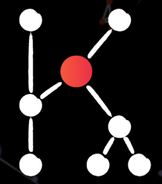


JAVASCRIPT PART 1

KIRELL BENZI, PH.D

@KirellBenzi

www.kirellbenzi.com

What is Javascript?

Dynamically weakly-typed, multi paradigm programming language for front-end and backend programming

JS supports imperative/procedural, object-oriented, and functional programming styles.

It is the world's most misunderstood programming language (and the most popular)!

Dynamic / static vs weak / strong typing

Dynamic typing: No need to declare types (int, float) for variables. The type is known when the code is run.

Static typing: Need to declare variable types (**int** a = 2) otherwise you'll have a compile error.

Strong typing: Once a variable is declared as a specific data type, it will be bound to that particular data type (int a is always an integer) but you can explicitely cast the type to something else.

Weak typing: Variables are not of a specific data type. You can choose to reassign it to something else completely.

Javascript is not Java

Java is to JavaScript as ham is to hamster. Jeremy Keith

The name Javascript was used a marketing ploy by Netscape in 1995 to give JavaScript the cachet of what was then the hot new Web programming language [Wikipedia]

JavaScript and Java differ greatly in design; JavaScript was influenced by programming languages such as Self and Scheme whereas Java was designed to be a simpler C++.

JavaScript core language features are defined in a standard called ECMA-262. The language defined in this standard is called ECMAScript.

[Segue Technologies]

Why use JS?

It is the franca lingua of web development and the only (serious) way of scripting the browser (excluding compile-to-js langs)

As soon as you want to add interactions, actions, events, or anything that is not static on the page, you need JS (excluding CSS or SVG animations)

The good and bad parts

javascript hate

Tous | Images | Vidéos | Actualités | Shopping | Plus | Paramètres | Outils

Environ 25 600 000 résultats (0,37 secondes)

Conseil : Recherchez des résultats uniquement en **français**. Vous pouvez indiquer votre langue de recherche sur la page Préférences.

Why do so many people seem to hate JavaScript? - Quora

https://www.quora.com/Why-do-so-many-people-seem-to-hate-Ja... ▾ Traduire cette page

In December of 2009/January of 2010, I decided to give this whole SSJS thing a go. I spent some relearning **JavaScript** and tried to write some code (an Avro ...

10 things we hate about JavaScript | InfoWorld

www.infoworld.com/.../javascript/146732-10-things-we-hate-abou... ▾ Traduire cette page

10 things we **hate** about **JavaScript**. Endless library reloading, cool tools that piggyback on **JavaScript** success, spaghetti code -- alert('Over it!') By Peter Wayner ...

Seems that now I hate javascript : javascript - Reddit

https://www.reddit.com/.../javascript/.../seems_that_now_i_hate_ja... ▾ Traduire cette page

12 août 2016 - 20 messages - 18 auteurs

Hey guys, For 3 years I've done front-end development but now I realized that how much I **hate** it day by day. You know it feels like I don't...

Do you hate JavaScript? - The Practical Dev

https://dev.to/reverentgeek/do-you-hate-javascript ▾ Traduire cette page

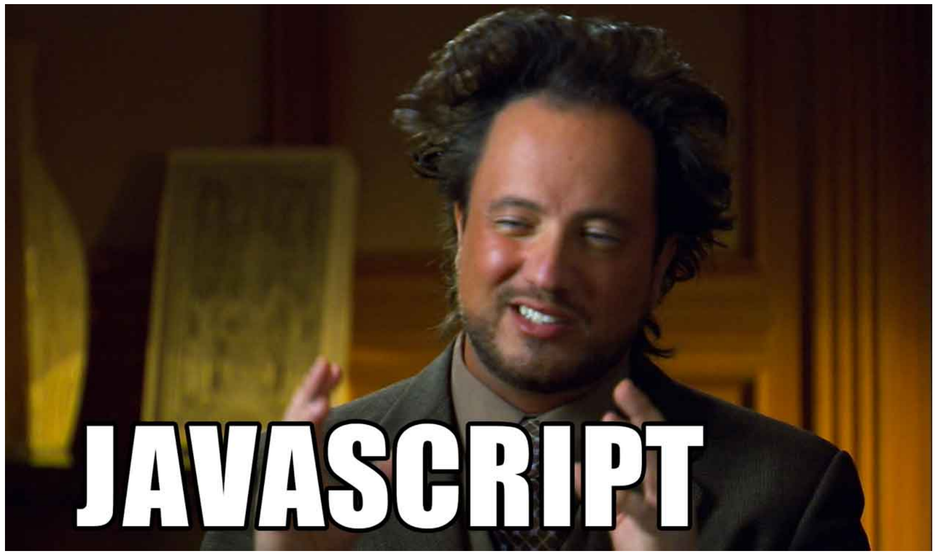
15 févr. 2017 - Every programming language I know has its own challenges and peculiarities. **JavaScript** has a reputation for being exceptionally quirky. And ...

"I Hate JavaScript" - Redo The Web

www.redotheweb.com/2015/12/04/i-hate-havascript.html ▾ Traduire cette page

4 déc. 2015 - I've heard that sentence a lot during the latest PHP conference I attended. I also heard “I **hate** PHP” many times during **JavaScript** conferences.

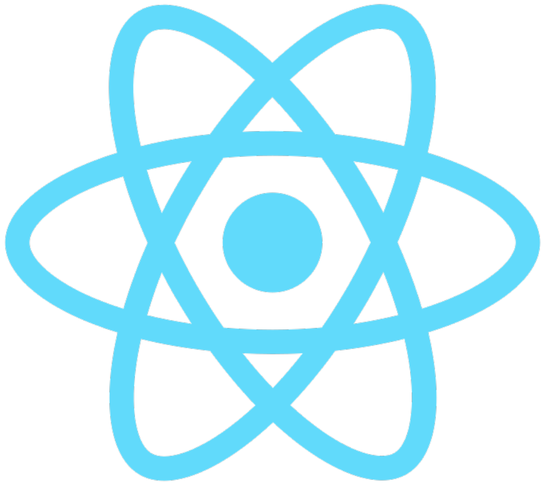
```
> '5' - 3
2         // weak typing + implicit conversions * headaches
> '5' + 3
'53'      // Because we all love consistency
> '5' - '4'
1         // string - string * integer. What?
> '5' + + '5'
'55'
> 'foo' + + 'foo'
'fooNaN' // Marvelous.
> '5' + - '2'
'5-2'
> '5' + - + - - + - - + + - + - + - + - - - '-2'
'52'      // Apparently it's ok

> var x * 3;
> '5' + x - x
50
```



