Web Development

COMP 431 / COMP 531

Lecture 3: JavaScript

Instructor: Mack Joyner

Department of Computer Science, Rice University

mjoyner@rice.edu

http://www.clear.rice.edu/comp431

Recap

- Administration
- HTML
- HTML and forms

Homework Assignment 1
(Simple Page)
Due **THURSDAY** by 11:59PM

In the beginning…

Netscape vs Microsoft

- 1995 – Sun’s Java just hit the scene

  Netscape desired a light-weight analog, similar to VB
- Sept 95 – Codenamed Mocha, LiveScript shipped with Navigator 2.0

  JavaScript appeared in 2.0B3

  (About the same time Java applets added to Navigator)
- July 1996 – IE had JScript, a port of JavaScript which included CSS
- Nov 1996 – Netscape submits for standardization
- June 1997 – ECMAScript as ECMA-262 specification
- June 1998 – ECMAScript v2 and ISO/IEC-16262 standardization
- June 2015 – ECMAScript v6

What *is* JavaScript?

- JavaScript = Java + Scheme + Self + Perl + …

- Single-threaded client-side scripting language with C-like syntax

- Semi-colons are optional…
  - Interpreters perform Automatic Semicolon Insertion
  - Watchout for unintended run-on statements
    - Generally solved by use of a semi-colon
    - Semi-colons are *statement separators* not *terminators*

- No requirements on organization
  - Functions, "objects," and modules can all be defined in the same file

What *is* JavaScript?

- Dynamically typed
- Prototype-based
- Delegatory
- Functional
- Variadic functions
- Engine evaluated script

What *is* JavaScript?

- Dynamically typed
- Prototype-based
- Delegatory
- Functional
- Variadic functions
- Engine evaluated script

- Types are associated with values not variables.
- We can use duck typing

```
> const a = "foo"
> let b = 5, c = 6
> a + b + c
"foo56"
> b = a + (b + c)
"foo11"
> b.doSomething()
```

What *is* JavaScript?

- Dynamically typed
- Prototype-based
- Delegatory
- Functional
- Variadic functions
- Engine evaluated script

- Object-oriented
- Inheritance is performed via cloning from prototype objects
- Dynamic = runtime prototype reassignment
- Just as powerful as your vanilla object-oriented languages

What *is* JavaScript?

- Dynamically typed
- Prototype-based
- Delegatory
- Functional
- Variadic functions
- Engine evaluated script

- Delegate functionality
- Dispatch to correct implementation by following pointers, i.e., prototype

```
> var parent = { lock:
         function(){ return ″locked″;}}
> var child = { name: "the child" }
> child.lock()
Uncaught TypeError: child.lock is not
a function
> child.__proto__ = parent
> child.lock()
"locked"
```

What *is* JavaScript?

- Dynamically typed
- Prototype-based
- Delegatory
- Functional
- Variadic functions
- Engine evaluated script

- Functions are "first-class" citizens
- In fact, a function *is* an object constructor

```
> function Parent() {
        this.name = "the parent"
  }
> new Parent().name
"the parent"
> Parent.name
"Parent"
```

What *is* JavaScript?

- Dynamically typed
- Prototype-based
- Delegatory
- Functional
- Variadic functions
- Engine evaluated script

- Functions can have *any* non-negative number of parameters
- i.e., every function has varargs

```
➢ function average() {
    function sum(a, b) {
      return a + b
    }
    return [].reduce.call(
      arguments, sum, 0) / arguments.length
  }
> average(4,5,6,7,8,9)
6.5
```

What *is* JavaScript?

- Dynamically typed
- Prototype-based
- Delegatory
- Functional
- Variadic functions
- Engine evaluated script

- JavaScript is interpreted at runtime by a JS engine
- Spidermonkey (Firefox)
- Mozilla's Rhino (Java impl)
- Google's V8 (C++ impl)
- …

Execution Order: Where does JS go?

- `<head>`
  - The body doesn't exist yet, so don't go looking for it
  - Execution here is before the loading of the body
- `<body onload="go()">`
  - When the body finishes loading, the onload function is executed
  - This is an example of obtrusive JavaScript
- `<script>` after body
  - No guarantee that the body is loaded

Data Types

- Boolean (true/false)
- null
- undefined
- Number, NaN, Infinity
- String
- Object
  - Array
  - Function

Conversions

- Automatic number to string
- parseInt()
- parseFloat()

- JSON.parse()
- JSON.stringify()

No need to write *new Object*()

JavaScript Objects

- Outside of Primitives, everything is an Object (even Functions)
- *Ex nihilo* literal object creation

  `var a = { foo:“bar” }`

- Field accessing

```
> a
Object {foo: "bar"}
> a.foo
"bar"
> a["foo"]
"bar"
> a.baz = "boo"
"boo"
> a
Object {foo: "bar", baz: "boo"}
```

No need to write *new Array()*

Arrays

- Sparse implementation under the covers
- Array literal

  var a = [24, “bar”, 42]
- Array traversal

  for-in provides index values

  forEach provides the values themselves

```
> sum=0;
> for (var i in a) {
      sum += a[i]
  };
> console.log(sum)
24bar42
```

Some under-utilized powerful array methods:
slice, splice, map, join, shift/pop, unshift/push

Callback function

```
> a.reduce( function(l, r) {
      return l + r
  })
"24bar42"
```

null vs. undefined

undefined

- The value given to anything that has not been defined, e.g., declared but not initialized variable, property, function, etc.

null

- A special value that explicitly indicates a variable has the null value.

- *When might we use or find each?*

typeof vs. instanceof

```
> typeof a == "object"
true
> typeof function foo() {} == "function"
true
> f = function foo() {}
> f instanceof Function
true
```

=, ==, ===

=

- Assignment operator

==, !=

- Equality operator, will coerce

===, !==

- Strict equality, no coercion

```
> 23 == "23"
true
> 23 === "23"
false

> null == undefined
true
> null === undefined
false
```

References

- Primitives are accessed by value
- Objects are accessed by reference

?

