Lecture 2

More JavaScript + CSS

Frontend Web Development

CSS

Formatting Contexts

MDN Intro to Formatting contexts

Formatting context

Formatting context is an area of a web-page where content is laid out according to some rules. Some examples of formatting contexts are:

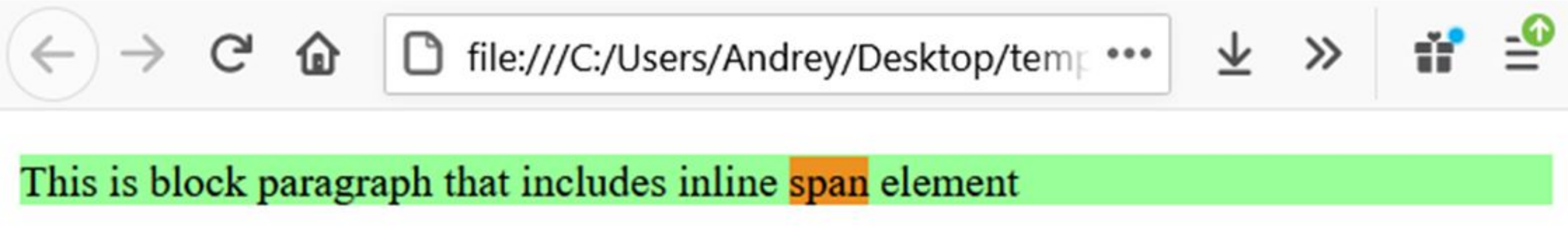
- inline
- block
- inline-block
- table
- flex
- grid

CSS Tricks: When to use inline-block

Inline vs. Block Formatting Contexts

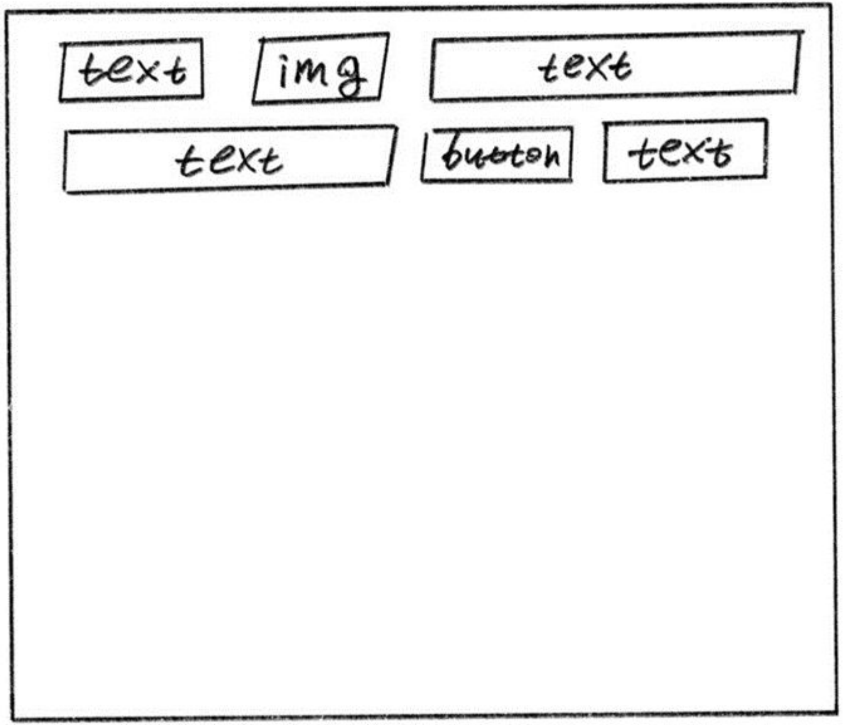
- **inline formatting context**
- Purpose: formats text and everything that is included
- Elements: text, images, buttons, inline-blocks, ets.
- Fills only space that is needed for element
- Examples of inline tags are span, a, img, label, code, etc.

- **block formatting context**
- Purpose: formats blocks of inline content
- Elements: block boxes, float boxes
- Fills full width of container
- Examples of block elements are div, p, ul, form, hr, h1, etc.

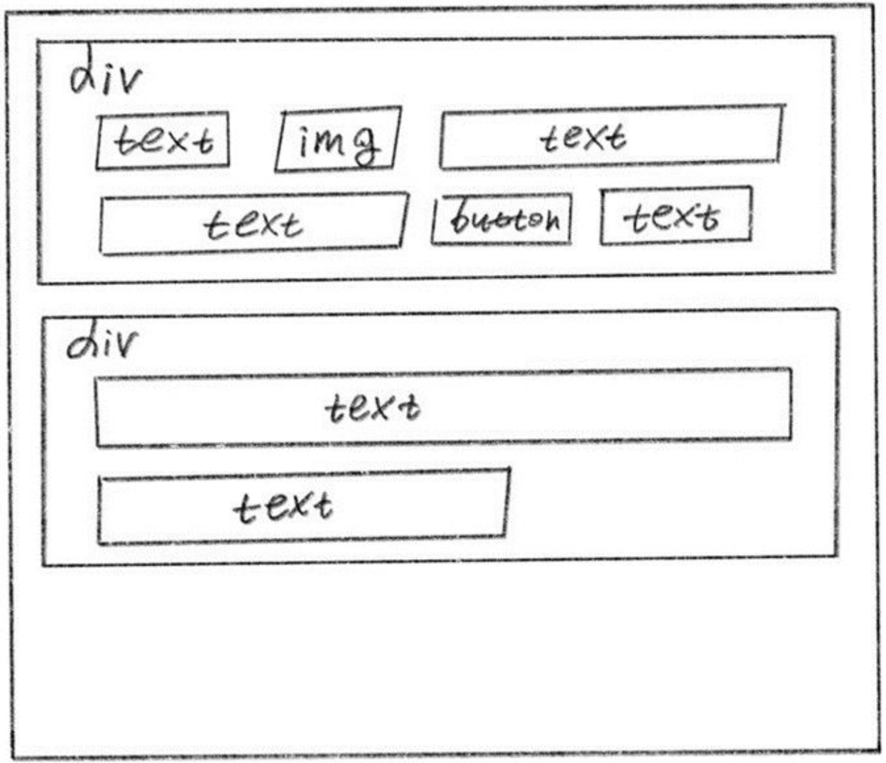


This is block paragraph that includes inline span element

Inline Formatting Context

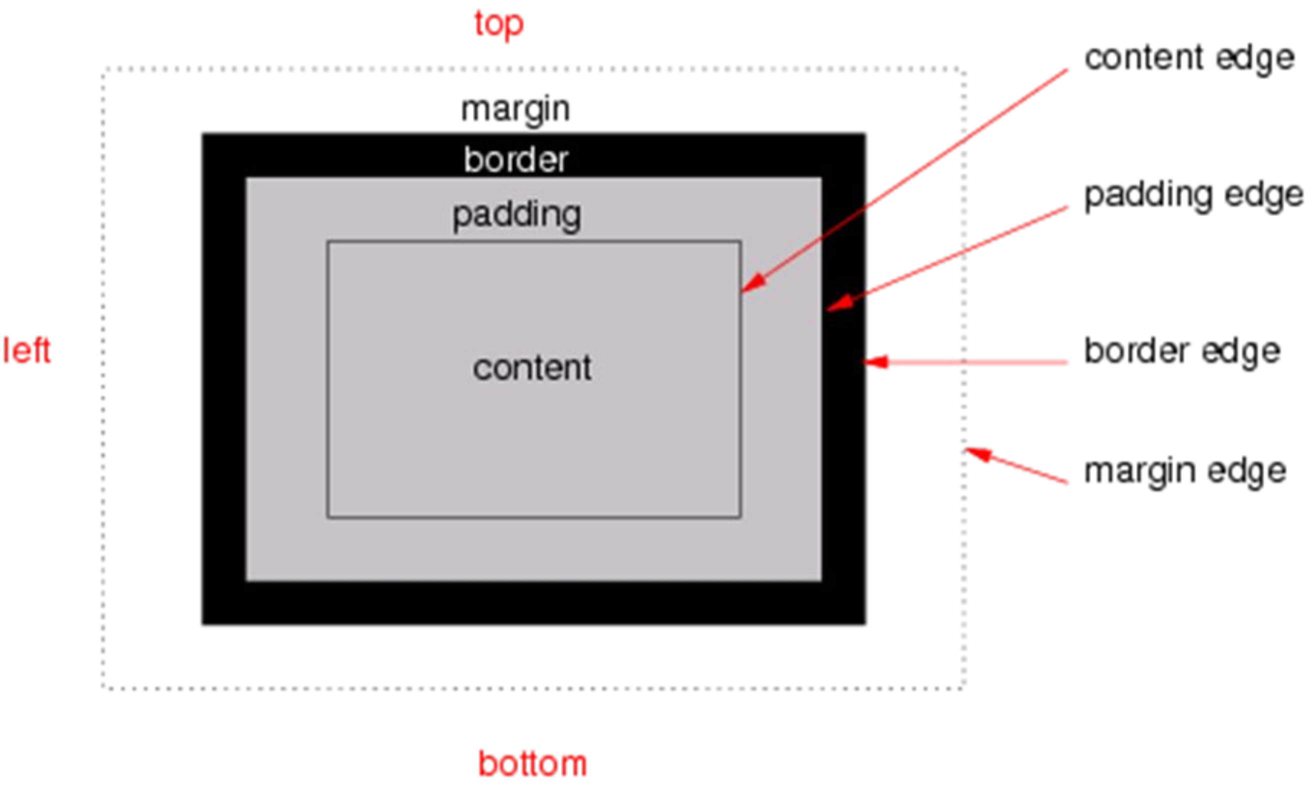


Block formatting context



CSS Box Model

CSS Box Model



Sizes and CSS Box Model

Some properties (e.g. sizing properties, such as `width`, `max-height`, `min-width`, etc.) directly depend on the value of `box-sizing` property. This property can have following values:

- `content-box`: height and width define only content size (default)
- `padding-box`: includes paddings width/height
- `border-box`: includes borders in width/height
- `margin-box`: you get the point...