JS renaissance

Google use of AJAX for webapps

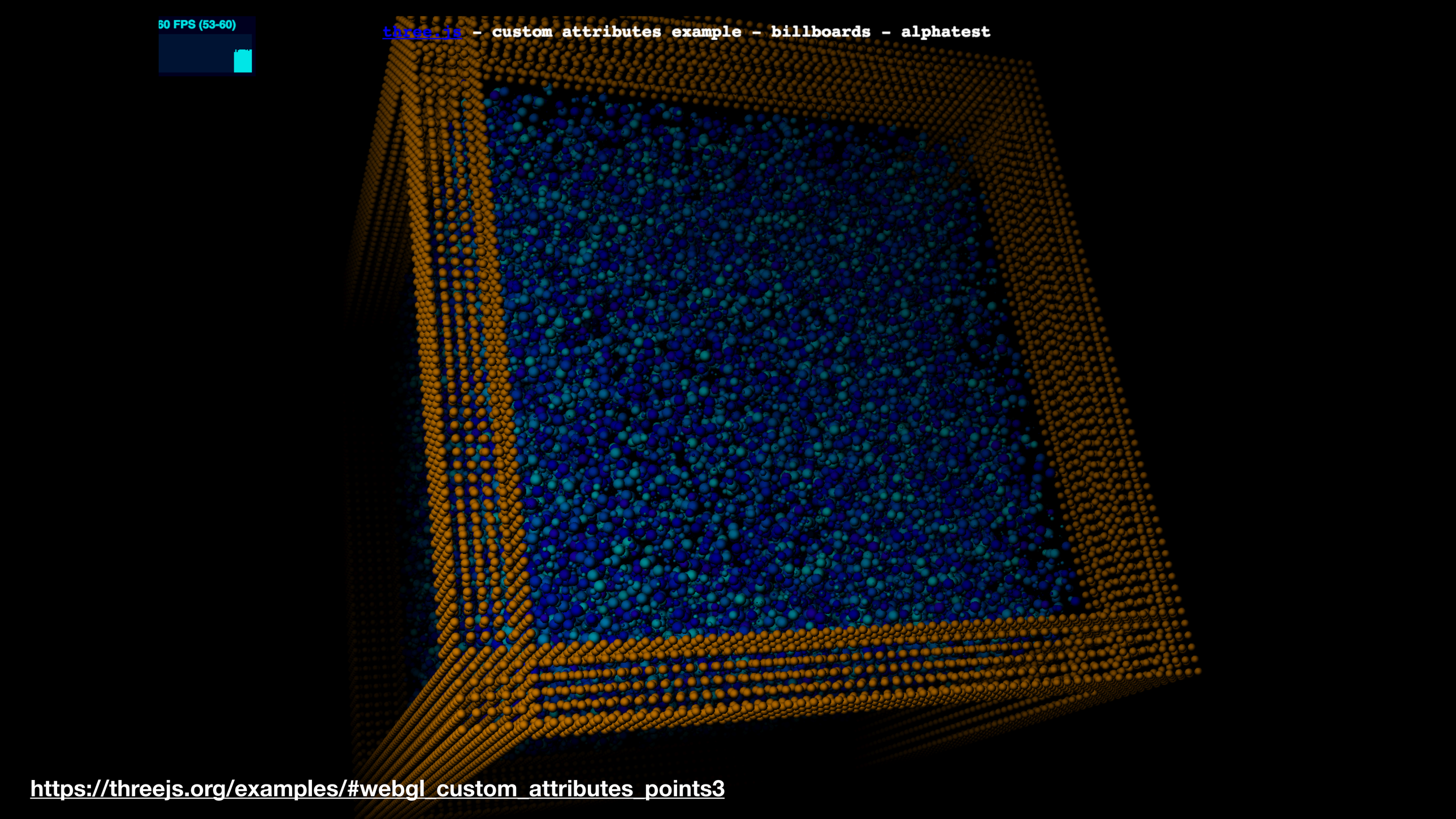
Creation of Node.js (2009): open-source, cross-platform JavaScript run-time environment for executing JavaScript code **server-side**



ECMAScript 6 or Javascript 2015: cleaner, saner, easier version of the language

Many JS frameworks : React, Angular, Express, etc.

Backed up by leading companies: Google, Facebook, LinkedIn, Netflix, Mozilla, etc.

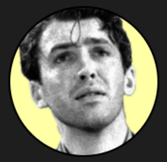


https://threejs.org/examples/#webgl_custom_attributes_points3

played out stories that were like experiments in life, in the homes and streets of an unglamorized America.

