JavaScript: the language of browser interactions

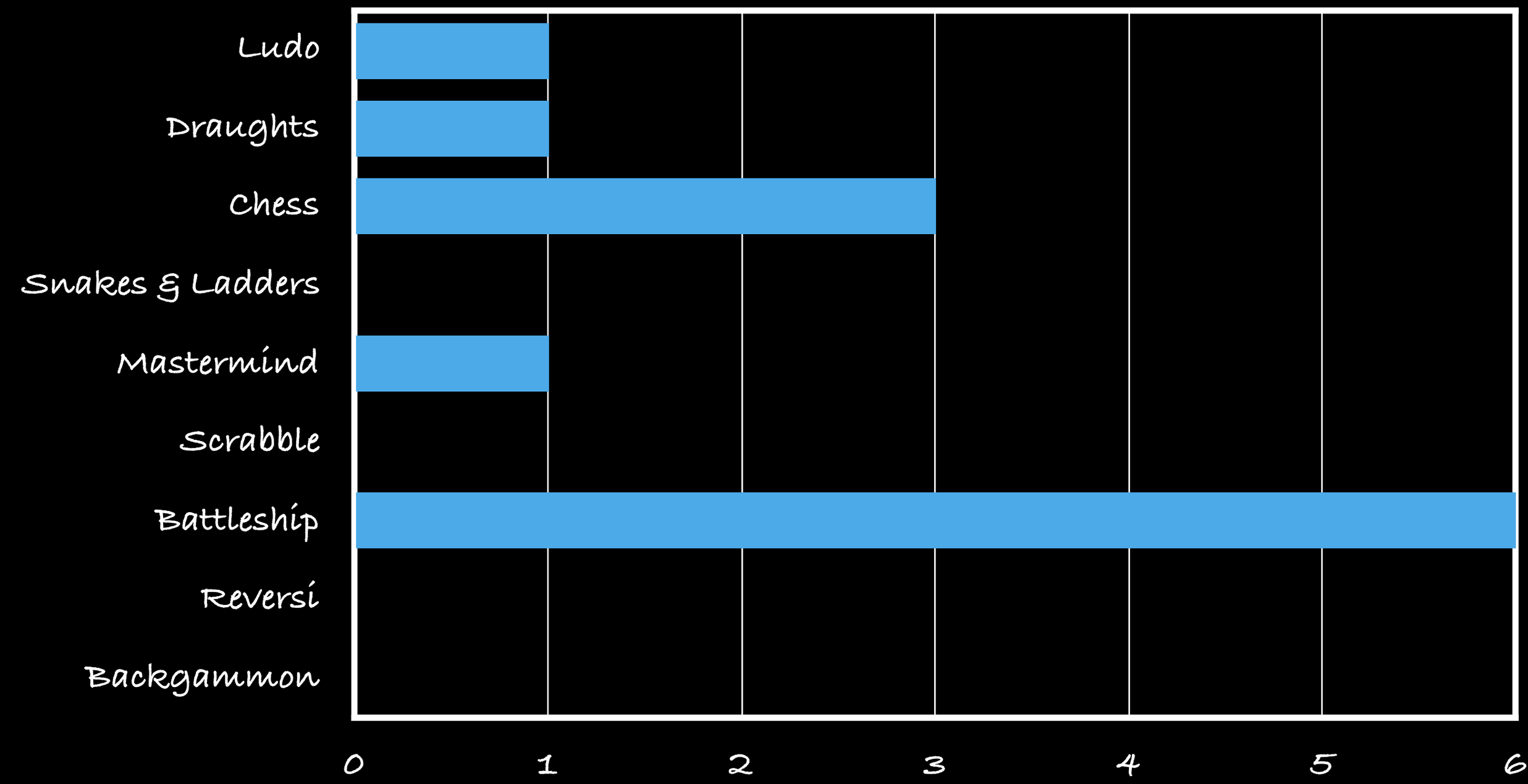
Claudia Hauff
cse1500-ewi@tudelft.nl

TU Delft

Web technology overview

1. HTTP: the language of web communication
2. HTML & web app design
3. **JavaScript**: interactions in the browser
4. **Node.js**: JavaScript on the server
5. **CSS**: adding style
6. Node.js: advanced topics
7. Cookies & sessions
8. Web security

12 submissions so far out of expected ~400



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.

Take-aways of book chapter 4

- JavaScript basics (variables, functions, …)
- JavaScript as part of a web application
- Where to place `<script>` and why
- `“use strict”`
- DOM
- jQuery basics

No emphasis on JavaScript objects.

