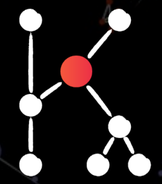


JAVASCRIPT PART 2

KIRELL BENZI, PH.D

@KirellBenzi

www.kirellbenzi.com

Questions?

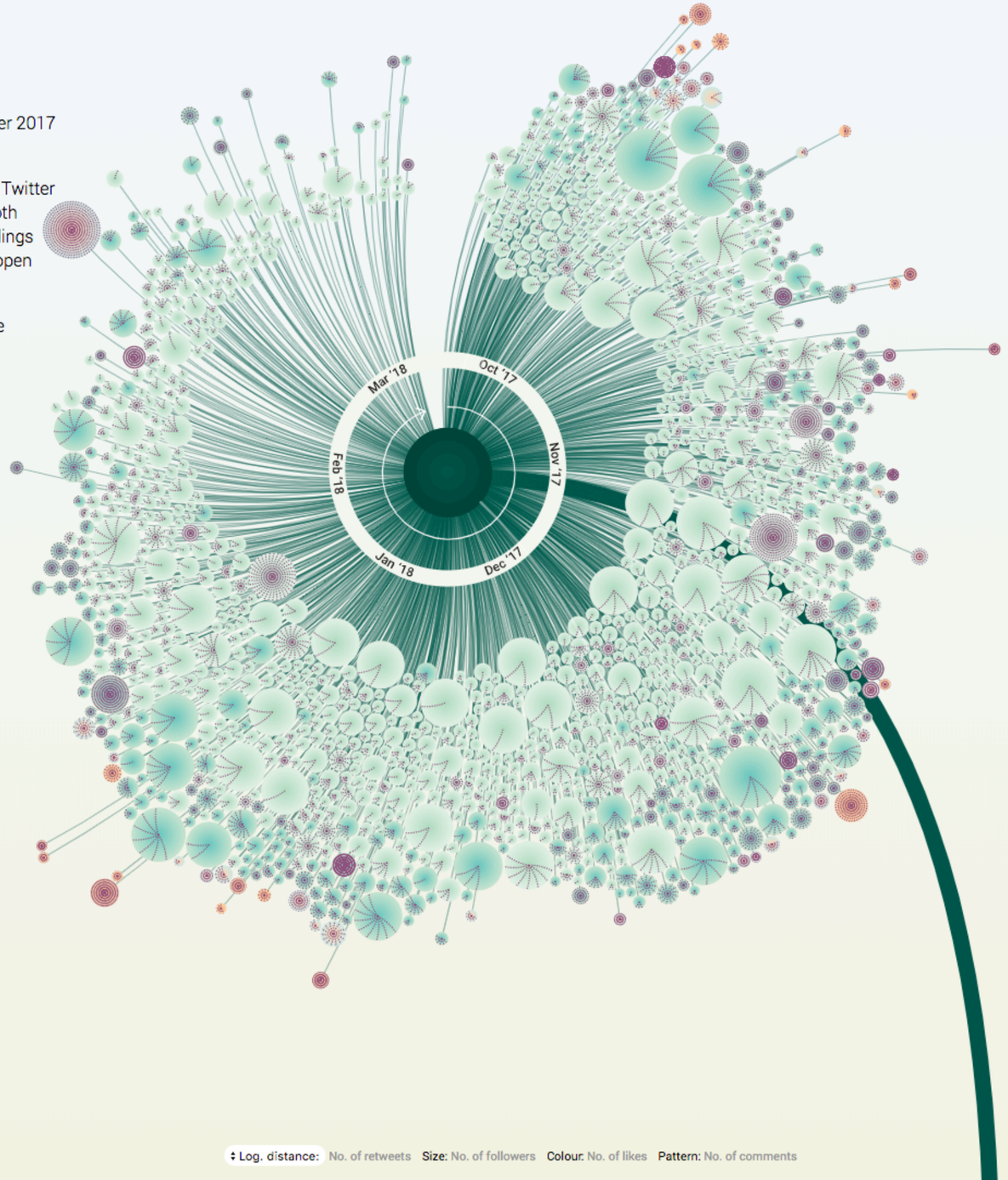
Trending seeds

Trending seeds are the most popular tweets from October 2017 through March 2018.

While media outlets generated significant awareness on Twitter regarding the issue, the hashtag #MeToo has allowed both celebrities and regular people to share their personal feelings and experiences – using social media as a platform for open discussion.

Hover over each seed to read individual tweets or explore conversations around key events.

EXPLORE TIMELINE



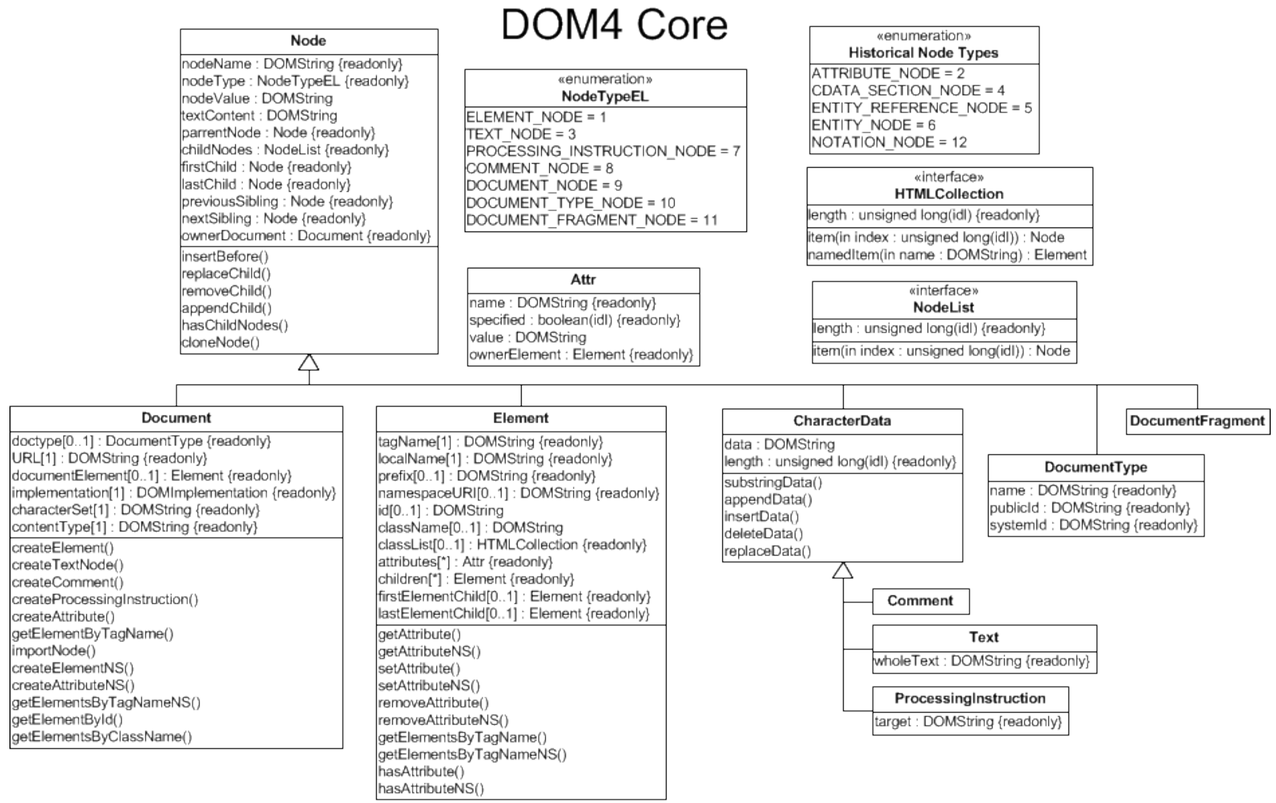
Log. distance: No. of retweets Size: No. of followers Colour: No. of likes Pattern: No. of comments

Let’s get back to the DOM

The Document Object Model (DOM) is a programming interface for HTML and XML documents.

