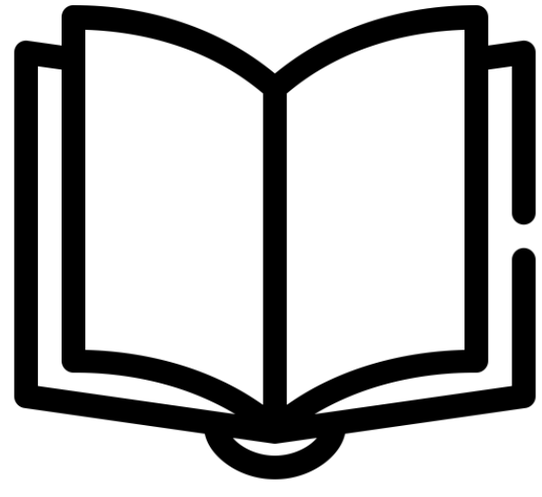
JavaScript: the language of browser interactions

Claudia Hauff
cse1500-ewi@tudelft.nl

TU Delft

Take-aways of chapter 4



- JavaScript basics (variables, functions, …)
- JavaScript as part of a web application
- Placement of `<script>`
- `strict` mode
- DOM
- jQuery basics

No emphasis on JavaScript objects.

Learning goals

- **Employ** JavaScript objects
- **Employ** the principle of callbacks
- **Write** interactive web applications based on click, mouse and keystroke events
- **Explain** and use `jQuery` (JavaScript library introduced in the book)

Not exam material!

A bit of context

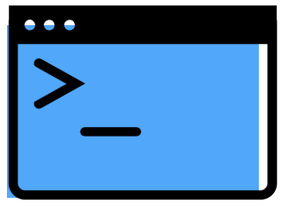
JavaScript's development

- Years ago it was considered a toy language
- Now: most important language of the **modern web stack**
- **Tooling** has vastly improved (debuggers, testing frameworks, etc.) - vital for a **dynamic** language
- JavaScript **runtime engines** are efficient :
  V8 - Google, SpiderMonkey - Mozilla, Chakra* - Microsoft
- JavaScript tracks **ECMAScript (ES)**

*Microsoft is moving to Chromium.

JavaScript: an interpreted language

- JavaScript runtime engine **interprets** JavaScript code line by line

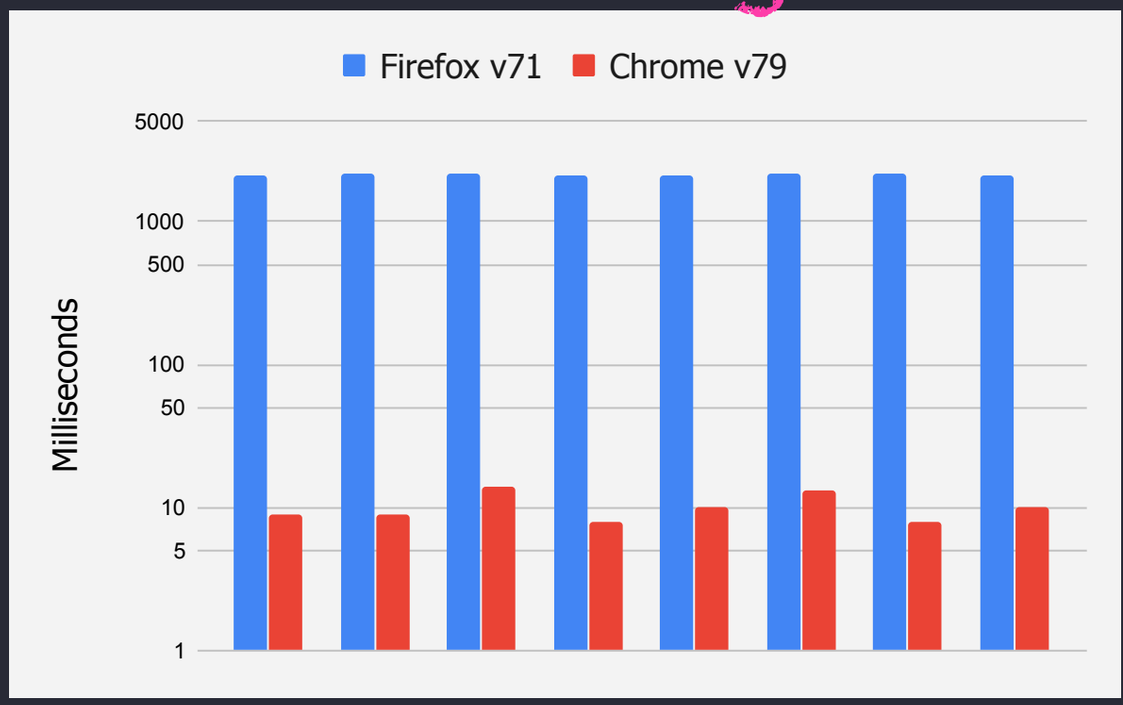


- Advantages: **quick upstart**, read-eval-print loop (REPL)
- Disadvantage: programs run slower than those written in a language that is compiled
- Modern JavaScript engines both interpret and compile: **just-in-time (JIT) compilation**

```
var limit = 10000000; //10 million
var t1, t2;

function Player(n, a) {
  this.name = n;
  this.age = a;
}

function func() {
  for (let i = 0; i < limit; i++) {
    let p = new Player("Sam", 21);
  }
}

t1 = performance.now();
func();
t2 = performance.now();
console.log("Time to create Player objects: %d ms.", t2 - t1);
```