A bit of context

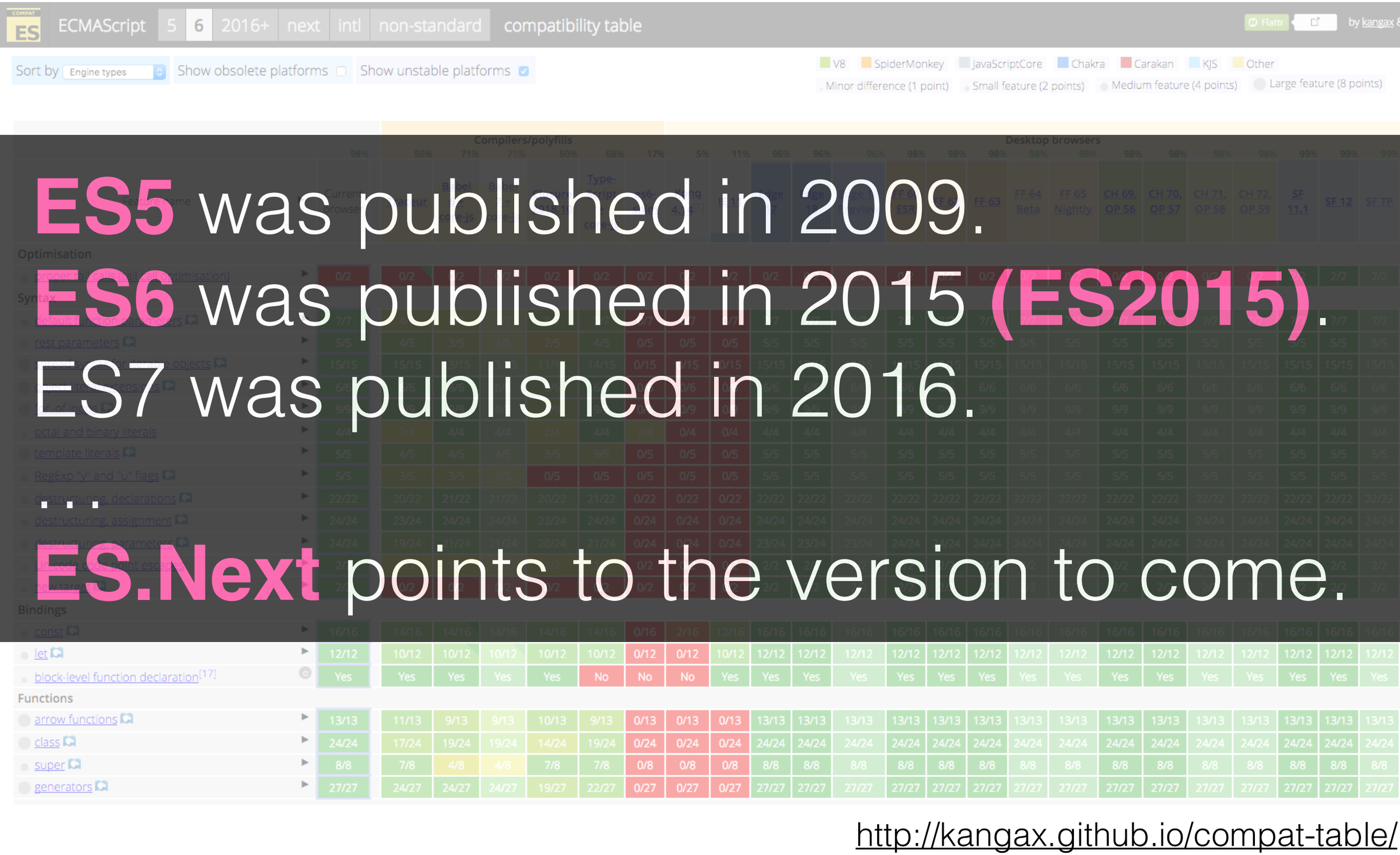
JavaScript's reputation

- Ten 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.)
- JavaScript **runtime engines** are efficient (V8, SpiderMonkey, Chakra)
- JavaScript tracks **ECMAScript (ES)**

JavaScript's reputation

- Ten years a
- Now: most **Web stack**
- **Tooling** h testing fra
- JavaScrip (V8, Spid
- JavaScrip

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

ECMAScript 5 6 2016+ next intl non-standard compatibility table

Flattr by kangax

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 points)

		Compilers/polyfills						Desktop browsers																
	98%	56%	71%	71%	50%	68%	17%	5%	11%	96%	96%	96%	98%	98%	98%	98%	98%	98%	98%	98%	98%	99%	99%	99%
Feature name	Current browser	Traceur	Babel 6 + core-js	Babel 7 + core-js	Closure 2018.10	Type-Script + core-js	es6-shim	Kong 4.14[1]	IE 11	Edge 17	Edge 18	Edge 19 Preview	FF 60 ESR	FF 62	FF 63	FF 64 Beta	FF 65 Nightly	CH 69, OP 56	CH 70, OP 57	CH 71, OP 58	CH 72, OP 59	SF 11.1	SF 12	SF TP
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	0/2	0/2	2/2	2/2	2/2
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	7/7	7/7	7/7	7/7	7/7
rest parameters	5/5	4/5	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	5/5	5/5	5/5	5/5	5/5
spread syntax for iterable objects	15/15	15/15	13/15	13/15	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	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	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	9/9	9/9	9/9	9/9	9/9
octal and binary literals	4/4	2/4	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	4/4	4/4	4/4	4/4	4/4
template literals	5/5	4/5	4/5	4/5	3/5	3/5	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	5/5	5/5	5/5	5/5	5/5
RegExp "y" and "u" flags	5/5	3/5	3/5	3/5	0/5	0/5	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	5/5	5/5	5/5	5/5	5/5
destructuring, declarations	22/22	20/22	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	22/22	22/22	22/22	22/22	22/22
destructuring, assignment	24/24	23/24	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	24/24	24/24	24/24	24/24	24/24
destructuring, parameters	24/24	19/24	21/24	21/24	20/24	21/24	0/24	0/24	0/24	23/24	23/24	23/24	24/24	24/24	24/24	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	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	2/2	2/2	2/2	2/2	2/2
Bindings																								
const	16/16	14/16	14/16	14/16	14/16	14/16	0/16	2/16	12/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16
let	12/12	10/12	10/12	10/12	10/12	10/12	0/12	0/12	10/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12	12/12
block-level function declaration[17]	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Functions																								
arrow functions	13/13	11/13	9/13	9/13	10/13	9/13	0/13	0/13	0/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13	13/13
class	24/24	17/24	19/24	19/24	14/24	19/24	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	24/24	24/24	24/24	24/24	24/24
super	8/8	7/8	4/8	4/8	7/8	7/8	0/8	0/8	0/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8
generators	27/27	24/27	24/27	24/27	19/27	22/27	0/27	0/27	0/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27	27/27

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

A language in flux

ECMAScript 5 6 2016+ **next** intl non-standard compatibility table

Flattr by kangax & webbedspace & zloirock Fork 687

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.