It represents the page so **that programs** can change the document structure, style and content.

The DOM represents the document as nodes and objects.



[Gerd Wagner]

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

DOM API

DOM was designed to be independent of any particular programming language.

The preferred language to access it is Javascript

Easy interface, just use the ***document*** or ***window*** global objects to manipulate the document itself or to get at the children of that document, which are the various elements in the web page.

Getting elements

getElementById / getElementsByTagName / getElementsByClassName

```
<!DOCTYPE html>
<html>
    <body>
      <p id="myText">Nice text</p>
      <p id="jsText"></p>

      <script type="text/javascript">
        const myElement = document.getElementById("myText");
        document.getElementById("jsText").innerHTML =
         "Filled from code: " + myElement.innerHTML;
      </script>
    </body>
</html>
```

querySelector()

Returns the first Element within the document that matches the specified selector, or group of selectors.

Works also to the list of children of an element.querySelector().

Use querySelectorAll() to get all matching elements

```
const el = document.querySelector(“div”); // Select by tag
const el2 = document.querySelector(“#myid”); // Select by ID
const el3 = document.querySelector(“.myclass”); // Select by class
const el4 = document.querySelector("div.user-panel.main input[name=‘login']");
```

Creating elements

createElement / createTextNode

```
// create a couple of elements in an otherwise empty HTML page
let heading = document.createElement("h1");
const headingText = document.createTextNode("Hello from JS!”);
heading.appendChild(headingText);
document.body.appendChild(heading);
```

Hello from JS!

Elements attributes

There are three ways of accessing the attributes of a DOM Element in JavaScript.

For simplicity we will just mention ***element.attributes*** which contains the list of all standard attributes

Hopefully, we will soon rely on D3.js selection mechanism, to perform all the operations we need to CRUD (Create, Read, Update, Delete) elements from the DOM

```
<p id="paragraph" style="color: green;">Sample Paragraph</p>
```

Asynchronous JS

Synchronous code is easier to follow and debug but async is generally better for performance and flexibility.

Modern web came with AJAX (Asynchronous JavaScript And XML). Asynchronous post or pull data in a web page without reloading it.

A common use case involves loading resources and do something when ready using **a callback.**

Dealing with async (old)

```
// Old way
function successCallback(result) {
  console.log("It succeeded with " + result);
}

function failureCallback(error) {
  console.log("It failed with " + error);
}

doSomethingAsync(successCallback, failureCallback);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

Callback pyramid of doom

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('Got the final result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

Promises

A Promise is an object representing the eventual completion or failure of an asynchronous operation to which we attach callbacks

Promises can be chained because the `then` function returns a new promise, different from the original

```
const promise = doSomething();
const promise2 = promise.then(successCallback, failureCallback);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

No callback hell!

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => {
  console.log(`Got the final result: ${finalResult}`);
})
.catch(failureCallback);
```

Object-oriented JS

Object-oriented JS?

When creating complex code, it is natural to organize it into groups of functions and modules.

If we create an object that contains functions, it seems like we are creating methods

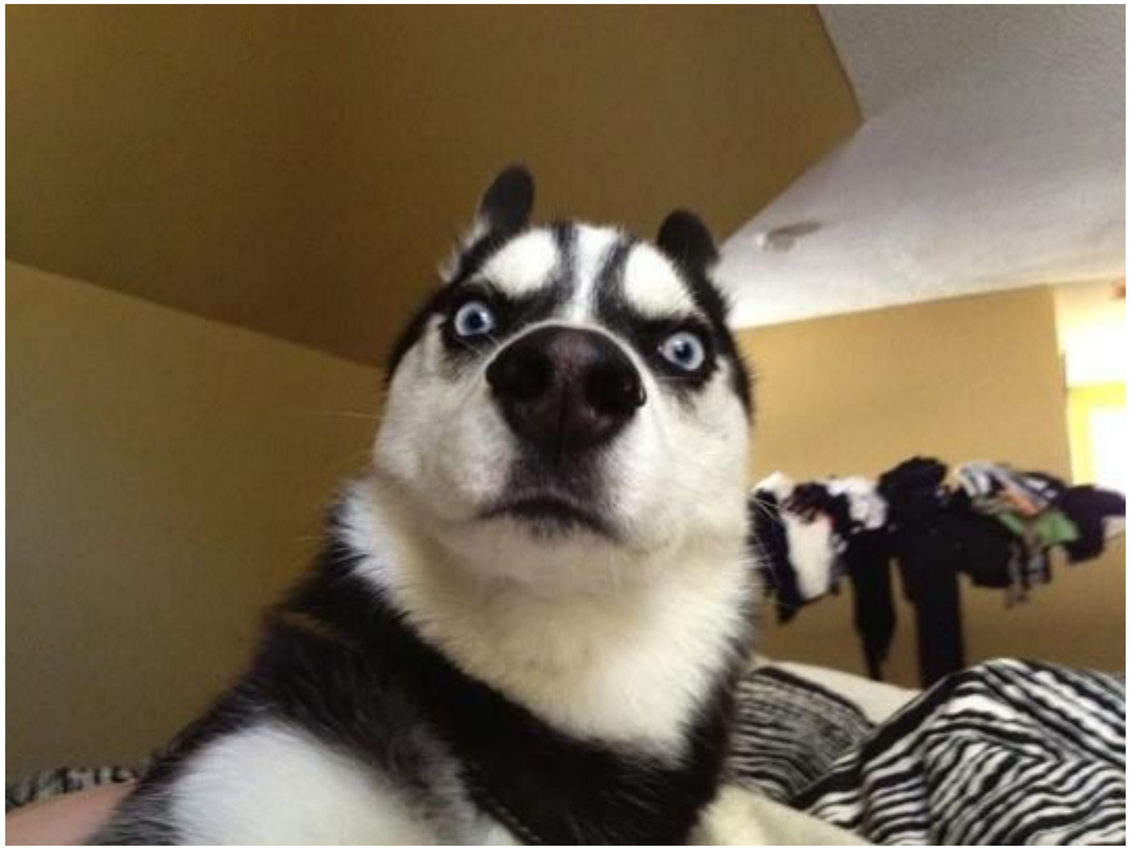
If we create a function that returns these objects, it looks like class contructors

```
function createDumbObject(name) {
  return {
    fakeMethod: () => console.log(name)
  };
};

createDumbObject('Yoda').fakeMethod(); // "Yoda"
```

Javascript does not have classes!

Javascript is NOT like other class-based language: C++, Java, Python..

It is the biggest source of confusion for developers discovering the language

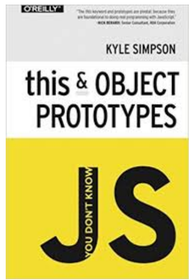


Call site

```
function baz() {
  // call-stack is: `baz`
  // so, our call-site is in the global scope
  console.log( "baz" );
  bar(); // <-- call-site for `bar`
}

function bar() {
  // call-stack is: `baz` -> `bar`
  // so, our call-site is in `baz`
  console.log( "bar" );
  foo(); // <-- call-site for `foo`
}

function foo() {
  // call-stack is: `baz` -> `bar` -> `foo`
  // so, our call-site is in `bar`
  console.log( "foo" );
}

baz(); // <-- call-site for `baz`
```

call-site

the location in code where a function is called (not where it's declared)



“This” (is not what you think)

The keyword ***this*** is a common source of confusion for developers because it is different from traditional languagues.

You can think of ***this*** as the context that is passed to when you call a function

```
function identify(context) {
  return context.name.toUpperCase();
}

function speak(context) {
  const greeting = "Hello, I'm " + identify( context );
  console.log( greeting );
}

identify( you ); // READER
speak( me ); // Hello, I'm Kirell
```

```
const me = {
  name: "Kirell"
};

const you = {
  name: "Reader"
};
```

You Don't Know JS: this & Object Prototypes Ch.1

“This” (is not what you think)

this mechanism provides a more elegant way of implicitly "passing along" an object reference.

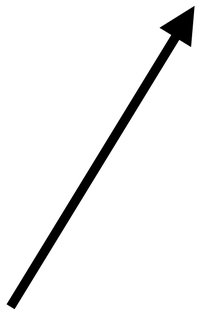
Cleaner API design and easier re-use.