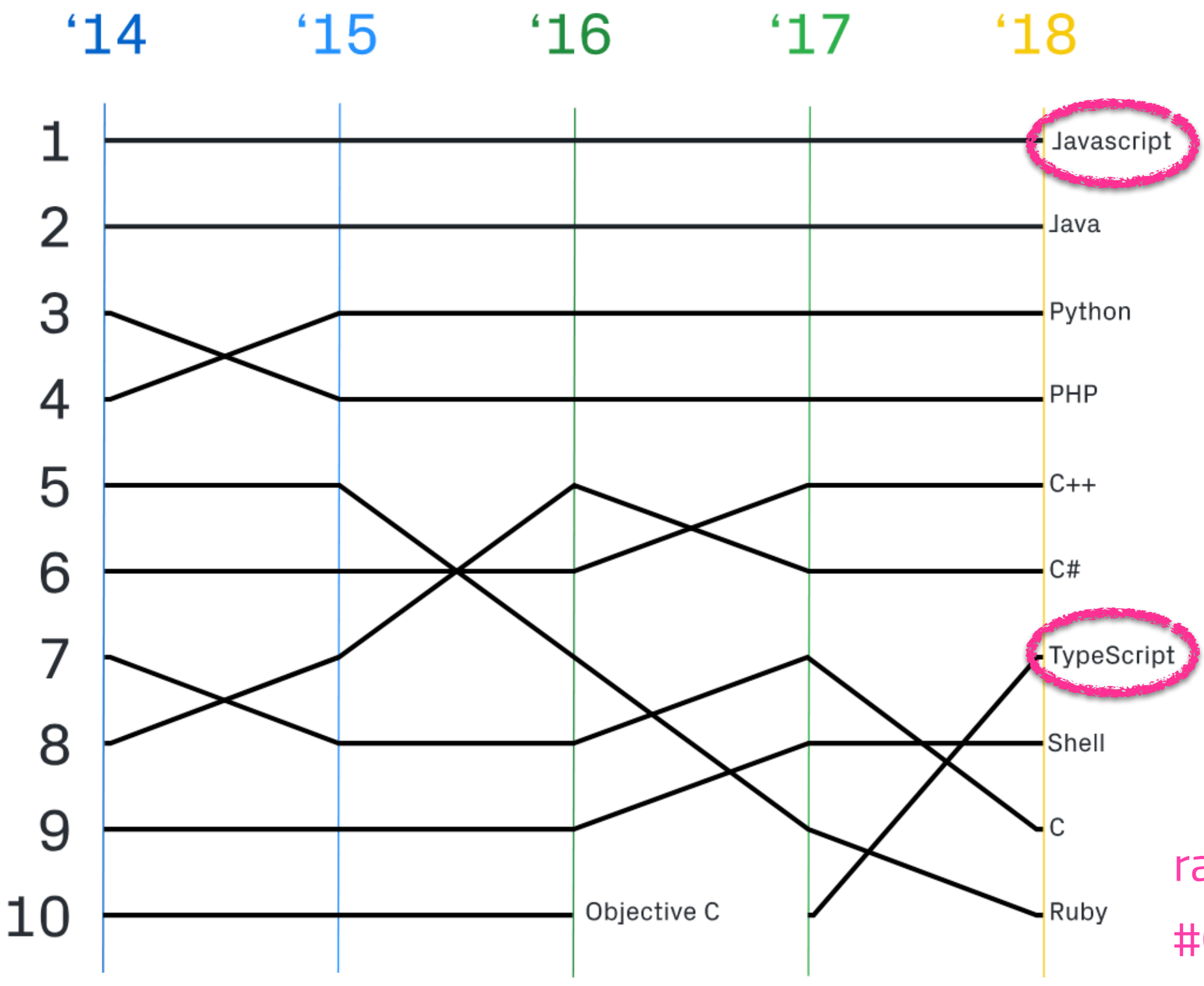
dynamic language: no types enforced

log scale!