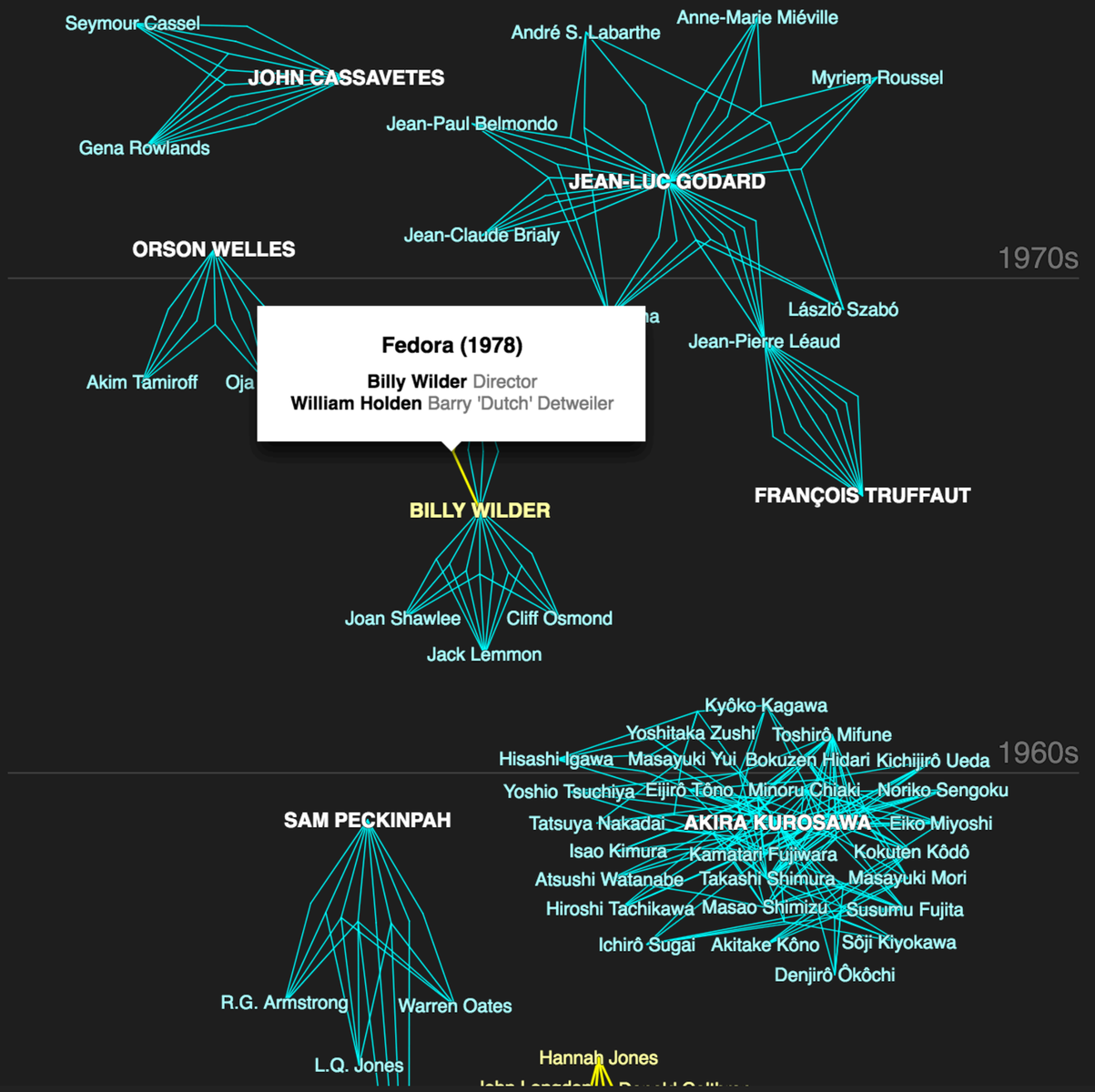
Alfred Hitchcock

CARY GRANT



JIMMY STEWART

As canny a promoter as he was an architect of suspense and imagery, Hitchcock was fully cognizant of the value of stars to his Hollywood films. Yet the draw of a beautiful face or well-known persona also had a crucial formal function: wrapping the audience in the psychological intrigue of his films. Watching Cary Grant in "North by Northwest," we almost feel as if we're in on a grand cosmic joke as he is put through multiple baffling adventures. Another recurring face, Jimmy Stewart, had a way of making Hitchcock's intrigues hit close to home, as the actor found amid extraordinary circumstances (as in "Vertigo" or "The Man Who Knew Too Much") a relatable, shifting emotional core.



http://www.nytimes.com/newsgraphics/2013/09/07/director-star-chart/

Using ECMA2015

As always when doing development, supporting old plateforms is difficult

The implementation of ES6 is not necessarily complete in all browsers

We need a tool that allows to deploy code anywhere and support new functions included in ECMA 2015 (also known as ECMA 6)

JS compatibility

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											
	97%	56%	70%	48%	59%	17%	5%	11%	93%	96%	94%	97%	97%	99%	99%	4%	95%	59%
Feature name	Current browser	Traceur	Babel + core-js[2]	Closure	Type-Script + core-js	es6-shim	Kong 4.14[3]	IE 11	Edge 14	Edge 15	FF 52 ESR	FF 54	CH 60, OP 47[1]	SF 10	SF 10.1	PJS	XS6	JXA
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	2/2	2/2	0/2	2/2	0/2
Syntax																		
default function parameters	7/7	4/7	4/7	4/7	5/7	0/7	0/7	0/7	7/7	7/7	6/7	7/7	7/7	7/7	7/7	0/7	7/7	0/7
rest parameters	5/5	4/5	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	0/5	5/5	0/5
spread (...) operator	15/15	15/15	13/15	12/15	4/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	0/15	15/15	11/15
object literal extensions	6/6	6/6	6/6	4/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	0/6	6/6	5/6
for..of loops	9/9	9/9	9/9	6/9	3/9	0/9	0/9	0/9	7/9	9/9	7/9	9/9	9/9	9/9	9/9	0/9	9/9	8/9
octal and binary literals	4/4	2/4	4/4	4/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	0/4	4/4	4/4
template literals	5/5	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	0/5	5/5	5/5
RegExp "y" and "u" flags	5/5	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	0/5	2/5	0/5
destructuring, declarations	22/22	20/22	21/22	19/22	15/22	0/22	0/22	0/22	21/22	22/22	21/22	22/22	22/22	22/22	22/22	0/22	21/22	19/22
destructuring, assignment	24/24	23/24	24/24	17/24	19/24	0/24	0/24	0/24	23/24	24/24	23/24	24/24	24/24	24/24	24/24	0/24	24/24	21/24
destructuring, parameters	23/23	19/23	20/23	18/23	15/23	0/23	0/23	0/23	22/23	23/23	20/23	23/23	23/23	23/23	23/23	0/23	23/23	18/23
Unicode code point escapes	2/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	1/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2
new.target	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	0/2	2/2	0/2
Bindings																		
const	16/16	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	1/16	16/16	10/16
let	12/12	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	0/12	12/12	0/12

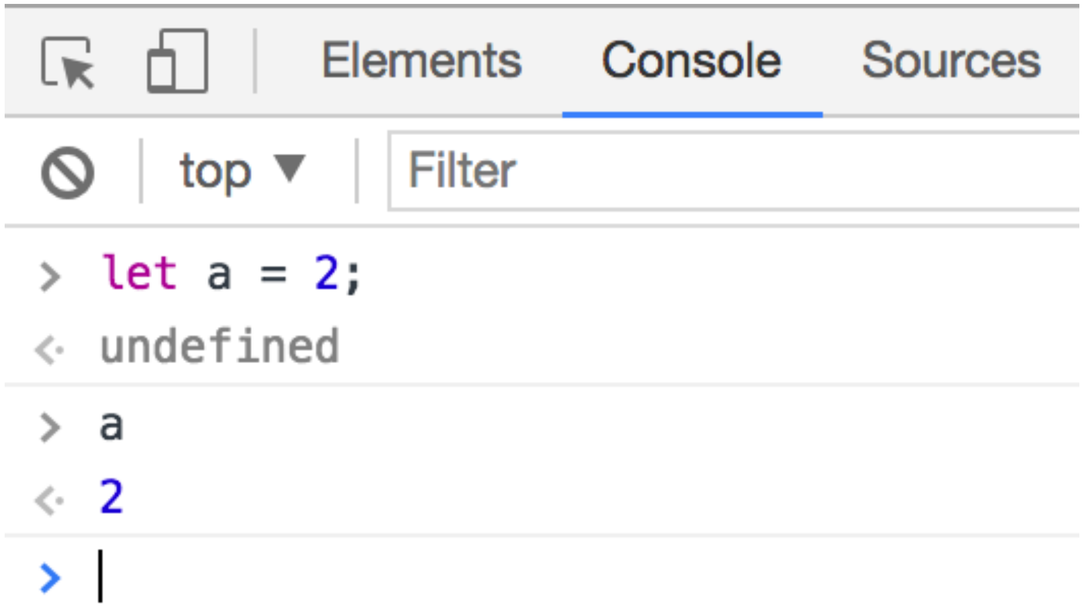
Introducing Babel

[GRC conseil]

Javascript console

Easy to get started, just open the DevTools on Chrome (Safari, etc.)

