



Formation JavaScript

Romain Bohdanowicz

Twitter : @bioub - <https://github.com/bioub>

<http://formation.tech/>



Introduction



- ▶ **Romain Bohdanowicz**

Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle

- ▶ **Expérience**

Formateur/Développeur Freelance depuis 2006

Plus de 8000 heures de formation animées

- ▶ **Langages**

Expert : HTML / CSS / JavaScript / PHP / Java

Notions : C / C++ / Objective-C / C# / Python / Bash / Batch

- ▶ **Certifications**

PHP 5 / PHP 5.3 / PHP 5.5 / Zend Framework 1

- ▶ **Particularités**

Premier site web à 12 ans (HTML/JS/PHP), Triathlète à mes heures perdues

- ▶ **Et vous ?**

Langages ? Expérience ? Utilité de cette formation ?



JavaScript IDEs



- ▶ Version orientée Web de IntelliJ IDEA de l'éditeur JetBrains

<https://www.jetbrains.com/webstorm/>

- ▶ Licence : Commercial

Licence entre 35 à 129 euros par an selon le profil et l'ancienneté.

Version d'essai 30 jours.

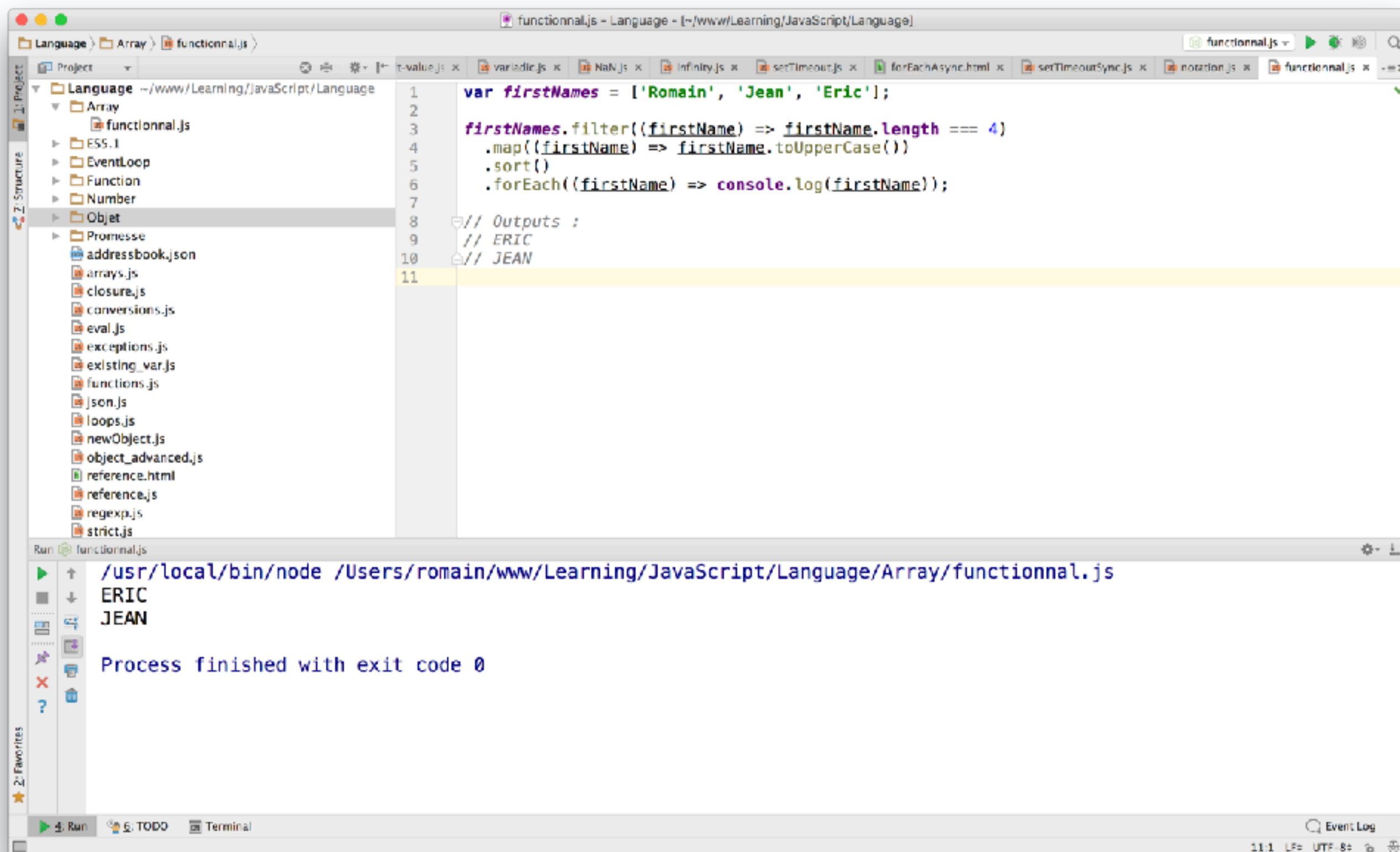
- ▶ Plugins :