Therefore:

```
> var a = { foo: "bar" }
undefined
> var b = a;
undefined
> b.foo = "zzz"
"zzz"
> a.foo
"zzz"
```

```
> function modify(o) {
    o.foo = "zzz"
}
> var a = { foo: "bar" }
> modify(a)

> a
Object {foo: "zzz"}
```

Control structures (exception handling too)

All the usuals:

```
if (condition) { … } else if (condition) { … } else { … }
var a = (condition) ? value : value;
for (initializer ; conditional ; update) { … }
while (conditional) { … }
do { … } while (conditional)
```

break and continue

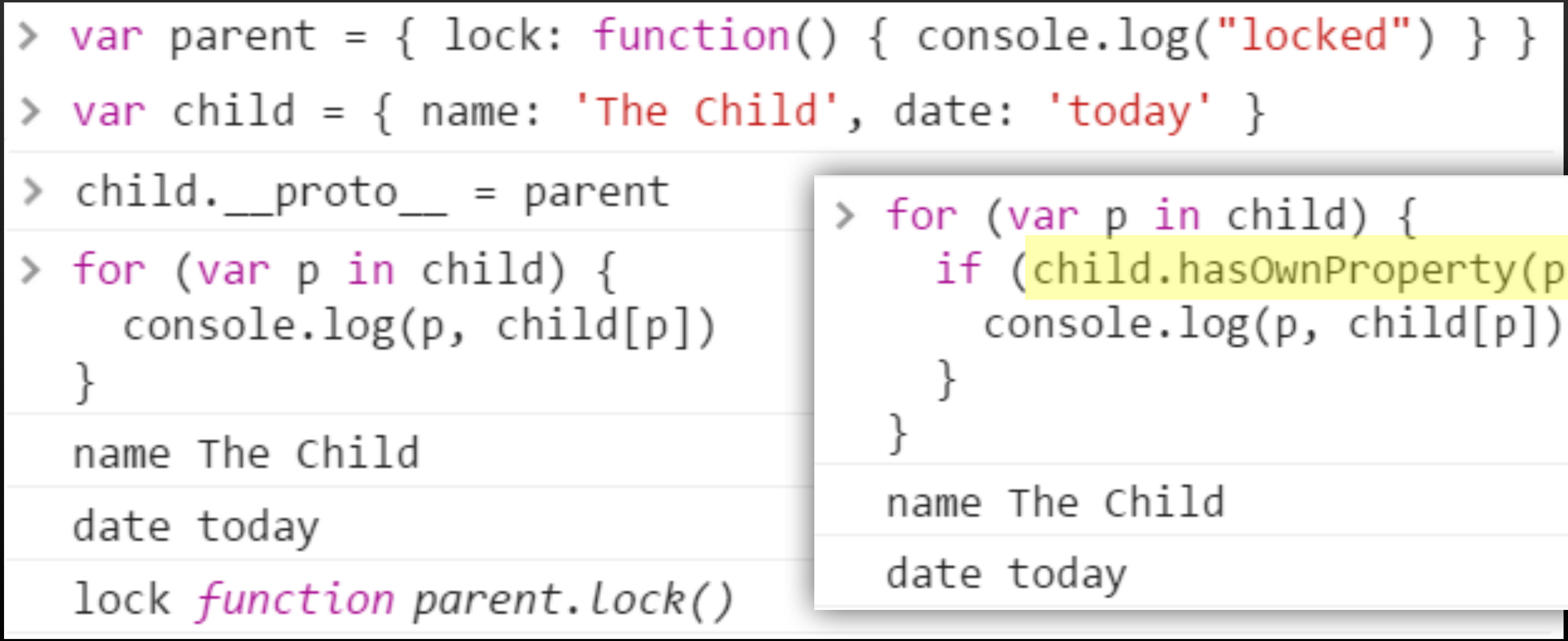
```
switch (value) { case <constant> : …; break; … default: … }
```

compares with ===

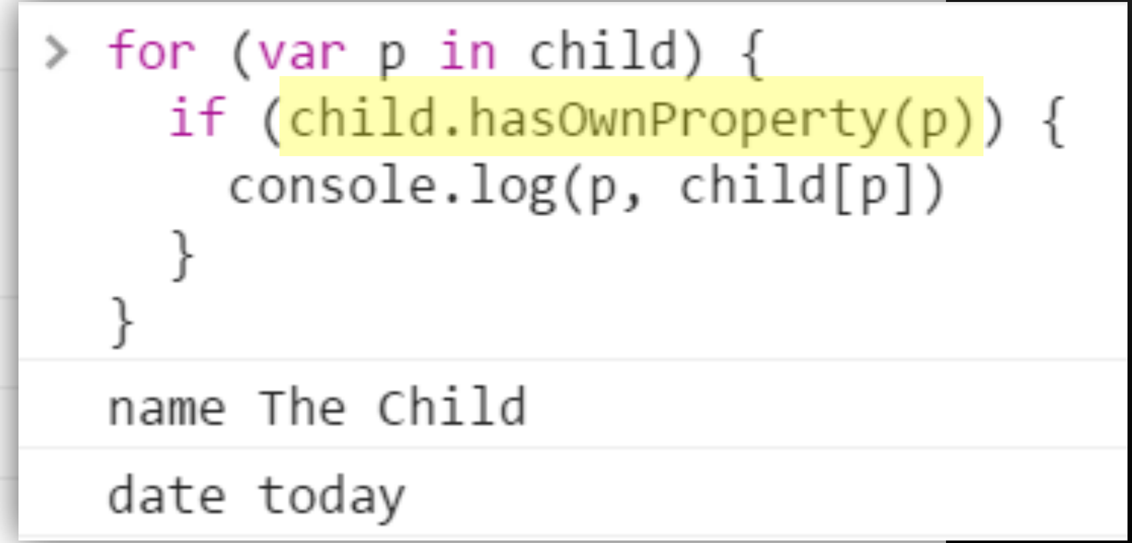
```
try {…;  throw foo; } catch (error) { … } finally { … }
```

for-in

- Iterates over indices for an Array
- Iterates over *all* properties of an Object
  - ALL really means ALL



```
> var parent = { lock: function() { console.log("locked") } }
> var child = { name: 'The Child', date: 'today' }
> child.__proto__ = parent
> for (var p in child) {
    console.log(p, child[p])
  }
  name The Child
  date today
  lock function parent.lock()
```



```
> for (var p in child) {
    if (child.hasOwnProperty(p)) {
      console.log(p, child[p])
    }
  }
  name The Child
  date today
```

