. For example, given this code:

```
console.log("This is the outer level");
console.group("First group");
console.log("In the first group");
console.group("Second group");
console.log("In the second group");
console.warn("Still in the second group");
console.groupEnd();
console.log("Back to the first group");
console.groupEnd();
console.debug("Back to the outer level");
```

The output looks like this:



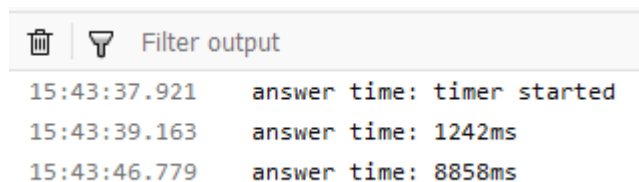
Timers

You can start a timer to calculate the duration of a specific operation. To start one, call the `console.time`()` method, giving it a name as the only parameter. To stop the timer, and to get the elapsed time in milliseconds, just call the `console.timeEnd()` method, again passing the timer's name as the parameter. Up to 10,000 timers can run simultaneously on a given page.

For example, given this code:

```
console.time("answer time");
alert("Click to continue");
console.timeLog("answer time");
alert("Do a bunch of other stuff...");
console.timeEnd("answer time");
```

Will log the time needed by the user to dismiss the alert box, log the time to the console, wait for the user to dismiss the second alert, and then log the ending time to the console:



The screenshot shows a browser console with a 'Filter output' button and three log entries. Each entry has a timestamp and a message. The messages are 'answer time: timer started', 'answer time: 1242ms', and 'answer time: 8858ms'.

```
15:43:37.921    answer time: timer started
15:43:39.163    answer time: 1242ms
15:43:46.779    answer time: 8858ms
```

Notice that the timer's name is displayed both when the timer is started and when it's stopped.

Note: It's important to note that if you're using this to log the timing for network traffic, the timer will report the total time for the transaction, while the time listed in the network panel is just the amount of time required for the header. If you have response body logging enabled, the time listed for the response header and body combined should match what you see in the console output.

Stack traces

The console object also supports outputting a stack trace; this will show you the call path taken to reach the point at which you call `console.trace()`. Given code like this:

```
function foo() {
  function bar() {
    console.trace();
  }
  bar();
}

foo();
```

The output in the console looks something like this:



The screenshot shows a browser console with a stack trace. The trace starts with 'console.trace():' and lists the call stack: 'bar()' at 'main.js:46', 'foo()' at 'main.js:48', and '<anonymous>' at 'main.js:42'.

```
console.trace():                                main.js:46
  bar()                                          main.js:46
  foo()                                          main.js:48
  <anonymous>                                   main.js:42
```

Specifications

Specification	Status	Comment
Console API	Living Standard	Initial definition.

JavaScript Stack Trace: Understanding It and Using It to Debug | Scalyr

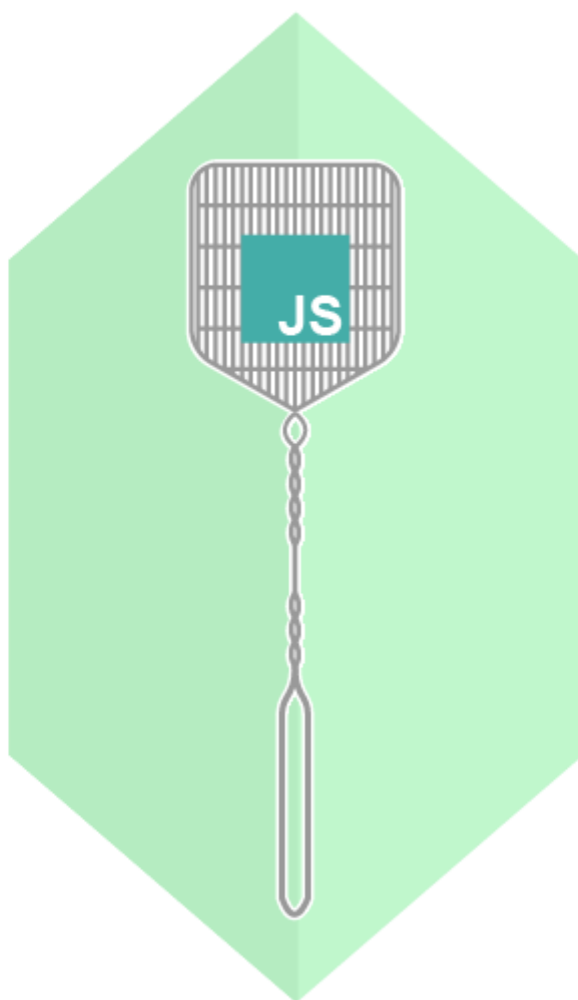
What's the stack trace in JavaScript? Learn that and all about the anatomy of the JavaScript stack trace —plus how to ensure good error reports for debugging