All modern browsers support ES6 out of the box

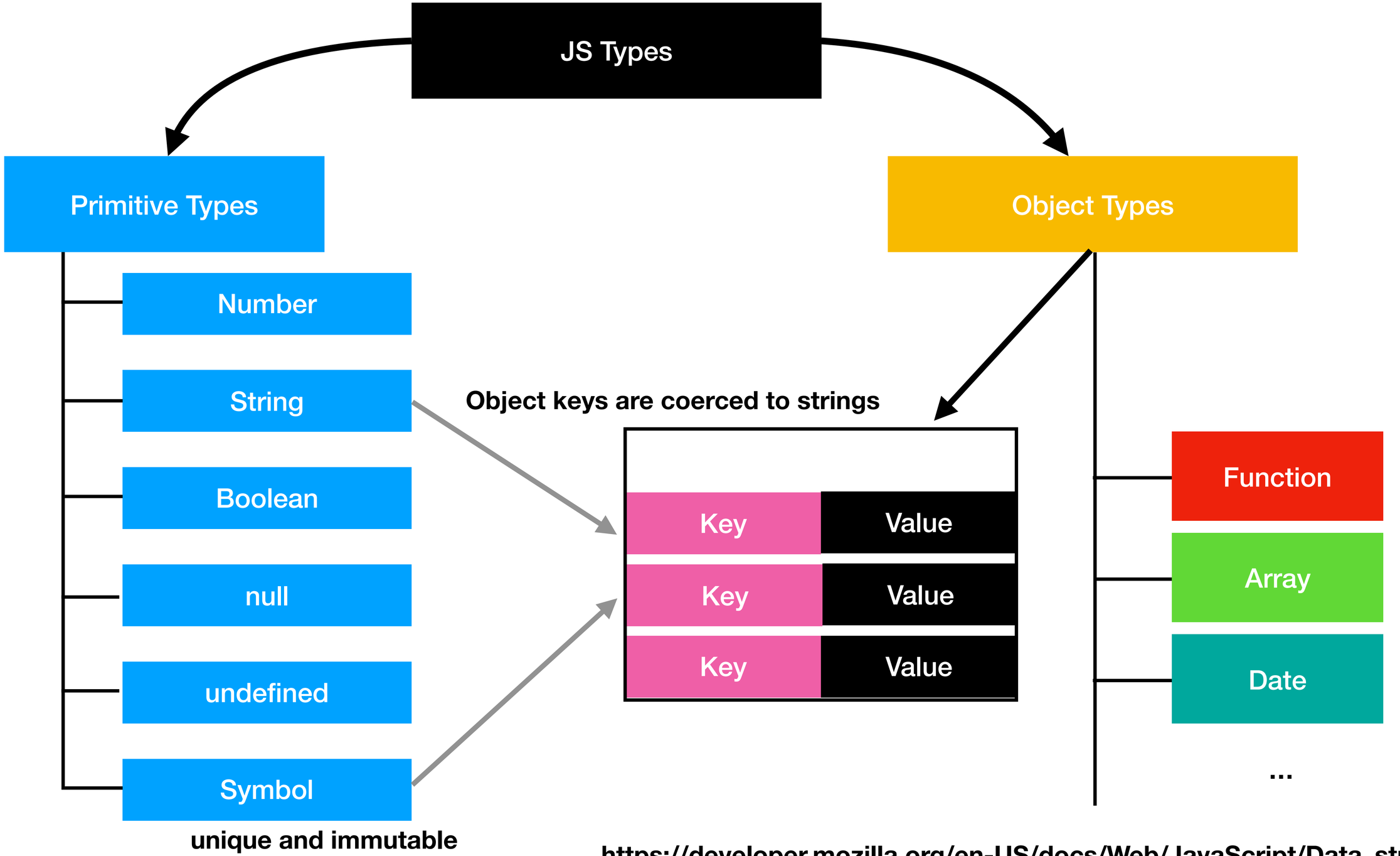


Comments

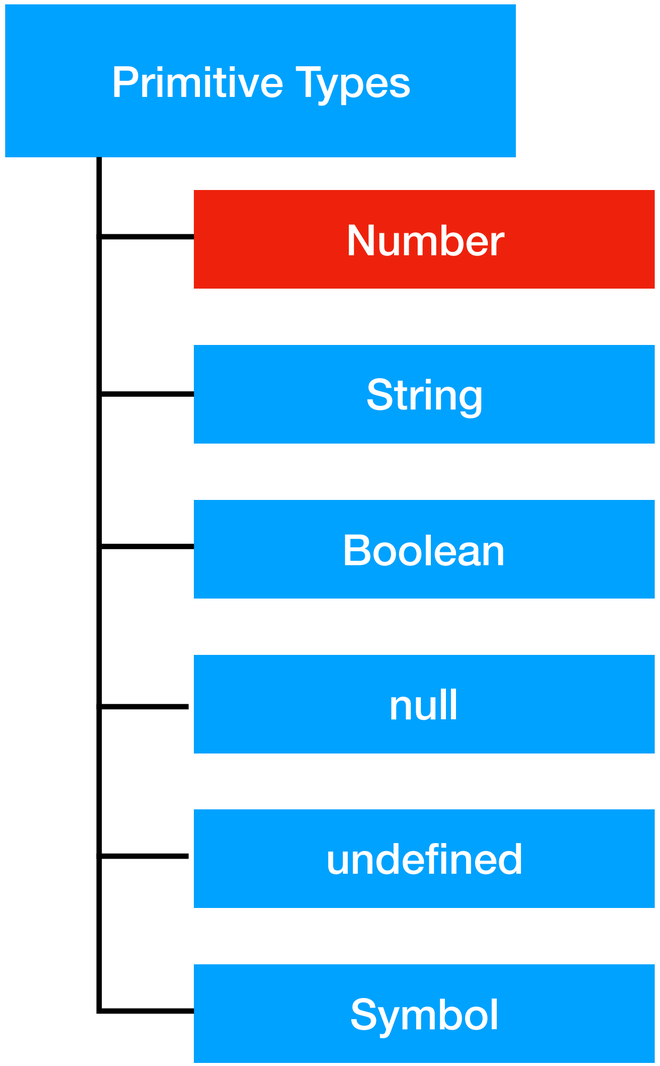
JavaScript borrows most of its syntax from Java, but is also influenced by Awk, Perl and Python.