```
function identify() {
  return this.name.toUpperCase();
}

function speak() {
  const greeting = "Hello, I'm " + identify.call( this );
  console.log( greeting );
}
```

```
identify.call( me ); // Kirell
identify.call( you ); // READER

speak.call( me ); // Hello, I'm Kirell
speak.call( you ); // Hello, I'm READER
```

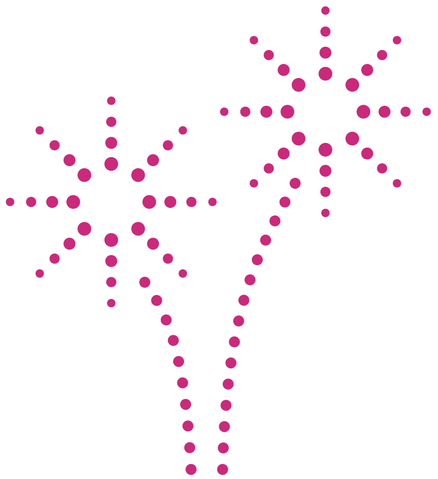
You Don't Know JS: this & Object Prototypes Ch.1

```
function createObject(value) {
  return {
      x: value,
      get: function () {
          return this.x;
      },
      set: function (newValue) {
          this.x = newValue;
      }
  };
}

let myObj = createObject(15);
myObj.x; // 15
myObj.get(); // 15
myObj.set(42);

myObj.get(); // 42
myObj.x; // 42
```



Gives context (“owns the function”)

Works as expected !?



```
function createObject(value) {
  return {
      x: value,
      get: function () {
          return this.x;
      },
      set: function (newValue) {
          this.x = newValue;
      }
  };
}

let myObj = createObject(15);

const objGetter = myObj.get;
objGetter(); // undefined
```

Alias of get()

Not the same context, fallback to global object, not defined

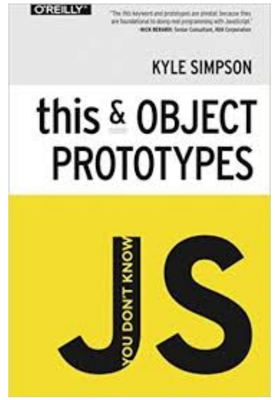
new keyword

A brand new object is created (aka, constructed) out of thin air

The newly constructed object is [[Prototype]] -linked (following slides)

The newly constructed object is **set as the this binding** for that function call unless the function returns its own alternate object

The new -invoked function call will automatically return the newly constructed object.



Real-life example

```
function Counter() {
  this.num = 0;
  // no return statement
}
const a = new Counter();
console.log(a.num); // 0
```

Real-life example

```
function Counter() {
  this.num = 0;
  this.timer = setInterval(function add() {
    this.num++;
    console.log(this.num);
  }, 1000);
  // no return statement
}
const b = new Counter();
// NaN
// NaN
// NaN
// ...
clearInterval(b.timer); // stop timer
```

How to fix this? (old)

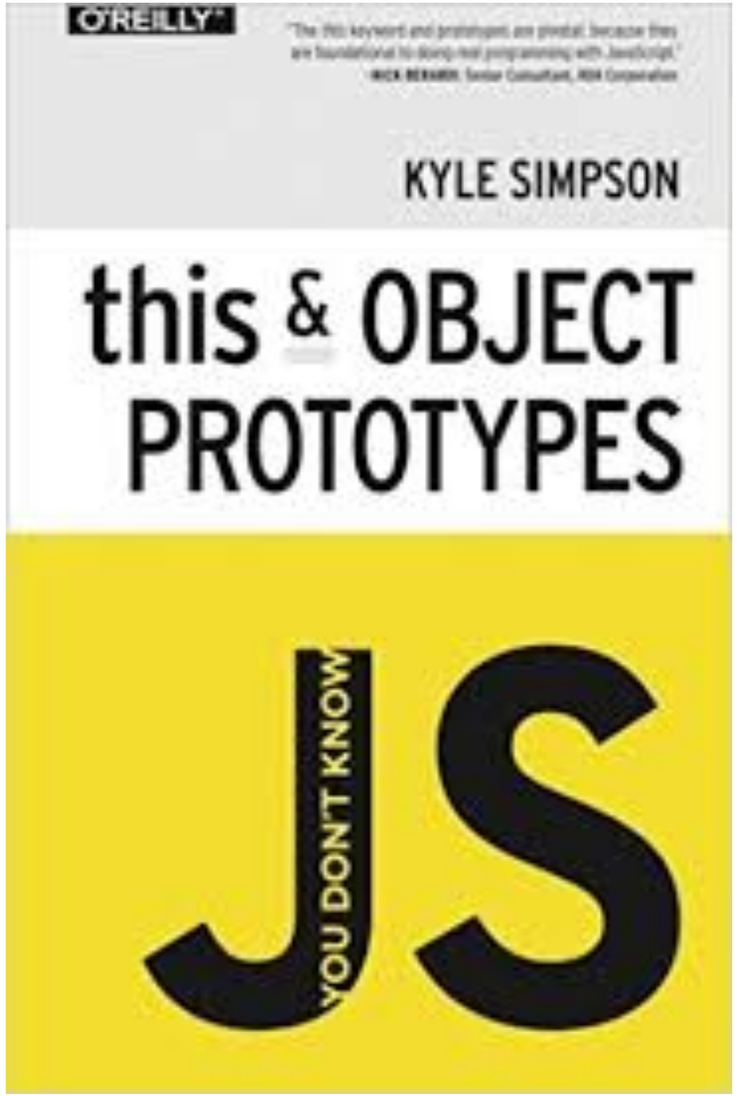
```
function Counter() {
  var that = this;  // old ES5 way
  that.num = 0;
  that.timer = setInterval(function add() {
    that.num++;
    console.log(that.num);
  }, 1000);
}
const c = new Counter();
// 1
// 2
// 3
// ...
```

How to fix this? (new)