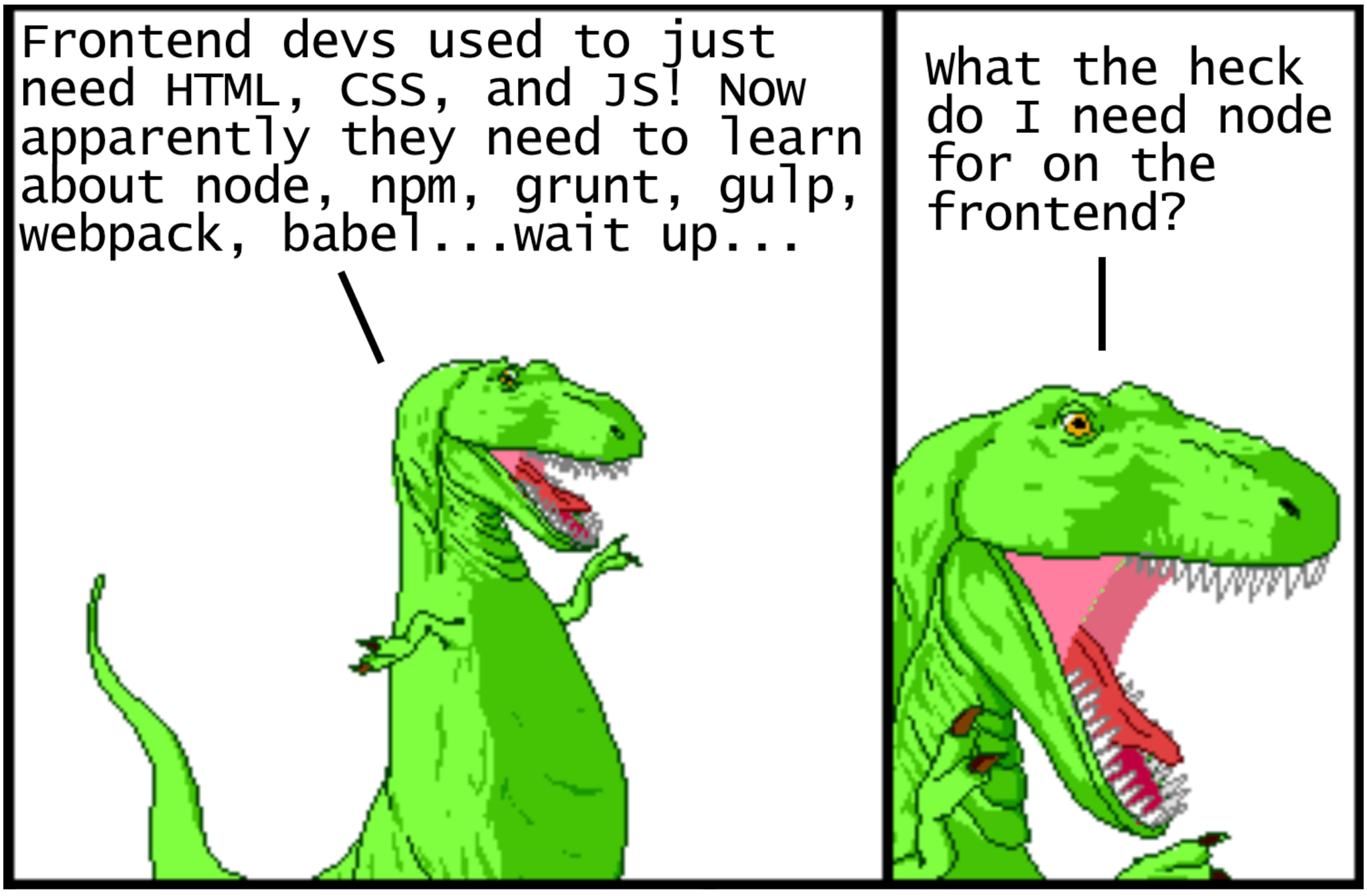


measuring object creation time

JavaScript's importance



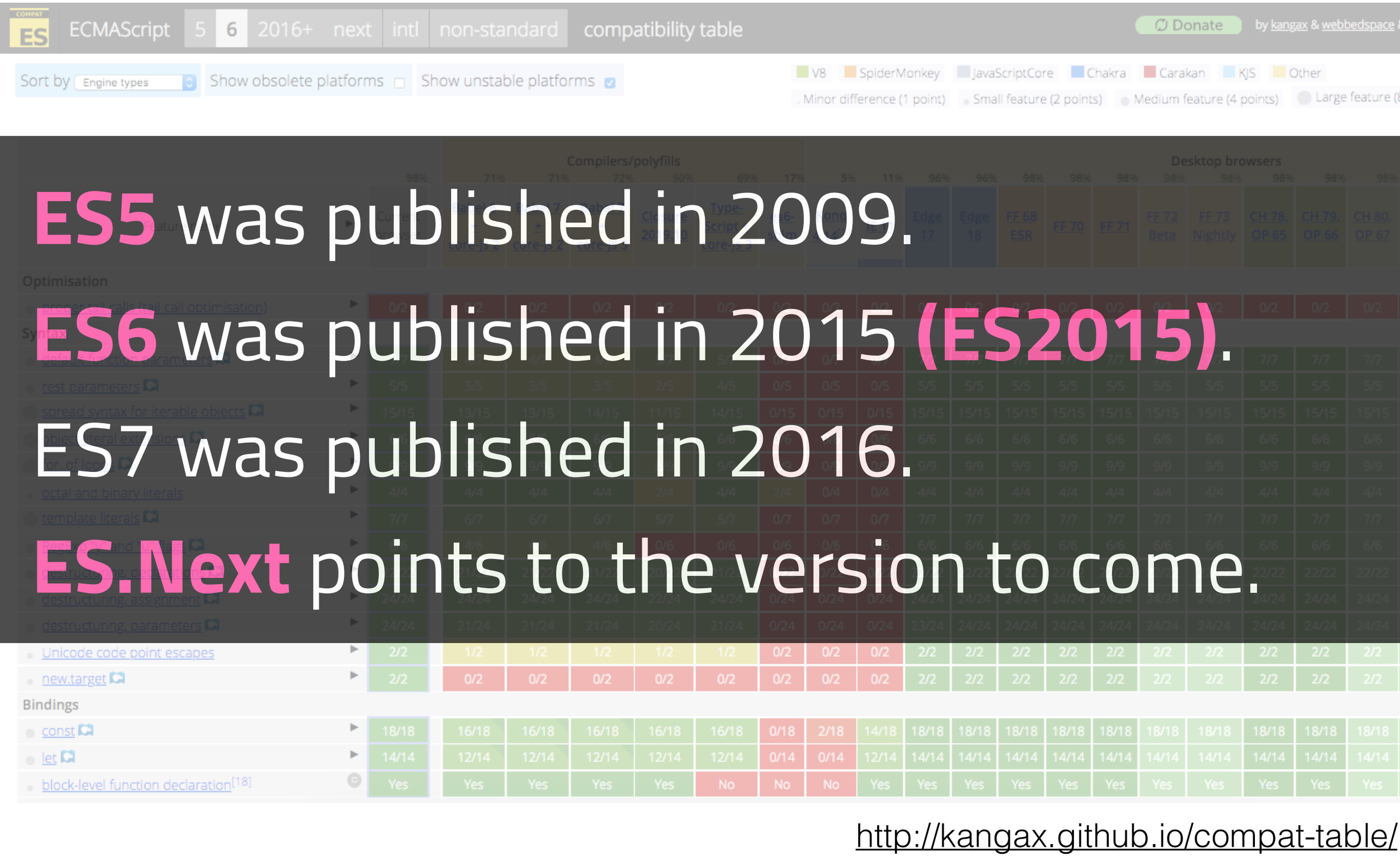
ranking based on
#GitHub repositories



“Learning modern JavaScript is tough if you haven’t been there since the beginning.”

https://medium.com/the-node-js-collection/modern-javascript-explained-for-dinosaurs-f695e9747b70

A language in flux



ES5 was published in 2009.

ES6 was published in 2015 **(ES2015)**.

ES7 was published in 2016.

ES.Next points to the version to come.

http://kangax.github.io/compat-table/

A language in flux

ES COMPAT | ECMAScript | 5 | 6 | 2016+ | next | intl | non-standard | compatibility table | Donate | by kangax & webbedspace

Sort by: Engine types | Show obsolete platforms ☐ | Show unstable platforms ☑

V8 | SpiderMonkey | JavaScriptCore | Chakra | Carakan | KJS | Other

Minor difference (1 point) | Small feature (2 points) | Medium feature (4 points) | Large feature (8

		Compilers/polyfills						Desktop browsers											
	98%	71%	71%	72%	50%	69%	17%	5%	11%	96%	96%	98%	98%	98%	98%	98%	98%	98%	98%
Feature name ▸	Current browser	Babel 6 + core-js 2	Babel 7 + core-js 2	Babel 7 + core-js 3	Closure 2019.10	TypeScript + core-js 3	es6-shim	Kong 4.14[1]	IE 11	Edge 17	Edge 18	FF 68 ESR	FF 70	FF 71	FF 72 Beta	FF 73 Nightly	CH 78, OP 65	CH 79, OP 66	CH 80, OP 67
Optimisation																			
proper tail calls (tail call optimisation) ▸	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2
Syntax																			
default function parameters ▸	7/7	4/7	4/7	4/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7
rest parameters ▸	5/5	3/5	3/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
spread syntax for iterable objects ▸	15/15	13/15	13/15	14/15	11/15	14/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15
object literal extensions ▸	6/6	6/6	6/6	6/6	5/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6
for..of loops ▸	9/9	9/9	9/9	9/9	6/9	9/9	0/9	0/9	0/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9
octal and binary literals ▸	4/4	4/4	4/4	4/4	2/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
template literals ▸	7/7	6/7	6/7	6/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7
RegExp "y" and "u" flags ▸	6/6	4/6	4/6	4/6	0/6	0/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6
destructuring, declarations ▸	22/22	21/22	21/22	21/22	20/22	21/22	0/22	0/22	0/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22
destructuring, assignment ▸	24/24	24/24	24/24	24/24	22/24	24/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24
destructuring, parameters ▸	24/24	21/24	21/24	21/24	20/24	21/24	0/24	0/24	0/24	23/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24
Unicode code point escapes ▸	2/2	1/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
new.target ▸	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
Bindings																			
const ▸	18/18	16/18	16/18	16/18	16/18	16/18	0/18	2/18	14/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18	18/18
let ▸	14/14	12/14	12/14	12/14	12/14	12/14	0/14	0/14	12/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14	14/14
block-level function declaration[18] C	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

http://kangax.github.io/compat-table/

A language in flux

ECMAScript 5 6 2016+ **next** intl non-standard compatibility table — Donate — by kangax & webbedspace

This table shows proposals which have not yet been included in the current ECMAScript standard, but are at one of the maturity stages of the TC39 process.

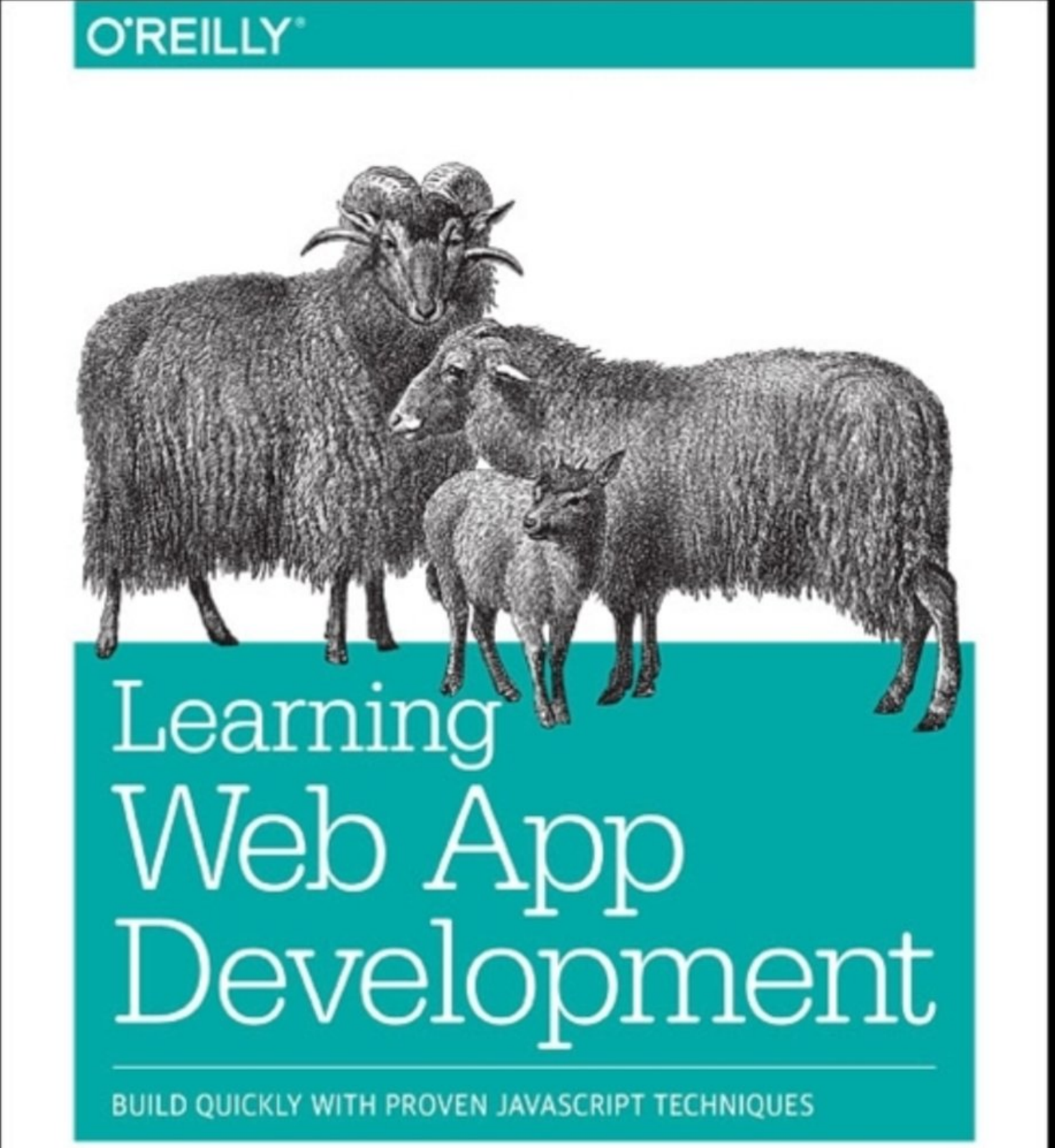
Sort by Engine types | Show obsolete platforms ☐ | Show unstable platforms ☑