JavaScript is case-sensitive and uses the Unicode character set.

```
/* this is a longer,
   multi-line comment
 */

// Inline comment

/* You can't, however, /* nest comments */ SyntaxError */
```



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



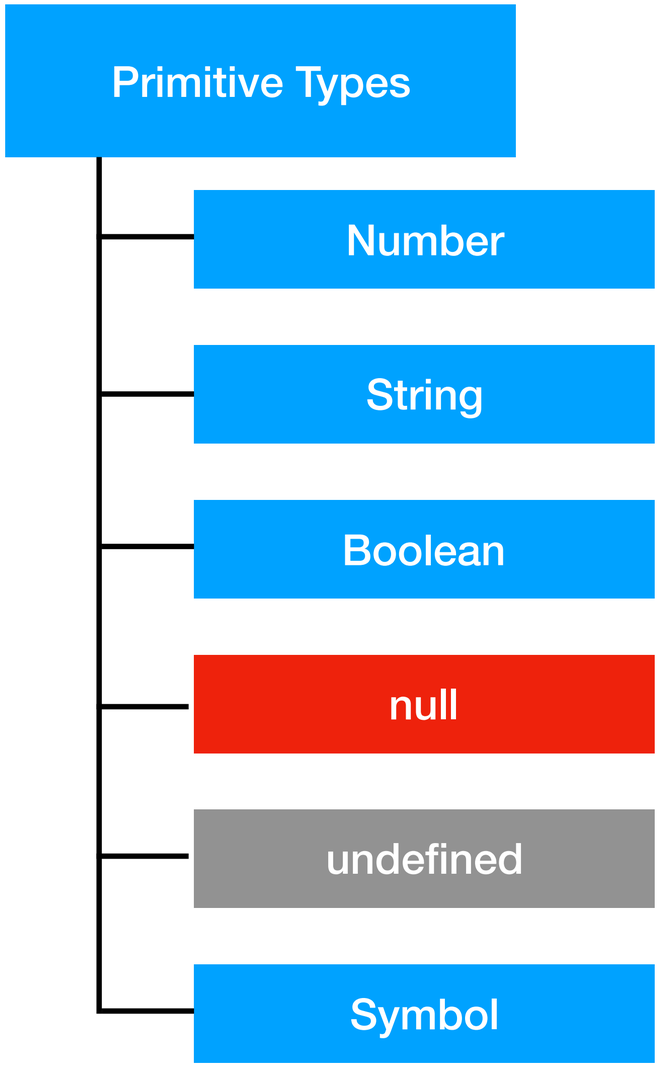
Double-precision 64bit float (no integer type)

Standard arithmetic operators (+, -, *, /, %)

Math.round(), Math.floor(), Math.cos(), etc.

parseInt(), parseFloat() to parse strings

Special numbers: **NaN**, **Infinity**



null denotes the absence of value

undefined denotes the absence of variable

Objects

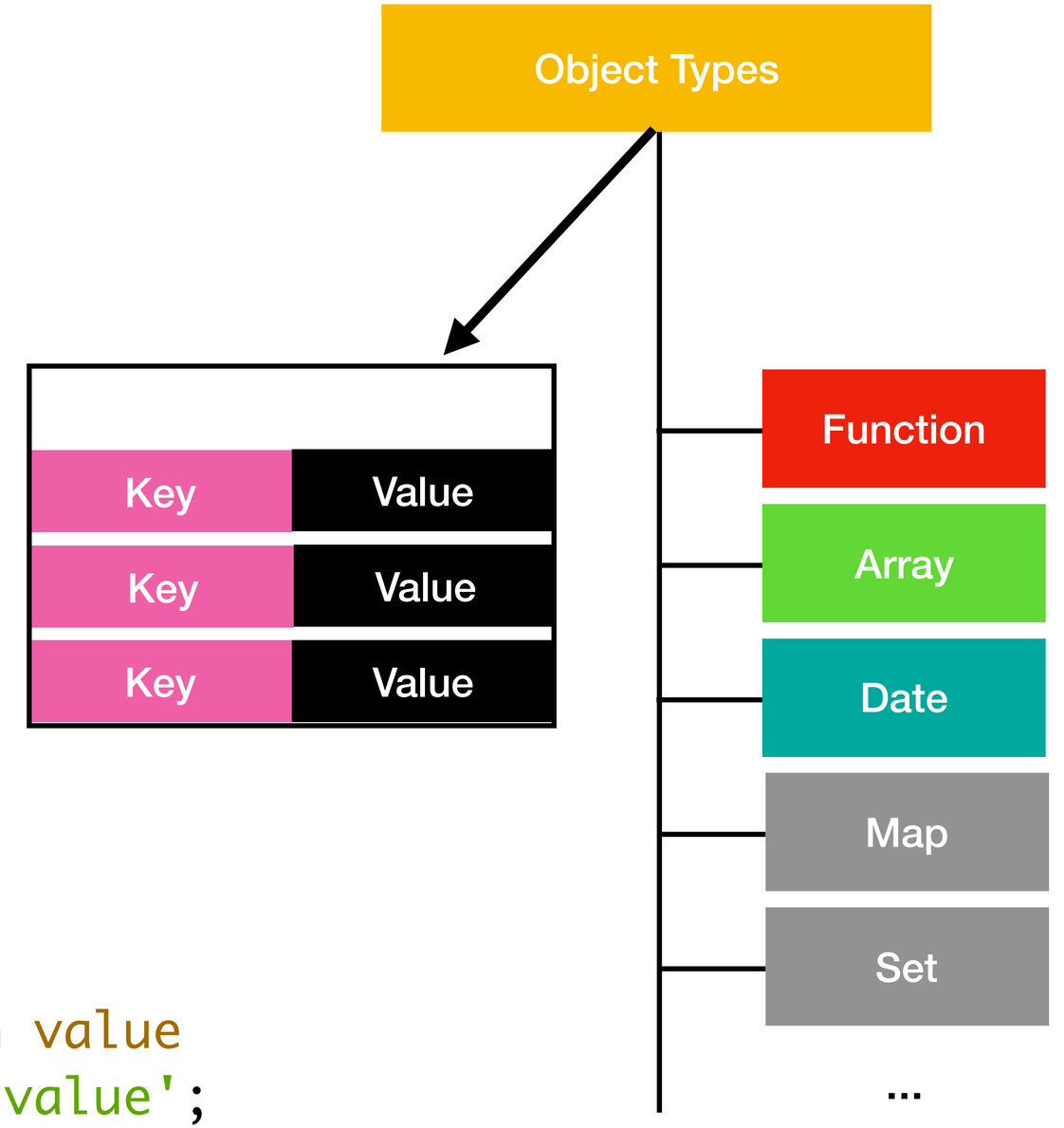
Most important type in JS

Everything is an Object (except primitives types)

Objects can be seen as a collection of properties (dictionaries/hash maps/associative arrays)

Functions are regular objects with the additional capability of being callable.

```
const myObject = {
  a: 12,
  b: 'I am a string'
};

myObject['a'] = 42;  // reassign value
myObject.c = 'dynamically added value';
```



Array

Similar to other languages

Declared using square brackets and addressed with square brackets and a reference to the index starting with 0.

```
const myArray = [1, 15, 42];
let empty = []; // empty array declaration
console.log(myArray.length);
>> 3
```

Variables

var x; Declares a variable, optionally initializing it to a value. (old deprecated)

let x; Declares a block-scoped, local variable, optionally initializing it to a value.

const x = 1; Declares a block-scoped, read-only named constant.