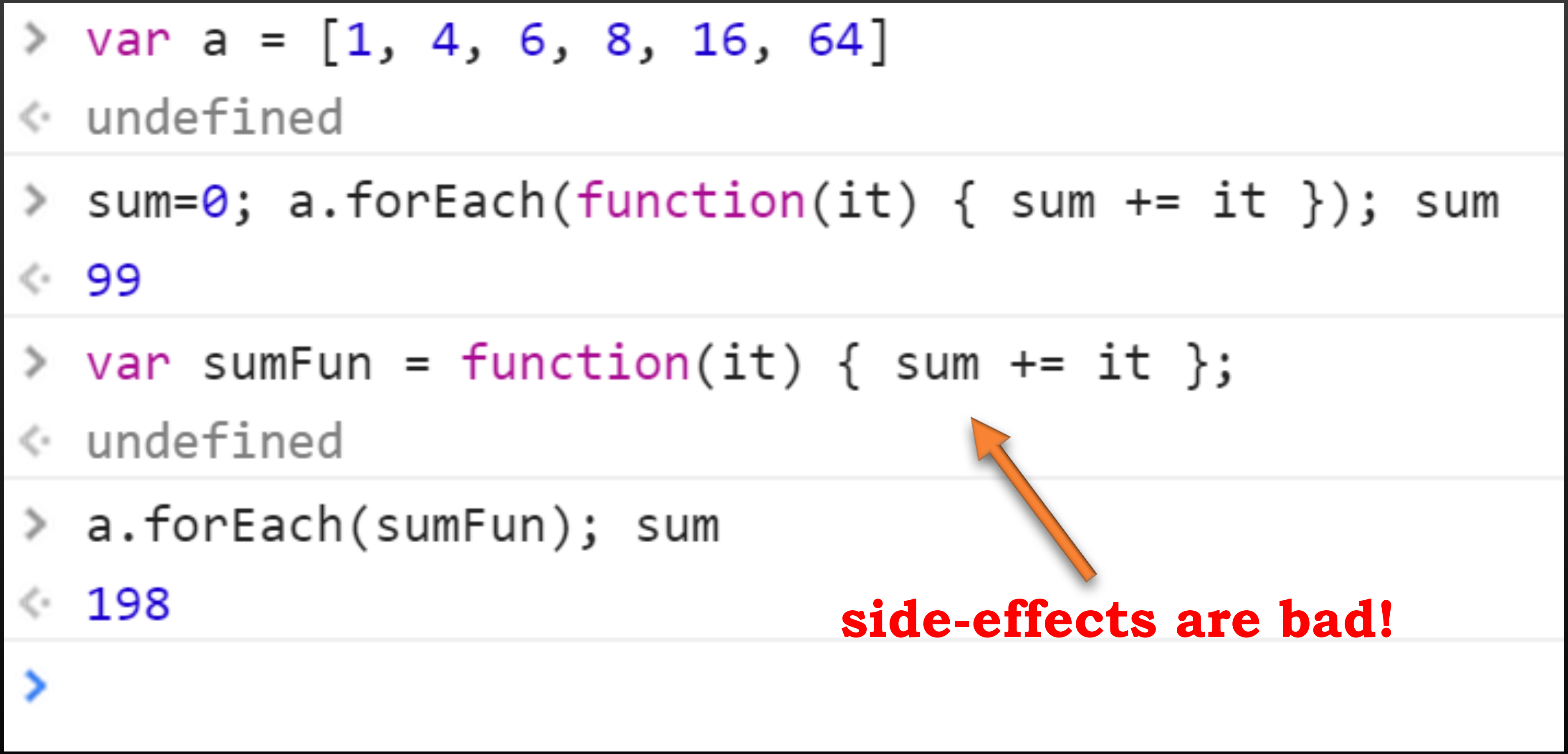
JavaScript *is* functional

- 1950s – Functional “in” (LISP)
- 1970s – Functional “out” (FORTRAN, C, COBOL, later Java, C#)
- 1987 – Haskell (Functional still out)
- 2003 – Scala
- 2005 – F#
- 2010s – Functional is back!

Functional means no side effects!

(among other things…)

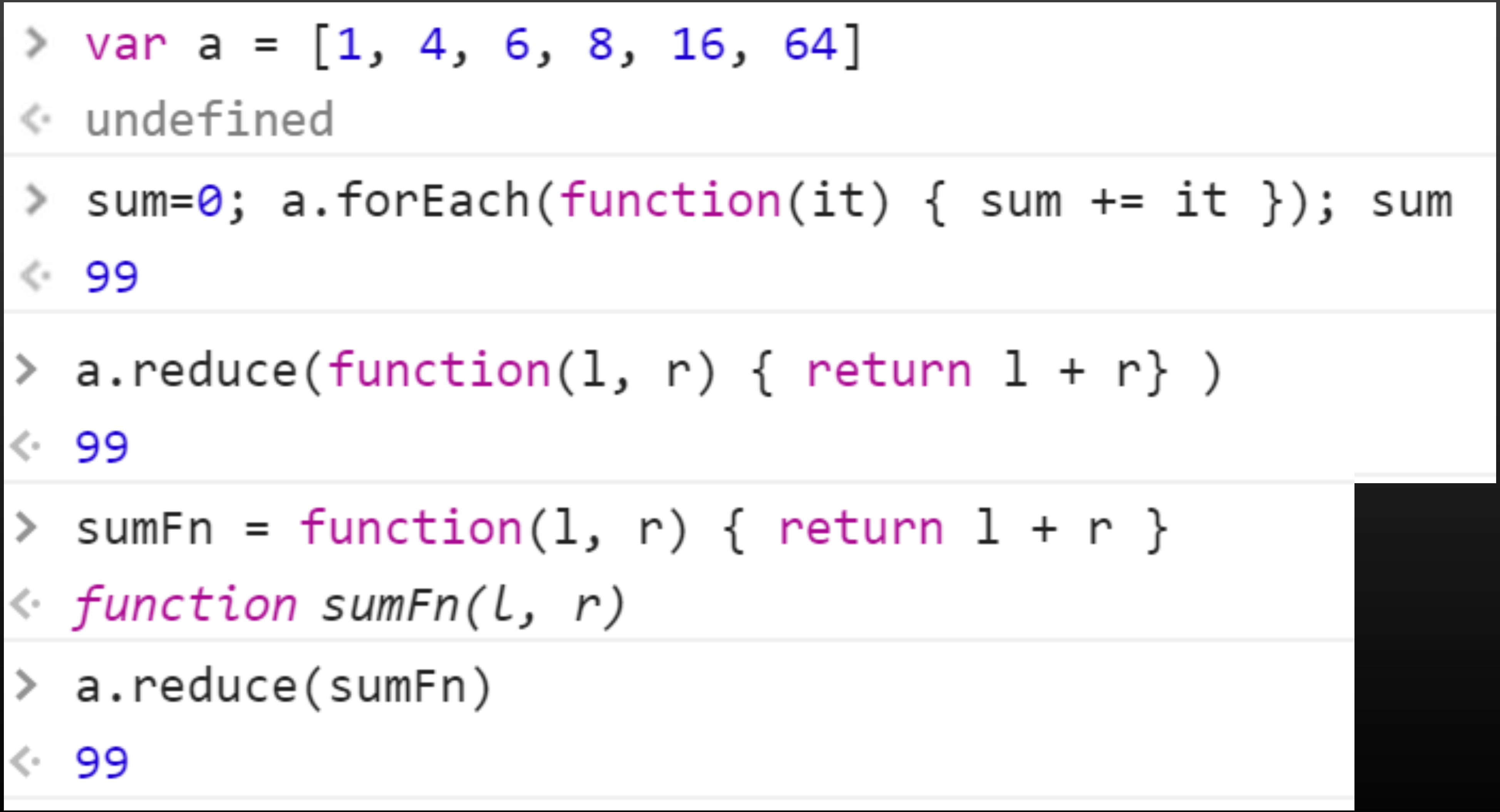
Array functions: forEach



```
> var a = [1, 4, 6, 8, 16, 64]
<· undefined
> sum=0; a.forEach(function(it) { sum += it }); sum
<· 99
> var sumFun = function(it) { sum += it };
<· undefined
> a.forEach(sumFun); sum
<· 198
>
```

