

Command Line API Reference



By Andi Smith

Andi is a contributor to WebFundamentals




By Meggin Kearney

Meggin is a Tech Writer

The Command Line API contains a collection of convenience functions for performing common tasks: selecting and inspecting DOM elements, displaying data in readable format, stopping and starting the profiler, and monitoring DOM events.

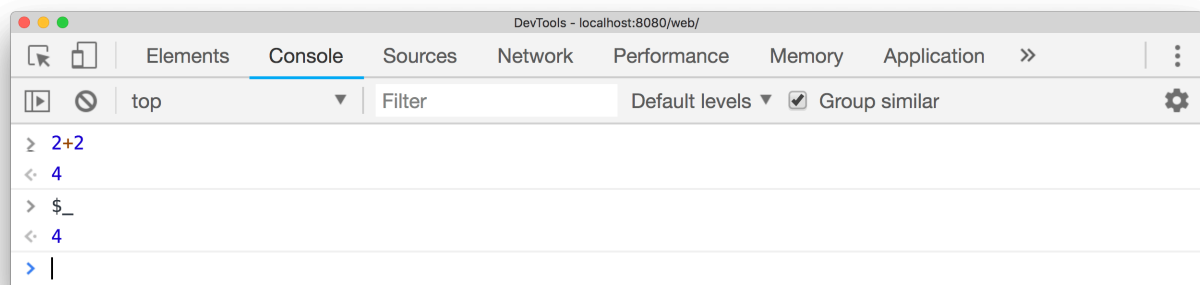
Note: This API is only available from within the console itself. You cannot access the Command Line API from scripts on the page.

Note: If you are looking for functions that write to the Console (functions that start with `console.*`), consult the  Console API instead.

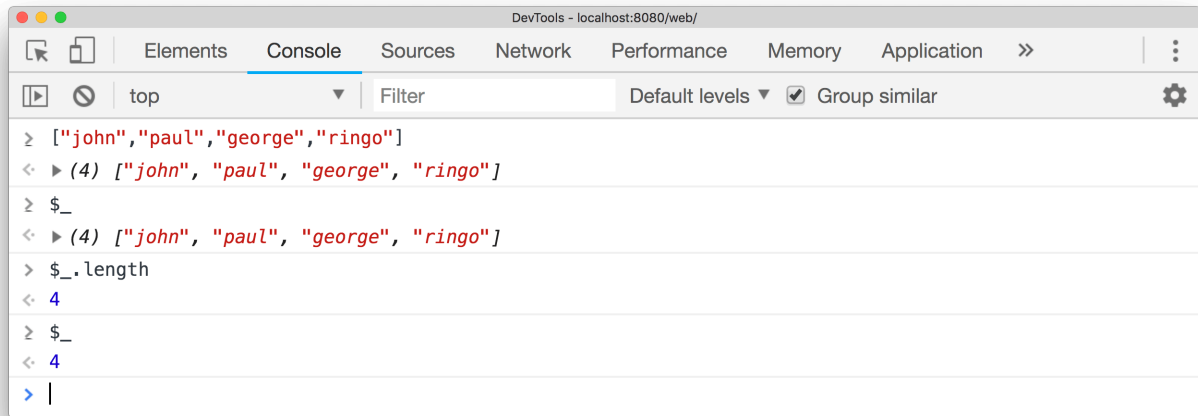
\$_

`$_` returns the value of the most recently evaluated expression.

In the following example, a simple expression (`2 + 2`) is evaluated. The `$_` property is then evaluated, which contains the same value:



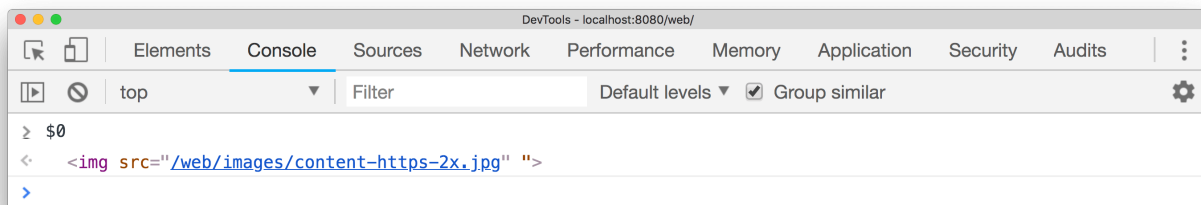
In the next example, the evaluated expression initially contains an array of names. Evaluating `$_ .length` to find the length of the array, the value stored in `$_` changes to become the latest evaluated expression, `4`:



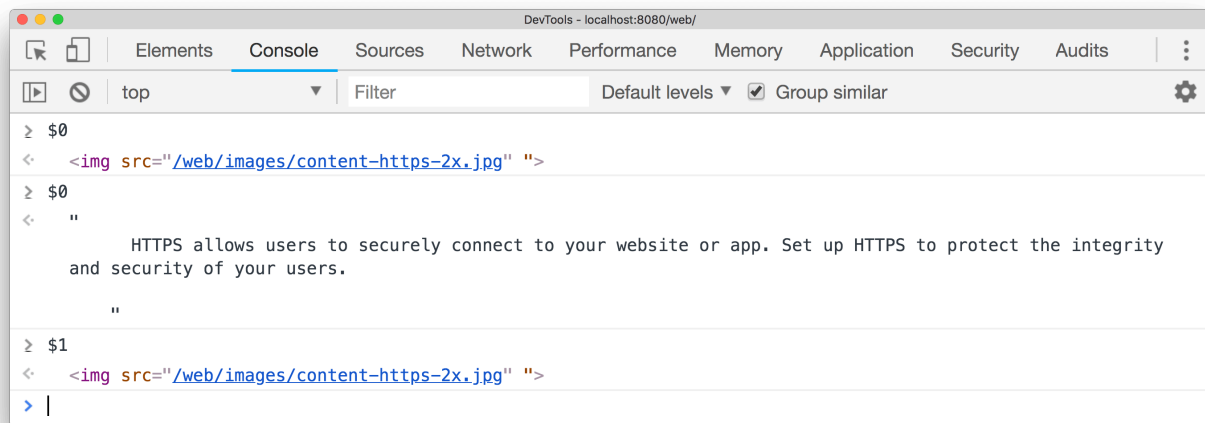
\$0 - \$4

The `$0`, `$1`, `$2`, `$3` and `$4` commands work as a historical reference to the last five DOM elements inspected within the Elements panel or the last five JavaScript heap objects selected in the Profiles panel. `$0` returns the most recently selected element or JavaScript object, `$1` returns the second most recently selected one, and so on.