There are also `stroke-box`, `fill-box`, and `view-box` values that are used in SVG images.

W3C CSS Box Model

MDN CSS Box Model

Positioning

Positioning

Another important CSS-property is the **`position`** property. It helps to control the location of an element on a page. The property can have following values:

- **`static`**: default value
- **`relative`**: position is calculated relatively to its default location
- **`fixed`**: position is calculated relatively to the viewport
- **`absolute`**: position is calculated relatively to the nearest positioned ancestor
- **`sticky`**: position is calculated as the relative one, but it sticks to edge of viewport when element reaches it during the scroll

MDN Positioning

Learn CSS Positioning

What else?

- Images
- Fonts
- Colors
- Media queries
- Grid/Flex boxes
- Animations
- Theming (light/dark)
- etc…

The amazing powers of CSS

JavaScript

C Programmers: All variables must be strictly typed

Javascript programmers:

91-"1" == 90
true+true == 2
0.1+0.2 != 0.3
[]+[] == ""
{}+[] == 0
[] == 0
9999999999999999 == 10000000000000000
VARIABLE IS VARIABLE
imgflip.com

Syntax

Primitive Types

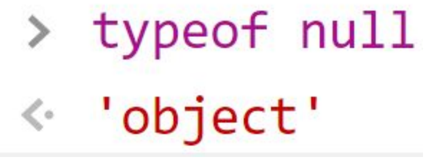
1. string
2. number
3. bigint
4. boolean
5. symbol
6. null
7. undefined

Non-primitive Types

- Object

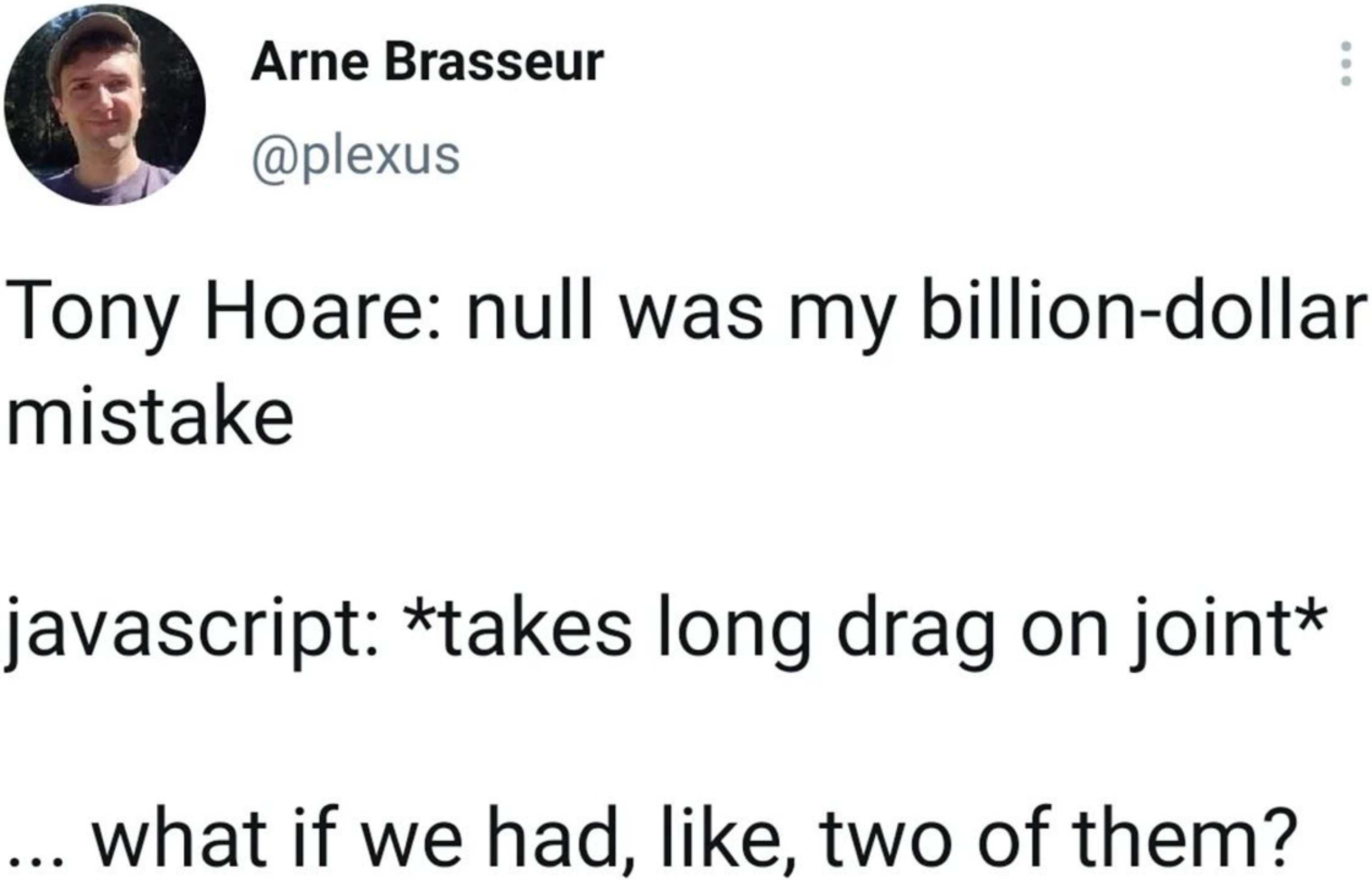
Anything that is constructed with **new** or an object/array/function literal

Non-primitives are compared/passed by reference!



```
> typeof null
< 'object'
```

Data Structures (MDN)



Arne Brasseur

@plexus

Tony Hoare: null was my billion-dollar mistake

javascript: *takes long drag on joint*

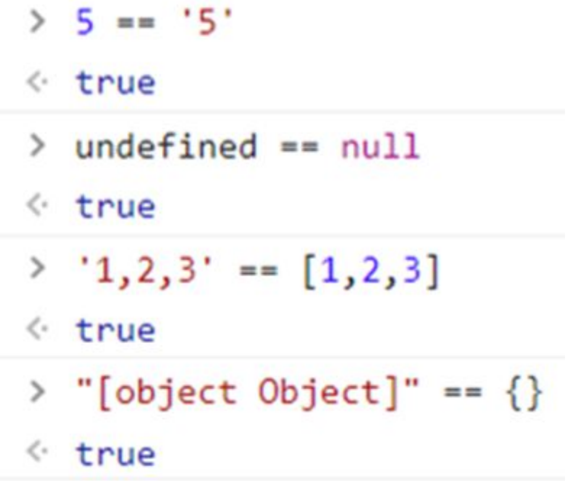
... what if we had, like, two of them?

Strict vs. Loose Equality

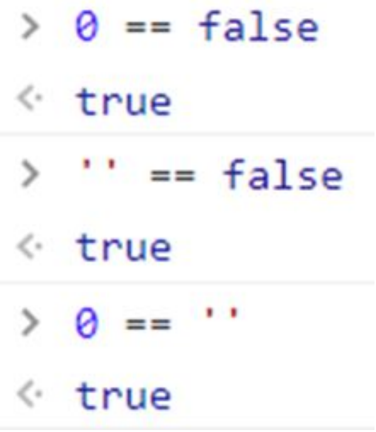
Loose (abstract) Equality

If types do not have to match, it will perform type coercion (implicitly)

Uses ==



```
> 5 == '5'
<· true
> undefined == null
<· true
> '1,2,3' == [1,2,3]
<· true
> "[object Object]" == {}
<· true
```

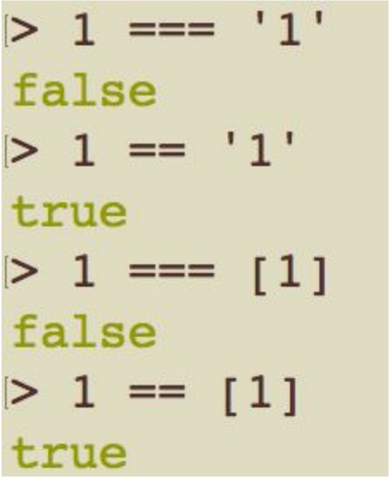


```
> 0 == false
<· true
> '' == false
<· true
> 0 == ''
<· true
```

Strict Equality

The good ol' equality we know and are used to.