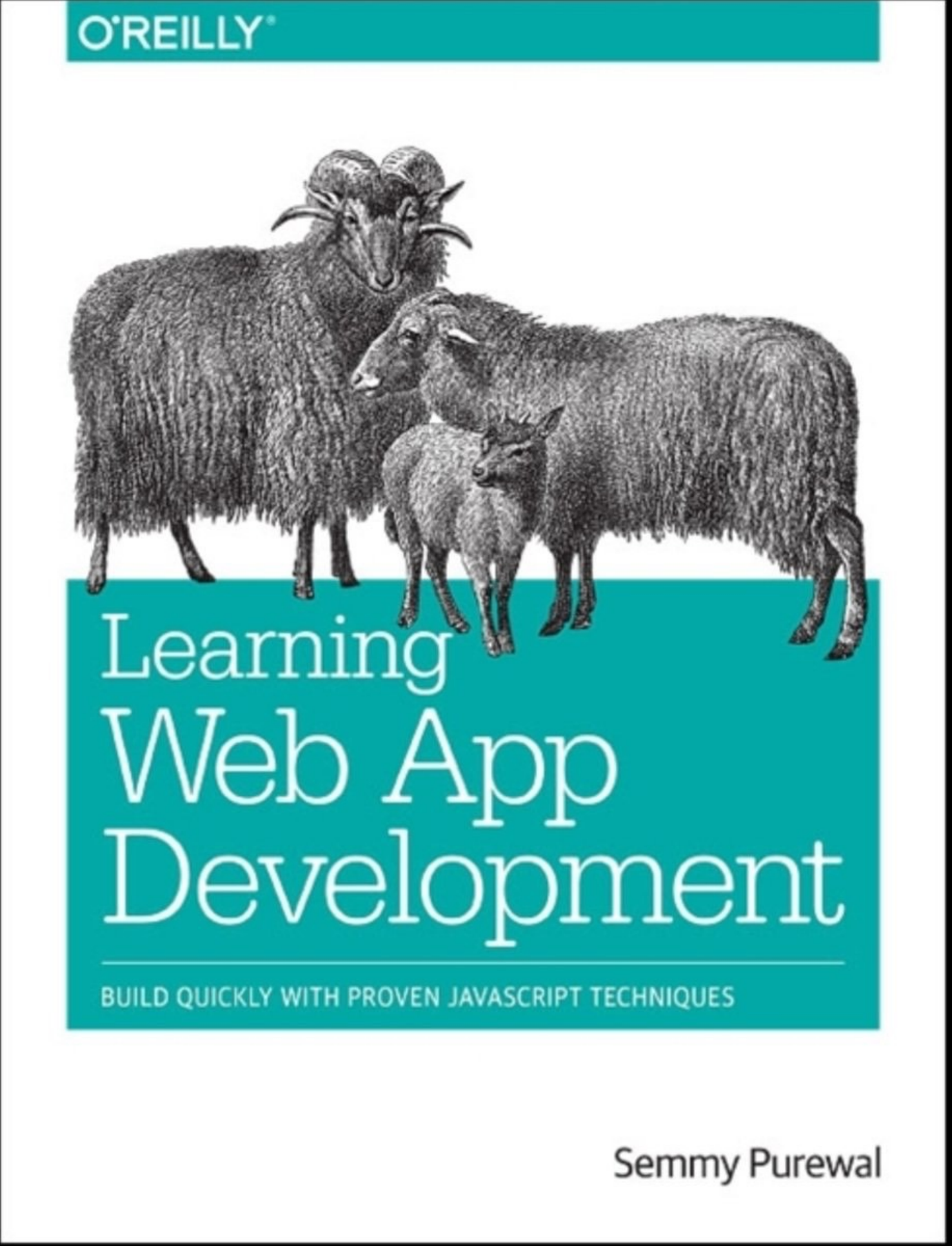
Sort by Engine types | Show obsolete platforms | Show unstable platforms

V8 SpiderMonkey JavaScriptCore Chakra Other

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

		Compilers/polyfills					Desktop browsers															Servers/runtime			
	8%	2%	25%	36%	0%	25%	1%	1%	1%	2%	5%	8%	9%	10%	8%	9%	9%	14%	1%	6%	7%	7%	0%	1%	1%	5%
Feature name ▸	Current browser	Traceur	Babel 6 + core-js	Babel 7 + core-js	Closure 2018.10	TypeScript + core-js	Edge 17	Edge 18	Edge 19 Preview	FF 60 ESR	FF 62	FF 63	FF 64 Beta	FF 65 Nightly	CH 69, OP 56	CH 70, OP 57	CH 71, OP 58	CH 72, OP 59	SF 11.1	SF 12	SF TP	WK	PJS	Node >=6.5 <7[2]	Node >=8.10 <9[2]	Node >=10.13 <11[2]
Candidate (stage 3)																										
string trimming ▸	4/4	0/4	4/4	4/4	0/4	4/4	2/4	2/4	2/4	2/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	2/4	4/4	4/4	4/4	2/4	2/4	2/4	4/4
globalThis ▸	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	2/2	0/2	0/2	0/2	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2
String.prototype.matchAll	No	No	Yes[4]	Yes[4]	No	Yes[5]	No	No	No	No	No	No	No	No	Flag[8]	Flag[8]	Flag[8]	Flag[8]	No	No	No	No	No	No	No	Flag[8]
instance 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	0/3	0/3	0/3	0/3	1/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3
static class fields ▸	0/2	1/2	1/2	1/2	0/2	1/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	1/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2
Function.prototype.toString revision ▸	7/7	0/7	0/7	0/7	0/7	0/7	4/7	4/7	4/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	4/7	4/7	4/7	4/7	1/7	4/7	4/7	7/7
Array.prototype.{flat, flatMap}[9] ▸	2/2	0/2	1/2	1/2	0/2	1/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	2/2	0/2	0/2	0/2	0/2
Symbol.prototype.description	Yes	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Flag[8]	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	No	No
BigInt ▸	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	8/8	8/8	8/8	8/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	8/8
Object.fromEntries	Yes	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes	No	No	No	No	No	No	Yes	Yes	No	No	No	No
Well-formed JSON.stringify	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	No	No	No	Yes	No	No	No	No	No	No	No	No
Draft (stage 2)																										
Generator function.sent Meta Property	No	No	Yes	Yes	No	No	No	No	No	No	No	No	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	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	No	No	No	No	No	No	No	No
weak references	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
throw expressions ▸	0/4	0/4	0/4	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	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4
numeric separators	No	No	No	Yes	No	Yes	No	No	No	No	No	No	No	No	Flag[8]	Flag[8]	Flag[8]	Flag[8]	No	No	No	No	No	No	No	Flag[8]
Set methods ▸	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	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4
Proposal (stage 1)																										
do expressions	No	No	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
Observable ▸	0/7	0/7	7/7	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	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7
Frozen Realms API	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No