Annuaire (642 en novembre 2016) : <https://plugins.jetbrains.com/webStorm>

Langage de création : Java



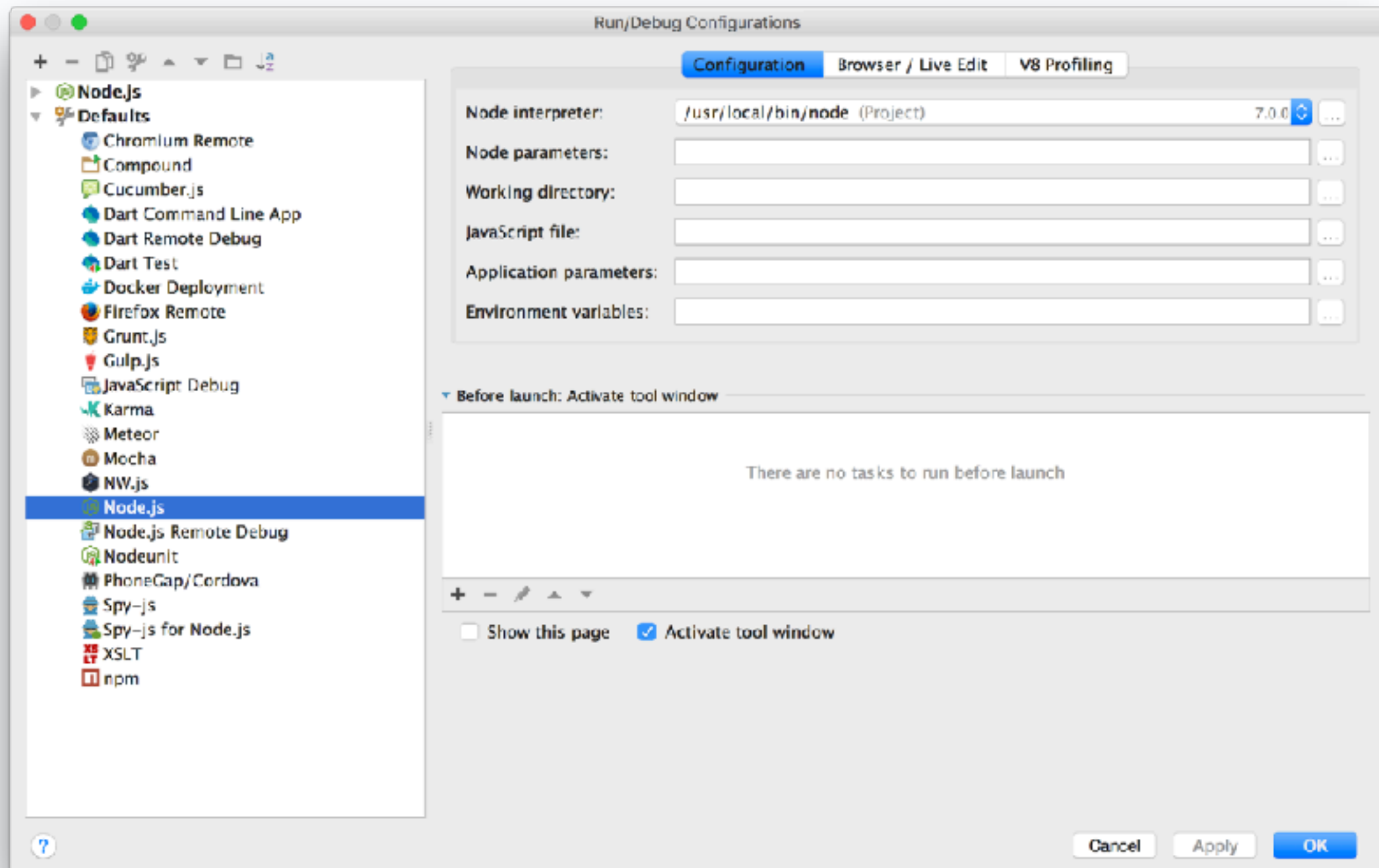
JavaScript IDEs - Webstorm



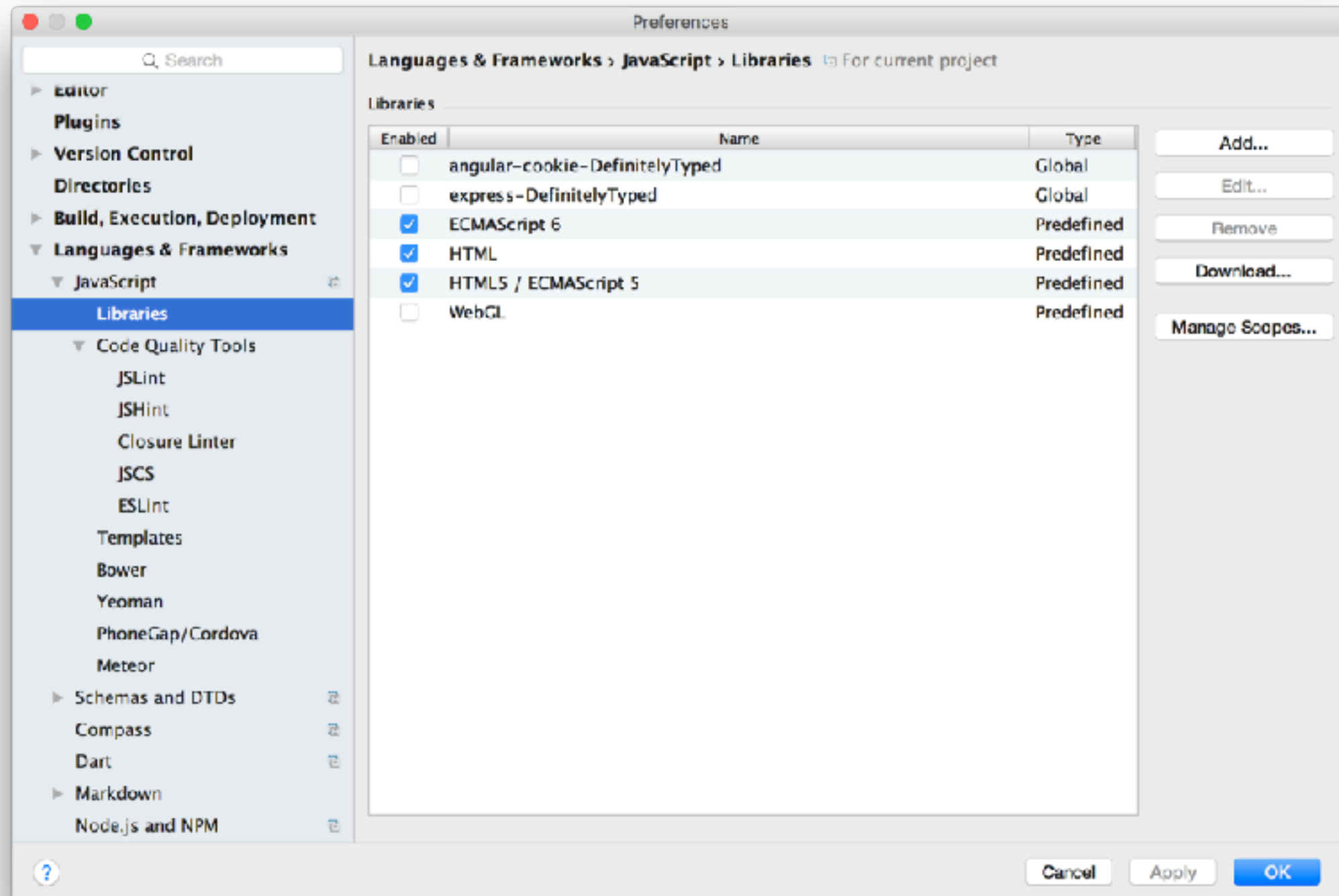
JavaScript IDEs - Webstorm



JavaScript IDEs - Webstorm



JavaScript IDEs - Webstorm





- ▶ IDE créé par Github, tourne sous Electron (Chromium + Node.js)
<https://atom.io>
- ▶ Licence : MIT
La licence open-source la plus permissive
- ▶ Plugins :
Annuaire (5232 en novembre 2016) : <https://atom.io/packages>
Langage de création : JavaScript sous Node.js
Exemples : atom-ternjs, linter, JavaScript Snippets, autocomplete+, autoprefixer...)



JavaScript IDEs - Atom

A screenshot of the Atom IDE interface. The left sidebar shows a project tree with folders like 'TestFlow', 'AmdLoader', 'Flow', 'UIKIT', 'TestFlowWidget', 'TodoDS', and 'UIKIT'. The 'index.html' file in the 'UI' folder of 'TodoDS' is selected. The main editor area shows the content of 'index.html', which is an HTML document with embedded JavaScript and CSS. The code includes comments for application metas, standalone emulation files, and UIKIT files. It also contains a JavaScript section with a 'main' function that uses 'require' to load 'DS/ToDoDS/ToDoDS' and sets up an 'onLoad' event. The status bar at the bottom indicates 'File 0', 'Project 0', 'No Issues', 'TodoDS/index.html', '1:1', '1 deprecation', 'UTF-8', 'HTML', and '1 update'.



- IDE créé par Microsoft, tourne sous Electron (Chromium + Node.js)

<http://code.visualstudio.com/>

- Licence : MIT

La licence open-source la plus permissive

- Plugins :