Uses ===



```
> 1 === '1'
false
> 1 == '1'
true
> 1 === [1]
false
> 1 == [1]
true
```

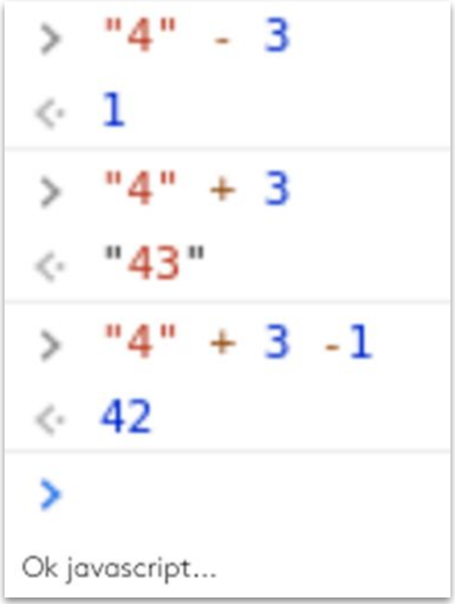
Abstract Equality Comparison Algorithm

Type Coercion?

Implicit type conversion

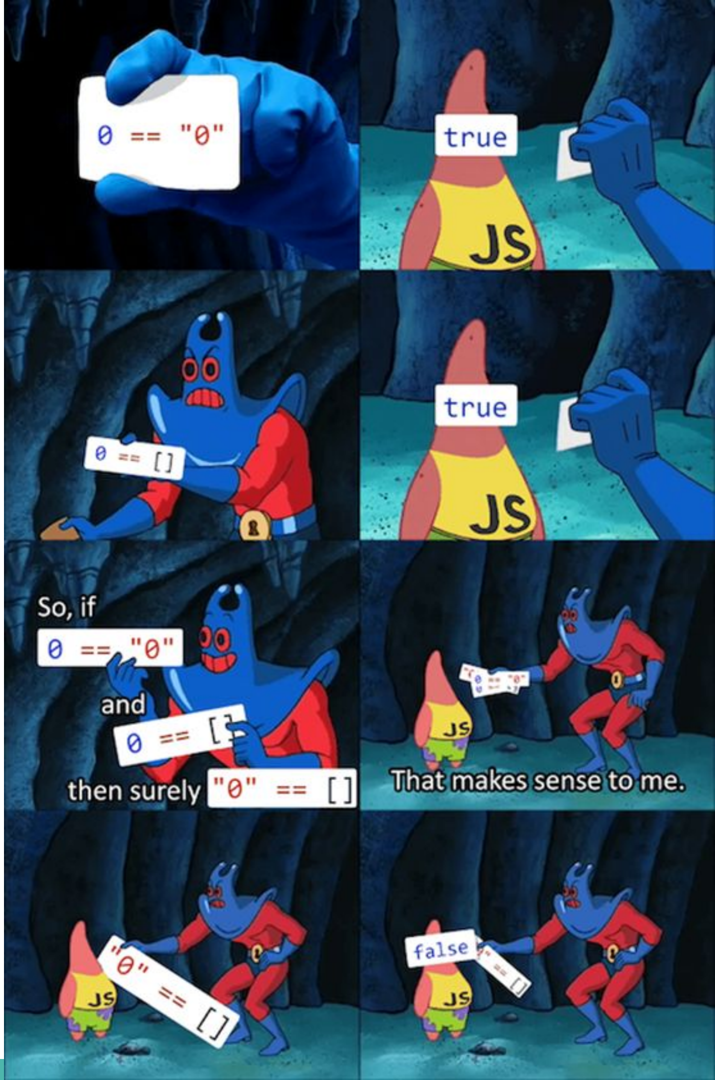
Its rules are weird (but generally straightforward)

It's better to avoid it altogether!

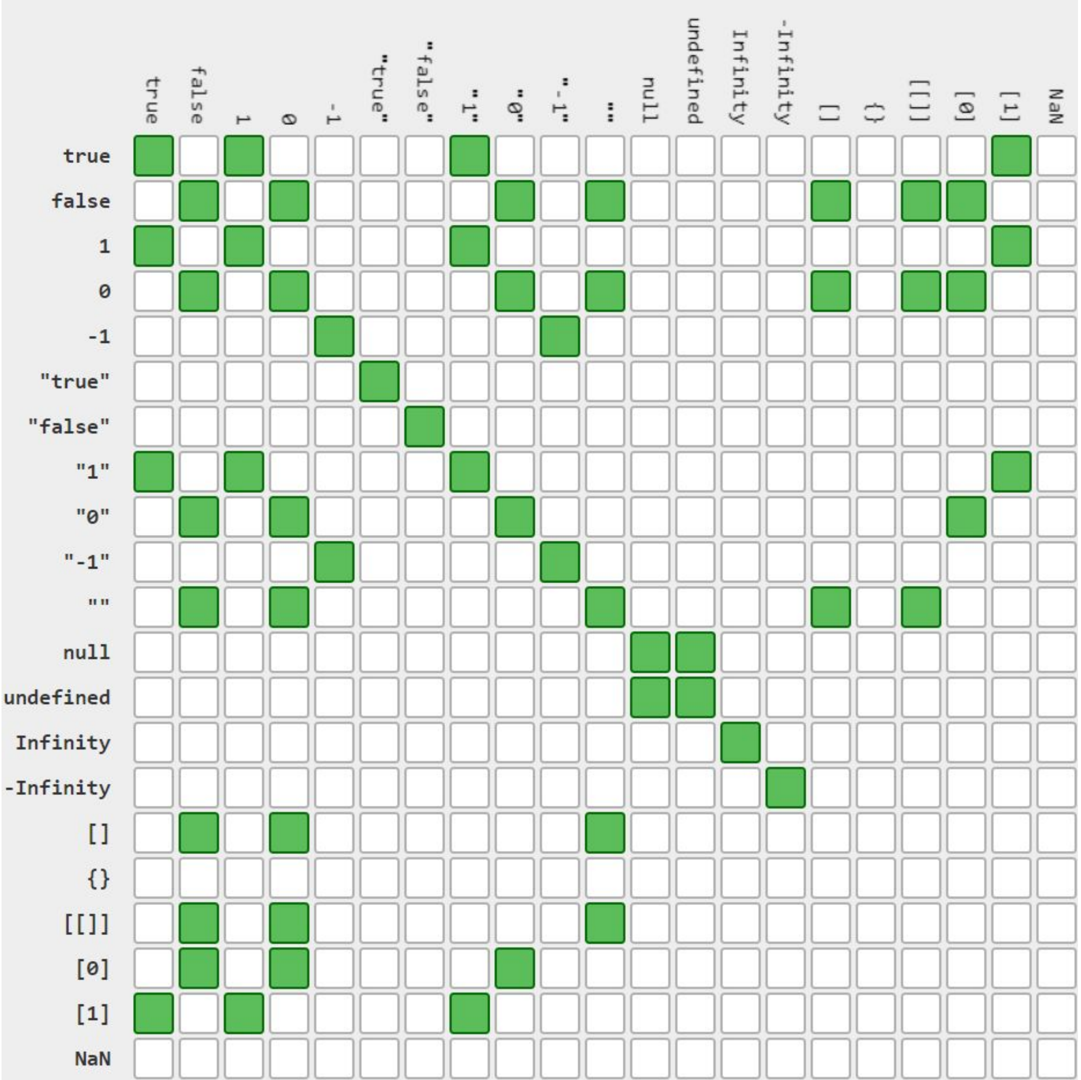


```
> "4" - 3
<· 1
> "4" + 3
<· "43"
> "4" + 3 -1
<· 42
>
```

Ok javascript...