In the following example, an `img` element is selected in the Elements panel. In the Console drawer, `$0` has been evaluated and displays the same element:



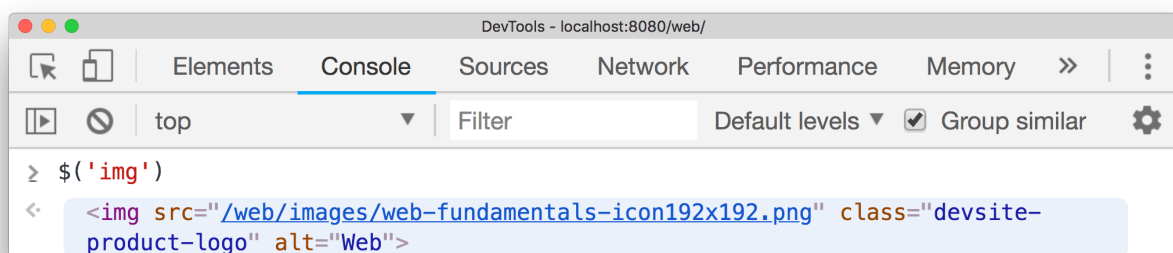
The image below shows a different element selected in the same page. The `$0` now refers to newly selected element, while `$1` returns the previously selected one:



\$(selector, [startNode])

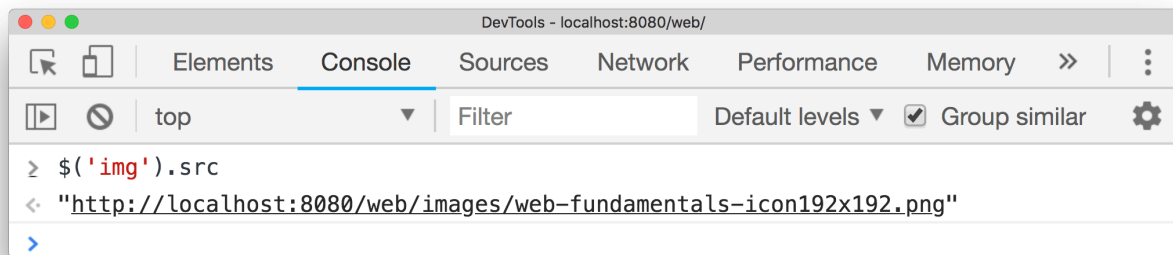
\$(selector) returns the reference to the first DOM element with the specified CSS selector. This function is an alias for the [document.querySelector\(\)](#) function.

The following example returns a reference to the first `` element in the document:



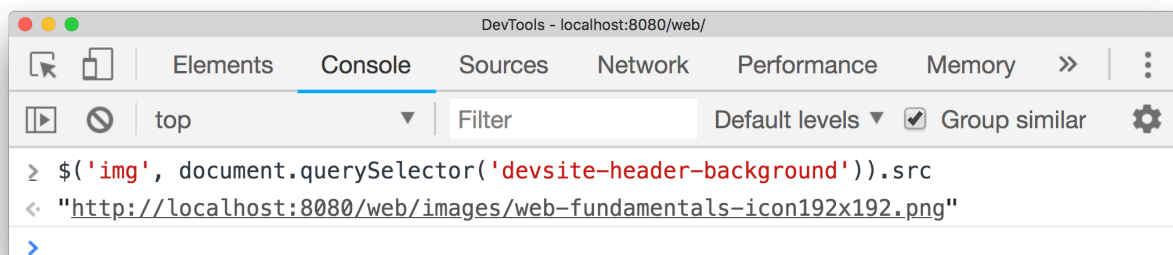
Right-click on the returned result and select 'Reveal in Elements Panel' to find it in the DOM, or 'Scroll in to View' to show it on the page.

The following example returns a reference to the currently selected element and displays its `src` property:



This function also supports a second parameter, `startNode`, that specifies an 'element' or Node from which to search for elements. The default value of this parameter is `document`.

The following example returns a reference to the first element after the currently selected Node and displays its `src` properly:



Note: If you are using a library such as jQuery that uses `$`, this functionality will be overwritten, and `$` will correspond to that library's implementation.