Never use ***var***, always start with ***const***, optionally change to ***let*** if you need to modify the value!

A JavaScript identifier must start with a letter, underscore (_), or dollar sign ($)

Variable scope

JavaScript before ES6 does not have block statement scope; rather, a variable declared within a block is local to the **function (or global scope)** that the block resides within.

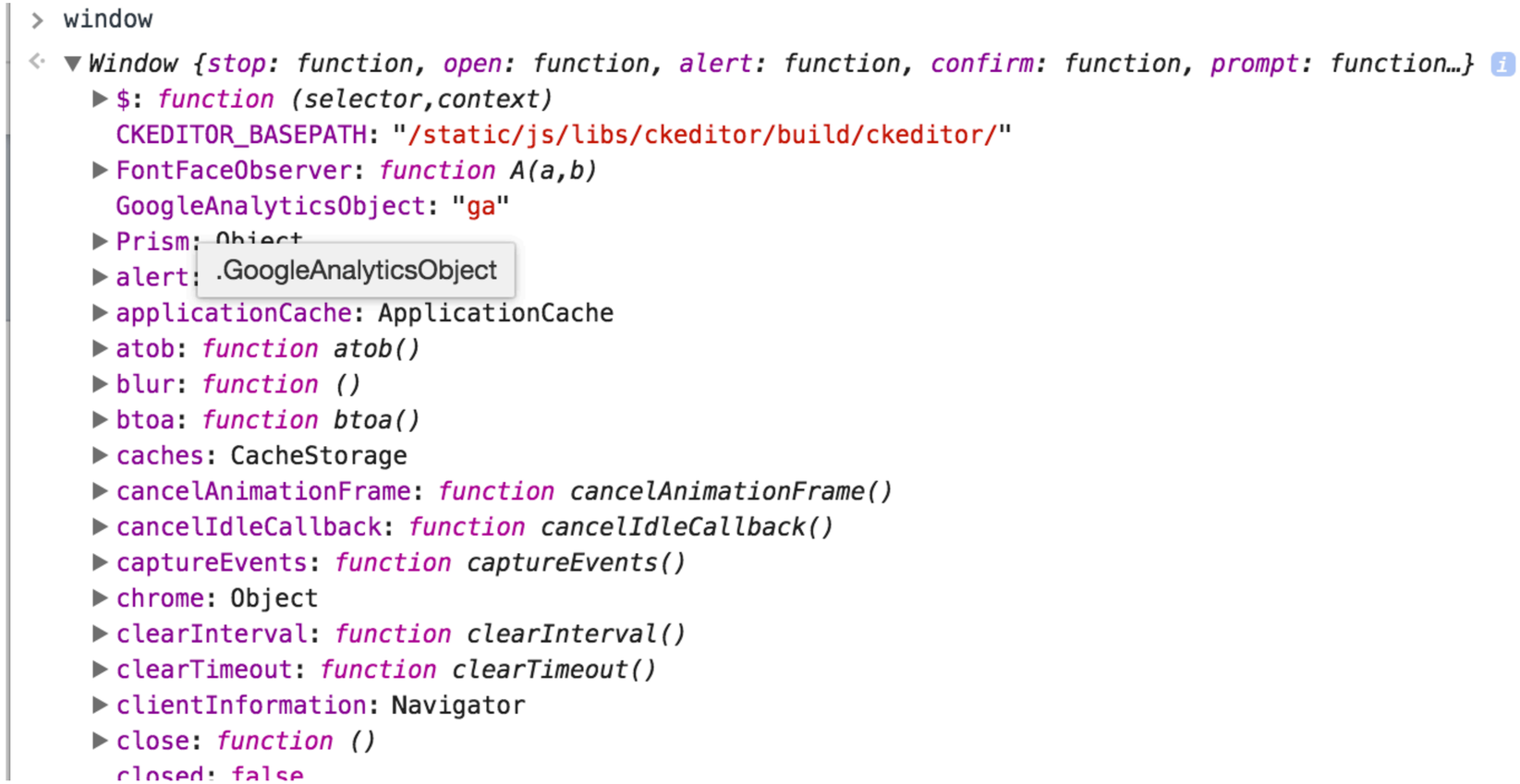
```
if (true) {
  var x = 5;
}
console.log(x);  // x is 5
```

With ***let*** or ***const*** (works like C)

```
if (true) {
  let y = 5;
}
console.log(y);  // ReferenceError: y is not defined
```

Global object: window

Global variables are properties of the *global* `window` *object*



Operators

Operator	Description	Example	Result
==	Equal to	1 == 1	true
===	Equal in value and type	1 === '1'	false
!=	Not equal to	1 != 2	true
!==	Not equal in value and type	1 !== '1'	true
>	Greater than	1 > 2	false
<	Less than	1 < 2	true
>=	Greater than or equal to	1 >= 1	true
<=	Less than or equal to	2 <= 1	false

http://apprize.info/

Conditions

```
if (expression) {
   // Statement(s) to be executed if expression 1 is true
}
else if (expression2) {
   // Statement(s) to be executed if expression 2 is true
}
else {
   // Statement(s) to be executed if no expression is true
}
```

For loops

```
for (let index = 0; index < myArray.length; index++) {
  console.log(myArray[index]);
}
```

```
// Iterate over values of an Iterable (Arrays, list, etc.), not Object
for (const value of myArray) {
  console.log(value);
}
```

```
// Iterate over keys of an Object
for (const key in myObj) {
  console.log(key);
}
```

For loops

```
let animals = ['🐔', '🐷', '🐑', '🐇'];
let names = ['Gertrude', 'Henry', 'Melvin', 'Billy Bob'];

for (let animal of animals) {
  // Random name for our animal
  let nameIdx = Math.floor(Math.random() * names.length);

  console.log(`${names[nameIdx]} the ${animal}`);
}

// Henry the 🐔
// Melvin the 🐷
// Henry the 🐑
// Billy Bob the 🐇
```

```
let oldCar = {
  make: 'Toyota',
  model: 'Tercel',
  year: '1996'
};

for (let key in oldCar) {
  console.log(`${key} --> ${oldCar[key]}`);
}

// make --> Toyota
// model --> Tercel
```

```
// Iterate throught objects key/val
for (const key of Object.keys(someObject)) {…}
```

https://alligator.io/js/for-of-for-in-loops/

Destructuring objects

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

[MDN]

```
let a, b, rest;
[a, b] = [10, 20];
console.log(a); // 10
console.log(b); // 20

[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // 10
console.log(b); // 20
console.log(rest); // [30, 40, 50]

({a, b} = {a: 10, b: 20});
console.log(a); // 10
console.log(b); // 20
```

Functions

https://stackoverflow.com/questions/336859/var-functionname-function-vs-function-functionname

Function declaration

Simplest form, it is a declaration not a statement, no semi-colon needed.