https://developer.mozilla.org/en-US/docs/Glossary/Type_coercion



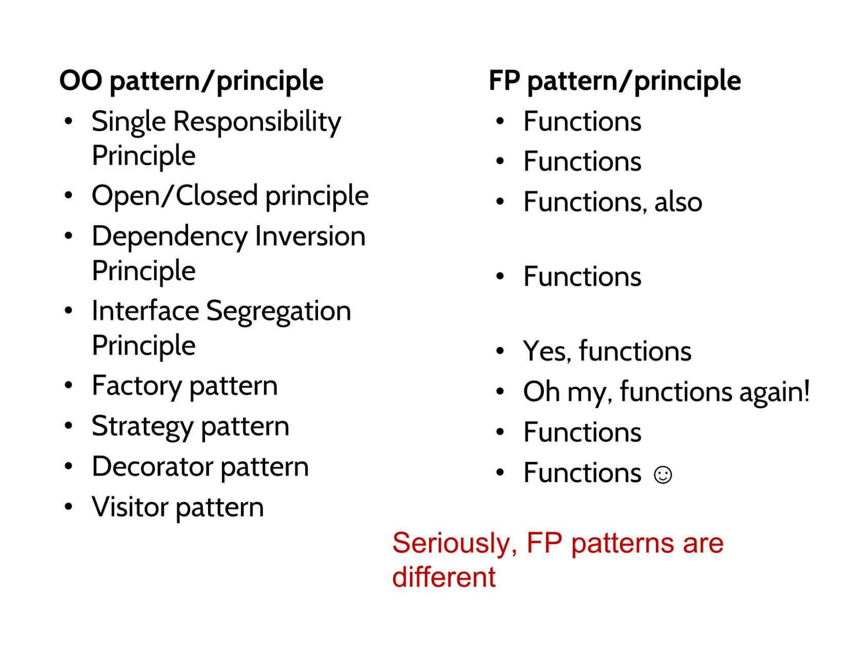
	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[[]]	[0]	[1]	NaN
true	🟩		🟩					🟩												🟩	
false		🟩		🟩					🟩		🟩					🟩		🟩	🟩		
1	🟩		🟩					🟩												🟩	
0		🟩		🟩					🟩		🟩					🟩		🟩	🟩		
-1					🟩					🟩											
"true"						🟩															
"false"							🟩														
"1"	🟩		🟩					🟩												🟩	
"0"		🟩		🟩					🟩										🟩		
"-1"					🟩					🟩											
""		🟩		🟩							🟩					🟩		🟩			
null												🟩	🟩								
undefined												🟩	🟩								
Infinity														🟩							
-Infinity															🟩						
[]		🟩		🟩							🟩										
{}																					
[[]]		🟩		🟩							🟩										
[0]		🟩		🟩					🟩												
[1]	🟩		🟩					🟩													
NaN																					

https://dorey.github.io/JavaScript-Equality-Table/

Popular JavaScript Programming Paradigms

- Functional
- Asynchronous
- Event-Driven
- Object-Oriented

Functional Programming



OO pattern/principle	FP pattern/principle
Single Responsibility Principle	Functions
Open/Closed principle	Functions
Dependency Inversion Principle	Functions, also
Interface Segregation Principle	Functions
Factory pattern	Yes, functions
Strategy pattern	Oh my, functions again!
Decorator pattern	Functions
Visitor pattern	Functions ☺

Seriously, FP patterns are different

Pure Functions

- Do not perform side-effects
- Do not depend on external data
- Same input → same output

```
const state = {
    currentNumber: 0
};

function sum(a, b) {
    state.currentNumber = a;  ✗
    return a + b * Math.random();  ✗
}

function pureSum(a, b) { ✓
    return a + b;
}
```

Immutability

Immutable variables cannot be changed once set

```
let obj = { prop: 42 };

Object.freeze(obj);

obj.prop = 33; // no error?

console.log(obj.prop); // 42

obj = ‘something else’;
```

```
const number = 0;
number = 2; // ✗ Uncaught TypeError

let anotherNumber = 0;
anotherNumber = 2; // ✓ Correct

const state = {
   number: 0
};
state.number = 2; // ✓ Correct
```

First-class Functions

Functions in functional programming are treated as a data type and can be used like any other value.

```
const numbers = [1, 2, 3];
function isEven(number) { return number % 2 === 0; }
const double = (number) => number * 2;
numbers.map(double); // [2, 4, 6]
numbers.filter(isEven); // [2]
```

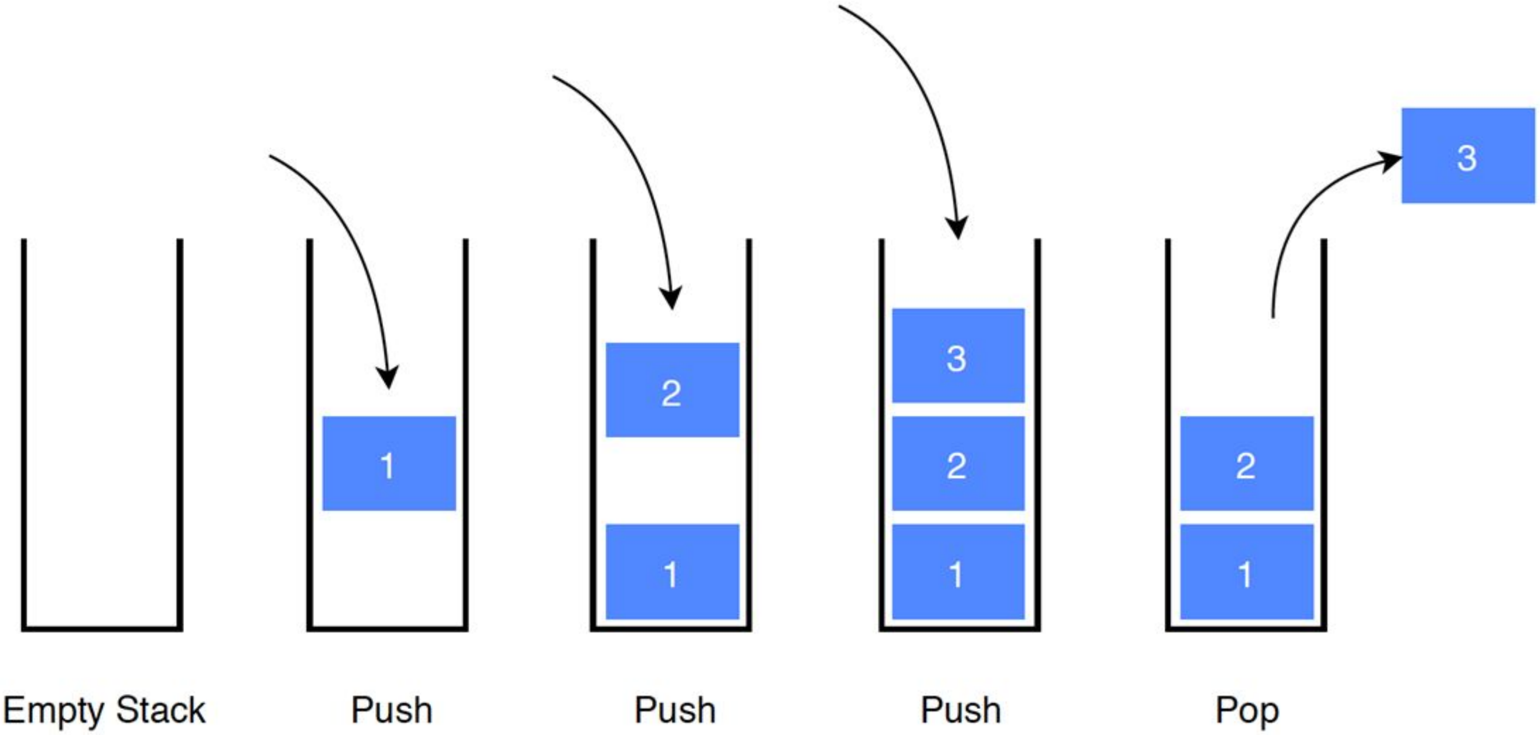
Higher-Order Functions

Higher-order functions can accept other functions as parameters or return functions as output

```
function attachLogger(fn) {
   return function (...args) {
       console.log("Function called");
       console.log(args);
       return fn(...args);
   }
}

function sum(a, b) {
   return a + b;
}

const loggedSum = attachLogger(sum);

sumWithLogger(1, 2);
```