Annuaire (1867 en novembre 2016) : <https://marketplace.visualstudio.com/VSCode>

Langage de création : JavaScript sous Node.js

- Documentation

<https://code.visualstudio.com/docs>



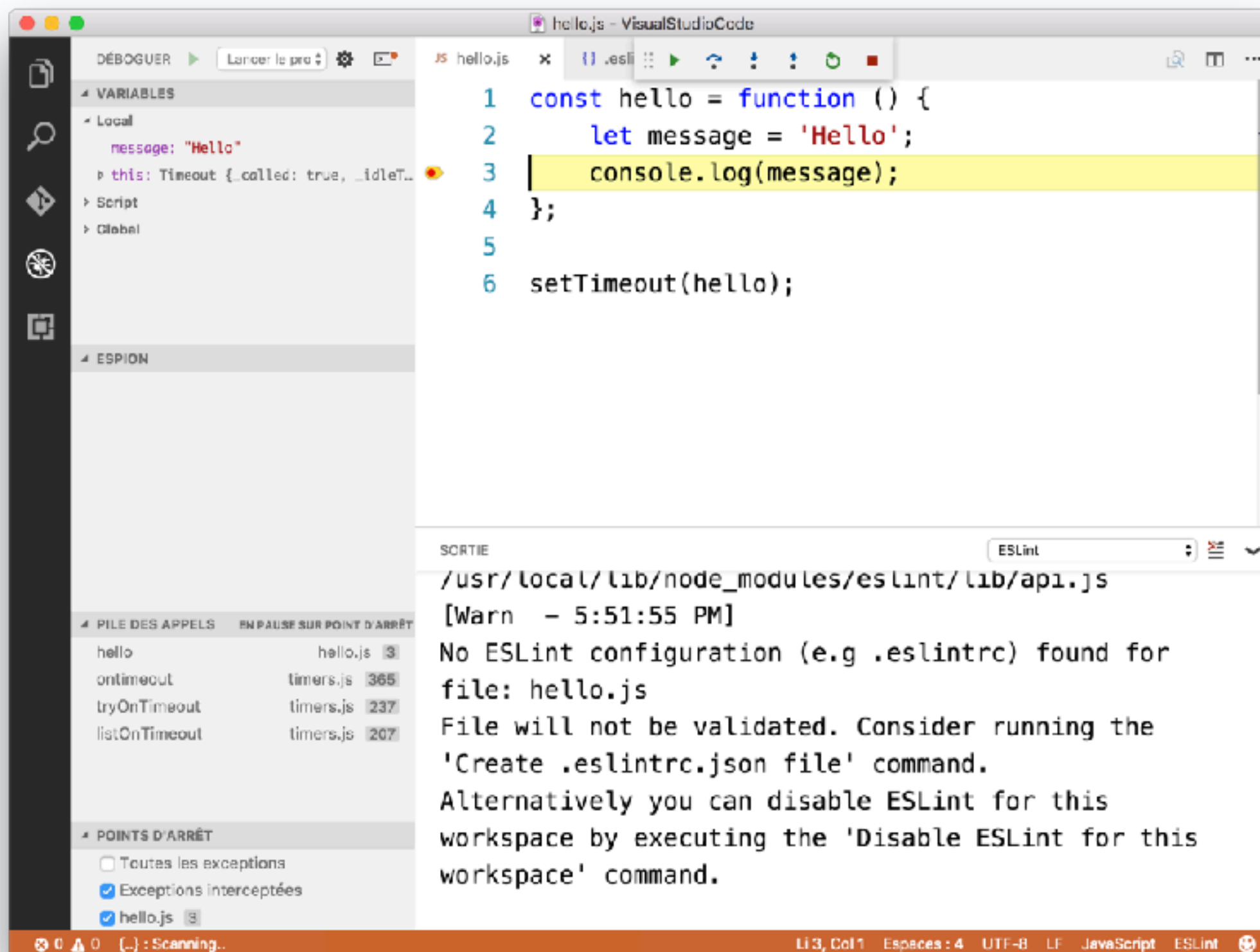
JavaScript IDEs - Visual Studio Code

A screenshot of the Visual Studio Code editor interface. The Explorer sidebar on the left shows a project structure with folders like 'src' and 'app'. The main editor area displays a TypeScript file named 'about.module.ts' with the following code:

```
1 import { Title } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AboutComponent } from './about.component';
5 import { AboutRoutingModule } from './about-routing.module';
6
7 @NgModule({
8   imports: [
9     AboutRoutingModule
10  ],
11   declarations: [
12     AboutComponent
13  ],
14   providers: [
15     Title
16  ],
17 })
18 export class AboutModule { }
```

The status bar at the bottom indicates the file is at line 18, column 1, using UTF-8 encoding and LF line endings.

JavaScript IDEs - Visual Studio Code





- Permet de standardiser les configs des IDEs sur l'indentation et les retours à la ligne
<http://editorconfig.org>
- Supporté par la plupart des IDE
- Il suffit de créer un fichier .editorconfig à la racine d'un projet

```
# EditorConfig is awesome: http://EditorConfig.org

# top-most EditorConfig file
root = true

# Unix-style newlines with a newline ending every file
[*]
end_of_line = lf
insert_final_newline = true
charset = utf-8
indent_style = space
indent_size = 4

# HTML + JS files
[*.{html,js}]
indent_size = 2
```