```
function Counter() {
  this.num = 0;
  this.timer = setInterval( () => {
    this.num++;
    console.log(this.num);
  }, 1000);
}
const d = new Counter();
// 1
// 2
// 3
// ...
```

Arrow function => lexical this binding

More on this (pun intended)



Inheritance?

In traditional languages

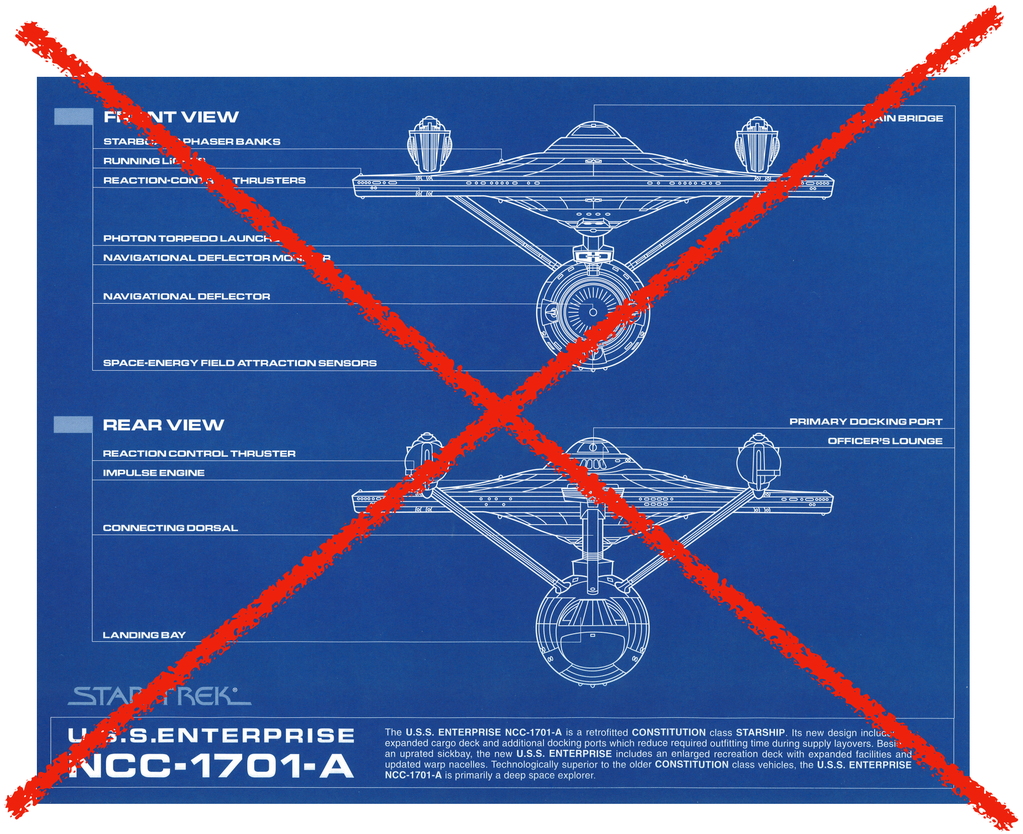
Classes are blueprints

Objects (instances) are copies of all the characteristics described by classes

In JS

There are just objects, linked together via their **prototypes**

Prototypes are the mechanism by which JavaScript objects inherit features from one another



Example: create a person

A Person “pseudo-factory” function

```
function createNewPerson(name) {
  let obj = {
    name: name,
    greeting: function() { console.log(`My name is ${this.name}!`); }
  };
  return obj;
}
const me = createNewPerson('Kirell');
me.greeting(); // My name is Kirell!
const you = createNewPerson('You');
```

Person ctor function?

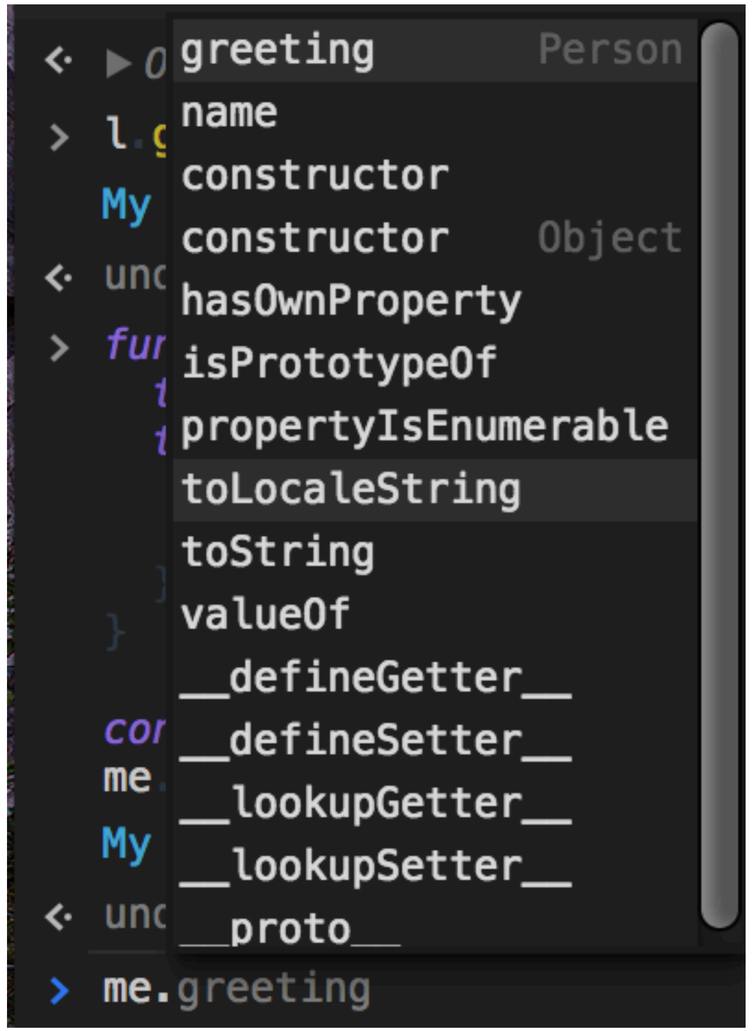
Let’s create a Person “class”. Capital case on the first letter to indicate a constructor function.

How can we share the greeting function between all Person objects?

```
function Person(name) {
  this.name = name;
  this.greeting = function() {
    console.log(`My name is ${this.name}!`);
  };
  // no return statement
}
const me = new Person('Kirell'); // use new keyword
me.greeting();
```

Enters the prototype object

Each object has a prototype object, which acts as a template object that it inherits methods and properties from. (Roughly)



https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

Prototype chain


```
> function Person(name) {
    this.name = name;
    this.greeting = function() {
      // Notice the backtick `` to have string substitution
      console.log(`My name is ${this.name}!`);
    };
  }

  const me = new Person('Kirell');
  me.greeting(); // My name is Kirell!
  My name is Kirell!

> me.valueOf()
<· ▶ Person {name: "Kirell", greeting: function}
> me.toString()
<·  [object Object]
```

These methods are available to the “**me**” object by walking the prototype chain upwards

They are defined in the **Object prototype**

Looking up the chain

```
function Person(name) {
  this.name = name;
}

const me = new Person('Kirell');

Person.prototype.greeting = function() {
  console.log(`My name is ${this.name}!`);
};

me.greeting(); // works: My name is Kirell!
```

Browser looks up the prototype chain

Array

See also

Standard built-in objects

Array

- Properties
  - Array.length
  - Array.prototype
  - Array.prototype[@@unscopables]
- Methods
  - Array.from()
  - Array.isArray()
  - Array.observe()
  - Array.of()
  - Array.prototype.concat()
  - Array.prototype.copyWithin()
  - Array.prototype.entries()
  - Array.prototype.every()

In This Article

The JavaScript **Array** object is a global object that is use
like objects.

Create an Array

```
var fruits = ['Apple', 'Banana'];

console.log(fruits.length);
// 2
```

Access (index into) an Array item

```
var first = fruits[0];
// Apple

var last = fruits[fruits.length - 1];
// Banana
```

Classical inheritance in JS

Before ES6 there were basically two ways

Constructor functions

Objects Linked to Other Objects (OLOO)

Object.create