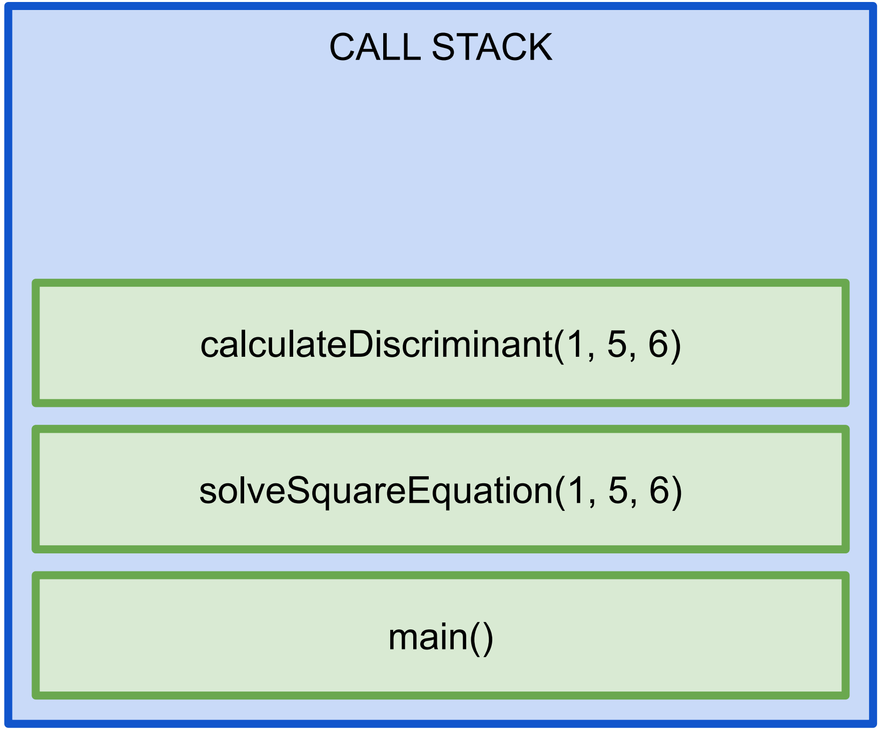
Asynchronous Programming

Stack



Call Stack

```
function calculateDiscriminant(a, b, c) {
  return b**2 - 4*a*c;
}

function solveSquareEquation(a, b, c) {
  const D = calculateDiscriminant(a, b, c);
  return {
      x1: (-b + Math.sqrt(D)) / 2*a,
      x2: (-b - Math.sqrt(D)) / 2*a,
  }
}

function main() {
  result = solveSquareEquation(1, 5, 6);
  console.log('The solutions for x^2+5x+6=0
are x1 = ' + result.x1 + ' and x2 = ' +
result.x2);
}

main();
```



CALL STACK

calculateDiscriminant(1, 5, 6)

solveSquareEquation(1, 5, 6)

main()

JavaScript engines are single-threaded

One thread ⇒ One call stack

MDN Call Stack

3 rules:
- no wishing for death
- no falling in love
- no bring back dead people
I want javascript to support multithreading.
There are 4 rules

[RU] Что такое Event Loop

Event Loop FULL Advanced JavaScript Concepts

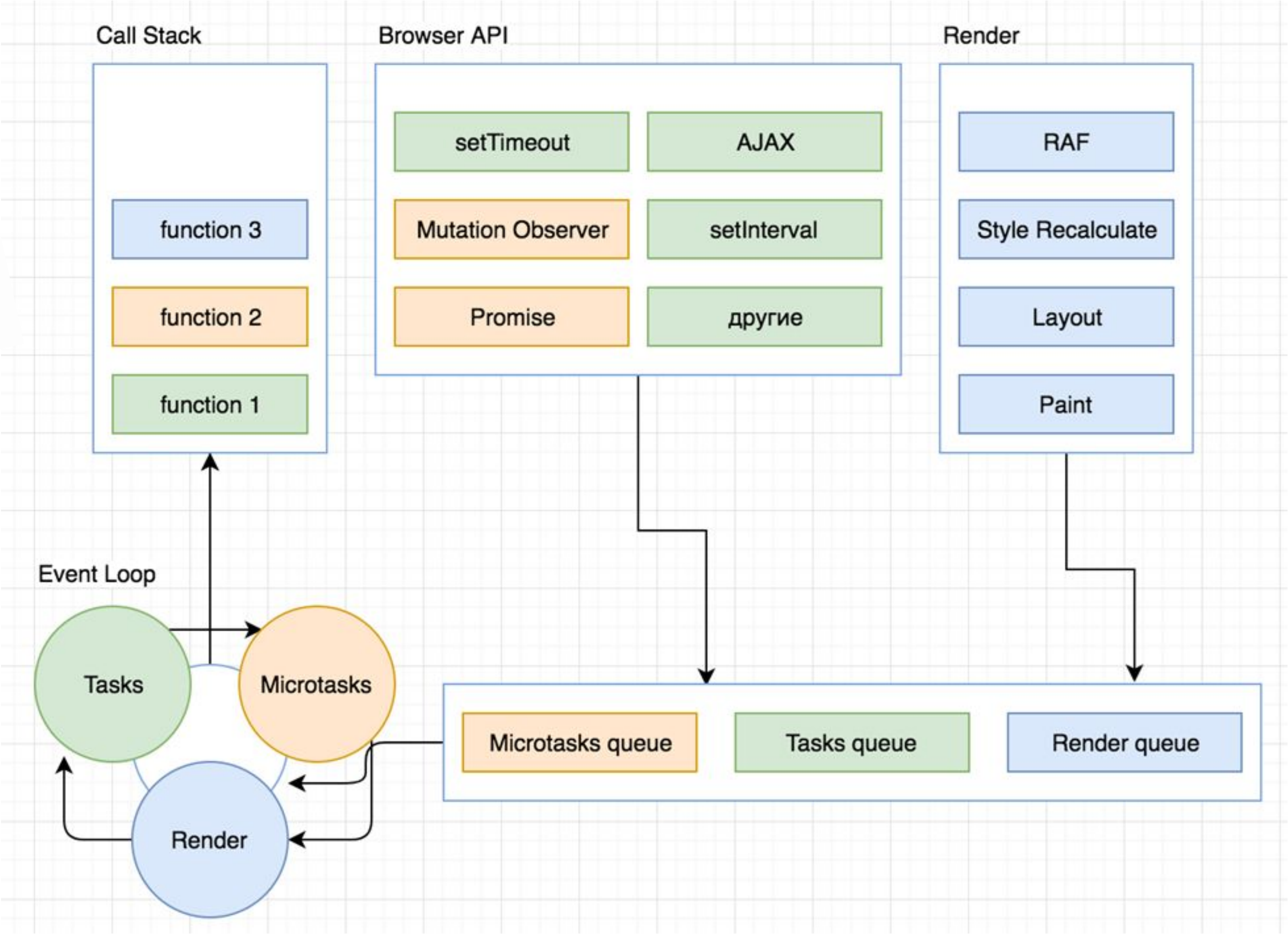


Image by anatoliy841993

Event Loop and Macrotasks

Microtasks

Macro vs. Micro Tasks

- `setTimeout`
- `setInterval`
- `setImmediate`
- `requestAnimationFrame`
- `I/O`
- `UI rendering`

- `process.nextTick`
- `Promises`
- `queueMicrotask`
- `MutationObserver`

Difference between Microtasks and Macrotasks

Most of the interactions with the different APIs are asynchronous.

This means that sometimes you can’t get the computation result immediately, but you can get it after some (unspecified) time