V8 | SpiderMonkey | JavaScriptCore | Chakra | Other

Minor difference (1 point) | Small feature (2 points) | Medium feature (4 points) | Large feature (8

		Compilers/polyfills					Desktop browsers											
	4%	19%	26%	50%	0%	43%	0%	0%	1%	4%	4%	4%	4%	8%	8%	8%	8%	1%
Feature name ▸	Current browser	Babel 6 + core-js 2	Babel 7 + core-js 2	Babel 7 + core-js 3	Closure 2019.10	TypeScript + core-js 3	Edge 17	Edge 18	FF 68 ESR	FF 70	FF 71	FF 72 Beta	FF 73 Nightly	CH 78, OP 65	CH 79, OP 66	CH 80, OP 67	CH 81, OP 68	SF 12.1
Candidate (stage 3)																		
WeakReferences ▸	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2
instance class fields ▸	2/4	1/4	1/4	1/4	0/4	1/4	0/4	0/4	0/4	2/4	2/4	2/4	2/4	4/4	4/4	4/4	4/4	0/4
static class fields ▸	0/3	1/3	1/3	1/3	0/3	1/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	3/3	3/3	3/3	3/3	0/3
numeric separators	Yes	No	Yes	Yes	No	Yes	No	No	Flag[8]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
String.prototype.replaceAll	No	No	No	Yes[9]	No	Yes[11]	No	No	No	No	No	Flag[8]	Flag[8]	No	No	Flag[12]	Flag[12]	No
Promise.any	No	No	No	Yes[9]	No	Yes[11]	No	No	No	No	No	Flag[8]	Flag[8]	No	No	No	No	No
Legacy RegExp features in JavaScript ▸	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
Draft (stage 2)																		
Generator function.sent Meta Property	No	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No
Class and Property Decorators ▸	0/1	0/1	0/1	0/1	0/1	1/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Realms	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
throw expressions ▸	0/4	0/4	4/4	4/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4
Set methods ▸	0/7	0/7	0/7	7/7	0/7	7/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7
ArrayBuffer.prototype.transfer ▸	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2
Map.prototype.upsert ▸	0/2	0/2	0/2	2/2	0/2	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2
Array.isTemplateObject	No	No	No	Yes[9]	No	Yes[11]	No	No	No	No	No	No	No	No	No	No	No	No
Iterator Helpers ▸	0/35	0/35	0/35	35/35	0/35	35/35	0/35	0/35	0/35	0/35	0/35	0/35	0/35	0/35	0/35	0/35	0/35	0/35

http://kangax.github.io/compat-table/

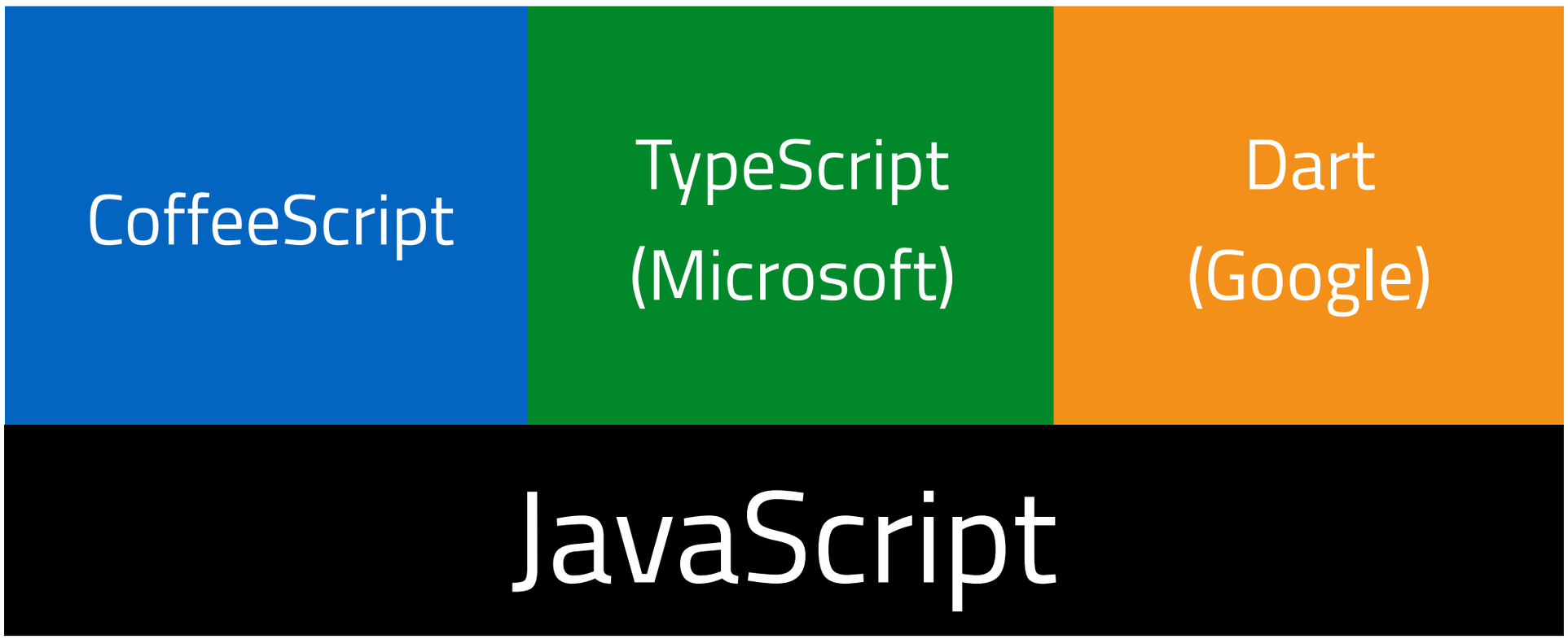


Published in 2014, thus no ES6.

Not a bug, but a feature!

ES6 introduced many features that make sense knowing the “old” JavaScript.