JavaScript

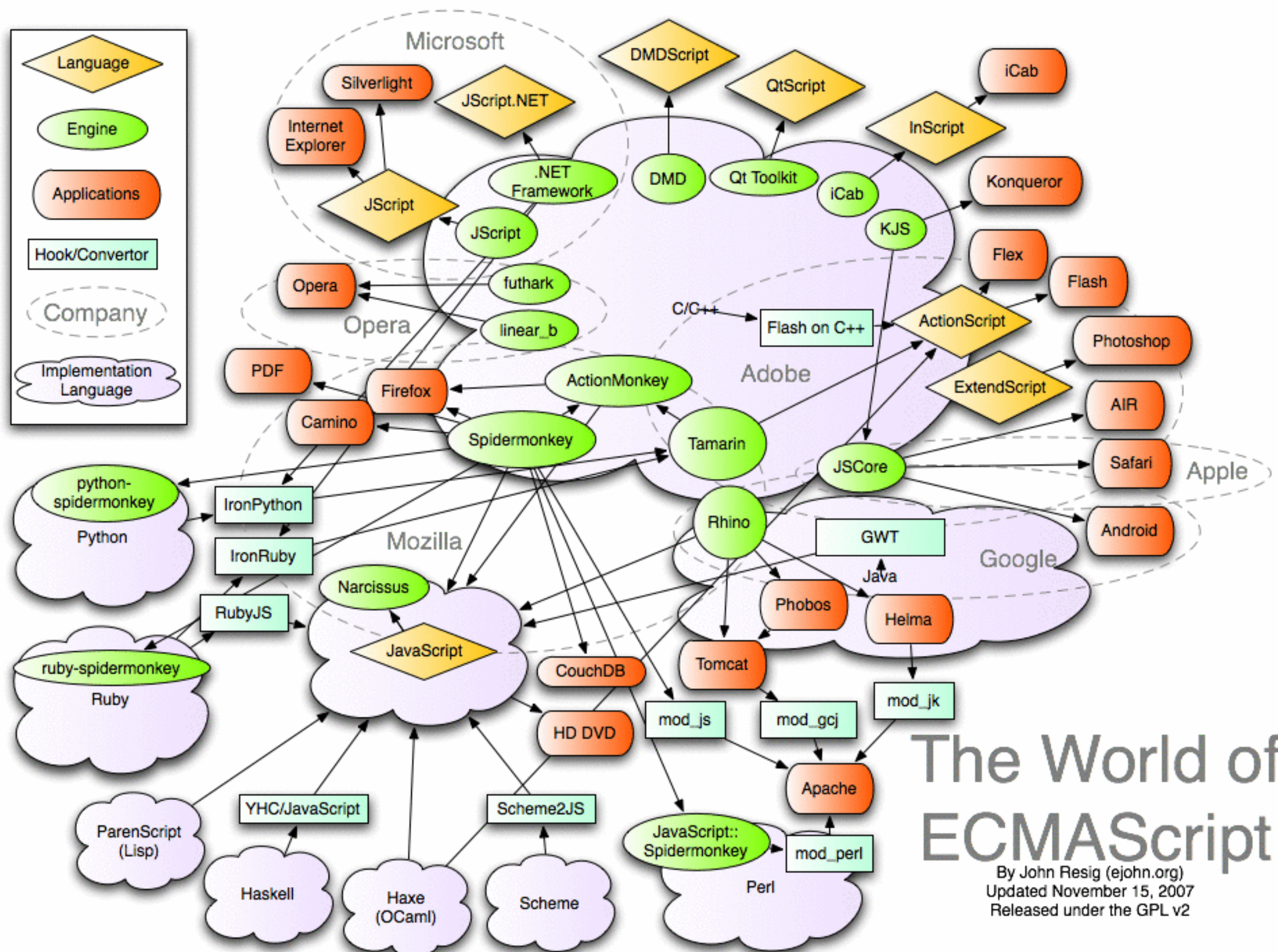


- Langage créé en 1995 par Netscape
- Objectif : permettre le développement de scripts légers qui s'exécutent une fois le chargement de la page terminé
- Exemples de l'époque :
 - Valider un formulaire
 - Permettre du rollover
- Netscape ayant un partenariat avec Sun, nomma le langage JavaScript pour qu'il soit vu comme le petit frère de Java (dont il est inspiré syntaxiquement)
- Fin 1995 Microsoft introduit JScript dans Internet Explorer
- Une norme se crée en 1997 : ECMAScript