DID YOU HEAR WHAT HAPPENED TO THE JAVASCRIPT DEVELOPER?
ME NEITHER. I'M STILL WAITING FOR HER CALLBACK.
imgflip.com

Callbacks

```
function animate(frameTime) { … }
```

Is that a function? **Yes**

Is that a functional object? **Yes**

Is that a reference to the functional object? **Yes**

Callbacks

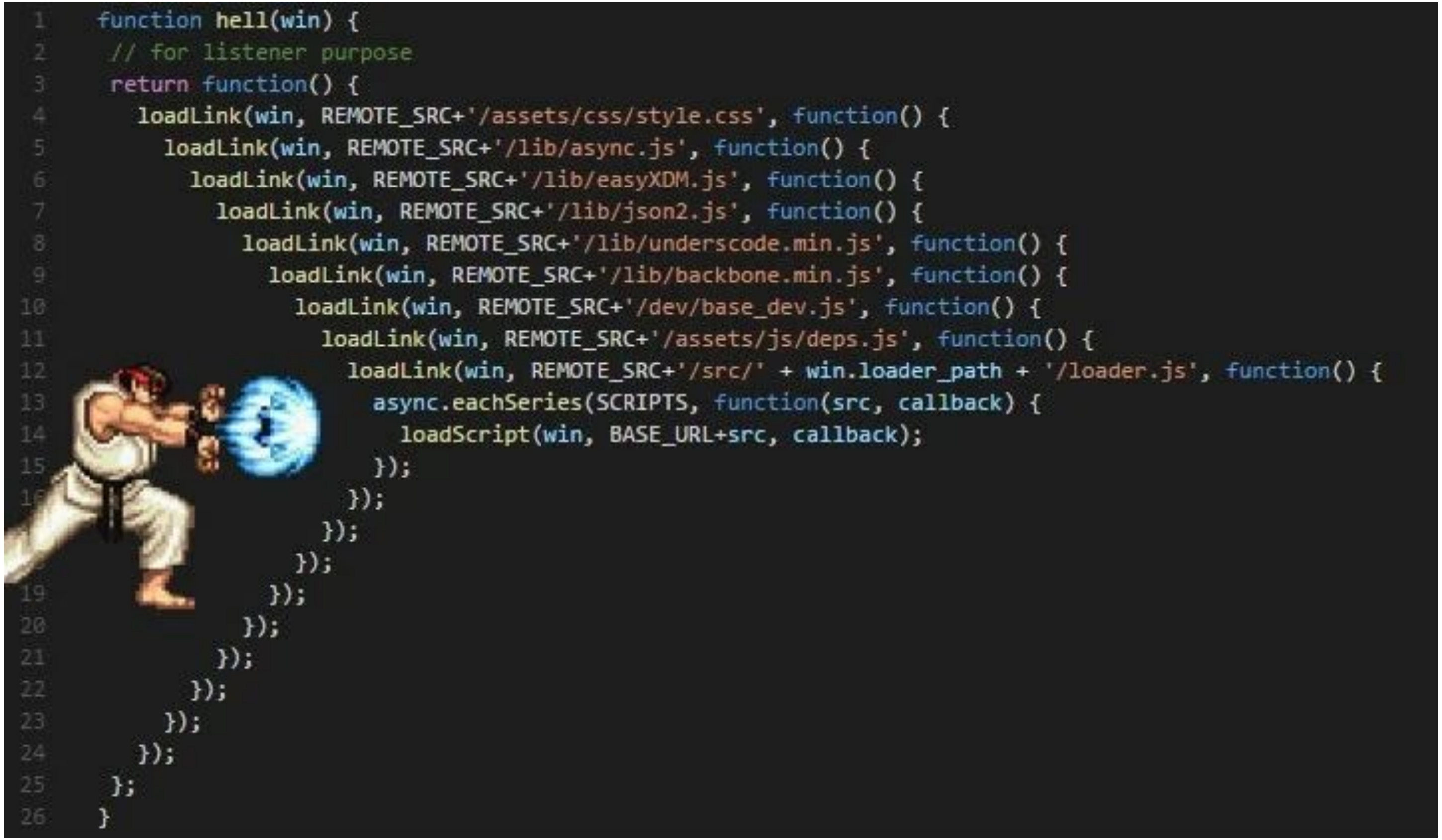
```
function animate(frameTime) { … }
```

```
window.requestAnimationFrame(animate);
```

So, we can pass it as an argument to another function

Introduction: Callbacks

Callback Hell



```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

Promise API

```
function loadScriptPromise(scriptName) {...}

loadScriptPromise('1.js')
    .then(() => loadScriptPromise('2.js'))
    .then(() => loadScriptPromise('3.js'))
    .then(() => console.log('All scripts are loaded!'))
    .catch((error) => console.log('An error occurred!'));
```

MDN - Promise

Promise API

```javascript
function loadScriptPromise(scriptName) {
    return new Promise((resolve, reject) => {
        loadScript('1.js', (error, script) => {
            if (error) reject(error);
            else resolve(script);
        });
    });
}

loadScriptPromise('1.js')
    .then(() => loadScriptPromise('2.js'))
    .then(() => loadScriptPromise('3.js'))
    .then(() => console.log('All scripts are loaded!'))
    .catch((error) => console.log('An error occurred!'));
```

Promise API

```
function loadScriptPromise(scriptName) {...}


Promise.all([
   loadScriptPromise('1.js'),
   loadScriptPromise('2.js'),
   loadScriptPromise('3.js')
])
   .then((results) => console.log('All scripts are loaded!'))
   .catch((error) => console.log('An error occurred!'));
```

CALLBACKS
PROMISES
ASYNC/AWAIT

async / await

```
function loadScriptPromise(scriptName) {...}

async function loadAllScripts() {
   const script1 = await loadScriptPromise('1.js');
   const script2 = await loadScriptPromise('2.js');
   const script3 = await loadScriptPromise('3.js');
   console.log('All scripts are loaded!');
}

loadAllScripts()
   .then(() => /* do something... */)
```

Events and Event-Driven Programming

Events are objects that implement the Event interface.
Those objects are dispatched by the user agent* and handled by the Event Listener.
Events are responsible for handling user interactions or network activity.

Browser Events Explained in Plain English

Handling Events

A modern guide to Events in JavaScript

MDN Event reference

Listening to Events

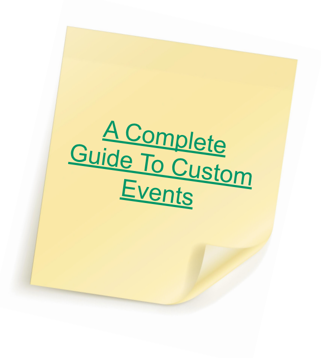
```
// HTML
<button onclick="someHandler()">Button</button>

// JavaScript
const button = document.getElementById("someButton");
function eventHandler(event) {
   console.log("I'm listening on a click event");
}

// Just one event
button.onclick = eventHandler;
// Many events
button.addEventListener("click", eventHandler);
```

Removing Listeners

```
// JavaScript
const button = document.getElementById("someButton");
const eventHandler = (event) => {
   console.log("I'm listening on a click event");
}

button.onclick = null;
button.removeEventListener("click", eventHandler);
```

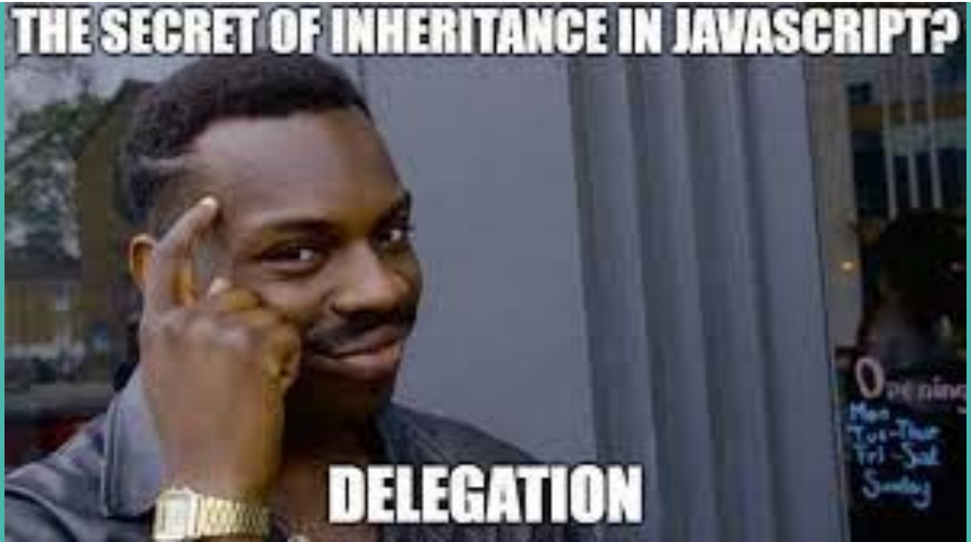
Custom Events

```
const myEvent = new CustomEvent("myevent", {
   detail: {},
   bubbles: true,
   cancelable: true,
   composed: false,
});

document.querySelector("#someElement")
        .dispatchEvent(myEvent);

document.querySelector("#someElement")
        .addEventListener("myevent", (event) => {
            console.log("I'm listening on a custom event");
        });
```



[A Complete Guide To Custom Events]

Object-Oriented

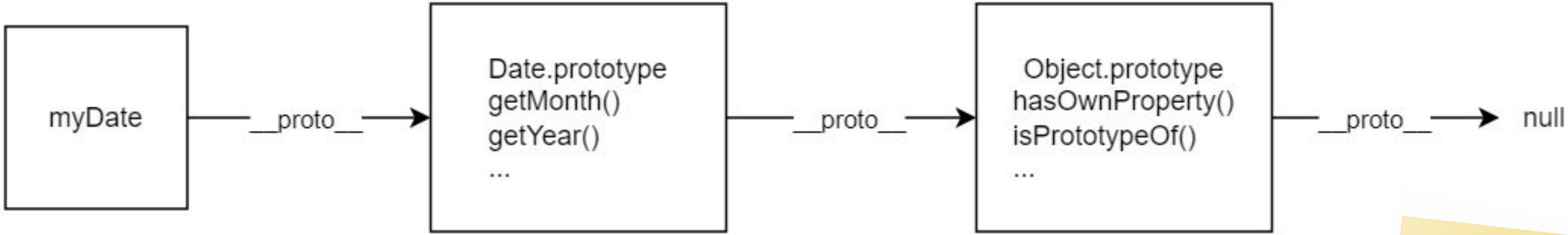


Objects, Inheritance and Prototypes