Compiling to JavaScript



All major (and not so major) languages compile to JS.

https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js

JavaScript vs. TypeScript

```
/* age is expected to be a number but we cannot enforce
 * this in plain JavaScript
 */
function increment(age){
  return (age+1);
}
```

In JavaScript, at runtime we can rely on the following snippet:

```
console.assert(Number.isInteger(age), “integer expected”);
```

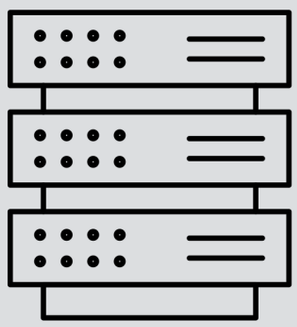
```
let a1 = 18;
a1 = increment(a1); //a1 -> 19

let a2 = "1";
a2 = increment(a2); //a2 -
```

```
let a2: string
Argument of type 'string' is not assignable to parameter of type
'number'. ts(2345)
Peek Problem    No quick fixes available
```

```
/*
 * TypeScript: function increment(age: number): number { ... }
 */
```

Scripting overview

Interactive web applications



2

execution of server-side scripts (PHP, ASP, Node.js, etc.)

1 browser sends a GET request to a web server

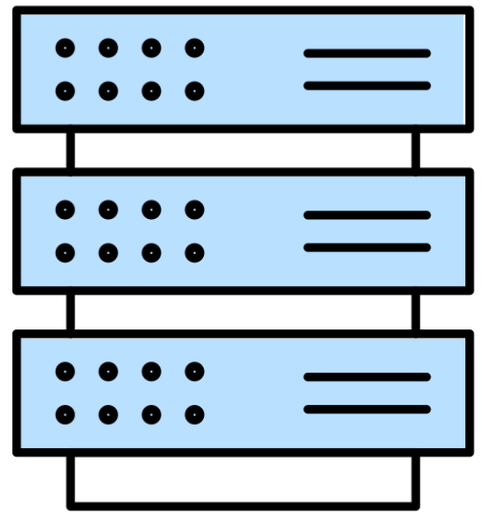
3 web server sends (generated) resources to the browser

4

execution of scripts (*.js) and rendering of the page

JavaScript makes web applications **interactive** and **responsive** to user actions.

Server-side scripting



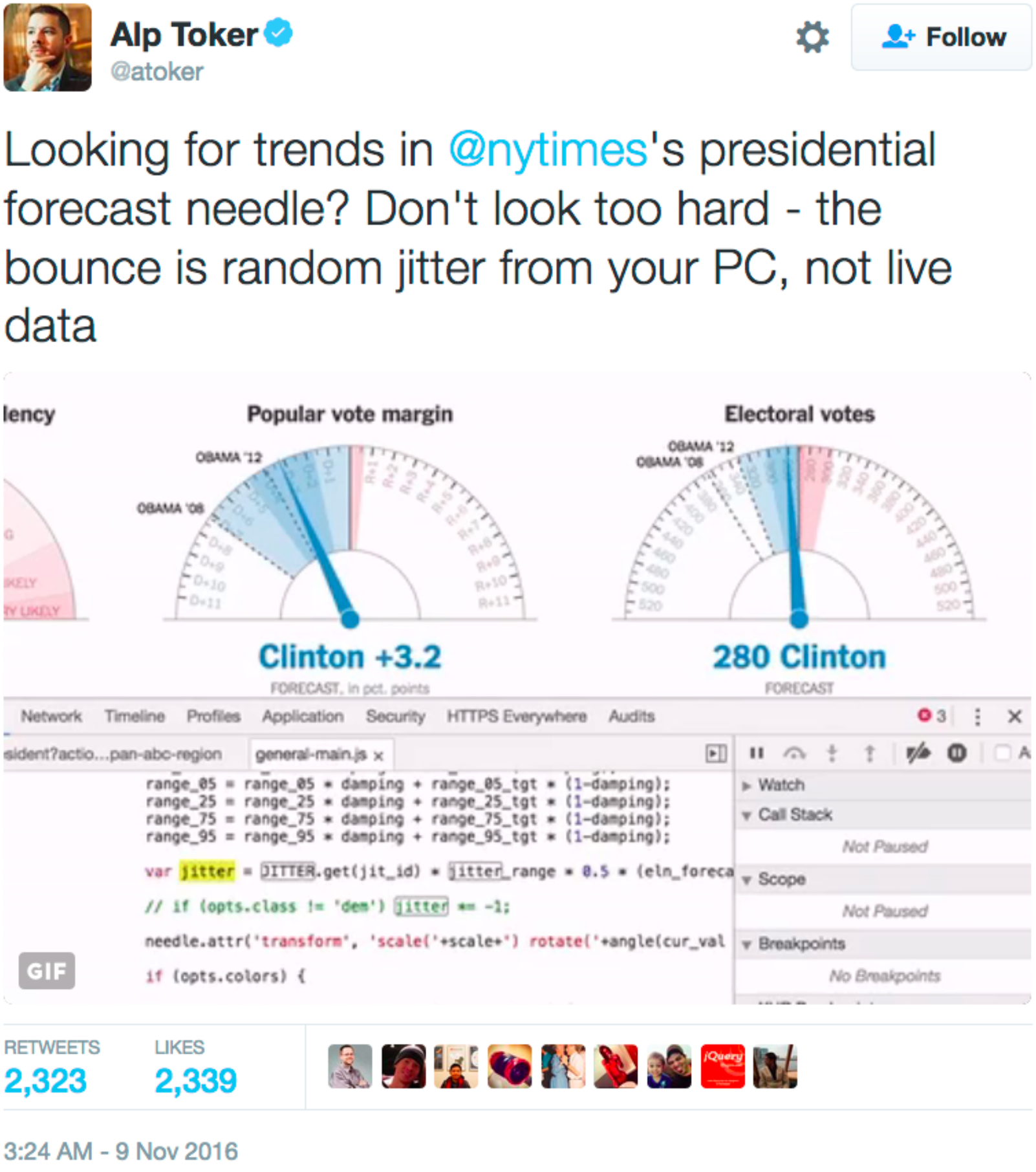
- Source code is **private**, only the result of a script's execution is returned
- HTML can be rendered by **any browser**
- Server-side scripts can **access additional resources** (e.g. databases)
- Server-side scripts can use **non-standard language features**

Client-side scripting

- Source code is **visible** to everyone
- Client-side script execution **reduces server load**
- All **raw data** needs to be downloaded and processed by the client (IndexedDB: powerful in-browser database system)
- JavaScript is **event-driven**: code blocks are executed in response to user actions

Client-side

- Source code is
- Client-side scri
- All **raw data** ne
  by the client (In
  database syste
- JavaScript is **ev**
  in response to



Hoisting, scoping and this

Hoisting principle:

declarations are processed before any code is executed.

Declarations are hoisted to the top (also inside of functions).

Expressions are not hoisted.

```
var x = six();

//function declaration
function six() {
  return 6;
}

var y = seven();

//function expression
var seven = function() {
  return 7;
};

var res = x + "" + y;
console.log(res);
```

Scoping: context in which values and expressions are visible.

JavaScript: global, local and block scope.