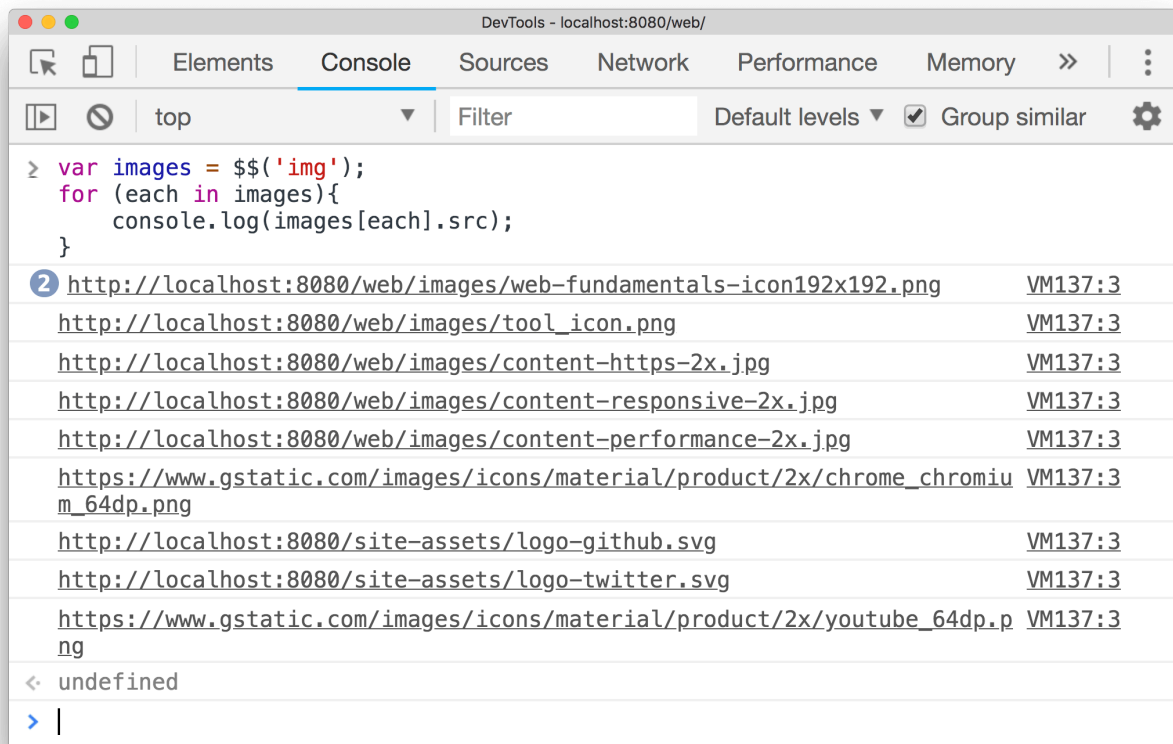
`$$ (selector, [startNode])`

`$$ (selector)` returns an array of elements that match the given CSS selector. This command is equivalent to calling `document.querySelectorAll()`.

The following example uses `$$()` to create an array of all `` elements in the current document and displays the value of each element's `src` property:

```
var images = $$('img');
for (each in images) {
  console.log(images[each].src);
}
```

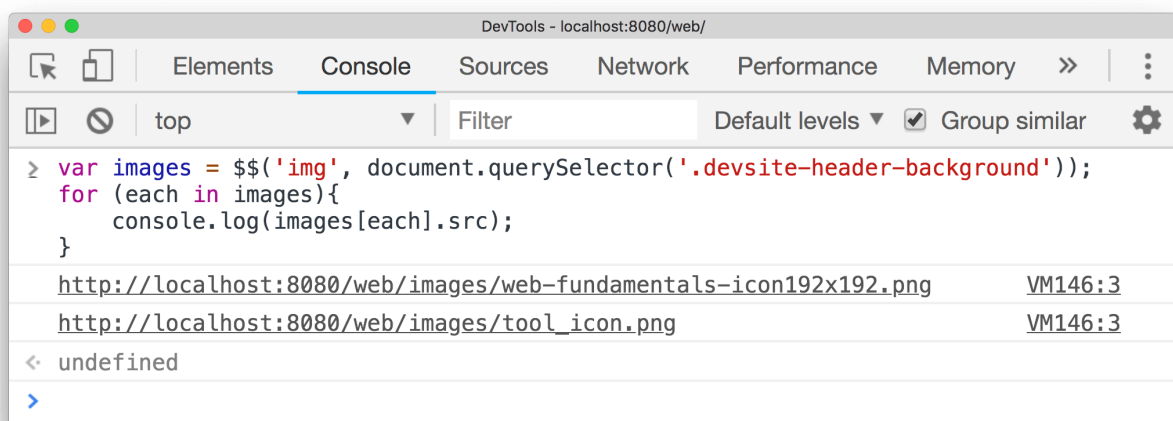




This function also supports a second parameter, `startNode`, that specifies an element or Node from which to search for elements. The default value of this parameter is `document`.

This modified version of the previous example uses `$$()` to create an array of all `` elements that appear in the current document after the selected Node:

```
var images = $('img', document.querySelector('.devsite-header-background'));
for (each in images) {
  console.log(images[each].src);
}
```



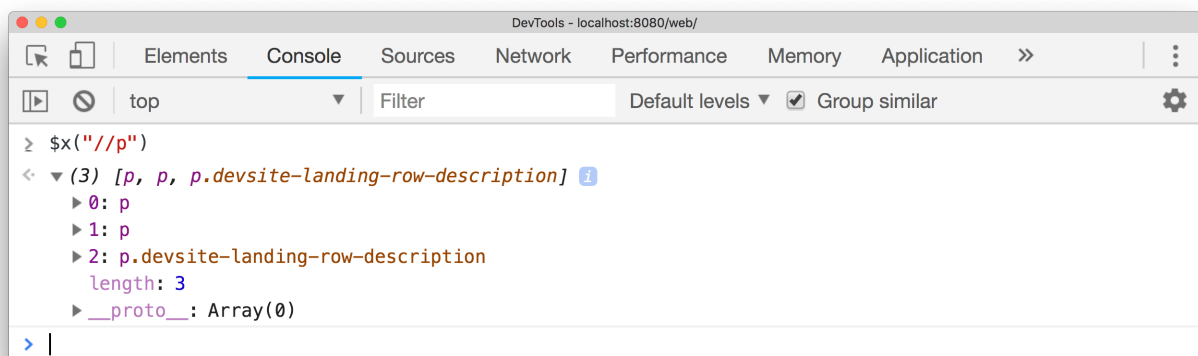
Note: Press Shift + Enter in the console to start a new line without executing the script.

`$x(path, [startNode])`

`$x(path)` returns an array of DOM elements that match the given XPath expression.