```
const a = {a: 1};
// a ---> Object.prototype ---> null

const b = Object.create(a);
// b ---> a ---> Object.prototype ---> null
console.log(b.a); // 1 (inherited)

const c = Object.create(b);
// c ---> b ---> a ---> Object.prototype ---> null

const d = Object.create(null);
// d ---> null
console.log(d.hasOwnProperty);
// undefined, because d doesn't inherit from Object.prototype
```

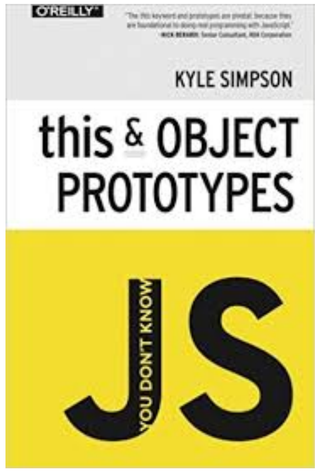
Object.create() method creates a new object with the specified prototype object and properties.

Constructor inheritance

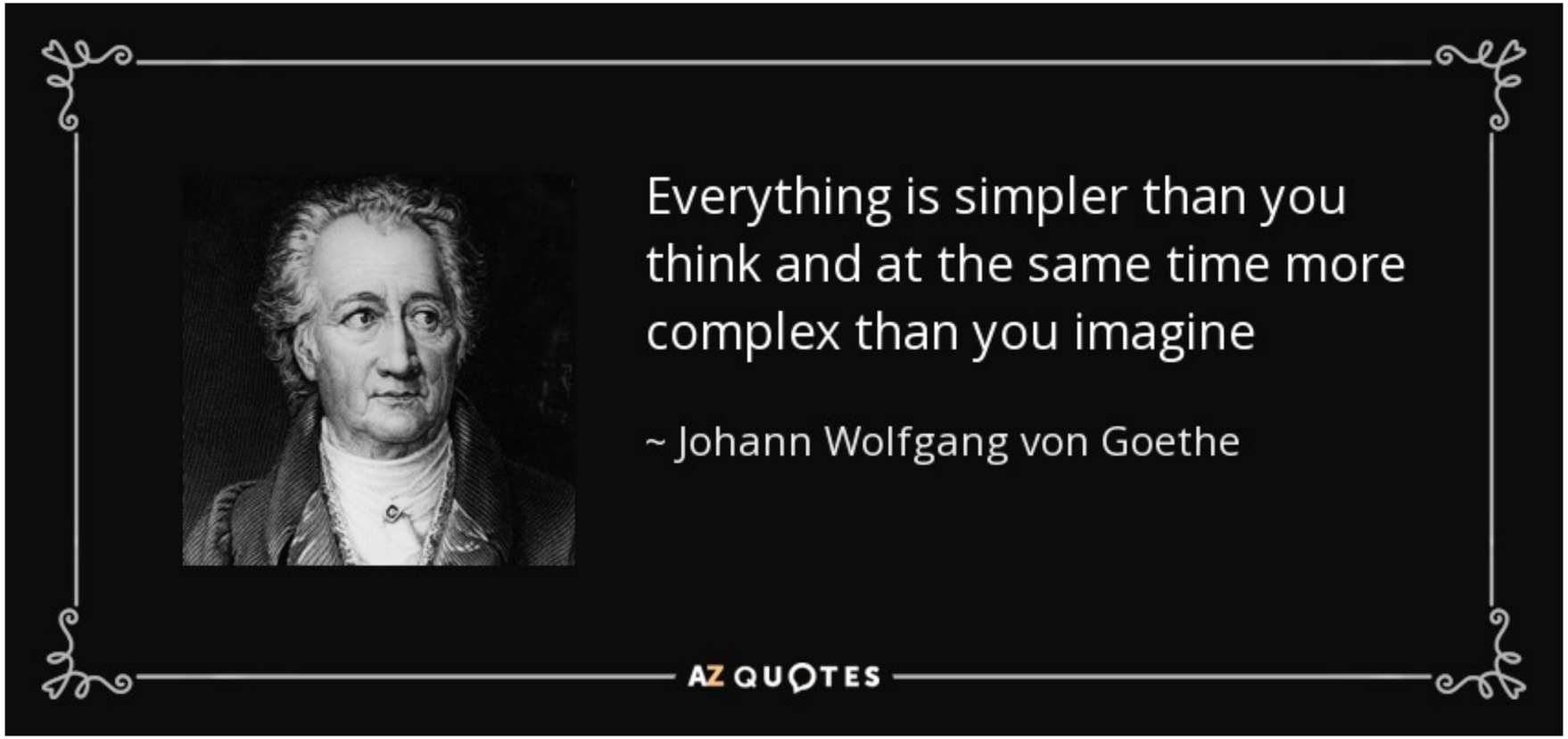
```
function Foo() {}
Foo.prototype.y = 11;

function Bar() {}
Bar.prototype = Object.create(Foo.prototype);
Bar.prototype.z = 31;

const x = new Bar();
x.y + x.z;  // 42
```

OLOO inheritance

```
const FooObj = { y: 11 };

let BarObj = Object.create(FooObj);
BarObj.z = 31;

const x = Object.create(BarObj);
x.y + x.z;  // 42
```

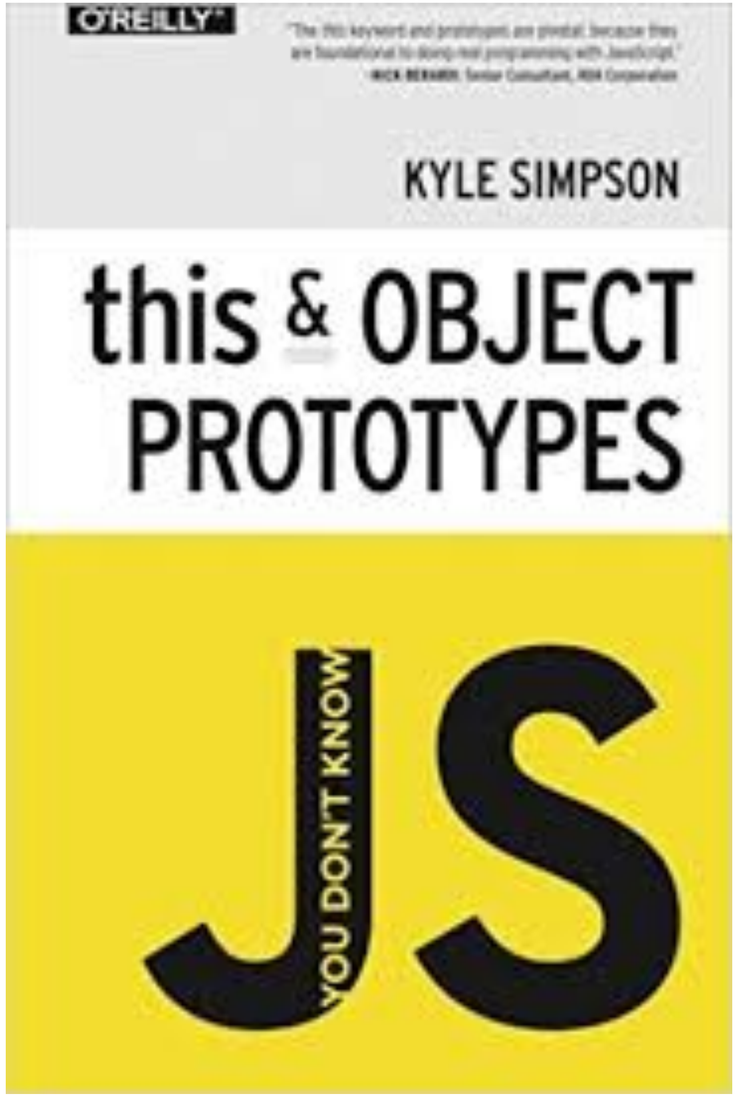


Prototypes are complex



https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

More on prototypes



ES2015 to the rescue

The traditional version of a “class” is useful and many people tried to emulate this features in Javascript

Since ES2015, new **syntaxic sugar** has been added to the language with new keyword: ***class*** to make our life easier.

The class syntax is not introducing a new object-oriented inheritance model to JavaScript!

Back to something familiar!

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  get area() {
    return this.calcArea();
  }
  calcArea() {
    return this.height * this.width;
  }
}

const square = new Rectangle(10, 10);
console.log(square.area);
```