- `var` declared within a function: **local**
- `var` declared outside a function: **global**
- no `var`: **global** (no matter where)
- `let/const` were introduced in **ES6**: **block** (statements grouped together with curly brackets { ... })

```
player1 = "Sam";
var player2 = "Pat";

  {
    let player3 = "Matt";
    var player4 = "Andrea";
  }

function f() {
    var player5 = "Martin";
    const player6 = "Sarah";
  }

f();

console.log(player1); console.log(player2);
console.log(player3); console.log(player4);
console.log(player5); console.log(player6);
```

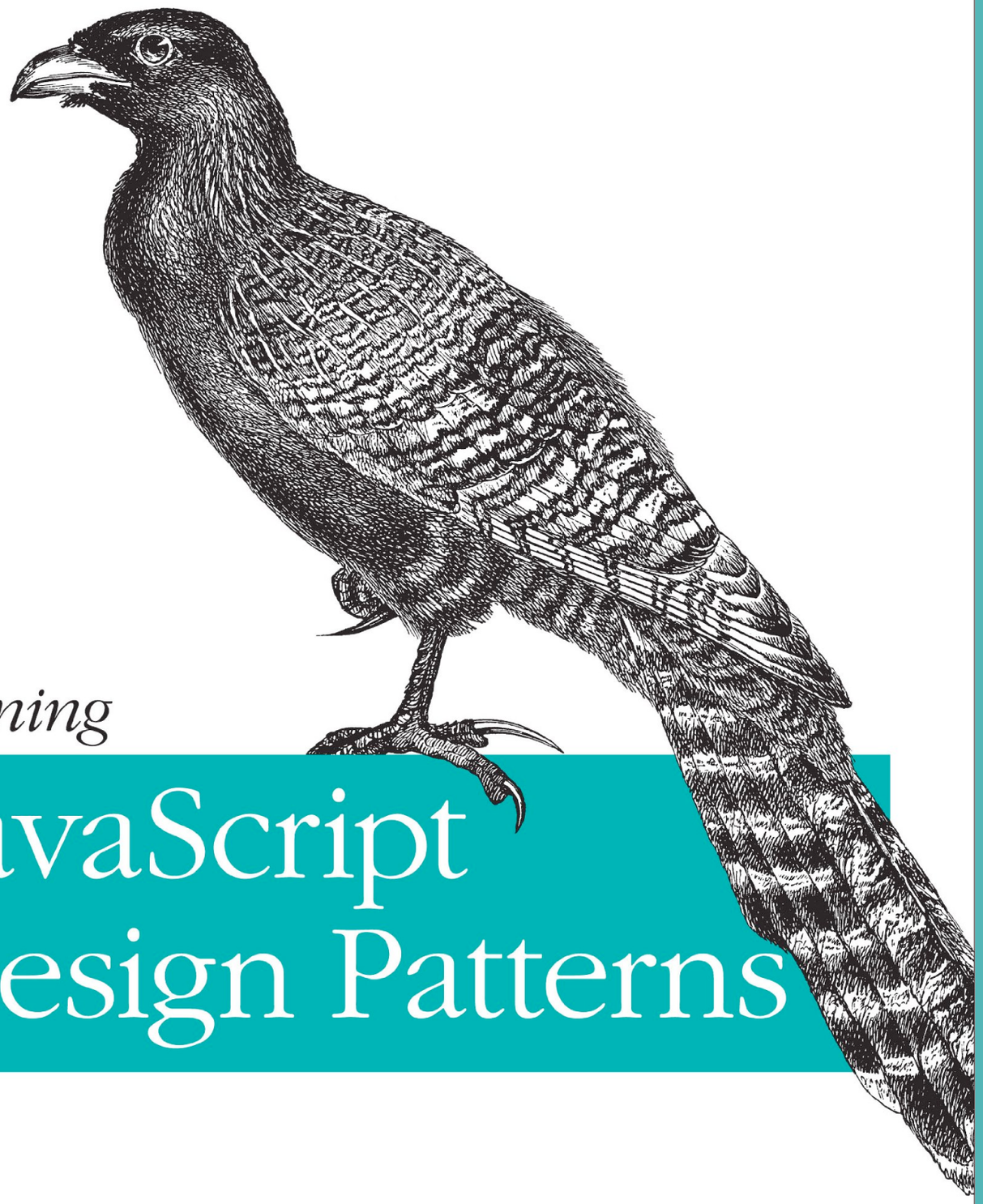
In JavaScript, what `this` refers to is dependent on **how** the function containing `this` was called.

- CASE 1: as a method of an object
  => "Reading"
- CASE 2: as a property of the global `window` object
  => "Sports"
- CASE 3: as a bound function
  => "Music"

`bind()`: set the value of a function's `this` independent of how it was called.

```
/* We assume execution in the browser's
 * Web Console, we thus know the global
 * window object exists (it is provided
 * by the browser).
 */

//h: property of the global `window` variable;
var h = "Sports";

function habit(s) {
  this.h = s;
  this.printHabit = function() {
    console.log(this.h);
  };
}

//CASE 1
var habitObj = new habit("Reading");
habitObj.printHabit();

//CASE 2
var printHabit = habitObj.printHabit;
printHabit();

//CASE 3
var boundPrintHabit =
    printHabit.bind({ h: "Music" });
boundPrintHabit();
```

A JavaScript and jQuery Developer's Guide



Learning

JavaScript Design Patterns

O'REILLY®

Addy Osmani

Objects in JavaScript

Basic constructor

Prototype-based constructor

Module pattern

JavaScript

> "A value has first-class status if it can be **passed** as a **parameter**, **returned** from a **subroutine**, or **assigned** into a **variable**." (Michael L. Scott)

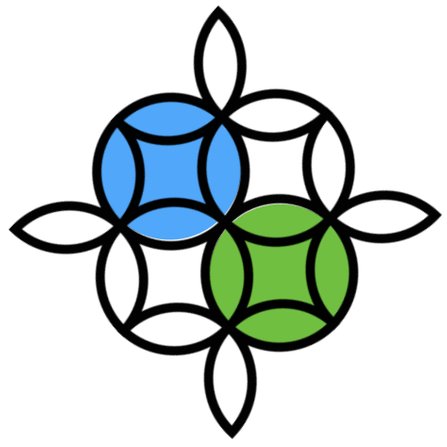
- JavaScript's **functions** (which are also objects) are **first-class citizens**

- Objects group together **related data and behaviour**

- Built-in objects: `String`, `Number`, `Array`, etc., but also `document` (when we talk about JS in the browser)

- **Objects can be created in ~7 different ways**

Design patterns



"Design patterns are **reusable solutions** to commonly occurring problems in software design." — **Addy Osmani**

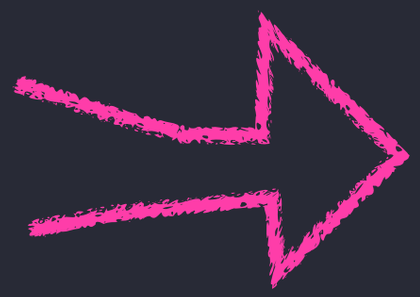
- Many design patterns exist, we focus on **three**
- Design patterns develop over time
- Design patterns often hold across programming languages

Object creation: new & obj. literals

- `new Object()` produces **an empty object**, ready to receive name/value pairs
  - Name (**property)**: a string
  - Value: anything apart from `undefined`
  - **Members** are accessed through
    - `[name]` (bracket notation)
    - `.name` (dot notation)

object literal

```
var game = new Object();
game["id"] = 1;
game["player1"] = "Alice";
game.player2 = "Bob";
console.log(game["player2"]); //prints "Bob"
console.log(game.player1); //prints "Alice"
```



```
var game = {
  id: 1,
  player1: "Alice",
  player2: "Bob"
};
```

```
var game = new Object();
game["id"] = 1;
game["player1"] = "Alice";
game.player2 = "Bob";

game["won lost"] = "1 12";

game.printID = function() {
  console.log(this.id);
};
game["printID"](); //prints out "1"
game.printID(); //prints out "1"
```

Adding a method is easy

```
////////////////////////////////////

var game = {
  id: 1,
  player1: "Alice",
  player2: "Bob",
  "won lost": "1 12",
  printID: function() {
    console.log(this.id);
  }
};
```

Object literals can be complex

```
var parameterModule = {
  /* parameter literal */
  Param: {
    minGames: 1,
    maxGames: 100,
    maxGameLength: 30
  },
  printParams: function() {
    console.table(this.Param);
  }
};
```

inner object
`Param`

(index)	Value
minGames	1
maxGames	100
maxGameLen…	30

Are object literals enough?