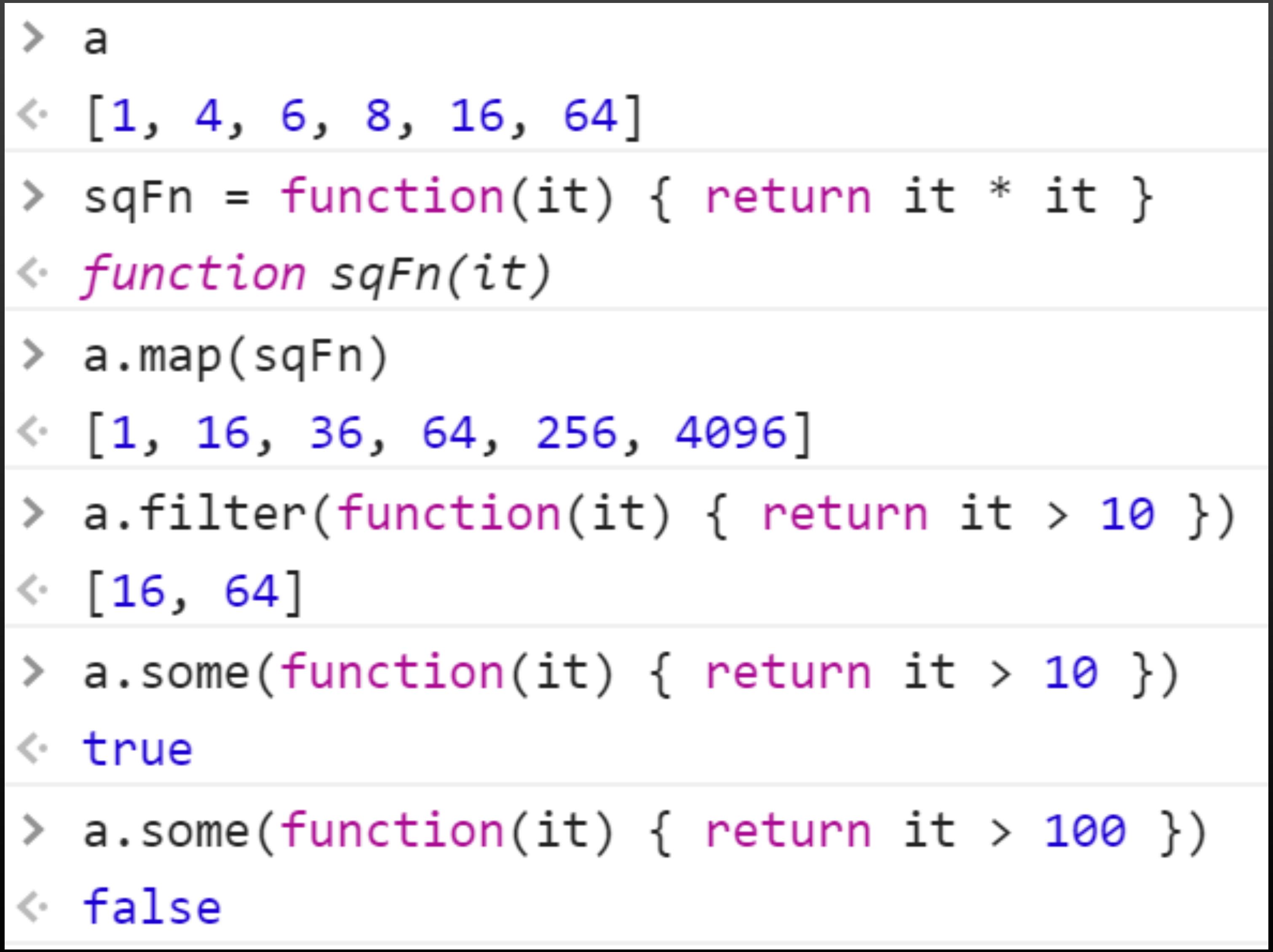
side-effects are bad!

Array functions: reduce



```
> var a = [1, 4, 6, 8, 16, 64]
<· undefined
> sum=0; a.forEach(function(it) { sum += it }); sum
<· 99
> a.reduce(function(l, r) { return l + r} )
<· 99
> sumFn = function(l, r) { return l + r }
<· function sumFn(l, r)
> a.reduce(sumFn)
<· 99
```

Array functions: map, filter, some



```
> a
<· [1, 4, 6, 8, 16, 64]
> sqFn = function(it) { return it * it }
<· function sqFn(it)
> a.map(sqFn)
<· [1, 16, 36, 64, 256, 4096]
> a.filter(function(it) { return it > 10 })
<· [16, 64]
> a.some(function(it) { return it > 10 })
<· true
> a.some(function(it) { return it > 100 })
<· false
```