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



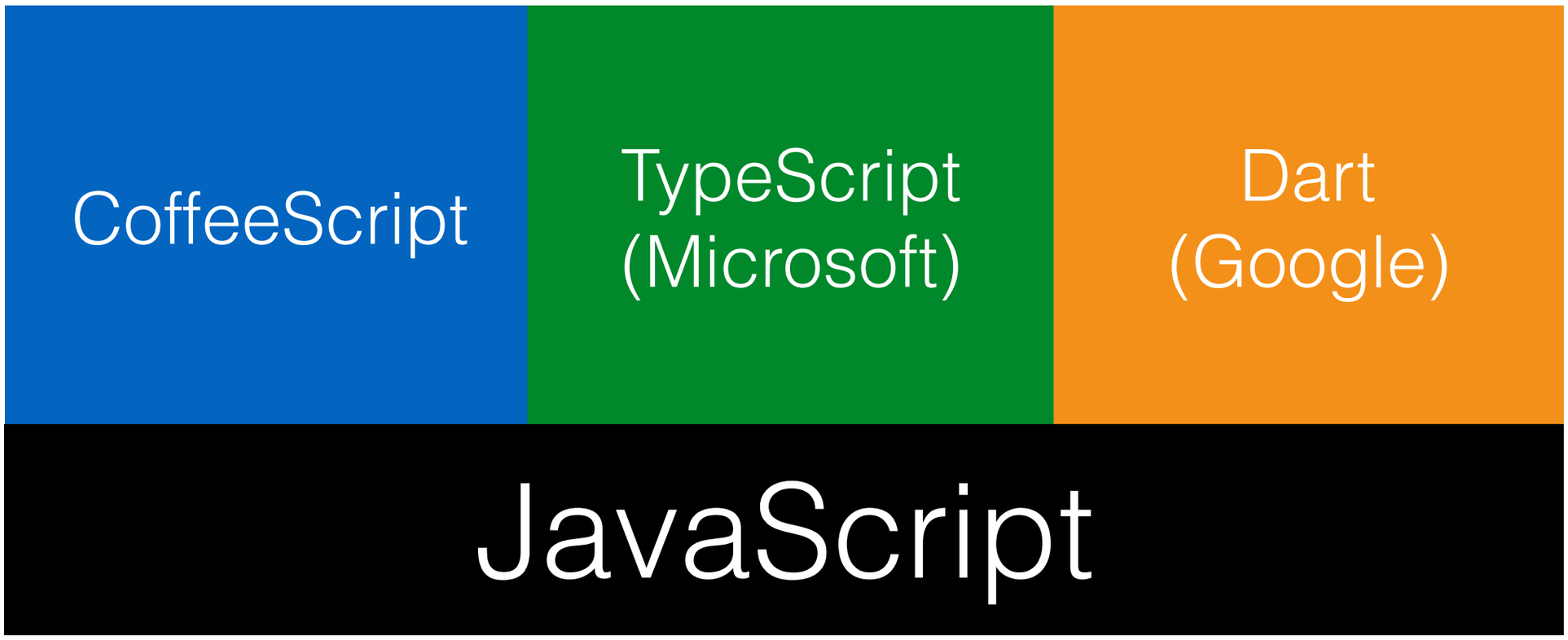
Published in 2014, thus no ES6.

Not a bug, but a feature!

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

We use two ES6 features: `let`,`const`

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 has quirks

Hoisting principle:
declarations are processed before any code is executed.

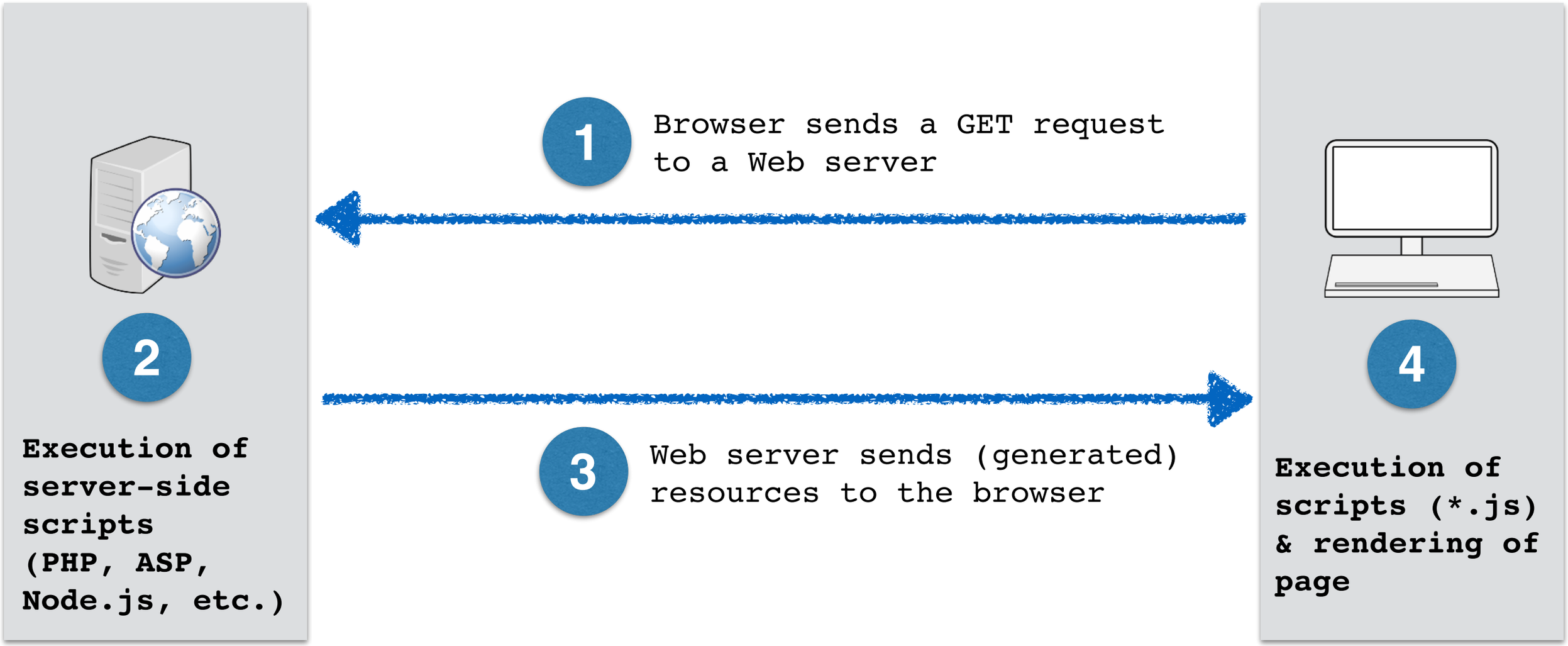
Declarations are hoisted to the top (also inside of functions).