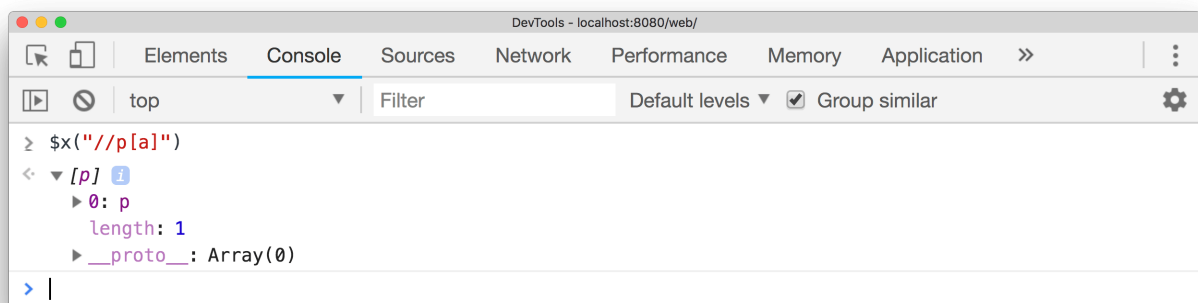
For example, the following returns all the `<p>` elements on the page:

```
$x("//p")
```

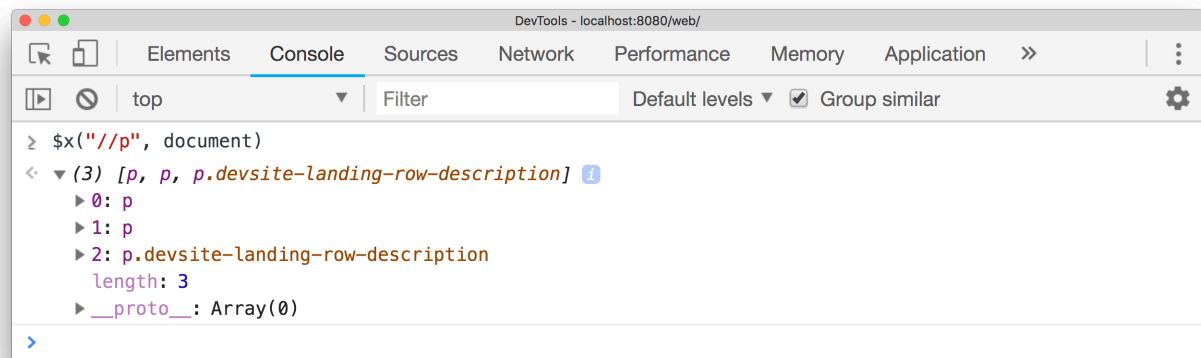


The following example returns all the `<p>` elements that contain `<a>` elements:

```
$x("//p[a]")
```



Similar to the other selector functions, `$x(path)` has an optional second parameter, `startNode`, that specifies an element or Node from which to search for elements.



clear()

`clear()` clears the console of its history.

```
clear();
```



copy(object)

`copy(object)` copies a string representation of the specified object to the clipboard.

```
copy($0);
```

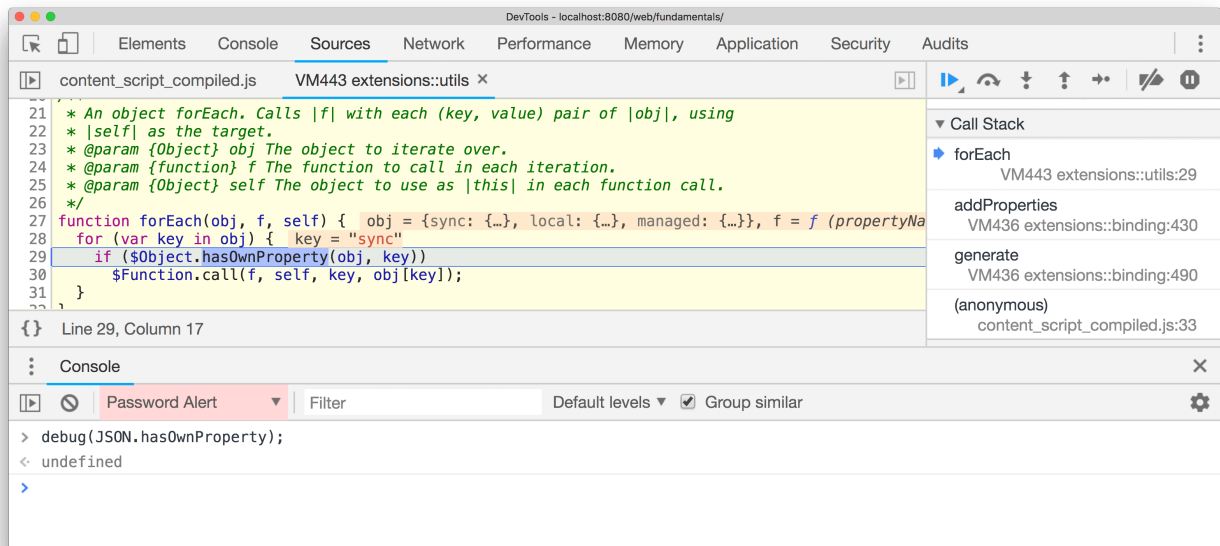


debug(function)

When the specified function is called, the debugger is invoked and breaks inside the function on the Sources panel allowing to step through the code and debug it.

```
debug(getData);
```





Use `undebug(fn)` to stop breaking on the function, or use the UI to disable all breakpoints.