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;
    return Math.hypot(dx, dy);
  }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);
console.log(Point.distance(p1, p2));
```

Inheritance is simpler (don’t abuse it)

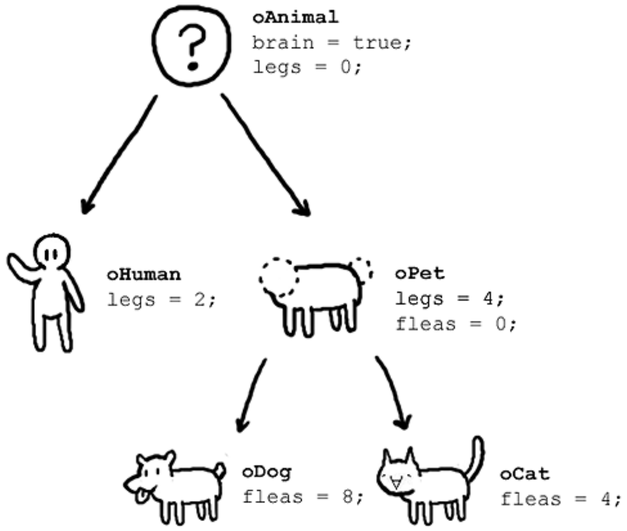
```
class Cat {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' makes a noise.');
  }
}

class Lion extends Cat {
  speak() {
    super.speak();
    console.log(this.name + ' roars.');
  }
}
```