Expressions are not hoisted.

```
1  var x = six();
2
3  //function declaration
4  function six(){
5      return 6;
6  }
7
8  var y = seven()
9
10 //function expression
11 var seven = function(){
12     return 7;
13 }
14
15 console.log(x+""+y);
```

Scripting overview

Interactive web applications



JavaScript makes Web apps **interactive** and **responsive** to user actions.

Server-side scripting

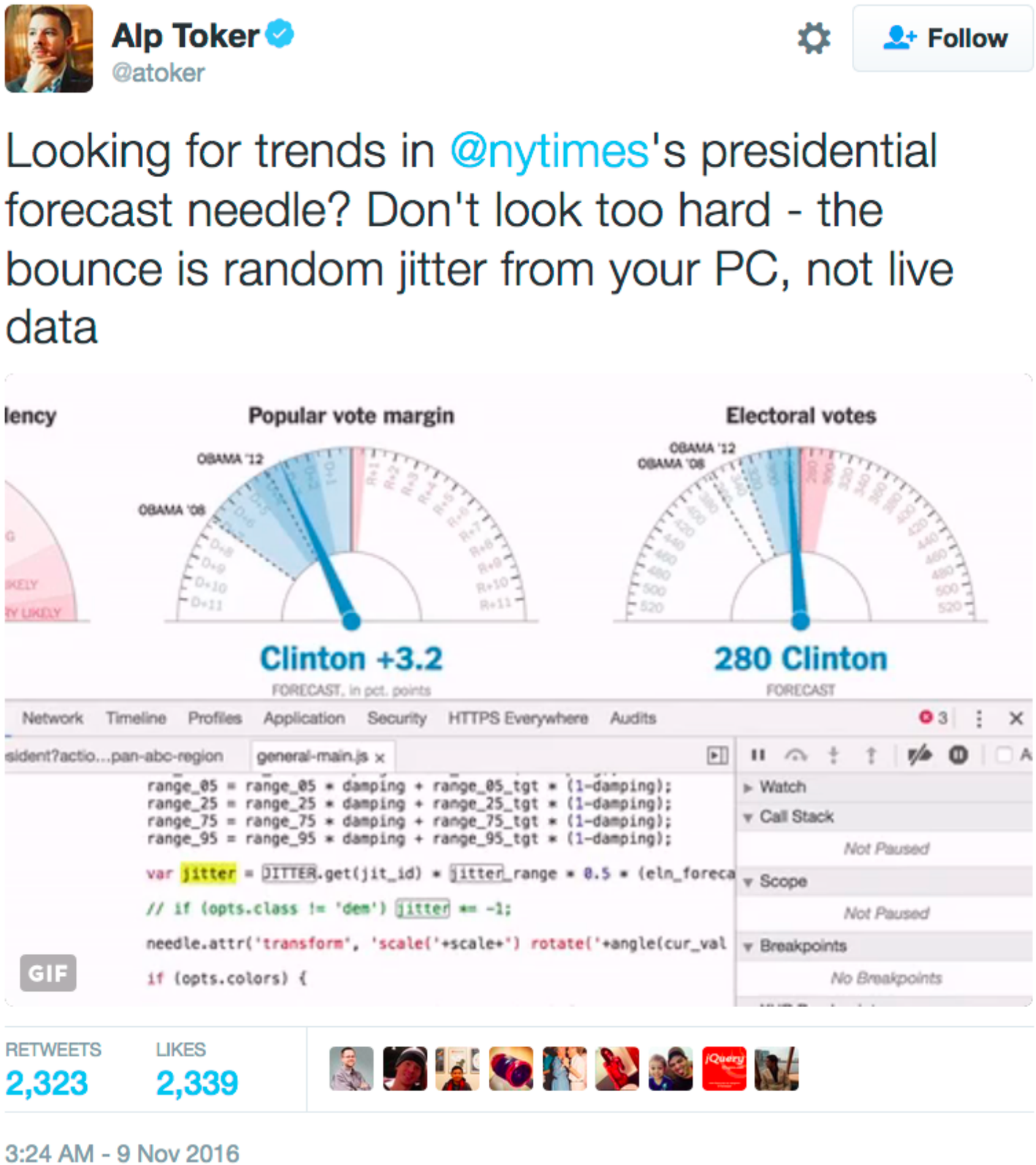
- Source code is **private**, only the result of a script's execution is returned
- HTML can be rendered by **any browser**
- Server-side scripts can **access additional resources** (e.g. databases)
- Server-side scripts can use **non-standard language features**