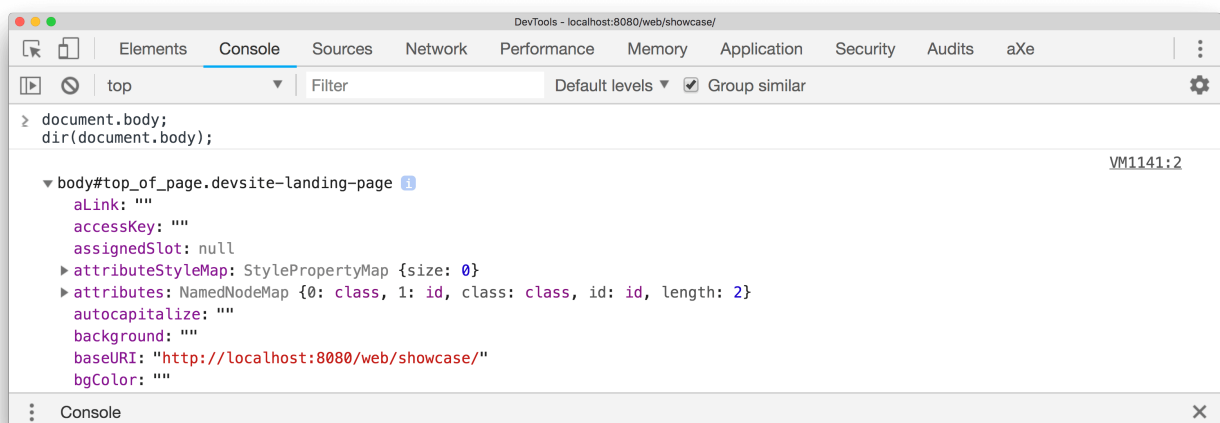
For more information on breakpoints, see [Pause Your Code With Breakpoints](#).

`dir(object)`

`dir(object)` displays an object-style listing of all the specified object's properties. This method is an alias for the Console API's `console.dir()` method.

The following example shows the difference between evaluating `document.body` directly in the command line, and using `dir()` to display the same element:

```
document.body;  
dir(document.body);
```



For more information, see the [`console.dir\(\)`](#) entry in the Console API.

`dirxml(object)`

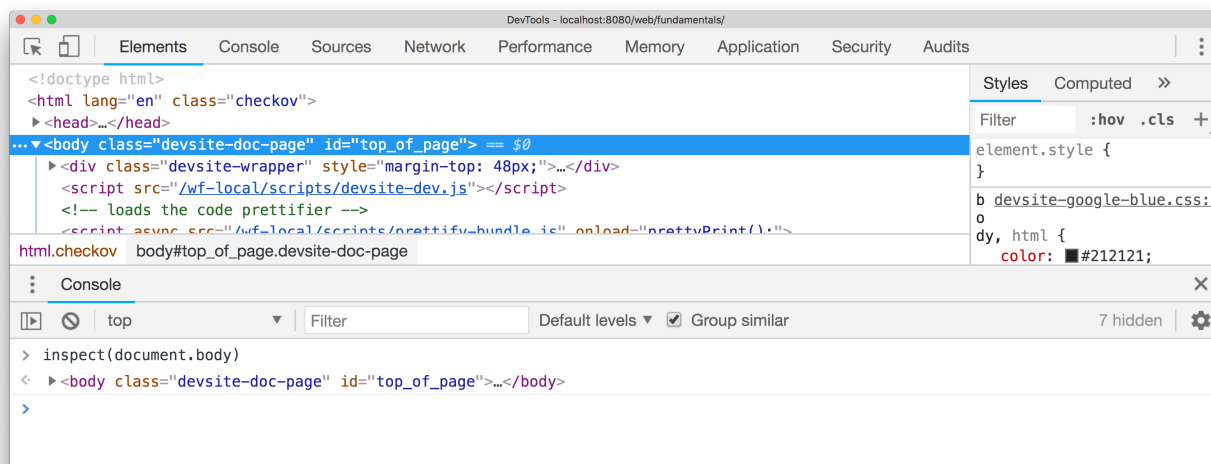
`dirxml(object)` prints an XML representation of the specified object, as seen in the Elements tab. This method is equivalent to the [`console.dirxml\(\)`](#) method.

`inspect(object/function)`

`inspect(object/function)` opens and selects the specified element or object in the appropriate panel: either the Elements panel for DOM elements or the Profiles panel for JavaScript heap objects.

The following example opens the `document.body` in the Elements panel:

```
inspect(document.body);
```



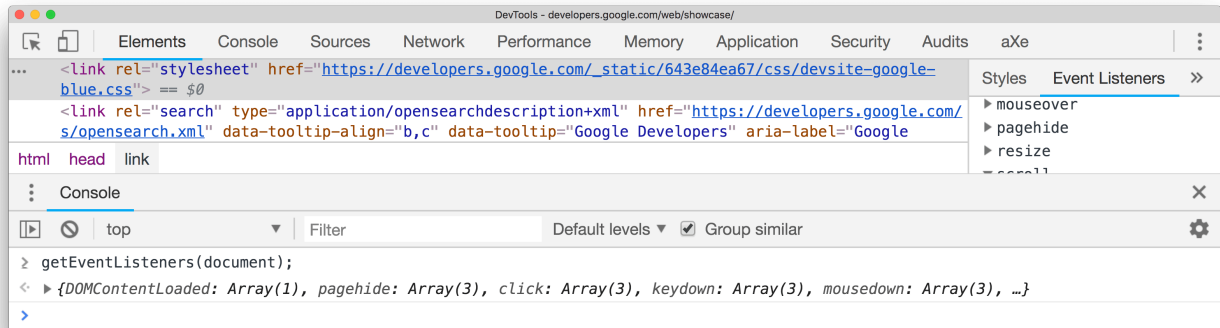
When passing a function to `inspect`, the function opens the document up in the Sources panel for you to inspect.