We've all been there. You've set up a new project, everything is going smoothly, and boom, it hits. You're staring at a big red error in your console. After trying everything you can think of, you just can't seem to resolve it. In desperation, you resort to posting the error up on the internet. Help!

Sure enough, an anonymous hero, "Linux412," rides in and informs you, "Looks like you've got a naming error on your function." Aha—that fixed it, but how did they know?! The secret to the anonymous developer's magic lies in their ability to read and make sense of application error reports.

If we learn how to read these errors, we'll be able to solve nearly all of our coding problems. In JavaScript, the error report is called a stack trace. We already covered the stack trace in [PHP](#) and [Java](#) in earlier articles. But today, it's the JavaScript stack trace that's under discussion.

By the end of this article, you should understand what the JavaScript stack trace is and why naming functions and error messages are important for debugging with stack traces.



What Is a Stack Trace?

As always, let's start by defining a stack trace.

A stack trace is a list of the functions, in order, that lead to a given point in a software program.

A stack trace is essentially a breadcrumb trail for your software. You can easily see the stack trace in [JavaScript](#) by adding the following into your code:

```
console.trace();
```

And you'll get an outputted stack trace.

But what does a stack trace actually look like? Let's cover that now.

A stack trace is essentially a breadcrumb trail for your software.

The Anatomy of a Stack Trace

To understand the stack trace better, let's take an example.

```
function controller(){ database(); }
```

```
function database(){ query(); }
```

```
function query () { throw new Error(); }
```

Here we see a series of functions that call each other. The last function will error, causing the program to halt and emit a stack trace.

When the last function errors, the following stack trace is emitted:

```
Uncaught Error at query (:10:11) at database (:6:5) at controller (:2:5)
```

And you can see from the above error, query was called by database, which in turn was called by controller. You can see then that the stack trace shows which functions called each other. Importantly, we can also see the line number and file that the function exists on, and the topmost line is the error itself.

The stack trace helps us to know the steps that lead up to our error. And the stack trace isn't only useful for humans. If you're using error tooling that tracks and stores errors, it can use these stack traces to count how often an error occurs.

But wait—before you take your newfound knowledge of stack traces and start using the console.trace method as a debugging tool, there are a few other things we should consider. In JavaScript, the stack trace may not always appear as you think. What do I mean? Let me explain...

Function Naming, and Why It's Important

One of the key components in the above stack trace is that it emits not only an error message and a set of line numbers, but the function names. However, in JavaScript, not every function has a name. Some functions can be anonymous.

These anonymous functions pose a threat to our debugging. If our stack traces include anonymous functions, we'll greatly reduce the value of the stack trace because it's much harder to see the steps that lead to an error. This will make our whole debugging process so much more painful.

Take, for instance, the following example:

```
const database = Promise.resolve(); database .then(function () { throw new Error(); });
```

When we run the above code, we see the following stack trace:

```
Uncaught (in promise) Error at
```

As you can see, the error is thrown, but the parent function that led to the error is anonymous. Not only that, but the error itself is quite cryptic. It tells us that an error happened, but no indication is made as to why the error occurred.

The stack trace contains a reference to an unnamed function because in the previous example the error that was thrown was in fact anonymous. Go ahead and take a look. We didn't tell JavaScript the name of the function, so it can't report it, which isn't very useful.

So how can we resolve the problem and go back to understanding the history of each of our called functions? By ensuring that we apply a name property to all of our functions.

Let's modify the above example with a name on our function.

```
const database = Promise.resolve(); database .then(function resolver () { throw new Error(); });
```

Which now results in the following stack trace:

```
Uncaught (in promise) Error at resolver (:1:81)
```

Aha! Much better.

Always Name Your Functions

Generally speaking, it's wise to always name your functions, no matter the situation. You can name a function by adding the name between the function keyword and your argument brackets. But don't be fooled—an anonymous function assigned to a variable is still anonymous.