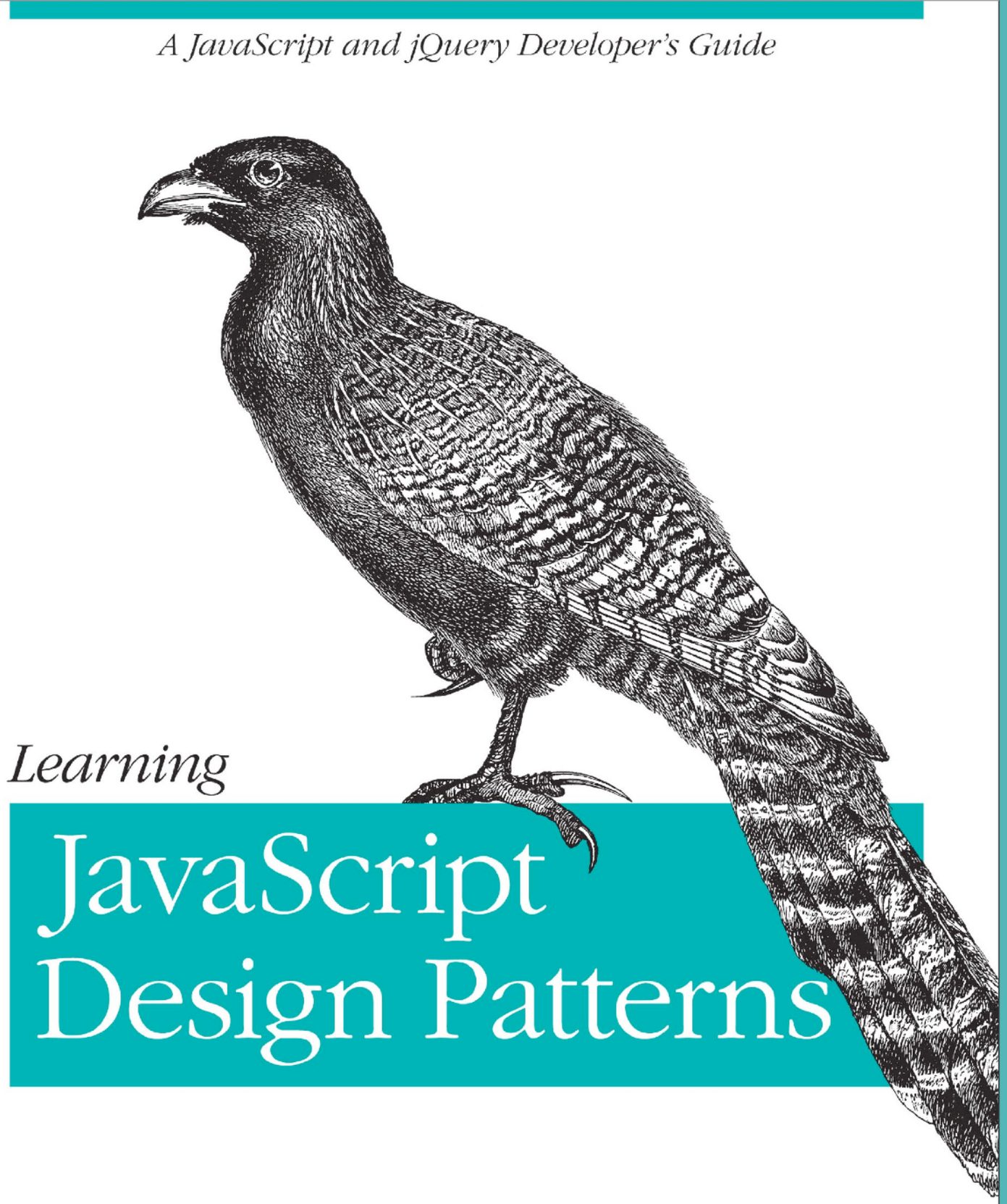
Client-side scripting

- 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
- JavaScript is **event-driven**: code blocks executed in response to user actions

Client-side scripting

- Source code is **vis**
- Client-side script e
- All **raw data** need processed by the
- JavaScript is **ever** executed in respo





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 has **functions** (which are also objects) as **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

Objects in JavaScript

- `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)

```
1 var game = new Object();
2 game["id"] = 1;
3 game["player1"] = "Alice";
4 game.player2 = "Bob";
5 console.log( game["player2"] ); //prints out "Bob"
6 console.log( game.player1 ); //prints out "Alice"
```

Another way: object literals

```
1 var game = new Object();
2 game["id"] = 1;
3 game["player1"] = "Alice";
4 game.player2 = "Bob";
5 console.log( game["player2"] ); //prints out "Bob"
6 console.log( game.player1 ); //prints out "Alice"
```

```
1 var game = {
2       id: 1,
3       player1: "Alice",
4       player2: "Bob"
5 };
```

Adding a method

```
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"
```

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

Object literals can be complex

```
1  var paramModule = {
2      /* parameter literal */
3      Param : {
4          minGames: 1,
5          maxGames: 100,
6          maxGameLength: 30
7      },
8      printParams: function(){
9          console.table(this.Param);
10     }
11 };
```

inner object
`Param`

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;
    };
}

//creating objects
var g1 = new Game(1);
g1.getID();
g1.setID(2);

var g2 = new Game(3);
```

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

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:
  - **Inheritance** (e.g. `NoteWithDueDate`)
  - **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)
```


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

Prototype-based constructor

```
1  function Game(id){
2      this.id = id;
3  }
4
5  /* new member functions are defined once in the prototype */
6  Game.prototype.getID = function(){ return this.id; };
7  Game.prototype.setID = function(id){ this.id = id; };
8
9  //using it
10 var g1 = new Game("1");
11 g1.setID("2"); //that works!
12
13 var g2 = new Game("2");
14 g2.setID(3); //that works too!
15
16 g1["setID"] = function(id){
17     this.id = "ID"+id;
18 }
19 //g1 and g2 now point to different setID properties
20 g1.setID(4);
21
22 console.log(g1.getID()); //ID4
23 console.log(g2.getID()); //3
```

Executing this JavaScript code snippet yields what output?

```
1  function Game(id){
2      this.id = id;
3  }
4
5  Game.prototype.getID = function(){ return this.id; };
6
7  var g1 = new Game("1");
8
9  Game.prototype.getID = function(){
10     return "0"+this.id;
11 }
12
13 console.log( g1.getID() );
```

A. 1
B. 01 ✔ Prototype changes are reflected in existing objects!
C. Error: a prototype property can only be assigned once.