`getEventListeners(object)`

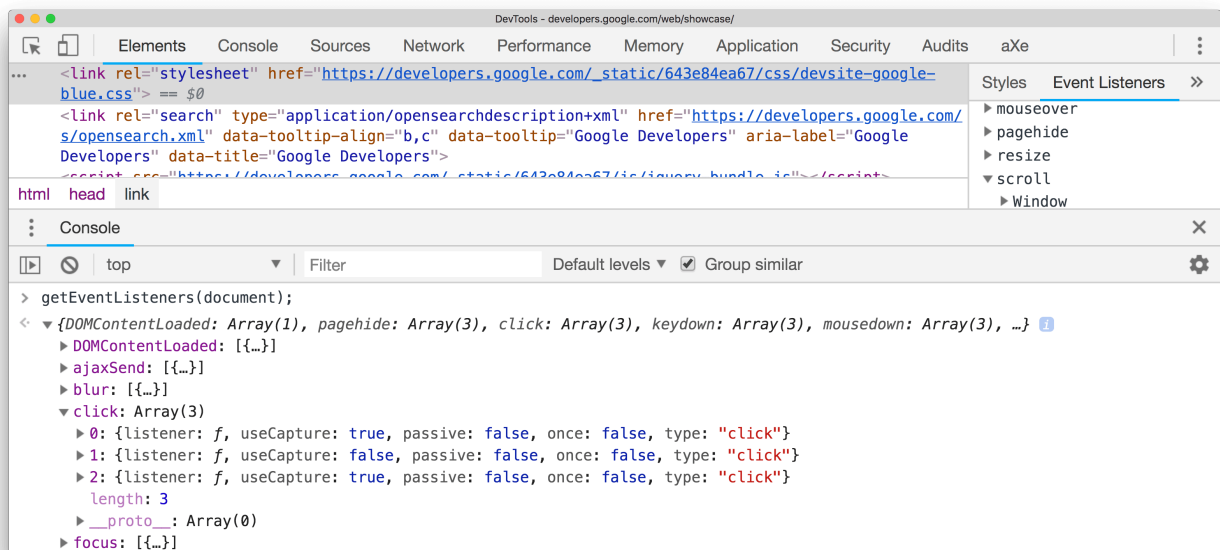
`getEventListeners(object)` returns the event listeners registered on the specified object. The return value is an object that contains an array for each registered event type (`click` or `keydown`, for example). The members of each array are objects that describe the listener

registered for each type. For example, the following lists all the event listeners registered on the document object:

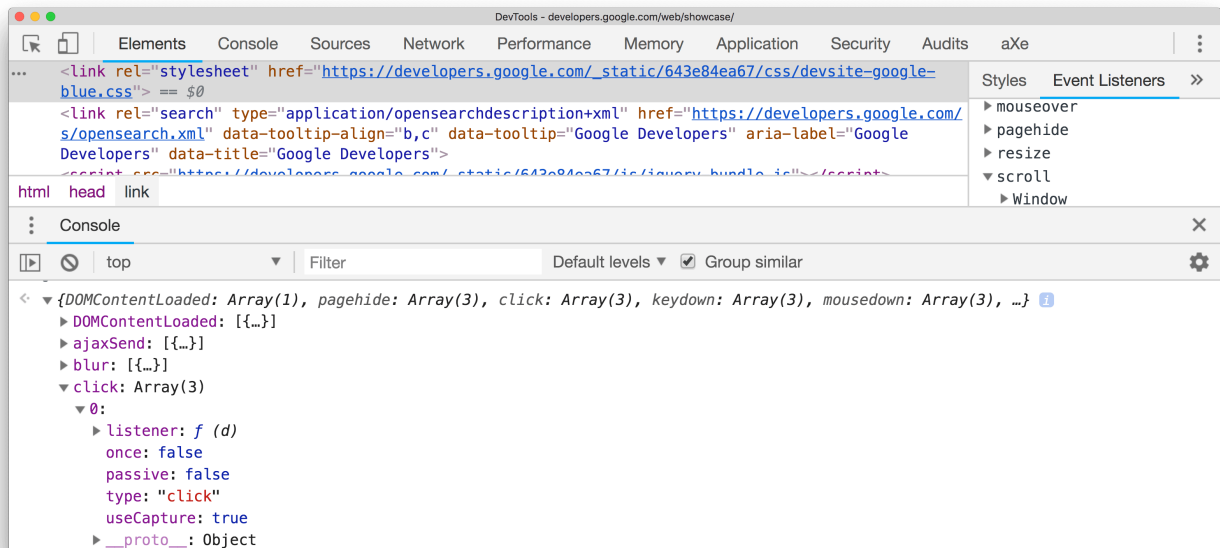
```
getEventListeners(document);
```



If more than one listener is registered on the specified object, then the array contains a member for each listener. In the following example, there are two event listeners registered on the document element for the `click` event:



You can further expand each of these objects to explore their properties:



keys(object)

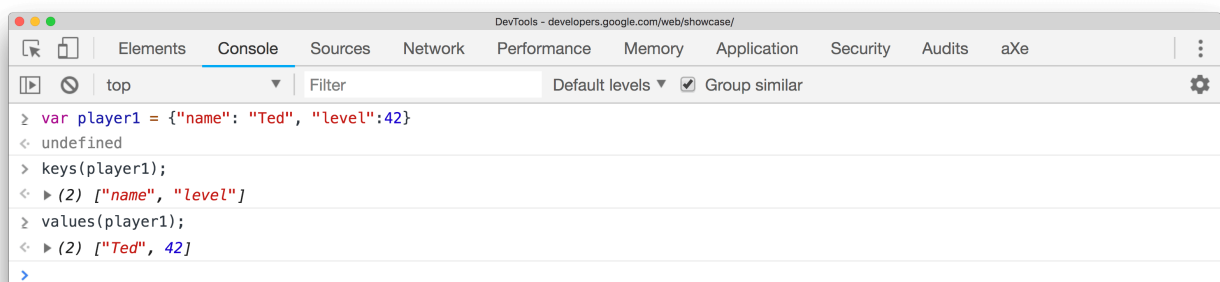
`keys(object)` returns an array containing the names of the properties belonging to the specified object. To get the associated values of the same properties, use `values()`.