- ▶ JavaScript est une implémentation de la norme ECMAScript 262
- ▶ La norme la plus récente est ECMAScript 2017
Aussi appelée ECMAScript 8 ou ES8 (juin 2017)
<https://www.ecma-international.org/ecma-262/8.0/>
- ▶ Le langage a très fortement évolué avec ECMAScript 2015
ECMAScript 6 / ES6 (juin 2015)
<https://www.ecma-international.org/ecma-262/6.0/>
- ▶ Pour connaître la compatibilité des moteurs JS :
<http://kangax.github.io/compat-table/>
- ▶ Compatibilité ES6
Navigateur actuels (octobre 2016) ~ 90% d'ES6
Node.js 6 ~ 90% d'ES6
Internet Explorer 11 ~ 10% d'ES6

JavaScript - ECMAScript





- La norme manque d'exemples et d'information sur les implémentations :
<http://www.ecma-international.org/ecma-262/7.0/>
- Mozilla fournit une documentation open-source sur le langage JavaScript et sur les APIs Web (utiliser la version anglaise qui est plus à jour) :
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- DevDocs permet de retrouver la documentation de Mozilla en mode hors-ligne
<http://devdocs.io/javascript/>



- La syntaxe s'inspire de Java (lui même inspiré de C)
- JavaScript est sensible à la casse, attention aux majuscules/minuscules !
- Les instructions se termine au choix par un point-virgule ou un retour à la ligne (même si les conventions incitent à la l'utilisation du point-virgule)
- 3 types de commentaires
 - `//` le commentaire s'arrête à la fin de la ligne
 - `/*` commentaire ouvrant/fermant `*/`
 - `/**` Documentation JSDoc `*/`



- Les identifiants (noms de variables, de fonctions) doivent respecter les règles suivantes :
 - Contenir uniquement lettres Unicode, Chiffres, \$ et _
 - Ne commencent pas par un chiffre
- Bonnes pratiques :
 - ne pas utiliser d'accents (passage d'un éditeur à un autre)
 - séparer les mots dans l'identifiant par des majuscules (camelCase), ou des _ (snake_case)
 - les identifiants qui commencent par des \$ ou _ sont utilisés par certaines conventions
- Exemples :
 - Valides
i, maVariable, \$div, v1, prénom
 - Invalides
1var, ma-variable



- ▶ Mots clés (ES7) :
break, case, catch, class, const, continue, debugger, default, delete, do, else, export, extends, finally, for, function, if, import, in, instanceof, new, return, super, switch, this, throw, try, typeof, var, void, while, with, yield
- ▶ Mots clés (mode strict) :
let, static
- ▶ Réservés pour une utilisation future :
enum, await
- ▶ Réservés pour une utilisation future (mode strict) :
implements, interface, package, private, protected, public



- Voici les types primitifs en JS
 - number
 - boolean
 - string
- Les types complexes
 - object
 - array
- Les types spéciaux
 - undefined
 - null



▸ Différence primitifs / complexes

En cas d'affectation ou de passage de paramètres, les primitifs ne sont pas modifiés, contrairement aux complexes

```
var boolean = false;
var number = 0;
var string = '';
var object = {};
var array = [];

var modify = function(b, n, s, o, a) {
  b = true;
  n = 1;
  s = 'Romain';
  o.prenom = 'Romain'; // object sera modifié également
  a.push('Romain'); // array sera modifié également
};

modify(boolean, number, string, object, array);

console.log(boolean); // false
console.log(number); // 0
console.log(string); // ''
console.log(object); // { prenom: 'Romain' }
console.log(array); // [ 'Romain' ]
```



- Pas de type spécifique pour les entiers ou les non-signés
- Implémentés en 64 bits en précision double
- Infinity et NaN sont 2 valeurs particulières de type number

```
// decimal
console.log(11); // 11
console.log(11.11); // 11.11

// binary
console.log(0b11); // 3 (ES6)

// octal
console.log(011); // 9
console.log(0o11); // 9 (ES6)

// hexadecimal
console.log(0x11); // 17

// exponentiation
console.log(1e3); // 1000

console.log(typeof 0); // number
```



