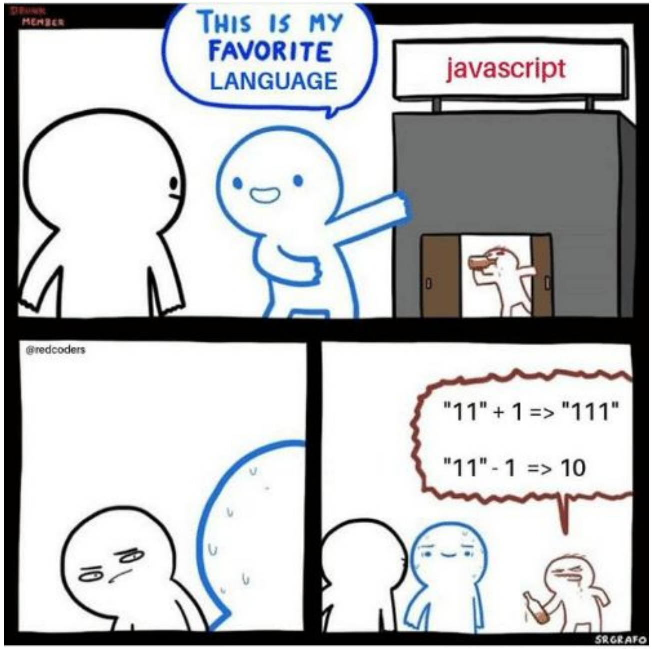
Js tips and tricks



javascript is **bad**

javascript is **good**

- You can change the appearance and behaviour of everything that you see in a webpage
- Extremely easy to make other people access your work
- You can write good code if you know how

What makes js special

- Everything you see in a website can be changed! In appearance and behavior

Starting a project

index.html

```
python3 -m http.server
```

Browser open on
127.0.0.1:8000

Running your code => loading page in the browser

Starting a project

index.html

You can open index.html directly from the browser without having a server running, but **you will encounter problems later**

```
python3 -m http.server
```

Run this in the root folder of your project

Browser open on 127.0.0.1:8000

Starting a project

index.html

python3 -m http.server

Browser open on
127.0.0.1:8000

style.css

script.js

Editor recommendations

All of them are pretty light, very customizable and ready out of the box

Sublime https://www.sublimetext.com/
- lightweight but you can obtain everything you need through plugins
- the only one in this list that is not open source

Vscode https://code.visualstudio.com/ (by Microsoft)
- some additional features like autocompletion are built in

Atom https://atom.io/ (by Github)
- runs on electron (very customizable but heavier than necessary on resources)

Brackets http://brackets.io/ (by Adobe)
- runs on electron too

Not ready out of the box:
Vim
- only recommended if you want to spend a good chunk of time configuring it and learning new shortcuts.

Where do I put my script?

Where do I put my script in an HTML page?

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="UTF-8">
                <title>title</title>
        </head>
        <body>
                <div>content…</div>
                <div>content…</div>
        </body>
</html>
```

http://htmlshell.com/

Ways to run a script

inline

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="UTF-8">
                <title>title</title>
        </head>
        <body>
                <div>content…</div>
                <div>content…</div>
        </body>
        <script>
                … your code ...
        </script>
</html>
```

From another file

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="UTF-8">
                <title>title</title>
                <script src=”./main.js”></script>
        </head>
        <body>
                <div>content…</div>
                <div>content…</div>
        </body>
</html>
```

Ways to run a script

inline

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="UTF-8">
                <title>title</title>
        </head>
        <body>
                <div>content…</div>
                <div>content…</div>
        </body>
        <script>
                … your code ...
        </script>
</html>
```

- does NOT scale
- will make you very confused when your code becomes longer
- only good for fast prototyping

From another file

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="UTF-8">
                <title>title</title>
                <script src="./main.js"></script>
        </head>
        <body>
                <div>content…</div>
                <div>content…</div>
        </body>
</html>
```

- much better, can add as many files as you want and divide your code effectively

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8">
		<title>title</title>
	</head>
	<body>
		<div>content…</div>
		<div>content…</div>
	</body>
</html>
```

Head (document metadata)

Body (content)