What happens if we need **1,000 objects** of this kind? What happens if a **method** needs to be **added to all objects**?

Design Pattern (1): Basic constructor

Constructors in Java vs. JS

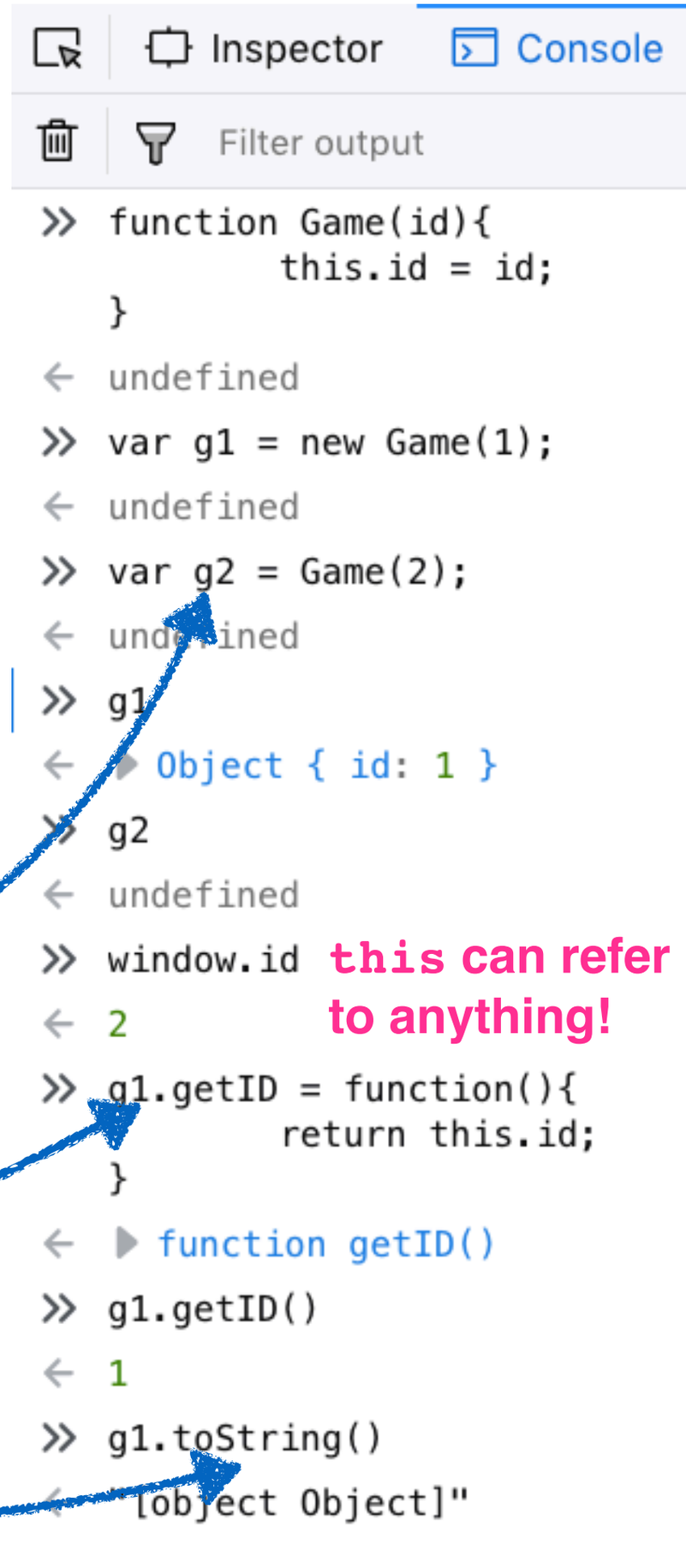
```java
public class Game {
    /* encapsulate private
     * members
     */
    private int id;

    /* constructor */
    public Game(int id){
        this.id = id;
    }

    public int getID(){
        return this.id;
    }

    public void setID(int id){
        this.id = id;
    }
}
```

- Functions as constructors
- `new` to initialise a new object

```javascript
function Game(id) {
  this.id = id;
  this.getID = function() {
    return this.id;
  };

  this.setID = function(id) {
    this.id = id;
  };
}
```

Basic constructor

- An object constructor is a **normal function**
- JavaScript runtime reacts to **`new`** keyword:
    1. anonymous empty object is created and used as **`this`**
    2. returns new object at the end of the function
- Object properties can be **added on the fly**
- Objects come with default properties (**prototype chaining**)



common error: forgetting "new"

Basic constructor

Remember:

In Java, this refers to the **current object.** In **JavaScript** what `this` refers to is dependent on **how the function containing this was called**.

Summary: basic constructor

- Advantage: **easy to use**
- Issues:

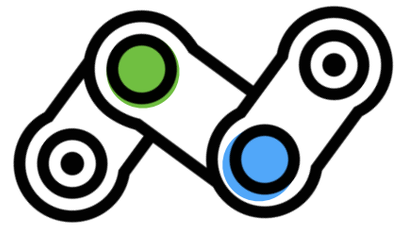
next …

- **Inheritance** (e.g. `TwoPlayerGame`)
- **Objects do not share functions**
- All members are **public:** any code can access, change or delete an object's properties

Design Pattern (2):

Prototype-based constructor

Prototype chaining explained



Objects have a **pointer** to another object: the object's **prototype**

- Properties of the **constructor's prototype** are also accessible in the new object

- If a property is not defined in the object, the **prototype chain** is followed

```
var name = “Daisy”;
typeof(name); //“string”
```

```
name.charAt(1)
```

__proto__

```
String.prototype
charAt()
indexOf()
…
```

Prototype-based constructor

```
function Game(id) {
  this.id = id;
}
/* new member functions are defined
 * just once in the prototype
 */
Game.prototype.getID = function() {
  return this.id;
};
Game.prototype.setID = function(id) {
  this.id = id;
};

var g1 = new Game("1");
g1.setID("2"); //that works!

var g2 = new Game("2");
g2.setID(3); //that works too!

g1["setID"] = function(id) {
  this.id = "ID" + id;
};

//g1 and g2 point to different setID properties
g1.setID(4);

console.log(g1.getID()); //ID4
console.log(g2.getID()); //3
```

Member functions are shared now

No more walking up the chain.