https://www.destroyallsoftware.com/talks/wat

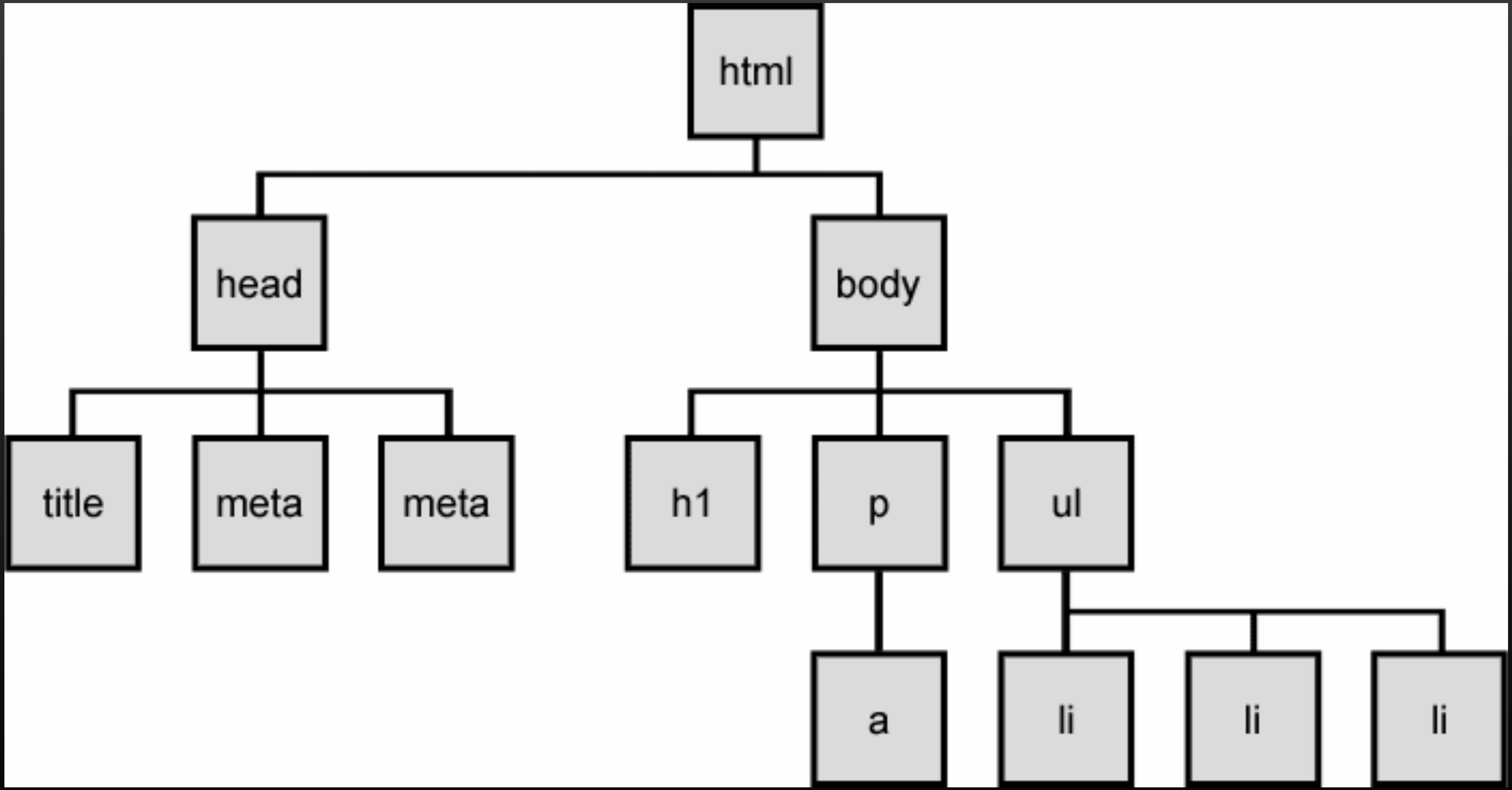
JavaScript is a *programming language* you say?

`Try these, one… at… a… time…`

- `[] + []`
- `[] + {}`
- `{} + []`
- `{} + {}`

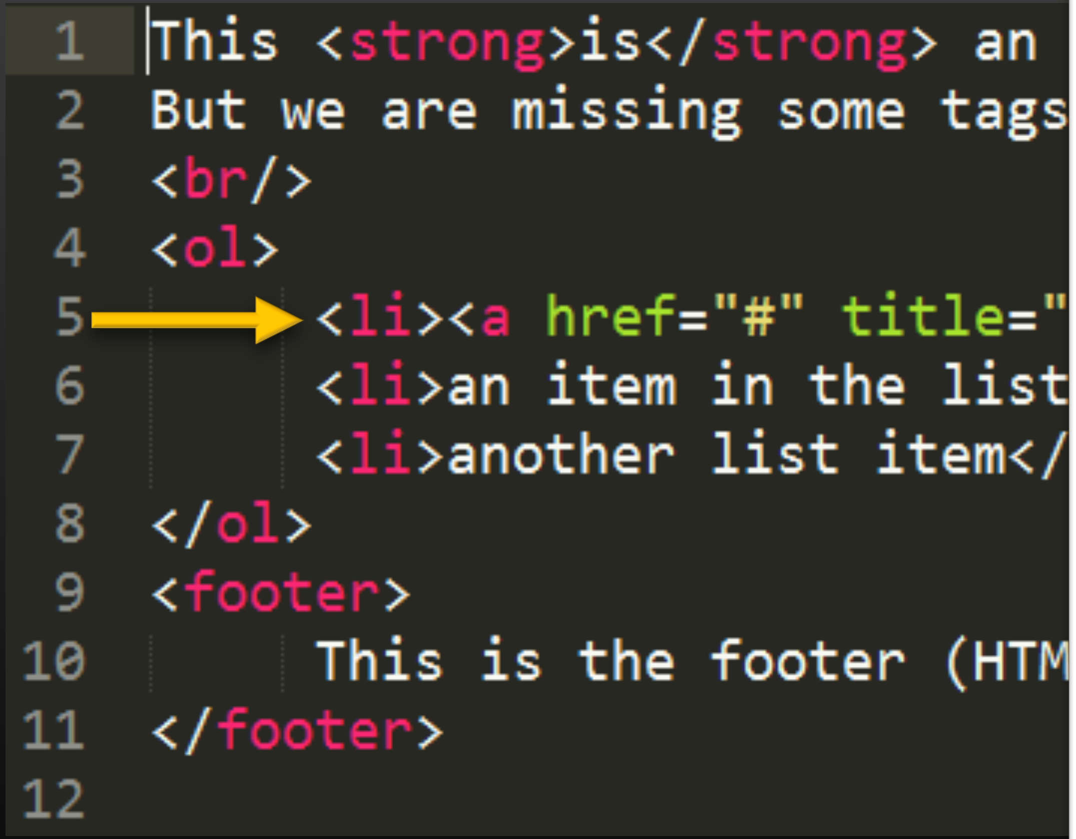
Document Object Model (DOM)