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="UTF-8">
                <title>title</title>
                <script src=”./main.js”></script>
                <script src=”./main2.js”></script>
        </head>
        <body>
                <div>content…</div>
                <script src=”./main.js”></script>
                <div>content…</div>
        </body>
        <script src=”./main.js”></script>
</html>
```

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="UTF-8">
                <title>title</title>
                <script src=”./main.js”></script>
                <script src=”./main2.js”></script>
        </head>
        <body>
                <div>content…</div>
                <script src=”./main.js”></script>
                <div>content…</div>
        </body>
        <script src=”./main.js”></script>
</html>
```

In head:

- Executed before everything else
- Can be used to make sure that some resources are accessible before everything else is loaded
- Can’t access DOM objects (because they have not been created yet) unless forced to wait
- Loading of this script is blocking towards the loading of the rest of the resources and scripts

In body:

- Executed after some content and before some other content
- Only useful for very small, localized scripts

After body:

- Able to access every DOM element created in body
- Executed after everything else, won’t block loading of the body

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8">
		<title>title</title>
		<script src=”./main.js”></script>
		<script src=”./main2.js”></script>
	</head>
	<body>
		<div>content…</div>
		<script src=”./main.js”></script>
		<div>content…</div>
	</body>
	<script src=”./main.js”></script>
</html>
```

Workarounds to keep in mind if you have issues with flow control:

Option 1:

```
document.addEventListener(
	'DOMContentLoaded', function() {/*fun code to run*/}
)
```

Use this as a starting point to wait for all content to have loaded in the DOM regardless of where you position your script

The event **DOMContentLoaded** is automatically dispatched by the browser as soon as all the resources are loaded.

Option 2:
Build system / task runner tool set up to do flow control
(out of the scope of this class, google if you want to know more)

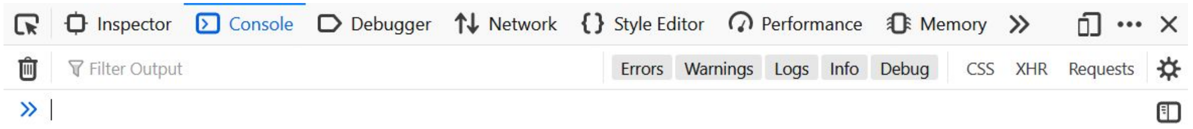
Using the browser console

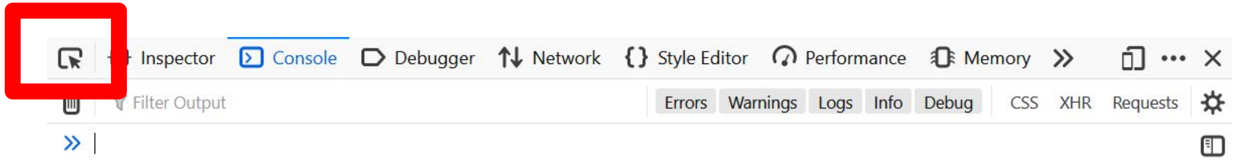
Open the browser console

Ctrl+shift+k on firefox

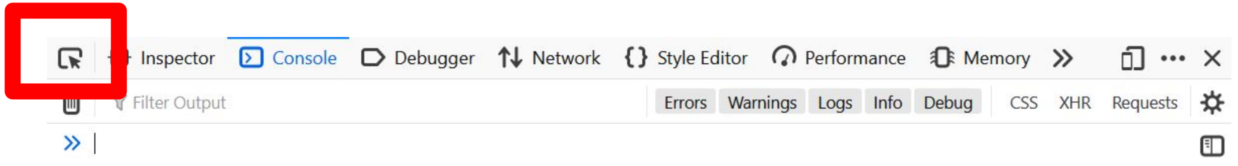
Ctrl+shift+j on chrome

Or click anywhere on the page with your right click and select “inspect element”





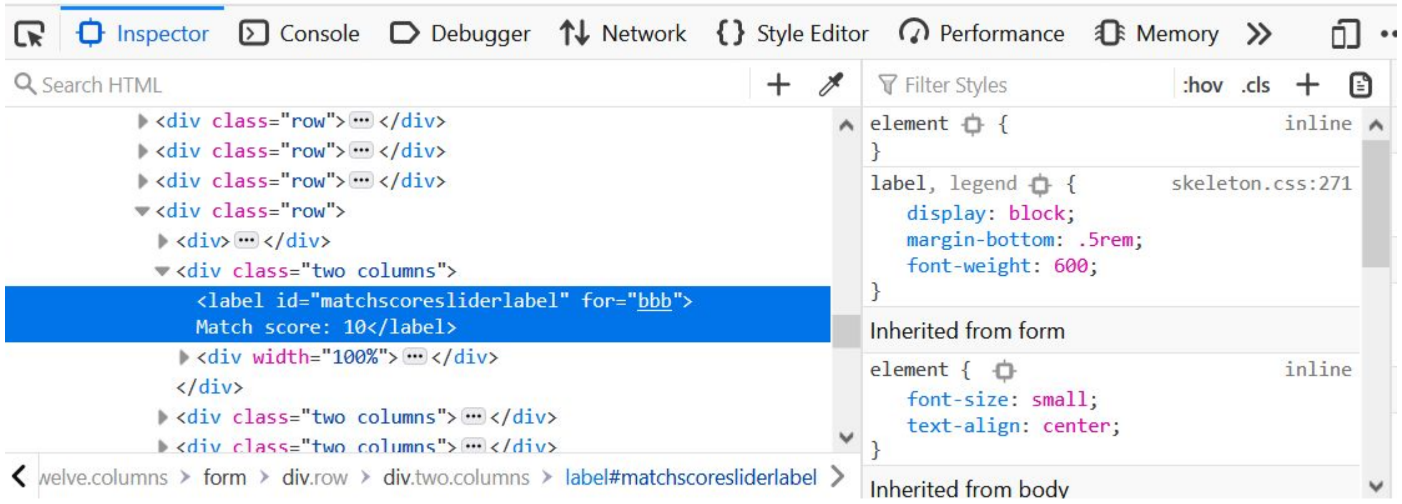
Will allow you to select any element in the page and see its properties, position in the DOM, etc.



Will allow you to select any element in the page and see its properties, position in the DOM, etc.

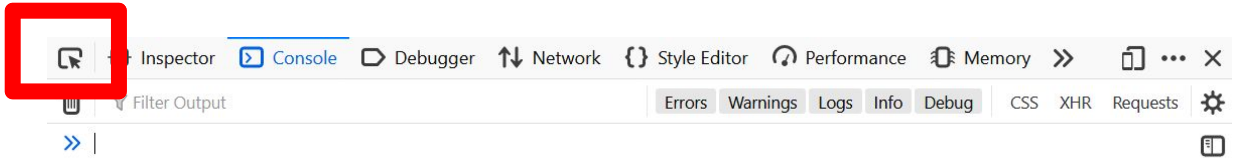
Css associated to selected element

Selected element in the DOM