A function declaration is processed when execution enters the context in which it appears, before any step-by-step code is executed.

If you call a function with too many parameters, JS will simply ignore the extra ones. Too few: JS gives the local parameters the special value undefined.

```
function foo([param,[, param,[..., param]]]) {
    [statements]
}
```

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

```
const result = foo(1, 2, 3);
console.log(result); // 3
```

“Anonymous” function expression

A function is a just a regular object that you can call. You can assign it to a variable!

Evaluated when it's reached in the step-by-step execution of the code.

ES2015, the function is assigned a name if possible by inferring it from context.

```
let y = function () {
  // whatever
};
y.name; // “y"
y(); // run func
```

Named function expression

The function has a proper name (fname in this case).

The name of the function is not added to the scope in which the expression appears; the name is in scope within the function itself

```
let z = function fname() {
    console.log(typeof fname); // "function"
};
console.log(typeof fname);     // "undefined"
```

Default function parameters

By default, parameters of functions are undefined.

With ES6 we can have default value for function parameters.

Parameters already encountered are available to later default parameters

```
function multiplyAdd(a, b = 1, c = b + 1) {
  return (a * b) + c;
}

multiplyAdd(5, 2); // 13
multiplyAdd(0);    // 2
```

JavaScript is a “functional” language

Functional programming?

Functional programming is a programming paradigm

Functional programming (often abbreviated FP) is the process of building software by composing pure functions, avoiding shared state, mutable data, and side-effects.

FP focuses on the task, not the implementation

https://blog.codeminer42.com/introduction-to-functional-programming-with-javascript-c06a2540a7c3

Functional programming

From Wikipedia, the free encyclopedia

For subroutine-oriented programming, see Procedural programming.

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are passed to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time; this is in contrast to procedures depending on a local or global state, which may produce different results at different times when called with the same arguments but a different program state. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its origins in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. Another well-known declarative programming paradigm, *logic programming*, is based on relations.[3]

In contrast, imperative programming changes state with commands in the source code, the simplest example being assignment. Imperative programming does have functions—not in the mathematical sense—but in the sense of subroutines. They can have side effects that may change the value of program state. Functions without return values therefore make sense. Because of this, they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program.[3]

Functional programming languages have largely been emphasized in academia rather than in commercial software development. However, prominent programming languages which support functional programming such as Common Lisp, Scheme,[4][5][6][7] Clojure,[8][9] Wolfram Language[10] (also known as Mathematica), Racket,[11] Erlang,[12][13][14] OCaml,[15][16] Haskell,[17][18] and F#[19][20] have been used in industrial and commercial applications by a wide variety of organizations. JavaScript, one of the world's most widely-distributed languages[21][22], has the properties of an untyped functional language[23], as well as imperative and object-oriented paradigms. Functional programming is also

Programming paradigms

- Action
- Agent-oriented
- Array-oriented
- Automata-based
- Concept
- Concurrent computing
  - Relativistic programming
- Data-driven
- Declarative (contrast: Imperative)
  - Constraint
    - Constraint logic
      - Concurrent constraint logic
  - Dataflow
    - Flow-based
    - Cell-oriented (spreadsheets)
    - Reactive
  - **Functional**
    - Functional logic
    - Purely functional
  - Logic
    - Abductive logic
    - Answer set
    - Concurrent logic
    - Functional logic
    - Inductive logic
- Dynamic
- End-user programming

Concrete example: forEach

Print the content of an array in an imperative fashion.

What if we want to do somthing other than print, can we **abstract** the **action** we want to **apply** on the array?

```
function printArray(array) {
  for (const v = 0; i < array.length; i++)
    console.log(array[i]);
}
```

http://eloquentjavascript.net/05_higher_order.html

forEach (cont')

Functions are first-class citizens in JS, as regular objects

The action is abstracted as forEach argument.

We name this action a **callback.**

```
function forEach(array, action) {
  for (const v of array)
    action(v);
}

forEach(["Luke", "Yoda", "Vader"], console.log);
forEach(["Luke", "Yoda", "Vader"], writeToFile);
forEach(["Luke", "Yoda", "Vader"], function(name) {
  console.log(name.toLowerCase());
});
```

forEach (cont')

Functions can be declared inside other functions

Very handy to encapsulate local computations

```
const numbers = [1, 2, 3, 4, 5];
function sum(numbers) {
  let total = 0;
  forEach(numbers, function(number) {
    total += number;
  });
  return total;
}
console.log(sum(numbers)); // 15
```

total is in the lexical scope of the anonymous function.

Higher-order functions

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.

```
function forEach(array, callback) {
  …
}
```

```
function myFunc() {
  const anotherFunc = function() { console.log("inner"); }
  return anotherFunc;
}

const innerFunc = myFunc();
innerFunc(); // "inner"
myFunc()(); // "inner"
```

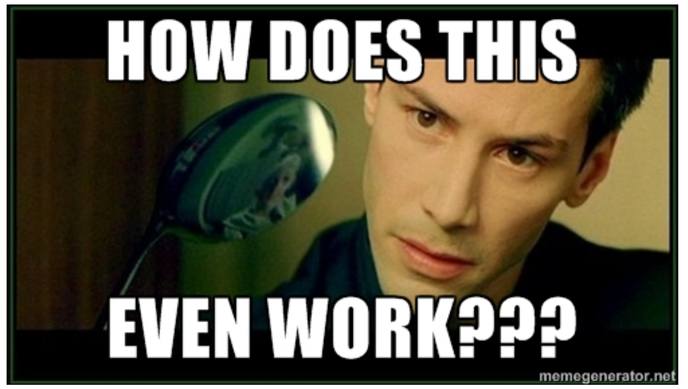
Closures

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

The function defined in the closure ‘remembers’ the environment in which it was created.

```
function greaterThan(n) {
  return function(m) { return m > n; };
}

const greaterThan10 = greaterThan(10);
greaterThan10(11); // true
```

```
let counter = (function() {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  };
})();

console.log(counter.value()); // logs 0
counter.increment();
counter.increment();
console.log(counter.value()); // logs 2
counter.decrement();
console.log(counter.value()); // logs 1
```



[ViralPatel.net]

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

Arrow functions

An arrow function expression has a shorter syntax than a function expression

Much simpler in ES6!!

```
let counter = (function() {
  let privateCounter = 0;
  changeBy = (val) => { return privateCounter += val; }
  return {
    increment: () => changeBy(1), // one liner can remove return and {}
    decrement: () => changeBy(-1),
    value: () => privateCounter,
    reset: (val=0) => { privateCounter = val; },
  };
})();
```

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

Essentials higher-order functions for arrays

Map

Applies a function to all the array's elements and returns a new array with the returned values.

Filter

Creates a new array with all elements that pass the test implemented by the provided function.