```
const l = new Lion('Fuzzy');
l.speak();
// Fuzzy makes a noise.
// Fuzzy roars.
```

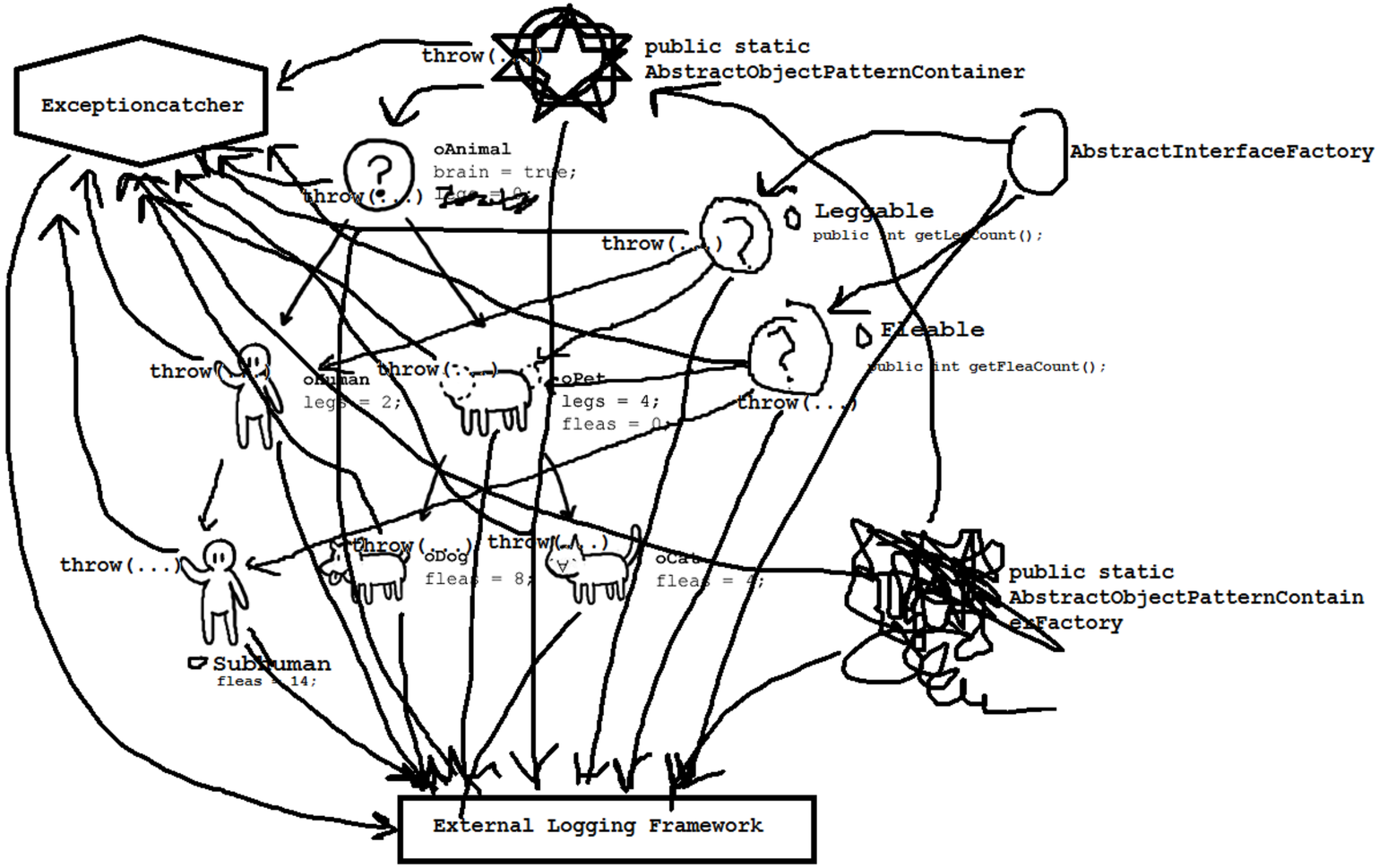
Favor composition over inheritance

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

What OOP users claim



What actually happens



No Silver Bullet

No Silver Bullet – Essence and Accident in Software Engineering by Fred Brooks in 1986

“there is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity.”

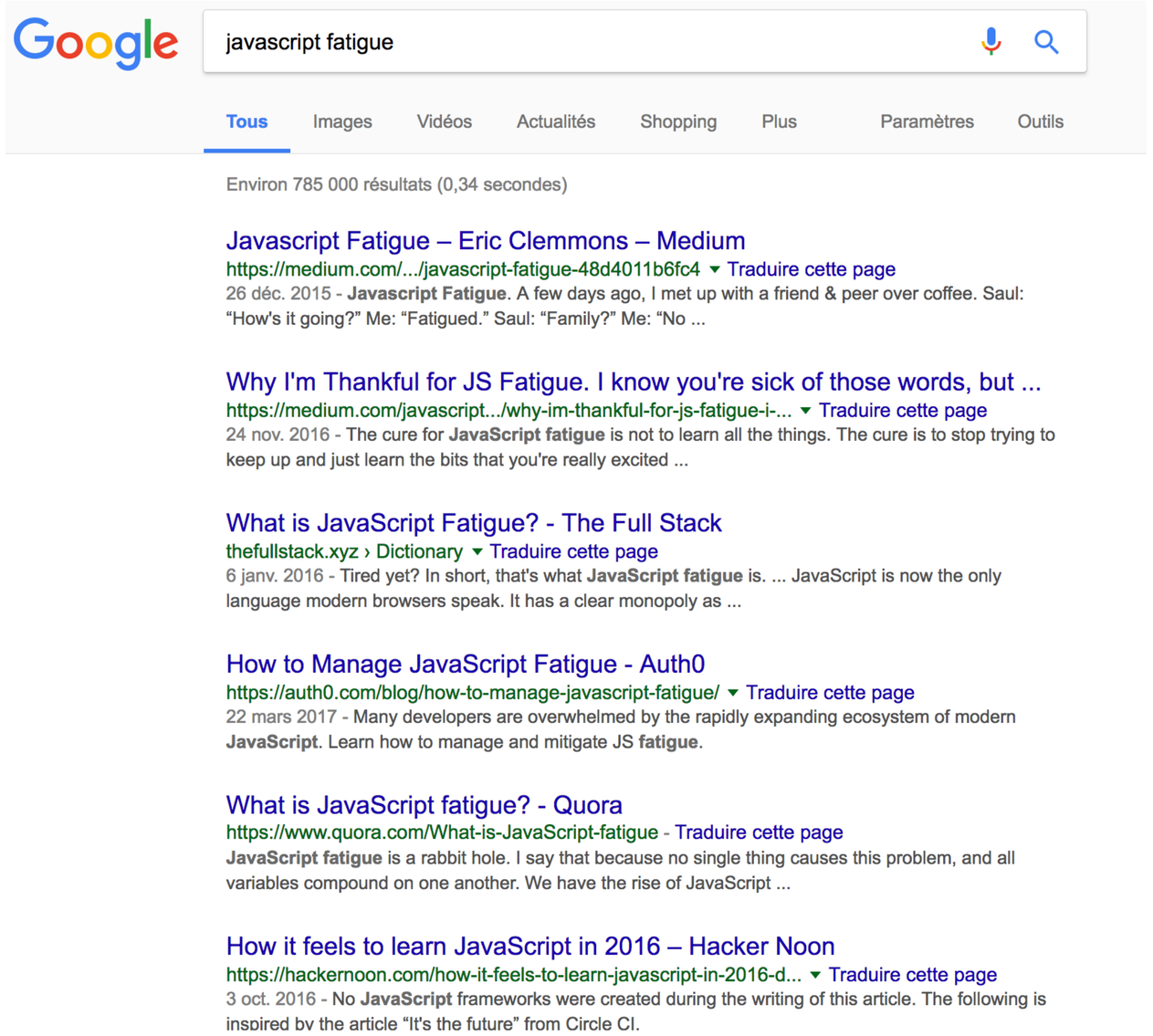
Use classes and functions altogether

Javascript ecosystem

The ecosystem is extremely rich in terms of tools, libraries or frameworks

It is so rich that a new term has been coined out: The javascript fatigue

There are so many things to know every year around JS => people are tired



open-source, cross-platform run-time environment for executing **JavaScript code server-side**.

event-driven architecture, asynchronous I/O

optimize throughput and scalability in Web applications with many input/output operations

Node is used in many tools to create and deploy JS apps

npm is the default and most popular package manager for Node.js

Command line client, ***npm***, searches from public database of Javascript modules

Much easier to share and reuse code

Takes care of updating the libraries you're using to the latest version (or not)

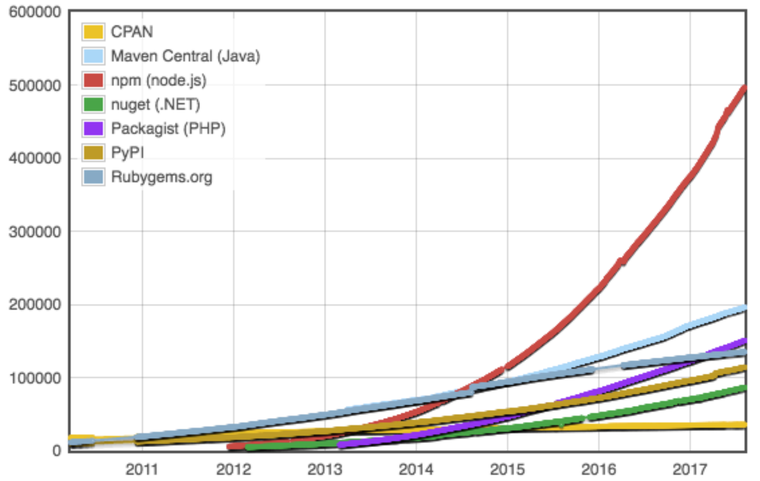
npm has a lot of modules

Most of them are low-quality

The number of modules is not a sign of greatest programming

However, it shows that the community is very active!

Module Counts



Include

- [ ] Clojars (Clojure)
- [x] CPAN
- [ ] CPAN (search)
- [ ] CRAN (R)
- [ ] Crates.io (Rust)
- [ ] Drupal (php)
- [ ] DUB (dlang)
- [ ] Gopm (go)
- [ ] Hackage (Haskell)
- [ ] Hex.pm (Elixir/Erlang)
- [ ] Julia
- [ ] LuaRocks (Lua)
- [x] Maven Central (Java)
- [ ] MELPA (Emacs)
- [x] npm (node.js)
- [x] nuget (.NET)
- [x] Packagist (PHP)
- [ ] Pear (PHP)
- [ ] Perl 6 Ecosystem (perl 6)
- [x] PyPI
- [x] Rubygems.org
- [ ] Vim Scripts

time period (•) all time ( ) last year ( ) last 90 days ( ) last 30 days ( ) last 7 days

	Aug 1	Aug 2	Aug 3	Aug 4	Aug 5	Aug 6	Aug 7	Avg Growth
Clojars (Clojure)	19500			19515		19519	19519	3/day
CPAN	35515			35537		35546	35551	6/day
CPAN (search)	35515			35538		35545	35551	6/day
CRAN (R)	11161			11179		11191	11195	6/day
Crates.io (Rust)	10588			10639		10666	10680	15/day
Drupal (php)	38190			38218		38232	38241	8/day
DUB (dlang)						1090	1091	1/day
Gopm (go)	19318			19330		19338	19340	4/day
Hackage (Haskell)	11614			11623		11631	11636	4/day
Hex.pm (Elixir/Erlang)	4603			4621		4631	4632	5/day
Julia	1478			1485		1489	1494	3/day
LuaRocks (Lua)	1501			1506		1508	1508	1/day
Maven Central (Java)				195714		195814	195879	55/day
MELPA (Emacs)	3713			3713		3713	3713	0/day
npm (node.js)	493505	494137	494696	495339	495729	496056	496511	508/day
nuget (.NET)	85765			86021		86147	86183	70/day
Packagist (PHP)	149699			150144		150351	150489	132/day
Pear (PHP)	602			602		602	602	0/day
Perl 6 Ecosystem (perl 6)	859			862		864	863	1/day
PyPI	113602			113857		113973	114034	72/day
Rubygems.org	134764			134888		134929	134951	31/day
Vim Scripts	5465			5465		5465	5465	0/day

http://www.modulecounts.com/

JS for desktop apps

The most popular framework to build cross-plateform apps with JS is Electron

Created by Github, it was initially developed for Github's Atom editor

Atom is a text editor that's modern, approachable, yet hackable to the core—a tool you can customize to do anything but also use productively without ever touching a config file.

https://atom.io/

Babel

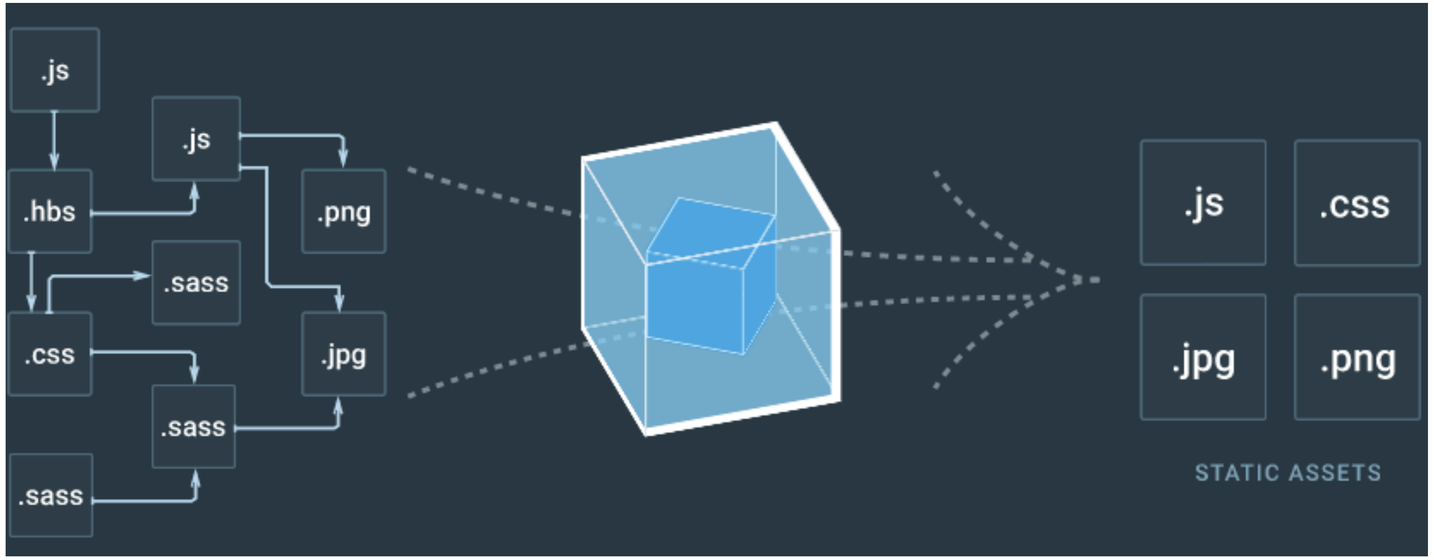

Babel is a JavaScript compiler.
Use next generation JavaScript, today.
Put in next-gen JavaScript
[1, 2, 3].map(n => n ** 2);
Get browser-compatible JavaScript out
[1, 2, 3].map(function (n) {
return Math.pow(n, 2);
});
Check out our REPL to experiment more with Babel!

Webpack

Webpack is a module bundler for JavaScript applications.

Recursively builds a dependency graph that includes every module your application needs.

Packages all of those modules into a small number of bundles to be loaded by the browser.



Including JS code in a HTML page