Inheritance through prototyping

```
function Game(id) {
  this.id = id;
}

Game.prototype.getID = function() {
  return this.id;
};
Game.prototype.setID = function(id) {
  this.id = id;
};

//constructor
function TwoPlayerGame(id, p1, p2) {
  /*
   * call(..) calls a function with a given
   * this value and arguments
   */
  Game.call(this, id);
  this.p1 = p1;
  this.p2 = p2;
}

//redirect prototype
TwoPlayerGame.prototype = Object.create(Game.prototype);
TwoPlayerGame.prototype.constructor = TwoPlayerGame;

//use it
var TPGame = new TwoPlayerGame(1, "Alice", "Bob");
console.log(TPGame.getID()); //prints out "1"
console.log(TPGame.p1); //prints out "Alice"
```

1. create a new constructor

2. redirect the prototype

= prototype chain

Inheritance through prototyping

```
function Game(id) {
  this.id = id;
}

Game.prototype.getID = function() {
  return this.id;
};
Game.prototype.setID = function(id) {
  this.id = id;
};

//constructor
function TwoPlayerGame(id, p1, p2) {
  /*
   * call(..) calls a function with a given
   * this value and arguments
   */
  Game.call(this, id);
  this.p1 = p1;
  this.p2 = p2;
}

//redirect prototype
TwoPlayerGame.prototype = Object.create(Game.prototype);
TwoPlayerGame.prototype.constructor = TwoPlayerGame;

//use it
var TPGame = new TwoPlayerGame(1, "Alice", "Bob");
console.log(TPGame.getID()); //prints out "1"
console.log(TPGame.p1); //prints out "Alice"
```

Firefox's Web Console

```
» TPGame
← ▼ {…}
    id: 1
    p1: "Alice"
    p2: "Bob"
  ▼ <prototype>: {…}
    ▶ constructor: function TwoPlayerGame()
    ▼ <prototype>: {…}
      ▶ constructor: function Game()
      ▶ getID: function getID()
      ▶ setID: function setID()
      ▼ <prototype>: {…}
        ▶ __defineGetter__: function __defineGetter__()
        ▶ __defineSetter__: function __defineSetter__()
        ▶ __lookupGetter__: function __lookupGetter__()
        ▶ __lookupSetter__: function __lookupSetter__()
          __proto__: »
        ▶ constructor: function Object()
        ▶ hasOwnProperty: function hasOwnProperty()
        ▶ isPrototypeOf: function isPrototypeOf()
        ▶ propertyIsEnumerable: function propertyIsEnumerable()
        ▶ toLocaleString: function toLocaleString()
        ▶ toSource: function toSource()
        ▶ toString: function toString()
        ▶ valueOf: function valueOf()
        ▶ <get __proto__()>: function __proto__()
        ▶ <set __proto__()>: function __proto__()
```

Summary: prototype-based constructor

- Advantages:
  - **Inheritance is easy** to achieve
  - **Objects share functions**
- Issue:
  - All members are **public:** any code can access, change or delete an object's properties

next …

Design Pattern (3):

Module

Scoping in practice

Task: print the numbers 1 to 10, each with a second delay

```
for (var i = 1; i <= 10; i++) {
  setTimeout(function() {
    console.log(i);
  }, 1000);
}
```

`setTimeout(fn,d)`:
executes function fn once after d ms

11 is printed ten times

```
function fn(i) {
  setTimeout(function() {
    console.log(i);
  }, 1000 * i);
}

for (var i = 1; i <= 10; i++) {
  fn(i);
}
```

callback

Works, but it looks tedious :(

```
for (let i = 1; i <= 10; i++) {
  setTimeout(function() {
    console.log(i);
  }, 1000 * i);
}
```

ES6 scoping to the rescue!

JavaScript scoping

```
function Game(p1, p2) {
  console.assert(typeof p1 === "string", "string expected");
  console.assert(typeof p2 === "string", "string expected");

  this.player1 = p1;
  this.player2 = p2;
  this.score = 0;
}

/*
 * game is complete if either player scores 100+ points;
 * pos. for p1, neg. for p2
 */

Game.prototype.isComplete = function() {
  return Math.abs(this.score) > 100;
};

let g1 = new Game("Alice", "Bob");
let g2 = new Game({ name: "Sam" }, "Rose"); //assertion will fail
```

What if another JavaScript library used in the project defines `Game`?

Module

- Goals:
  - **Do not declare global variables** unless required
  - Emulate **private/public** membership
    - Expose only the **necessary** properties (as API)
- Results:
  - Reduce potential **conflicts** with other libraries
  - Public API **minimises** unintentional side-effects

Module

Only the returned object has access to gamesStarted, etc. through JavaScript's concept of **closures**.

A closure is the combination of a function and the lexical environment within which that function was declared.

```
var statModule = (function() {
  //private members
  var gamesStarted = 0;
  var gamesCompleted = 0;
  var gamesAbolished = 0;

  //public members: return accessible object
  return {
    incrGamesStarted: function() {
      gamesStarted++;
    },
    getNumGamesStarted: function() {
      return gamesStarted;
    }
  };
})();

//using the module
statModule.incrGamesStarted();
console.log(statModule.getNumGamesStarted()); //prints out "1"
console.log(statModule.gamesStarted); //prints out "undefined"
```

public

private

Module

```
var statModule = (function(s, c, a) {
  //private members
  var gamesStarted = 0;
  var gamesCompleted = 0;
  var gamesAbolished = 0;

  //public members: return accessible object
  return {
    incrGamesStarted: function() {
      gamesStarted++;
    },
    getNumGamesStarted: function() {
      return gamesStarted;
    }
  };
})(15, 10, 5);

//using the module
statModule.incrGamesStarted();
console.log(statModule.getNumGamesStarted()); //prints out "1"
console.log(statModule.gamesStarted); //prints out "undefined"
```

The encapsulating function can contain arguments.

```
(function(){
  ...
})();
```

Immediately Invoked Function Expression (IIFE)

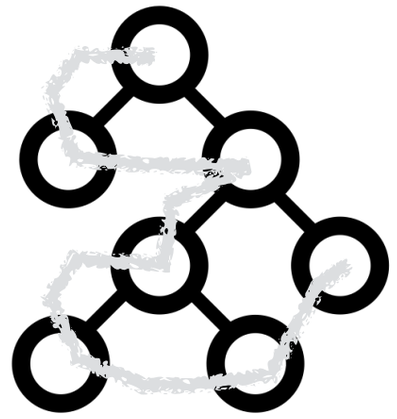
Summary: module

- Advantages:
  - **Encapsulation** is achieved
  - Object properties are **public** or **private**
- Issues:
  - Changing the membership type costs time
  - **Methods added on the fly later on cannot access 'private' members**

Events & the DOM

represents the structure of a document as a tree

A look at book chapter 4



```
var main = function() {
  "use strict";
  $(".comment-input button").on("click", function(event) {
    var $new_comment = $("<p>"),
      comment_text = $(".comment-input input").val();
    $new_comment.text(comment_text);
    $(".comments").append($new_comment);
  });
};
$(document).ready(main);
```

callback principle: we define what happens when an event fires

jQuery: uniform DOM element access pattern $(..)

The **`document` object** (*in the browser*) has different methods to access one or more DOM elements:

- `document.getElementById`, `document.getElementsByClassName`, `document.getElementsByTagName`
- **`document.querySelector`**: returns the first element within the DOM tree according to a **depth-first pre-order traversal** that matches the selector
- **`document.querySelectorAll`**: returns all elements that match the selector

```
                    });
                });
            });
        });
    });
});
```

callback hell

Step-by-step: making a responsive UI control

1. Pick a control (e.g. a button)

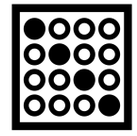
2. Pick an event (e.g. a click on a button)

3. Write a function: what happens when the event fires?

4. Attach the function to the event **on** the control

Interactions via the DOM

DOM: entrypoint to interactive web applications

- **Extract an element's state**
  - Is the Play button disabled?
  - Is the checkbox for level *easy* checked? ☑
  - Do four elements of class .red exist? 
- **Change an element's state**
  - Enable the button.
  - Create an element.
- **Change an element's style** (*later: CSS lecture*)
  - Change the colour of an element.
  - Change the position of an element.
  - Animate an element.

Client-side JS examples

Simple, small and self-contained.

Available in the lecture notes (not the PDF)!

`getElementById`

1

creating DOM nodes

2

`this`

3

mouse events

4

crowdsourcing interface

5

typing game

6

- Read **chapters 5 & 6** (Node.js) of the web development book. **Goal: get a first idea what Node.js is about.**

  Note: we will cover Node.js **from scratch** in the next lecture!

- Work on assignments 4 and 5!

flickr@courtneyrian