Prototype-based constructor

Inheritance through prototyping.

```
1  function Game(id){
2      this.id = id;
3  }
4
5  /* new member functions are defined once in the prototype */
6  Game.prototype.getID = function(){ return this.id; };
7  Game.prototype.setID = function(id){ this.id = id; };
8
9  /* constructor */
10 function TwoPlayerGame(id, p1, p2){
11     /*
12      * call(...) calls a function with a given this value and arguments.
13      */
14     Game.call(this, id);
15     this.p1 = p1;
16     this.p2 = p2;
17 }
18
19 /* redirect prototype */
20 TwoPlayerGame.prototype = Object.create(Game.prototype);
21 TwoPlayerGame.prototype.constructor = TwoPlayerGame;
22
23 /* use it */
24 var TPGame = new TwoPlayerGame(1, "Alice", "Bob");
25 console.log( TPGame.getID() ); //prints out "1"
26 console.log( TPGame.p1 ); //prints out "Alice"
```

`call()` calls a function with a given `this` value and arguments (one by one)

1. create a new constructor

2. redirect the prototype

= prototype chain

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

Design Pattern (3):

Module

JavaScript scoping

- All code enters the **same namespace**
- **Limited scoping**:
  - `var` in function: **local**, limited scope
  - `var` outside of a function: **global** scope
  - no `var`: **global** scope
  - `let` (**ES6**): block scope
  - `const` (**ES6**): block scope, no reassignment or redeclaration

JavaScript scoping

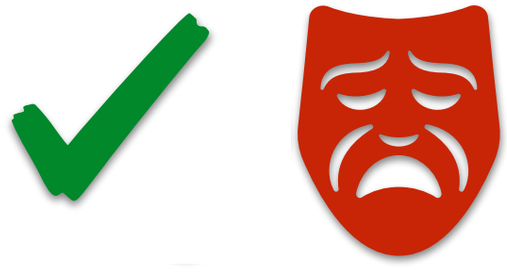
setTimeout: sets a timer and executes a piece of code once after the timer expires

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

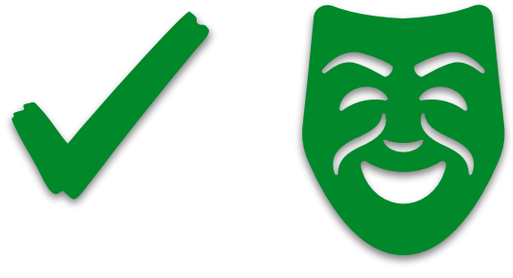
```
1 for (var i = 1; i <= 10; i++) {
2     setTimeout(function() {
3         console.log(i);
4     }, 1000);
5 }
```

11 is printed ten times

```
1 function fn(i) {
2     setTimeout(function() {
3         console.log(i);
4     }, 1000 * i);
5 }
6
7 for (var i = 1; i <= 10; i++)
8     fn(i);
```



```
1 for (let i = 1; i <= 10; i++)
2     setTimeout( function() {
3         console.log(i)
4     }, 1000 * i)
```



JavaScript scoping

```
1  function Note(n, t){
2      this.note = n;
3      this.type = t;
4  }
5
6  var note1 = new Note("Algorithms",1);
7  var note2 = new Note("Databases",3);
8
9  note1.setDueIn = function(x){
10     this.dueIn = x;
11 }
12
13 note2.setDueIn = function(x){
14     this.dueIn = x;
15 }
16
17 note1.setDueIn == note2.setDueIn; //true or false?
18 note1.setDueIn.toString() == note2.setDueIn.toString(); //true or false?
```

What if another JavaScript library used in the project defines Note?

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 JS libraries
  - Public API **minimizes** unintentional side-effects

Module

```
1  /* creating a module */
2  var gameStatModule = ( function() {
3
4      /* private members */
5      var gamesStarted = 0;
6      var gamesCompleted = 0;
7      var gamesAbolished = 0;
8
9      /* public members: return accessible object */
10     return {
11         incrGamesStarted : function(){
12             gamesStarted++;
13         },
14         getNumGamesStarted : function(){
15             return gamesStarted;
16         }
17     }
18 })();
19
20 /* using the module */
21 gameStatModule.incrGamesStarted();
22 console.log( gameStatModule.getNumGamesStarted() );
23 console.log( gameStatModule.gamesStarted ); //prin
```

public

private

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

Immediately Invoked Function Expression (IIFE)

Module

The encapsulating function can contain arguments.

The pattern can be arbitrarily complex.

```
1  /* creating a module */
2  var gameStatModule =  function(s, c, a) {
3
4      /* private members */
5      var gamesStarted = s;
6      var gamesCompleted = c;
7      var gamesAbolished = a;
8
9      /* public members: return
10      * accessible object
11      */
12     return {
13         incrGamesStarted : function(){
14             gamesStarted++;
15         },
16         getNumGamesStarted : function(){
17             return gamesStarted;
18         }
19     }
20 }(1, 1, 1);
```

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

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);
```

- Course book uses **jQuery** extensively
- **Important to understand** what jQuery covers up

A look at book chapter 4