- *document provides a reference to the root of the tree.*



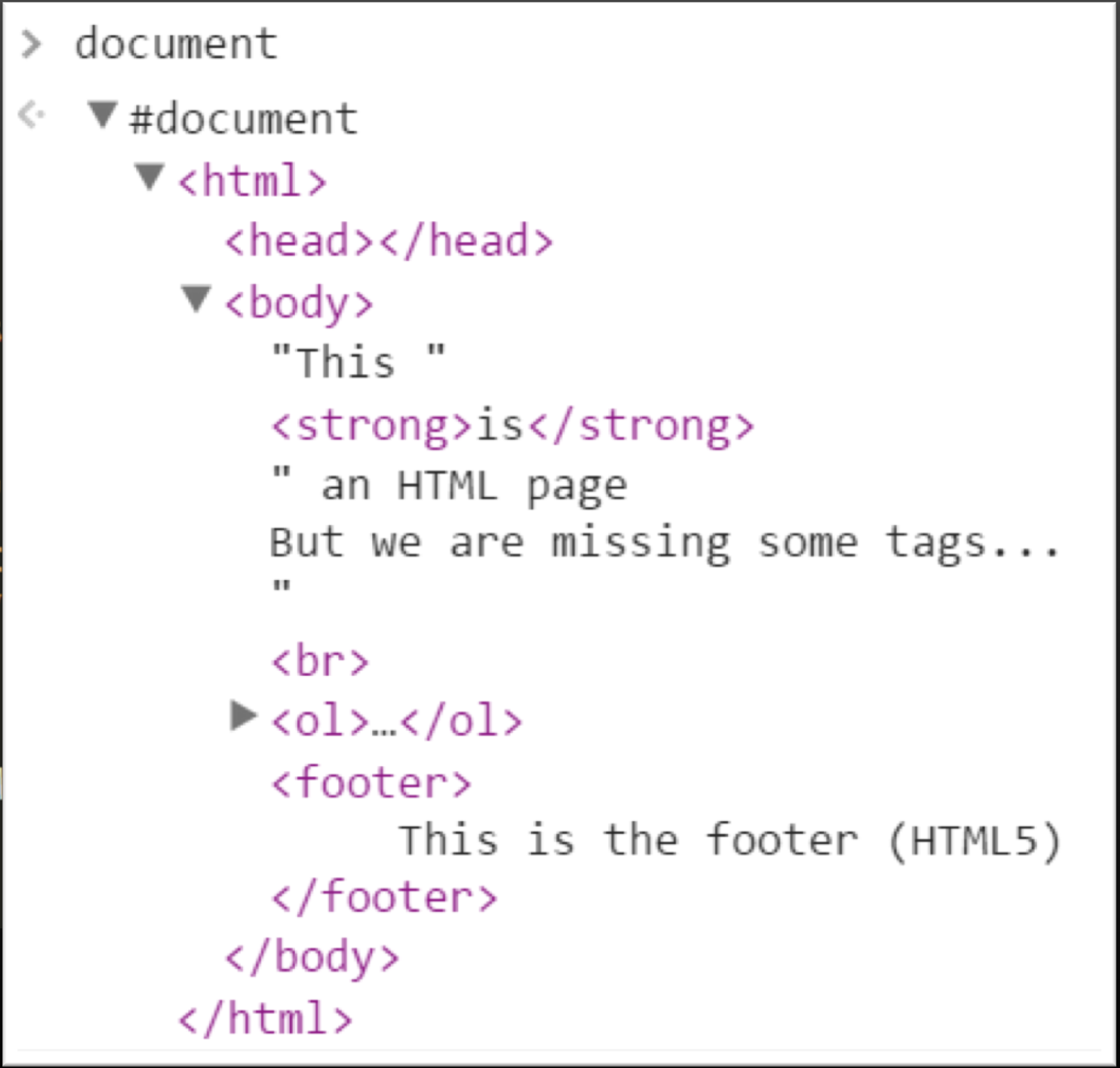
https://developer.mozilla.org/en-US/docs/Web/API/document

The DOM



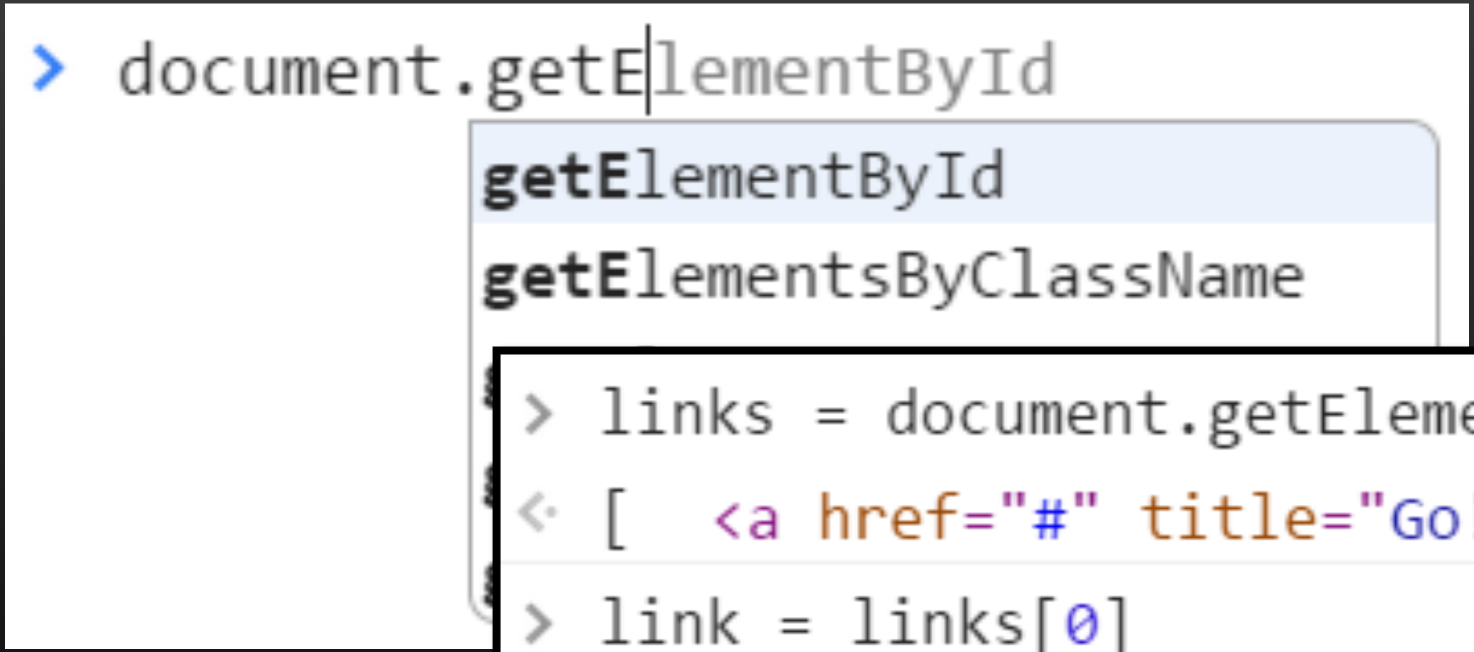
```
This <strong>is</strong> an
But we are missing some tags
<br/>
<ol>
	<li><a href="#" title="
	<li>an item in the list
	<li>another list item</
</ol>
<footer>
	This is the footer (HTM
</footer>