```
▶ <div class="row">…</div>
▶ <div class="row">…</div>
▶ <div class="row">…</div>
▼ <div class="row">
  ▶ <div>…</div>
  ▼ <div class="two columns">
      <label id="matchscoresliderlabel" for="bbb">
      Match score: 10</label>
    ▶ <div width="100%">…</div>
    </div>
  ▶ <div class="two columns">…</div>
  ▶ <div class="two columns">…</div>
```

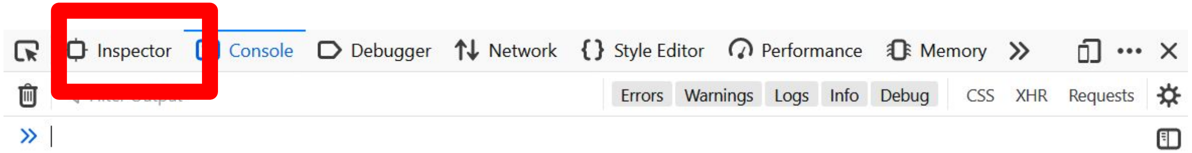
```
element {                          inline
}
label, legend {           skeleton.css:271
    display: block;
    margin-bottom: .5rem;
    font-weight: 600;
}
Inherited from form
element {                          inline
    font-size: small;
    text-align: center;
}
Inherited from body
```



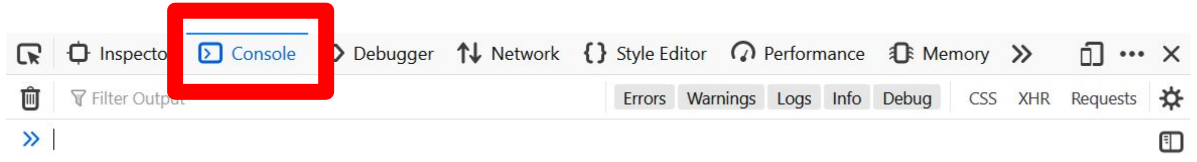
Will allow you to select any element in the page and see its properties, position in the DOM, etc.

Will allow you to answer questions such as:

- What is the id of this element that I am seeing?
- Is this element in the correct position in the DOM?
- What events are associated to this element?
- How would this element look like if I make it red without having to re-run the whole page?

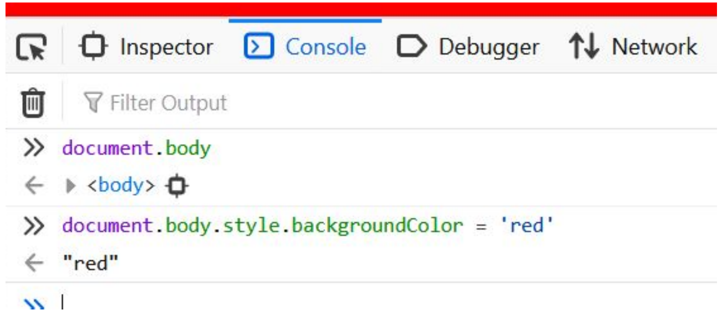


Shows the structure of the page plus css style associated with it



Shows print output
Can run scripts after page is loaded

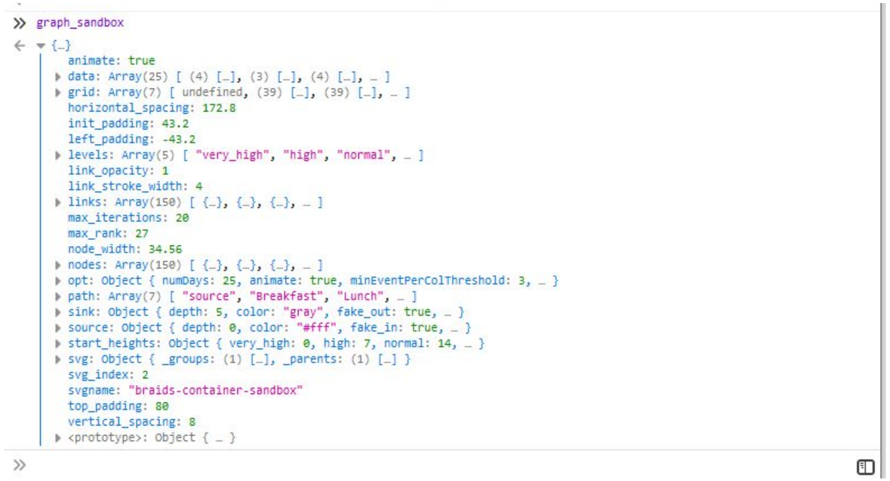
example:



```
>> document.body
← ▸ <body>
>> document.body.style.backgroundColor = 'red'
← "red"
```

Everything is an object

And everything can be printed in the console
If you **print an object in the browser console**, you can **navigate the fields of the object** and the functions associated with it



```
>> graph_sandbox
<- ▼ {…}
       animate: true
     ▶ data: Array(25) [ (4) […], (3) […], (4) […], … ]
     ▶ grid: Array(7) [ undefined, (39) […], (39) […], … ]
       horizontal_spacing: 172.8
       init_padding: 43.2
       left_padding: -43.2
     ▶ levels: Array(5) [ "very_high", "high", "normal", … ]
       link_opacity: 1
       link_stroke_width: 4
     ▶ links: Array(150) [ {…}, {…}, {…}, … ]
       max_iterations: 20
       max_rank: 27
       node_width: 34.56
     ▶ nodes: Array(150) [ {…}, {…}, {…}, … ]
     ▶ opt: Object { numDays: 25, animate: true, minEventPerColThreshold: 3, … }
     ▶ path: Array(7) [ "source", "Breakfast", "Lunch", … ]
     ▶ sink: Object { depth: 5, color: "gray", fake_out: true, … }
     ▶ source: Object { depth: 0, color: "#fff", fake_in: true, … }
     ▶ start_heights: Object { very_high: 0, high: 7, normal: 14, … }
     ▶ svg: Object { _groups: (1) […], _parents: (1) […] }
       svg_index: 2
       svgname: "braids-container-sandbox"
       top_padding: 80
       vertical_spacing: 8
     ▶ <prototype>: Object { … }
>>
```

Note: you can access any DOM element too as javascript objects

Callbacks and events

Callbacks and events

“Event-driven architecture”: the flow of a program is defined by **events**.

Events can be generated by the user or by the browser. Examples of events that you will want to use a callback for:

- user interacts with an element
- loading of a resource is completed
- browser window is resized
- request to some API is returned

  ...

Callbacks and events

Most of the events that you will use are already defined by the browser.

Examples:

- **mouseover**: cursor enters the bounding box of a specified element
- **mouseout**: cursor exits the bounding box of a specified element
- **onclick**: user clicks on specified element
- **onWindowResize**: browser window is resized
- **onDocumentReady**: all resources in document are loaded

You can also define and dispatch your own events

Callbacks and events

Adding an event listener to an item:

```
item.on(‘mouseover’, function(){
        console.log(‘hello’);
})
```

a callback

Events are usually managed using callbacks.
Callbacks are nameless functions that are executed after a condition is verified.

Callbacks and events

Callbacks are not only for events:

```
myArray = [1, 2, 3, 4, 5, 6]
result = myArray.filter( function(a){
        return a%2==0
})
// returns [2, 4, 6]
```

In this case, we use a callback to filter an array, keeping only even numbers

Callbacks and events

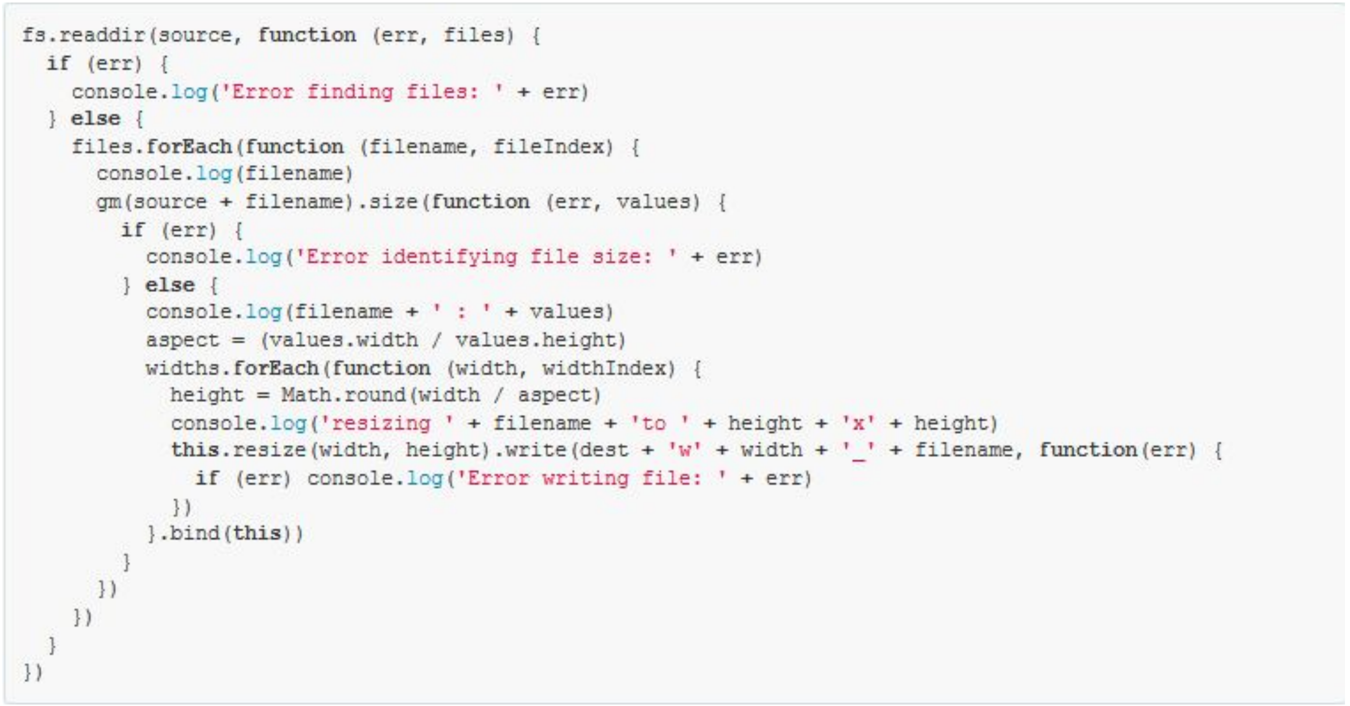
Similar to **lambdas** in python

JS

```
myArray = [1, 2, 3, 4, 5, 6]
result = myArray.filter( function(a){
        return a%2==0
})
// returns [2, 4, 6]
```

Python