```
<script type="text/javascript" src="myscript.js"></script>

<script type="text/javascript">
 // run this function when the document is loaded
  window.onload = () => {
   …
  }
</script>
```

Include JS code in <head> or just before </body> not to slow the display of HTML elements

Organizing complex code

As our codebase grows, we need a simple way to organize code

ES2015 introduces standard modules to separate and isolate blocks of relevant code

```
// lib.js
// Can only have one default export
export default function myfunc() {
  return 'whatever';
};
export function anotherOne() {
  return 'anotherOne';
};

// main.js
import myfoo from 'lib'; // name is irrelevant since it is a default export
import {anotherOne} from 'lib'; // named import
```

What is not exported is “private “ to lib.js

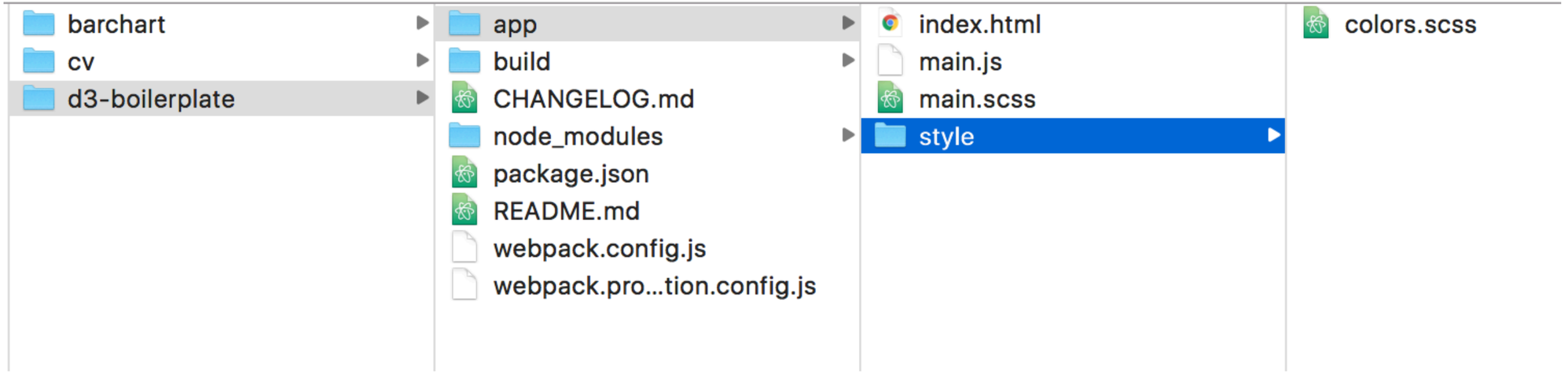
Organizing complex code

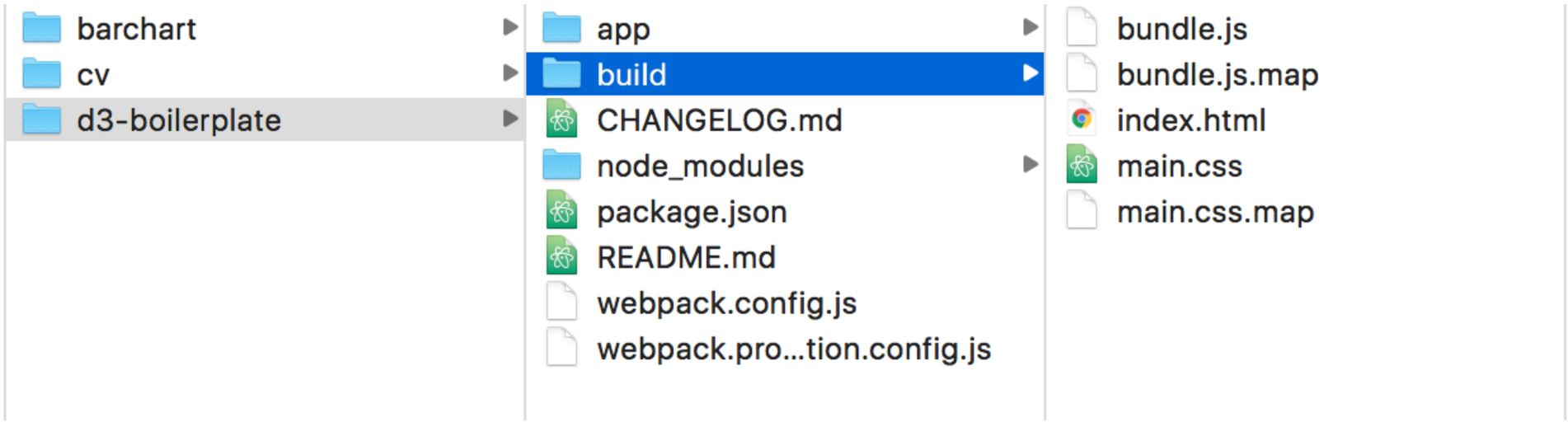
Browsers are only accepting modules natively

Only works with the latest versions

Current practices rely on the combination of a transpiler and bundle manager to create a single app.js / bundle.js file

Anatomy of a JS app





Homework

Read Mozilla tutorial

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_JS

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Inheritance