For example, suppose your application defined the following object:

```
var player1 = { "name": "Ted", "level": 42 }
```



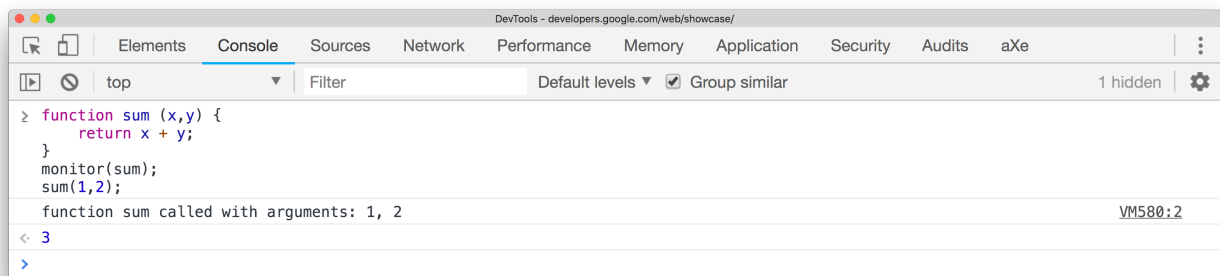
Assuming `player1` was defined in the global namespace (for simplicity), typing `keys(player1)` and `values(player1)` in the console results in the following:



monitor(function)

When the function specified is called, a message is logged to the console that indicates the function name along with the arguments that are passed to the function when it was called.

```
function sum(x, y) {  
    return x + y;  
}  
monitor(sum);
```



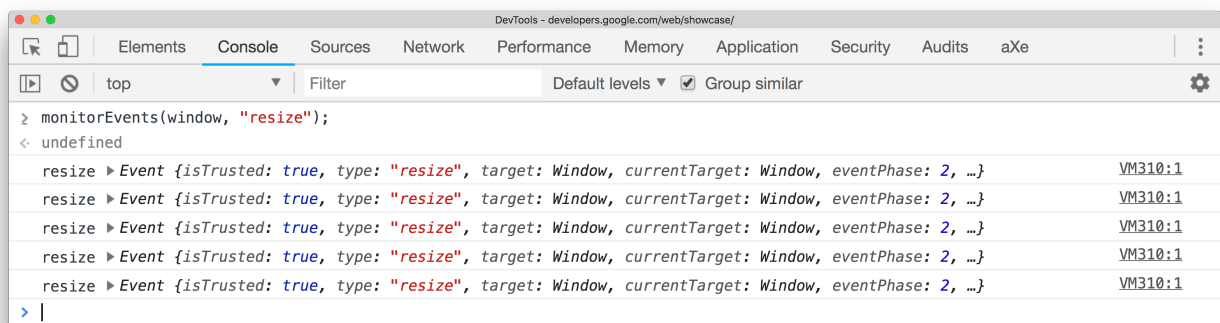
Use `unmonitor(function)` to cease monitoring.

monitorEvents(object[, events])

When one of the specified events occurs on the specified object, the Event object is logged to the console. You can specify a single event to monitor, an array of events, or one of the generic events "types" mapped to a predefined collection of events. See examples below.

The following monitors all resize events on the window object.

```
monitorEvents(window, "resize");
```



The following defines an array to monitor both "resize" and "scroll" events on the window object:

```
monitorEvents(window, ["resize", "scroll"])
```



You can also specify one of the available event "types", strings that map to predefined sets of events. The table below lists the available event types and their associated event mappings:

Event type & Corresponding mapped events

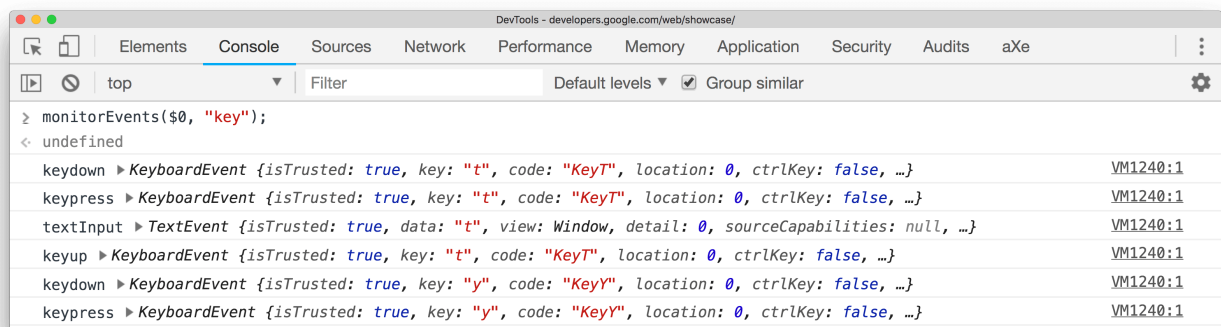
mouse	"mousedown", "mouseup", "click", "dblclick", "mousemove", "mouseover", "mouseout", "mousewheel"
key	"keydown", "keyup", "keypress", "textInput"
touch	"touchstart", "touchmove", "touchend", "touchcancel"
control	"resize", "scroll", "zoom", "focus", "blur", "select", "change", "submit", "reset"

For example, the following uses the "key" event type all corresponding key events on an input text field currently selected in the Elements panel.

```
monitorEvents($0, "key");
```



Below is sample output after typing a characters in the text field:



profile([name]) and profileEnd([name])

`profile()` starts a JavaScript CPU profiling session with an optional name. `profileEnd()` completes the profile and displays the results in the Profile panel. (See also [Speed Up JavaScript Execution](#).)