For instance, take the following:

```
const named = function(){};
```

Here, it appears that the function is named, but it isn't. Because there's no name between the function keyword and the round brackets, the function has no name and will appear as anonymous in our stack trace. Some JavaScript engines can infer our name in some scenarios, but it's not an officially supported feature, so don't be fooled. Always name your functions.

Give Names to Your JavaScript Errors

The keen-eyed among you will have noticed that in all the examples shown so far, the error messages themselves have been pretty useless. For instance, "Uncaught (in promise) Error" gives us very little context or understanding to fix the error.

But there is hope! We can actually fix these error messages. Let me show you how.

In order for stack traces to be their most effective, we should anticipate as many error states as possible and throw descriptive errors. Say, for instance, that JavaScript encountered a missing piece of data—let's say a username not in a database. We wouldn't want our code to continue running, because it would try to perform actions on the missing data and would eventually throw some unhelpful message like:

Uncaught ReferenceError: user is not defined

Seem familiar? It will be to most developers. Because JavaScript is quite flexible, running into these cryptic error messages is all too common.

So, how do we fix it? By throwing errors with descriptive error messages.

Let's go back to our first example to look at throwing named errors in action. If we take our original example, rather than throwing a generic error, let's go ahead and add an error message to our error object constructor, as follows:

```
function controller(){ database(); }
```

```
function database(){ query(); }
```

```
function query () { throw new Error('Error occurred in query, please try X'); }
```

Which will result in the following stack trace:

Uncaught Error: Error occurred in query, please try X at query (:5:27) at database (:3:22) at controller (:1:24)

Now that's a lot more readable!

Notice that we even modified our error message to include a suggestion for a fix.

When you're writing the code at that point in time, you'll have a good understanding of all the different ways that your function can fail. So, if you can, explicitly throw errors with meaningful messages to help out your future self when you're debugging your code.

No more will you need to mindlessly paste errors onto the internet.

And That's All on the Stack Trace

That concludes our look at JavaScript stack traces. I hope it helps you feel more confident when you see stack traces and that you understand how they're created, how to read them, and ultimately how to use them confidently to debug your application.

Remember, always name functions, and don't be fooled by assigning them to variables, which are still unnamed. And also, try to predict failure points in your application and throw reasonable error messages that will make your life easier later on.

No more will you need to mindlessly paste errors onto the internet. Instead, you can sift through them yourself like an investigator solving a mystery. Debugging applications in a much more logical and straightforward manner should save you plenty of headaches in the long run, and make writing code a little bit more fun.

Good luck!

This post was written by Lou Bichard. [Lou](#) is a JavaScript full stack engineer with a passion for culture, approach, and delivery. He believes the best products emerge from high performing teams and practices. Lou is a fan and advocate of old-school lean and systems thinking, XP, continuous delivery, and DevOps.

[Source](#)

Stack trace in JavaScript

When you are done check out how else we might help you!

When you are done [check out](#) how else we might help you!

We have already seen 5 levels of [logging in JavaScript](#). Each one showed the line number where the function was called. That's very nice, but if the same function can be called in multiple places then having that context can improve your understand what has happened.

Printing a full stack trace can help a lot.

Using console.trace

Luckily the console object also has a method called trace.

In the following example we have a few totally useless functions calling each other and then calling the add function, that calls console.trace.

examples/javascript/logging/logging_trace.js

```
1. function add(x, y) {  
2.   console.trace('add called with ', x, 'and', y);  
3.   return x+y;  
4. }  
5. function calc() {  
6.   return add(8, 11) + add(9, 14);  
7. }  
8. function main() {  
9.   var x = add(2, 3);  
10.  var y = calc();
```

```
11. }  
  
12. main();
```

The output in [the JavaScript web console](#) of Chrome looks like this:

 console trace in Chrome

In Firefox it looks like this:

 console trace in Firefox

If you'd like to try it yourself, here is the HTML that will load the above JavaScript file.

examples/javascript/logging/logging_trace.html

```
1.  
2.  
3.  
4.  
5.  
6.  
7.  
8. <a href="http://code-maven.com/open-javascript-console" target="_blank">Open the JavaScript  
   console  
9. in order to see the logging messages.  
10.  
11.
```

[Try!](#)

Stack trace as a string

There are cases when you might not necessarily want to print the stacktrace immediately, or you might even want to save it or send it to the server. For such cases, it would be nice to be able to get a string version of the stack trace.