```
const personPrototype = {
    name: "person",
    greet: function() {
        console.log("Hello! My name is " + this.name);
    },
};
const carl = { name: "Carl" };
carl.__proto__ = personPrototype;           // ❌ Not recommended!
Object.setPrototypeOf(carl, personPrototype); // ✅ Recommended
carl.greet(); // Hello! My name is Carl
// Alternatively (even better ✅)
const mike = Object.create(personProtoype);
mike.name = "Mike";
```

MDN - Inheritance and the Prototype Chain

Prototype chain



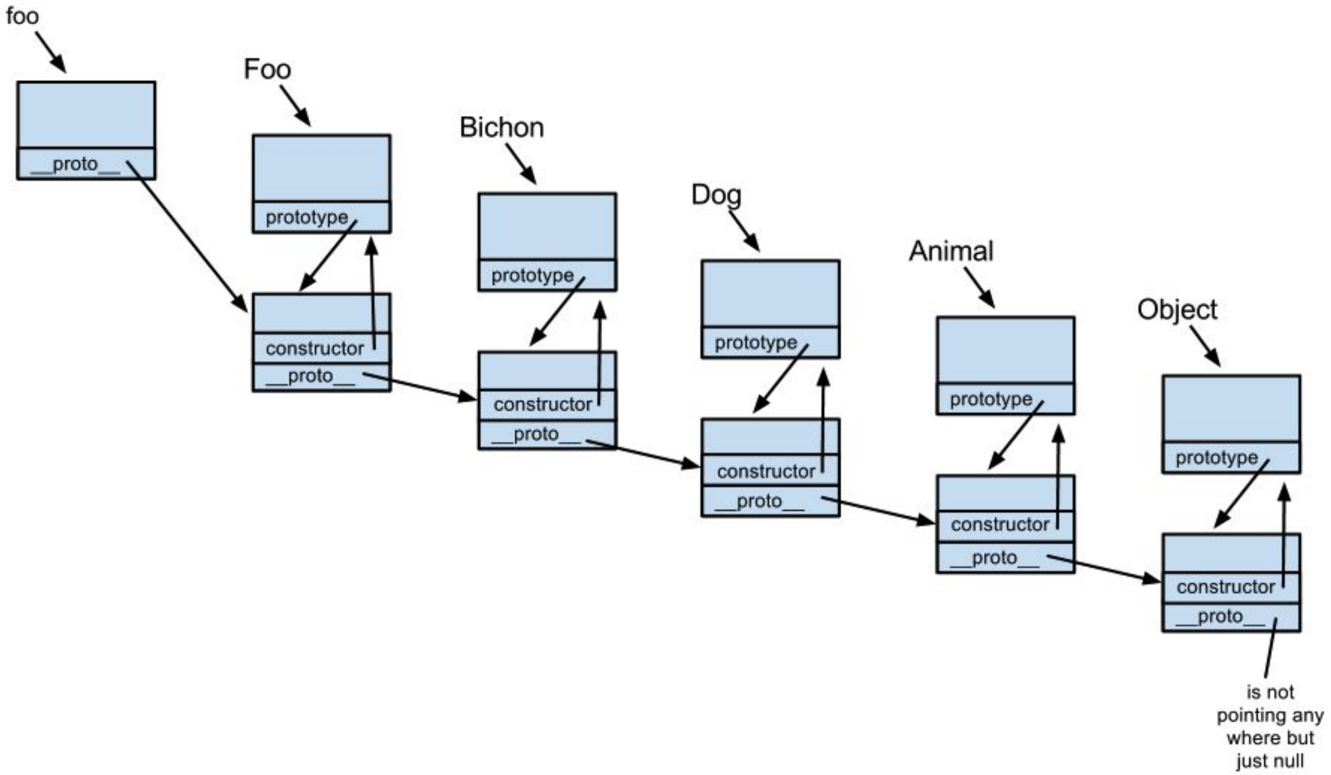
MDN - Object Prototypes

Constructor Function

Constructor in JS is a simple function that instantiates a new object when called with the **`new`** operator.

```
function Box(value) {
    this.value = value;
}
Box.prototype.getValue = function () {
    return this.value;
};
const box = new Box(1);

// Mutate Box.prototype after an
instance has already been created
Box.prototype.getValue = function () {
    return this.value + 1;
};
box.getValue(); // 2
```

Constructor Function - Inheritance

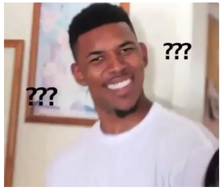


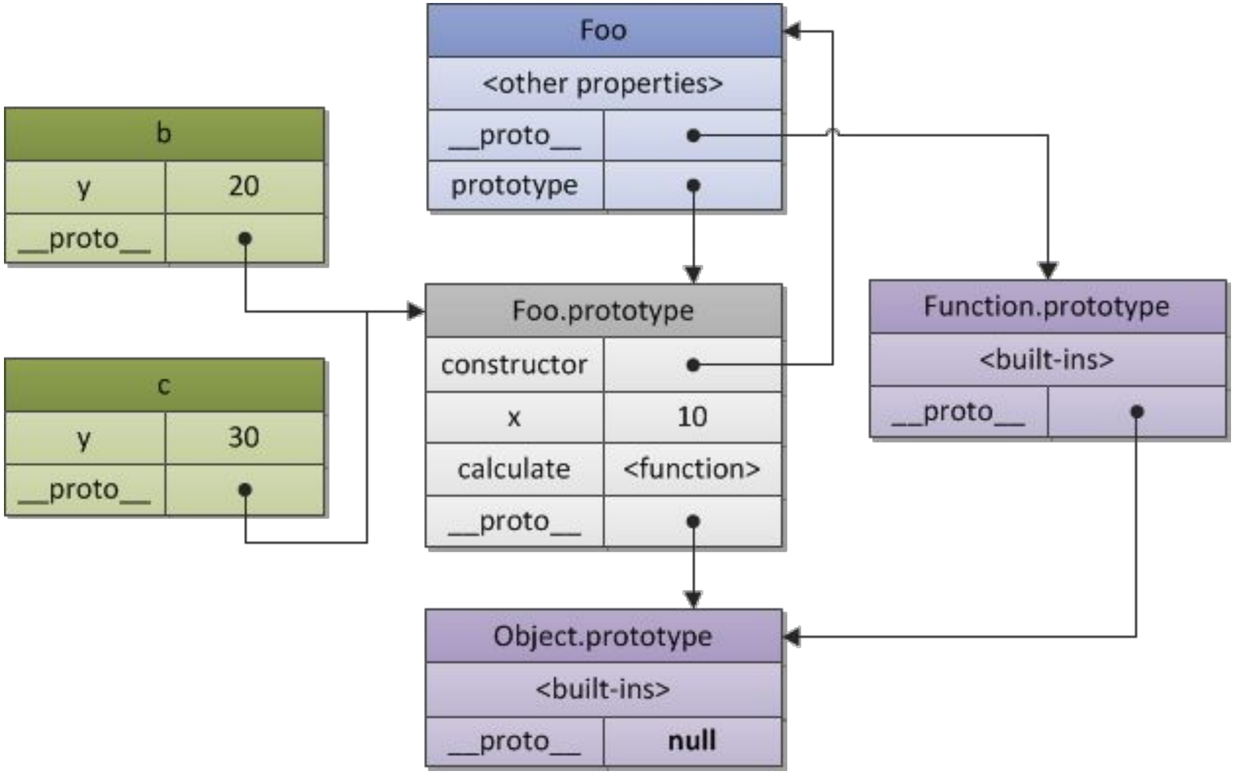
Constructor Function - more examples

```
function Foo(y) {
  this.y = y;
}

Foo.prototype.x = 10;
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};

var b = new Foo(20);
var c = new Foo(30);

b.calculate(30); // 60
c.calculate(40); // 80
```

```
console.log(
  b.__proto__ === Foo.prototype, // true
  c.__proto__ === Foo.prototype, // true

  b.constructor === Foo, // true
  c.constructor === Foo, // true
  Foo.prototype.constructor === Foo, // true

  b.calculate === b.__proto__.calculate, //
true
  b.__proto__.calculate ===
Foo.prototype.calculate // true
);
```

Constructor Function - Prototype





b	
y	20
__proto__	●

c	
y	30
__proto__	●

Foo	
<other properties>	
__proto__	●
prototype	●

Foo.prototype	
constructor	●
x	10
calculate	<function>
__proto__	●

Function.prototype	
<built-ins>	
__proto__	●

Object.prototype	
<built-ins>	
__proto__	**null**

https://stackoverflow.com/questions/9959727/proto-vs-prototype-in-javascript

Meme time

Dad: Why are your eyes red son?

Son: I smoke weed

Dad: Don’t lie, you’re crying because you have been coding in JavaScript

YOU GET A PROTO
YOU GET A PROTO
YOU GET A PROTO
EVERYONE GETS A PROTO

Using `this` in the global context

In global context (outside any object), `this` keyword points to the global object (`window` in browsers).

```
function testFn () {
    return this;
}

testFn() === window // true
```

MDN this

Using `this` in methods of objects

In the context of any object, `this` points to the object from which the method is called.