```

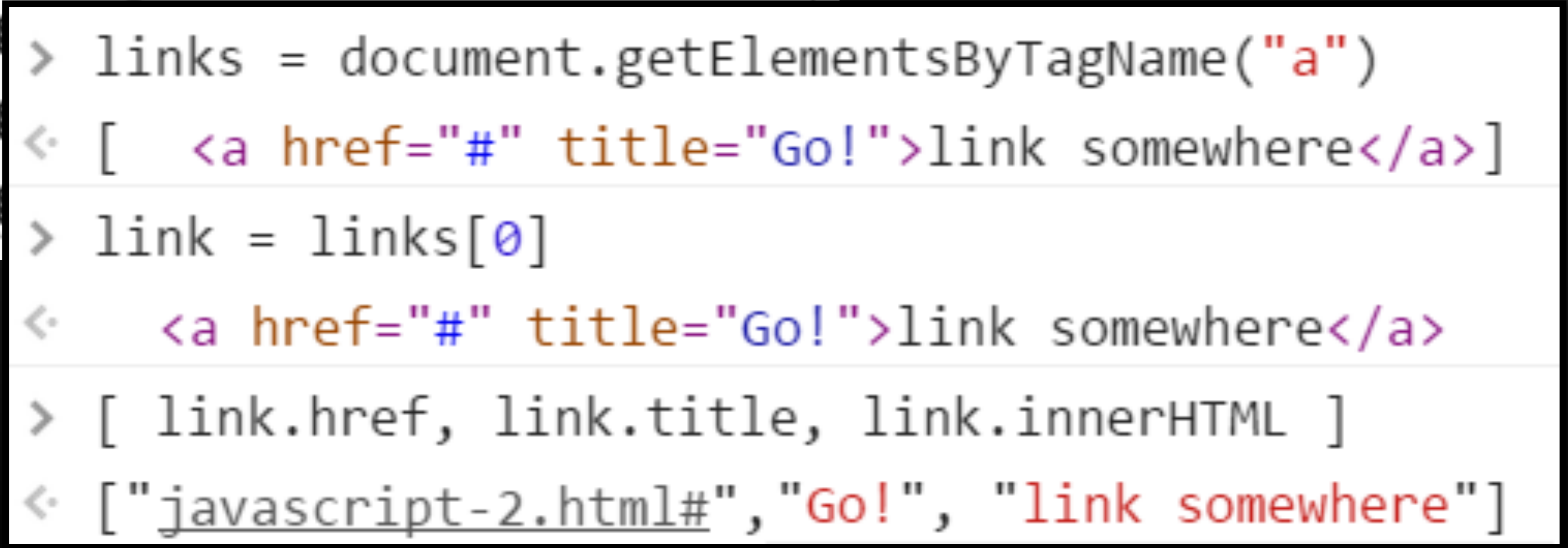


```
> document
<· ▼#document
    ▼<html>
       <head></head>
      ▼<body>
         "This "
         <strong>is</strong>
         " an HTML page
         But we are missing some tags...
         "
         <br>
        ▶<ol>…</ol>
         <footer>
             This is the footer (HTML5)
         </footer>
       </body>
    </html>
```

DOM Access



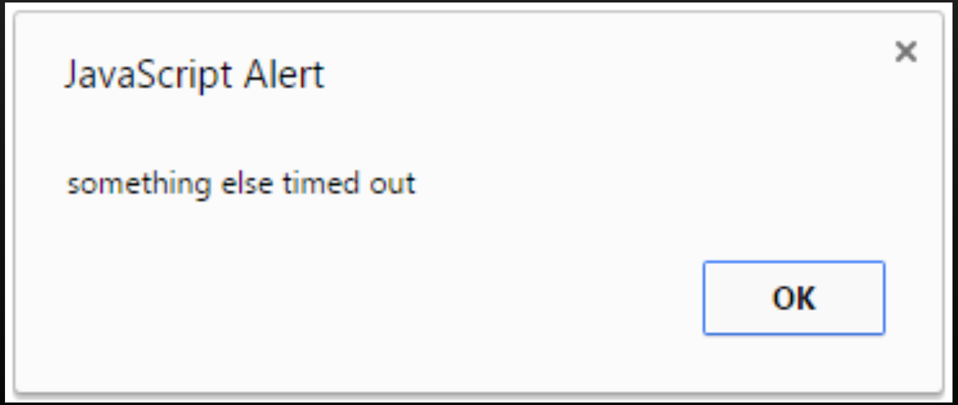
```
> document.getElementById
  getElementById
  getElementsByClassName
```



```
> links = document.getElementsByTagName("a")
<· [  <a href="#" title="Go!">link somewhere</a>]
> link = links[0]
<·   <a href="#" title="Go!">link somewhere</a>
> [ link.href, link.title, link.innerHTML ]
<· ["javascript-2.html#","Go!", "link somewhere"]
```