To start profiling:

```
profile("My profile")
```



To stop profiling and display the results in the Profiles panel:

```
profileEnd("My profile")
```

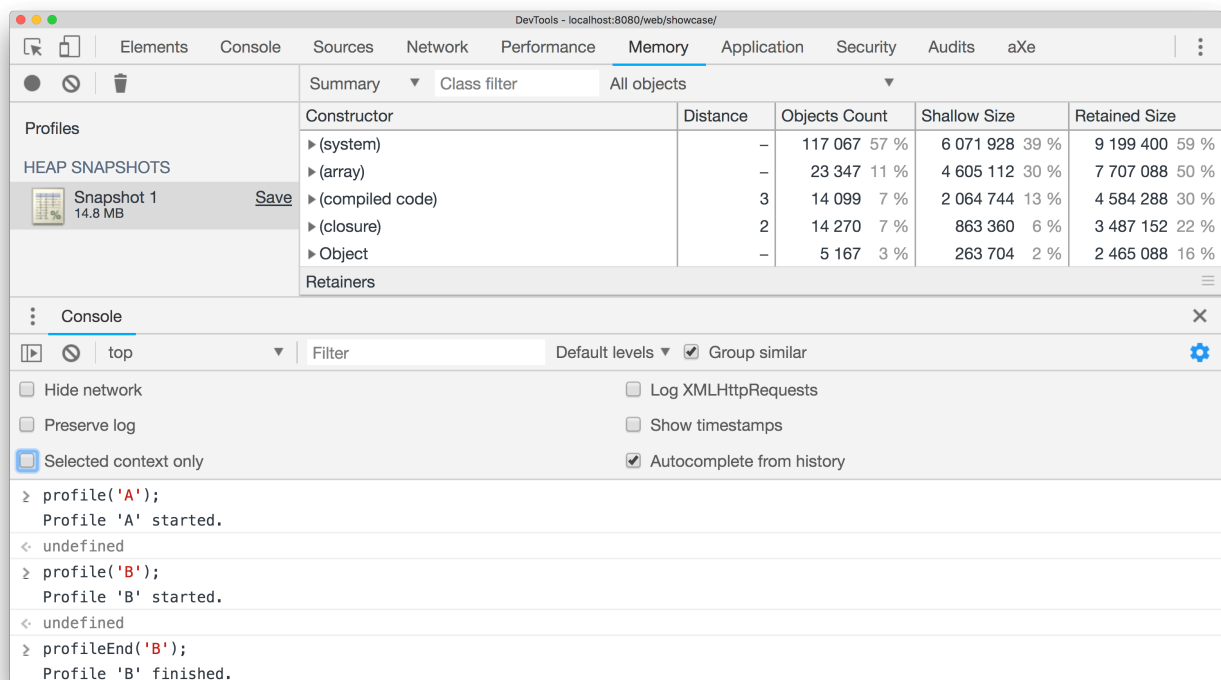


Profiles can also be nested. For example, this will work in any order:

```
profile('A');  
profile('B');  
profileEnd('A');  
profileEnd('B');
```



Result in the profiles panel:

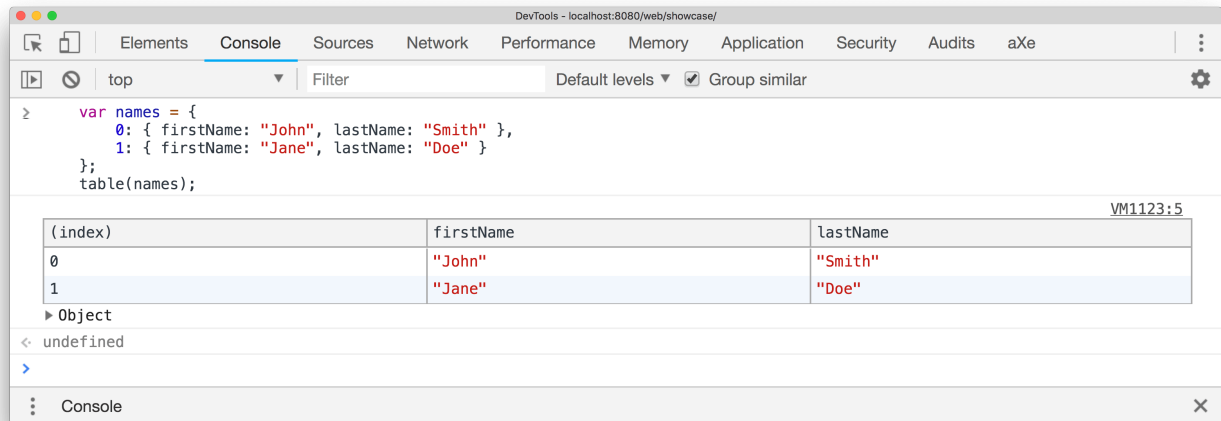


Note: Multiple CPU profiles can operate at once and you aren't required to close them out in creation order.

table(data[, columns])

Log object data with table formatting by passing in a data object in with optional column headings. For example, to display a list of names using a table in the console, you would do:

```
var names = {
  0: { firstName: "John", lastName: "Smith" },
  1: { firstName: "Jane", lastName: "Doe" }
};
table(names);
```



undebug(function)

`undebug(function)` stops the debugging of the specified function so that when the function is called, the debugger is no longer invoked.

```
undebug(getData);
```



unmonitor(function)

`unmonitor(function)` stops the monitoring of the specified function. This is used in concert with `monitor(fn)`.

```
unmonitor(getData);
```



unmonitorEvents(object[, events])

`unmonitorEvents(object[, events])` stops monitoring events for the specified object and events. For example, the following stops all event monitoring on the window object:

```
unmonitorEvents(window);
```



You can also selectively stop monitoring specific events on an object. For example, the following code starts monitoring all mouse events on the currently selected element, and then stops monitoring "mousemove" events (perhaps to reduce noise in the console output):

```
monitorEvents($0, "mouse");  
unmonitorEvents($0, "mousemove");
```



values(object)

`values(object)` returns an array containing the values of all properties belonging to the specified object.

```
values(object);
```



Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 6, 2018.