```
1 /* jQuery's way of accessing DOM elements */
2 $(".comment-input button").on("click", function(e){
3     ...
4 });
```

- **With jQuery**: no matter if `class` or `id` or ..., the access pattern is the same, i.e. `$()`
- **Without jQuery**:
  - `document.getElementById`
  - `document.getElementsByClassName`
  - `document.getElementsByTagName`
  - `document.querySelector`
  - `document.querySelectorAll`
- **Callback principle: we define what happens when an event fires**

$ ( )

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

callback hell

Step-by-step: making a responsive UI control

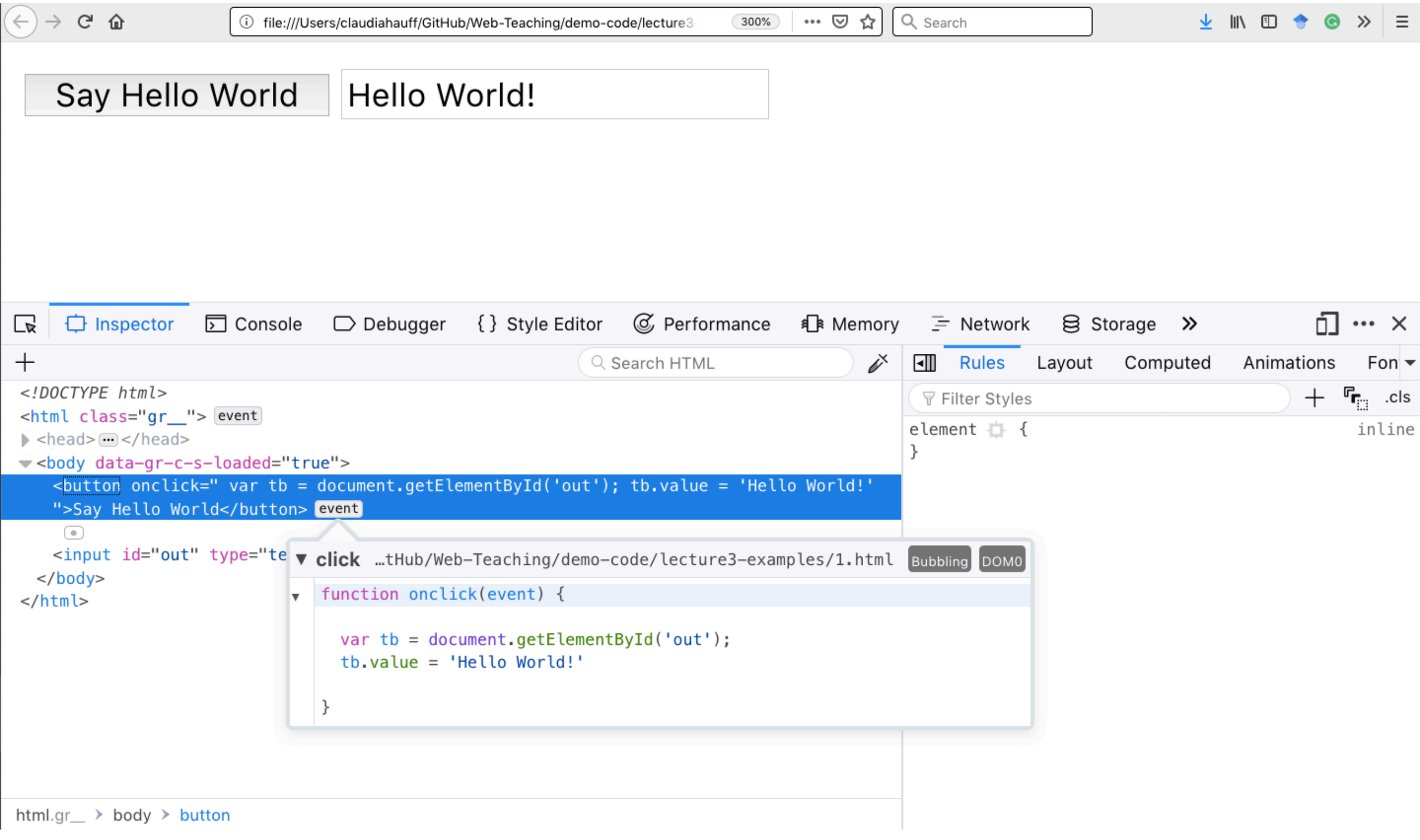
1. Pick a control (e.g. a button)

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

3. Write a function: what happens when the event fires? (e.g. a popup appears)

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

Browser developer tooling



Interaction via the DOM

The DOM is our entry point to interactive web applications. It allows use to:

- **Extract an element's state**
  - Is the checkbox checked?
  - Is the button disabled?
  - Is a `<h1>` appearing on the page?
- **Change an element's state**
  - Check a checkbox
  - Disable a button
  - Create an `<h1>` element on a page if none exists
- **Change an element's style** (material for a later lecture)
  - Change the color of a button
  - Change the size of a paragraph
  - Change the background color of a web application

Client-side JS examples

small and self-contained

`getElementById`

1

creating DOM nodes

2

`this`

3

mouse events

4

crowdsourcing interface

5

typing game

6

getElementById

1

```
<html>
    <head>
        <title>Example 1</title>
        <script>
            /* we define the function */
            function sayHello() {
              var tb = document.getElementById("out");
              tb.value = 'Hello World';
            };

            /* we attach a function to a button's click event
             * after the DOM finished loading
             */
            window.onload = function() {
                document.getElementById("b").onclick = sayHello;
            };
        </script>
    </head>

    <body>
        <button id="b">Say Hello World</button>
        <input id="out" type="text">
    </body>
</html>
```

creating DOM nodes

2

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Example 2</title>
5          <script>
6              window.onload = function() {
7                document.getElementById("b").onclick = addElement;
8              };
9
10             function addElement() {
11               var ul = document.getElementById('u');
12               var li = document.createElement('li');
13               li.innerHTML = 'List element ' + (ul.childElementCount+1) +' ';
14               ul.appendChild(li);
15             };
16         </script>
17     </head>
18
19     <body>
20         <button id="b">Add List Element</button>
21         <ul id="u"></ul>
22     </body>
23 </html>
```

1. create node
2. add as child of existing node

empty list element

this

3

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example 3</title>
        <script>
          window.onload = function() {
            document.getElementById("button10").onclick = computeTimes;
            document.getElementById("button23").onclick = computeTimes;
            document.getElementById("button76").onclick = computeTimes;
          };

          function computeTimes() {
            /*
             * this.innerHTML returns to us "N times",
             * parseInt() then strips out the " times" suffix
             * as it stops parsing at an invalid number character
             */
            var times = parseInt(this.innerHTML);
            var input = parseFloat(document.getElementById("input").value);
            var res = times * input;
            alert("The result is " + res);
          }
        </script>
    </head>

    <body>
        <input type="text" id="input">
        <button id="button10">10 times</button>
        <button id="button23">23 times</button>
        <button id="button76">76 times</button>
    </body>
</html>
```

points to the element the event handler is bound to

mouse events

4

A number of different mouse events exist (`mouseup`, `mousedown`, `mousemove`, ...) and some are defined as a series of simpler mouse events, e.g.

- A click of the mouse button in-place consists of:
  i. `mousedown`
  ii. `mouseup`
  iii. `click`
- A click of the mouse button while moving the mouse ("dragging") consists of:
  i. `mousedown`
  ii. `mousemove`
  iii. ... n-1. `mousemove` n. `mouseup`

Mouse events can be tricky, the more complex ones are not consistently implemented across browsers.

- Read **chapters 5 & 6** (Node.js) of the Web development book before this Thursday's lecture!

- Wrap up **Assignment 1**.

- Get started on **Assignment 2**.

flickr@courtneyrian