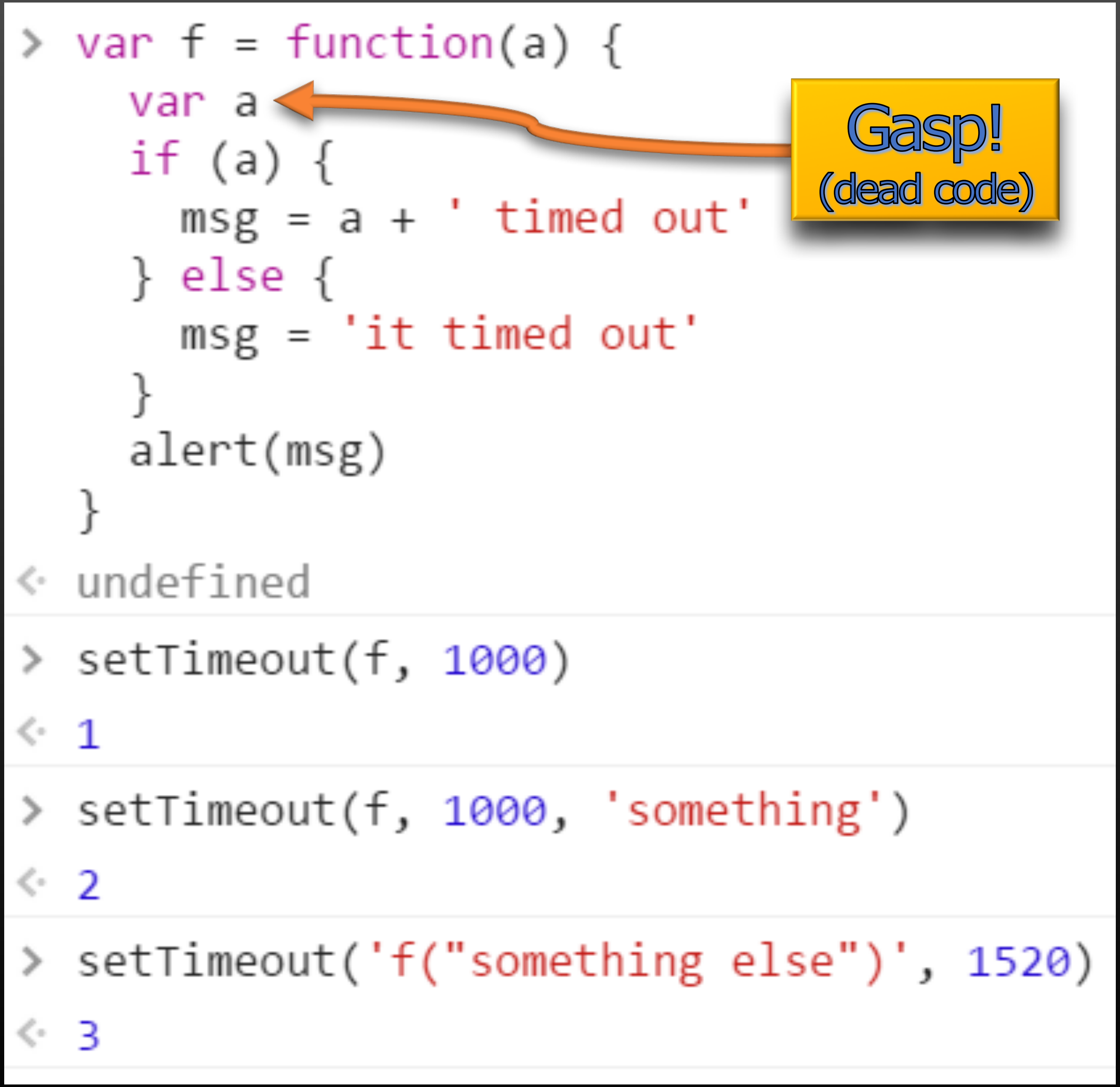
Timeout

- Used when we want something to occur after a certain amount of time.
- Call with either the function name or inline code



JavaScript Alert

something else timed out

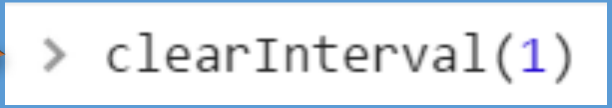


```
> var f = function(a) {
    var a
    if (a) {
      msg = a + ' timed out'
    } else {
      msg = 'it timed out'
    }
    alert(msg)
  }
<· undefined
> setTimeout(f, 1000)
<· 1
> setTimeout(f, 1000, 'something')
<· 2
> setTimeout('f("something else")', 1520)
<· 3
```

Interval

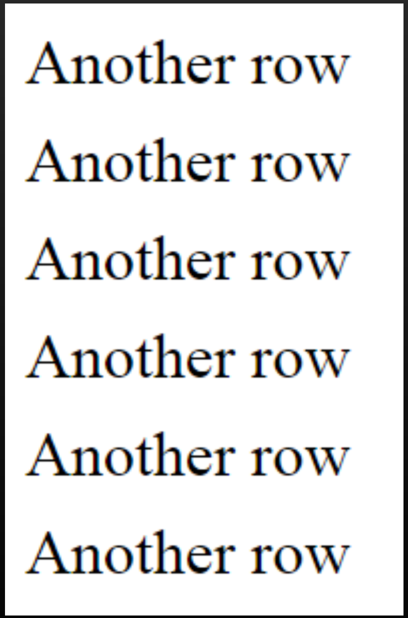
(use a variable)

To stop the interval



```
> clearInterval(1)
```

- Used to create periodic executions
- Best effort execution = The are not *actually* intervals!
    - The time between interval executions is not guaranteed (timeout is better)



Another row

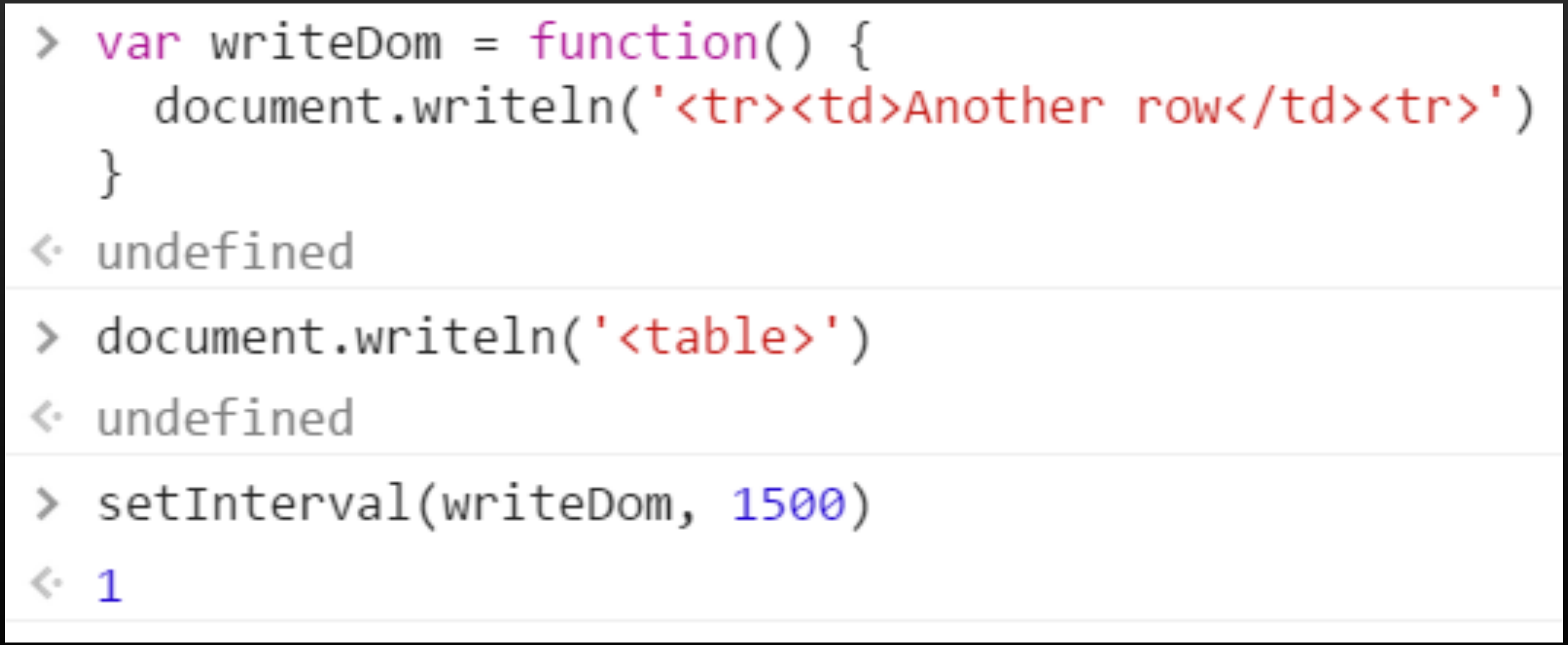
Another row

Another row

Another row

Another row

Another row



```
> var writeDom = function() {
    document.writeln('<tr><td>Another row</td><tr>')
  }
<· undefined
> document.writeln('<table>')
<· undefined
> setInterval(writeDom, 1500)
<· 1
```