The following solutions are based on ideas found on [this](#) and [this](#) page.

Stack trace with Error object

In this solution we create an Error object and then return (and print) the stack attribute.

The full JavaScript code looks like this:

examples/javascript/logging/logging_trace_with_error.js

```
1. function add(x, y) {  
  
2.   console.log(new Error().stack);  
  
3.   return x+y;
```

```
4. }  
  
5. function calc() {  
  
6. return add(8, 11) + add(9, 14);  
  
7. }  
  
8. function main() {  
  
9. var x = add(2, 3);  
  
10. var y = calc();  
  
11. }  
  
12. main();
```

The corresponding HTML file is not very interesting, but it is included to make it easier for you to try it:

`examples/javascript/logging/logging_trace_with_error.html`

```
1.  
2.  
3.  
4.  
5.  
6.  
7.  
8. <a href="http://code-maven.com/open-javascript-console" target="_blank">Open the JavaScript  
   console  
9. in order to see the logging messages.  
10.  
11.
```

Try!

The output in Chrome:  console trace in Chrome

and in FireFox:  console trace in Firefox

Stack trace using caller object

In this solution we have implemented a function called `stacktrace` that will return a string representing the call history to the point where `stacktrace()` was called.

Internally it uses another function called `st2` that will be called recursively traversing the call-tree up till the point where we reach the main body of our JavaScript script.

`examples/javascript/logging/logging_stacktrace.js`

```
1. function add(x, y) {  
  
2. console.log(stacktrace());
```

```
3. return x+y;

4. }

5. function calc() {

6. return add(8, 11) + add(9, 14);

7. }

8. function main() {

9. var x = add(2, 3);

10. var y = calc();

11. }

12. main();

13. function stacktrace() {

14. function st2(f) {

15. var args = [];

16. if (f) {

17. for (var i = 0; i < f.arguments.length; i++) {

18. args.push(f.arguments[i]);

19. }

20. var function_name = f.toString().split('(')[0].substring(9);

21. return st2(f.caller) + function_name + '(' + args.join(', ') + ')' + "\n";

22. } else {

23. return "";

24. }

25. }

26. return st2(arguments.callee.caller);

27. }
```

At the end of the declaration of `stacktrace` we call `st2` with the `arguments.callee.caller`. [arguments](#) is a special object that belongs to the current function call and that contains a lot of information about the current call. For example the `callee` attribute refers to the currently executing function which is `stacktrace`.

We can probably replace `return st2(arguments.callee.caller);` by `return st2(stacktrace.caller);`, but then we repeat the function name which will make it harder to rename the function.

the `st2` function is then called recursively and it returns the `stacktrace` string so far. When it reaches the top-most function call (in our case `main`) the next recursive call will be with an undefined value received from `f.caller`. That will make it return an empty string without further recursive calls.

If `f` is not yet undefined we build up the current call at the current level of the `stacktrace`. We use the `arguments` object of the current function. Because it is not a real Array we cannot use `join` and we have to loop over the elements to build up the list of arguments received by this call. That's what is saved in the `args` array.

`f.toString()` returns the string representation of the function `f`. Each such string representation starts with `function some_name(param, param) {`. `split('(')` cuts that string at the `(` characters in the source code of the function, including the first `(` in the argument declaration.

The `[0]` means we take the first element from the returned array. That returns function `some_name`. Calling `substring(9)` take all the characters except the first 9 and returns that string. The returned string is the name of the function. In our example it is `some_name`. That's how we can extract the name of the currently called function.

Now that we have both the name of the current function and the list of parameters it received we can create a string that represents the call.

The result looks like this in Chrome:

 console trace in Chrome

and this in FireFox:

 console trace in Firefox

If you'd like to try it yourself, here is the corresponding html file:

`examples/javascript/logging/logging_stacktrace.html`

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.
8. `Open the JavaScript`
console
9. in order to see the logging messages.
- 10.
- 11.

Try!

The console API

For even further details check out the [console API of Chrome](#) and the [console API of Firefox](#).

If this article helped you, [check out](#) what else can you learn here!

In the comments, please wrap your code snippets within

tags and use spaces for indentation.

If you have any comments or questions, feel free to post them on the source of this page in GitHub. [Source on GitHub.](#)

[Source](#)