```
myArray = [1, 2, 3, 4, 5, 6]
result = list(filter(lambda a: (a%2 == 0), myArray))
// returns [2, 4, 6]
```



```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

http://callbackhell.com/

Ways to declare a variable

- x = 5;	
- **var** x = 5, y = 6, z = 7;	Global
- **let** x = 5;	Scope of the variable is constrained to the scope in which it has been declared.
- **const** x = 5;	Scope limited, x has to be constant.

```
if (true) {

    var foo = 5;

}


console.log(foo);   // 5
```

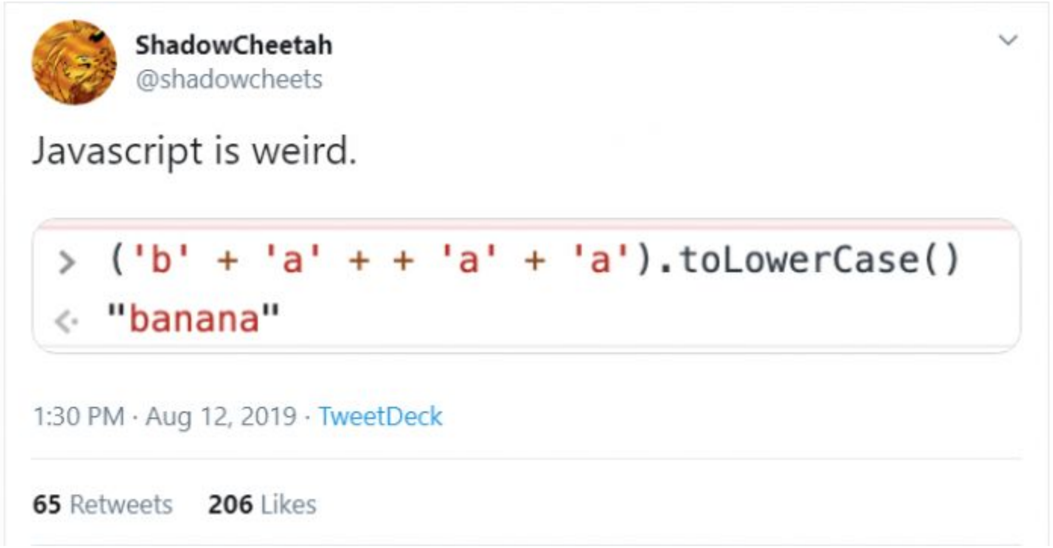
```
if (true) {

    let foo = 5;

}


console.log(foo);   // undefined
```

Always be aware of the data type that you are dealing with



ShadowCheetah
@shadowcheets

Javascript is weird.

```
> ('b' + 'a' + + 'a' + 'a').toLowerCase()
<· "banana"
```

1:30 PM · Aug 12, 2019 · TweetDeck

65 Retweets 206 Likes

https://github.com/denysdovhan/wtfjs

Ways to declare a function

Function declaration

```
function name (params) {
        ...
}
```

Function expression

```
let name = function (params) {
        ...
}
```

Arrow function

```
let name = (params) => {
        ...
}
```

All of these will have almost the same effect

Hoisting: a function will be positioned at the top of the scope and made available at any point of its own scope even before its own declaration

Arrow functions will let you write a lot of fun oneliners:

```
// custom sorting function
[3, 1, 2, 4].sort((a, b) => a < b)
→ [1, 2, 3, 4]

// custom filtering function
[1, 2, 3, 4].filter(a => a%2 == 0)
→ [2, 4]

// sum of all elements in an array
[1, 2, 3, 4].reduce((a, b) => a + b, 0)
→ 10

// sort then filter then sum
[3, 1, 2, 4].sort((a, b) => a < b).filter(a => a%2 == 0).reduce((a, b) => a + b, 0)
→ 6
```

Style guides

Google style guide: https://google.github.io/styleguide/javascriptguide.xml

Airbnb: https://github.com/airbnb/javascript

Standardjs: https://standardjs.com/#the-rules

Idiomatic: https://github.com/rwaldron/idiomatic.js

Linting

Linters force you to write code following some pre-established policies.

Jslint: http://www.jslint.com/

jshint: https://jshint.com/ started as a fork of jslint, customizable

prettier: https://prettier.io/ customizable

Automated code review

one of many tools to check issues in your code:

https://www.codacy.com/

(example)
https://app.codacy.com/app/picorana/sparqling/files?bid=7480002

Thank you!