Reduce

Applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

Map

The map() method creates a new array with the results of calling a provided function on every element in the calling array.

```
let numbers = [1, 5, 10, 15];
const doubles = numbers.map(x => x * 2);
// doubles is now [2, 10, 20, 30]
// numbers is still [1, 5, 10, 15]

numbers = [1, 4, 9];
const roots = numbers.map(Math.sqrt);
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

Filter

The filter() method creates a new array with all elements that pass the test implemented by the provided function

```
const words = ["spray", "limit", "elite", "exuberant", "destruction",
“present"];

const longWords = words.filter(word => word.length > 6);

console.log(longWords); // ["exuberant", "destruction", "present"]
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

Reduce

The reduce() method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

```
arr.reduce(callback[, initialValue]);
```

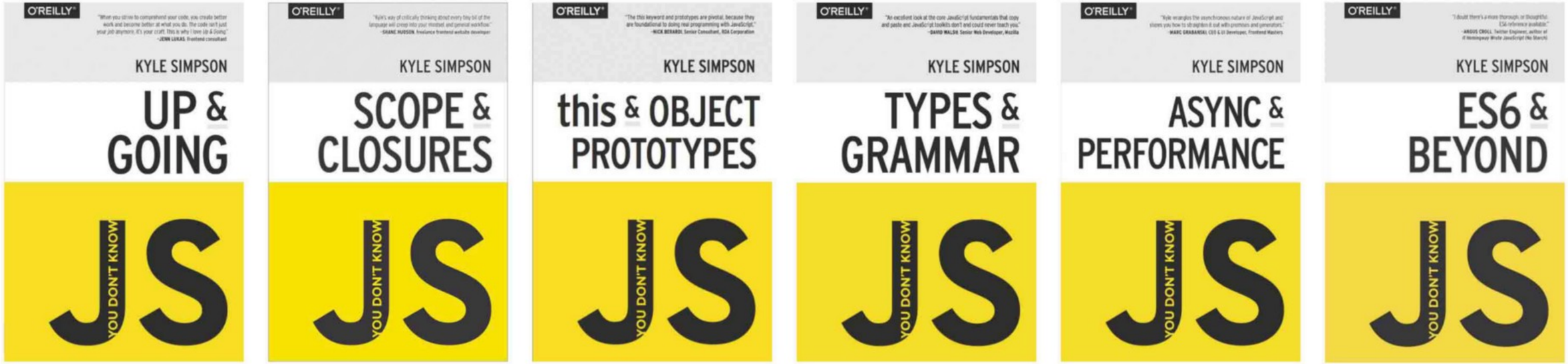
```
const total = [0, 1, 2, 3].reduce((sum, value) => {
  return sum + value;
}, 0);
// total is 6, initial value was 0
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce

All combined

```
const animals = [
  { name: 'Waffles',  type: 'dog', age: 12 },
  { name: 'Fluffy',   type: 'cat', age: 14 },
  { name: 'Spelunky', type: 'dog', age: 4 },
  { name: 'Hank',     type: 'dog', age: 11 },
];


const totalDogYears = animals
  .filter((x) =>  x.type === 'dog')
  .map((x) => x.age)
  .reduce((prev, cur) => prev + cur, 0)
// totalDogYears will be the integer 27 (Waffles 12 +
// Spelunky 4 + Hank 11)
```

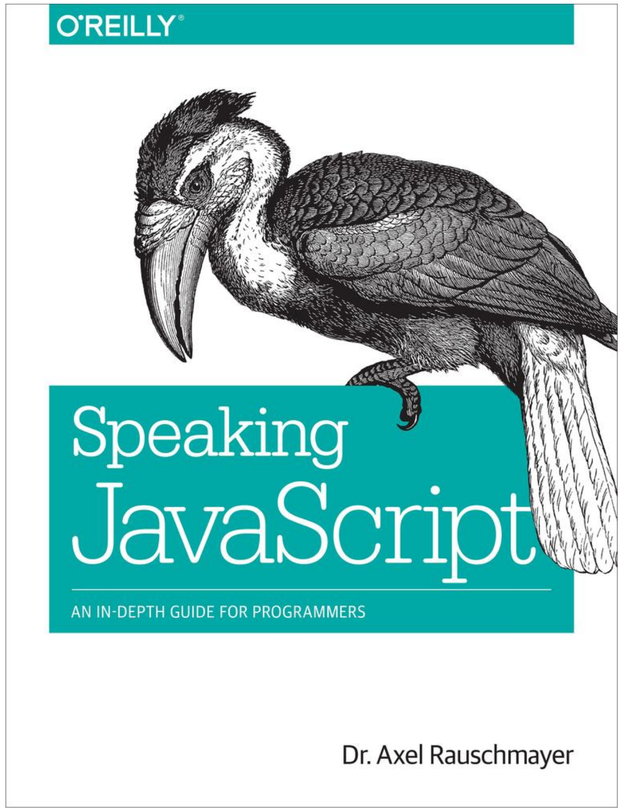
Going further (free resources)

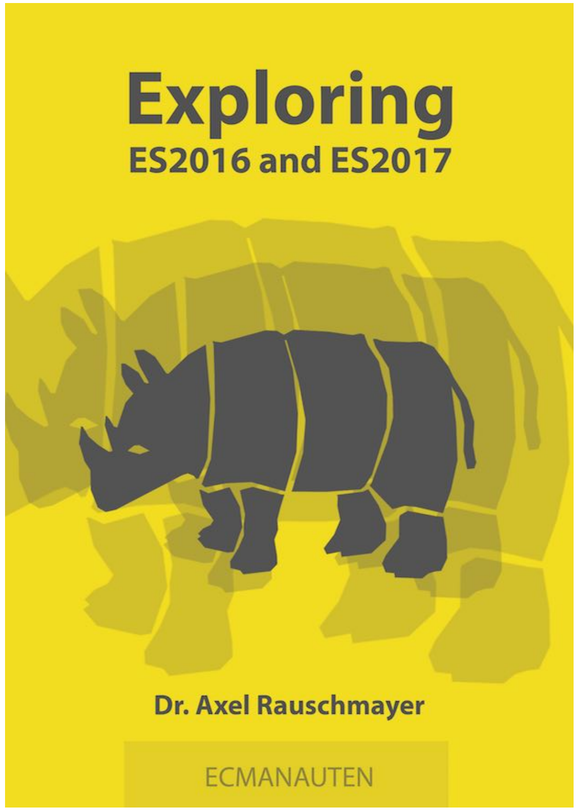


You don’t know Javascript series

https://github.com/getify/You-Dont-Know-JS

Going further (free resources)





Exploring JS series

http://exploringjs.com/

Going further (free resources)

Javascript Allongé

https://leanpub.com/javascriptallongesix/read

Homework

Read YDKJS up & going chapter 2

https://patrickfatrick.gitbooks.io/you-don-t-know-js-up-going/content/ch2.html

Read Functional Light JS chapter 2

https://github.com/getify/Functional-Light-JS/blob/master/manuscript/ch2.md