```
const o = {
   prop: 37,
   f: function() {
       return this.prop;
   }
};

console.log(o.f()); // 37

const f = o.f;
console.log(f()); // undefined
```

Using `this` in methods of objects

In the context of any object, **`this`** points to the object from which the method is called.

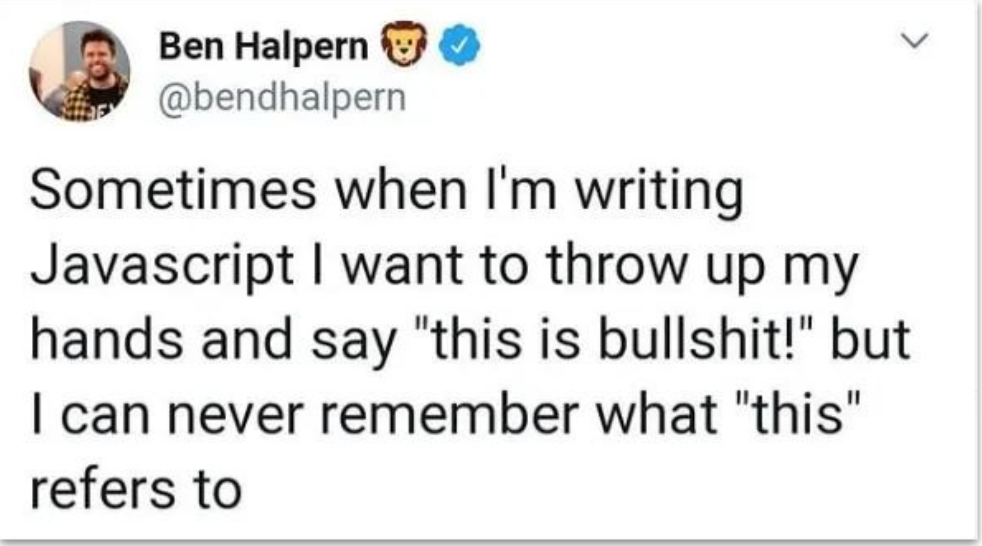
Moreover, the method doesn’t have to be defined in the object from the moment of its creation, you can add it there later.

```
const o = { prop: 37 };

function independent() {
   return this.prop;
}

o.f = independent;

console.log(o.f()); // 37

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

Using `this` in arrow functions

`this` in arrow functions *always* points to the context in which it was created

```
const obj = {
    foo: function() {
        return this;
    },
    bar: () => this;
};


console.log(obj.foo()); // obj
console.log(obj.bar()); // window
```



Ben Halpern 🦁 ✓
@bendhalpern

Sometimes when I'm writing Javascript I want to throw up my hands and say "this is bullshit!" but I can never remember what "this" refers to

ES6 Classes

```
class Animal {
     constructor() {
          console.log('New Animal was born!');
          this.eats = true;
     }

     walk() {
          alert("Animal walk");
     }
}

class Rabbit extends Animal {
     constructor() {
          console.log('New Rabbit was born!');
          this.jumps = true;
     }
}

const rabbit = new Rabbit();
rabbit.walk(); // Animal walk
```

Classes

ES6 Classes

```
class Dog {
    set weight(val) {
        this.#weight = val;
    }

    get weight() {
        return this.#weight;
    }

    static compare(dogA, dogB) {
        return dogA.weight - dogB.weight;
    }
}
```

Browser Environment

The global object `window` provides variables and functions that are available anywhere. By default, those that are built into the language or the environment.

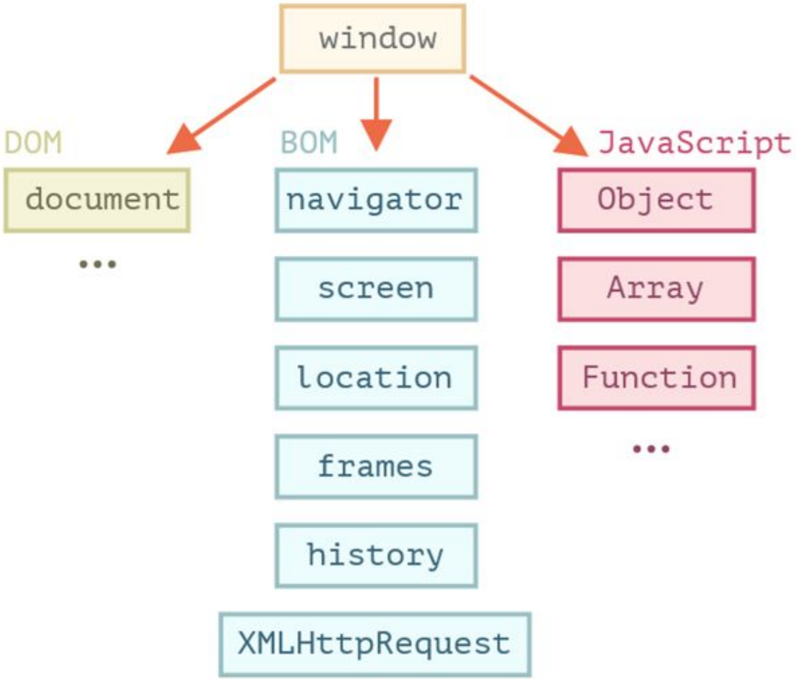
Global object

Browser environment, specs

MDN Window

Node.js Global objects

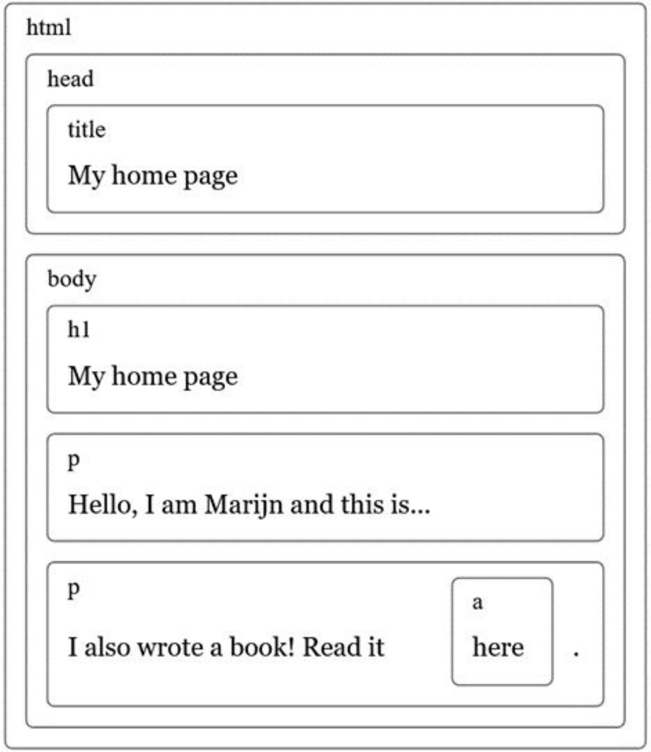
Window Object



What can we do with the DOM?

- Access elements (e.g. `getElementById`)
- Access elements using CSS selectors (e.g. `querySelector`)
- Create new DOM nodes (`createElement`)
- Modify element styles (e.g. `div.style.color` or `div.setAttribute()`)
- Modify element attributes and classes
- Modify a node’s content

Document Object Model (DOM)



html

head

title

My home page

body

h1

My home page

p

Hello, I am Marijn and this is...

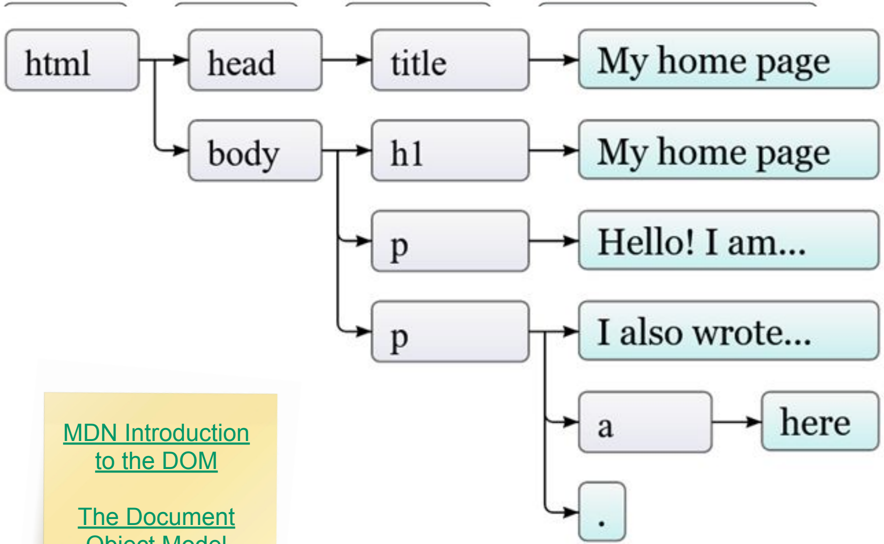
p

I also wrote a book! Read it

a

here

.



html

head

title

My home page

body

h1

My home page

p

Hello! I am...

p

I also wrote...

a

here

.

MDN Introduction to the DOM

The Document Object Model

Summary