- ▶ NaN est une valeur de type number pour les opérations impossibles (conversions, nombres complexes...)
- ▶ Une comparaison avec NaN donne systématiquement false (y compris NaN === NaN)

```
console.log(NaN); // NaN
console.log(Math.sqrt(-1)); // NaN
console.log(Number('abc')); // NaN
console.log(Number(undefined)); // NaN

console.log(typeof Math.sqrt(-1)); // number

console.log(NaN == NaN); // false
console.log(NaN === NaN); // false

console.log(isNaN(Math.sqrt(-1))); // true
console.log(Number.isNaN(Math.sqrt(-1))); // true (ES6)

console.log(isFinite(Math.sqrt(-1))); // false
console.log(Number.isFinite(Math.sqrt(-1))); // false (ES6)

console.log(0 < NaN); // false
console.log(0 > NaN); // false
console.log(0 == NaN); // false
console.log(0 === NaN); // false
```



- Infinity est une valeur de type number, une division par zéro est donc possible en JS

```
console.log(Infinity); // Infinity
console.log(1 / 0); // Infinity

console.log(typeof (1 / 0)); // number

console.log(isFinite(1 / 0)); // false
console.log(Number.isFinite(1 / 0)); // false (ES6)

console.log(isNaN(1 / 0)); // false
console.log(Number.isNaN(1 / 0)); // false (ES6)

console.log(0 < Infinity); // true
console.log(0 > Infinity); // false
console.log(0 == Infinity); // false
console.log(0 === Infinity); // false
```

JavaScript - Déclaration de variable



- ▶ Mot clé var

Contrairement à certains langages, on ne déclare pas le type au moment de la création

```
var firstName = 'Romain';  
var lastName = 'Bohdanowicz';
```

- ▶ Déclaration sans var

En cas de déclaration sans le mot clé var, la variable devient globale. Le mode strict apparu en ECMAScript 5 empêche ce comportement.

- ▶ ECMAScript 6

En ES6 une variable peut également se déclarer avec le mot clé let (portée de block), ou const (constante)



- Un identifiant qui n'est pas déclaré est typé undefined

```
var firstName;  
  
console.log(firstName === undefined); // true  
console.log(typeof firstName); // 'undefined'  
  
console.log(lastName === undefined); // ReferenceError: lastName is not defined  
console.log(typeof lastName); // 'undefined'
```



▸ Affectation

| Nom | Opérateur composé | Signification |
|----------------------------------|----------------------|-------------------------|
| Affectation | <code>x = y</code> | <code>x = y</code> |
| Affectation après addition | <code>x += y</code> | <code>x = x + y</code> |
| Affectation après soustraction | <code>x -= y</code> | <code>x = x - y</code> |
| Affectation après multiplication | <code>x *= y</code> | <code>x = x * y</code> |
| Affectation après division | <code>x /= y</code> | <code>x = x / y</code> |
| Affectation du reste | <code>x %= y</code> | <code>x = x % y</code> |
| Affectation après exponentiation | <code>x **= y</code> | <code>x = x ** y</code> |



► Comparaison

| Opérateur | Description | Exemples qui renvoient true |
|-----------------------------|---|---|
| Égalité (==) | Renvoie true si les opérandes sont égaux après conversion en valeurs de mêmes types. | <code>3 == var1</code> <code>"3" == var1</code> <code>3 == '3'</code> |
| Inégalité (!=) | Renvoie true si les opérandes sont différents. | <code>var1 != 4</code> <code>var2 != "3"</code> |
| Égalité stricte (===) | Renvoie true si les opérandes sont égaux et de même type. Voir <code>Object.is()</code> et égalité de type en JavaScript. | <code>3 === var1</code> |
| Inégalité stricte (!==) | Renvoie true si les opérandes ne sont pas égaux ou s'ils ne sont pas de même type. | <code>var1 !== "3"</code> <code>3 !== '3'</code> |
| Supériorité stricte (>) | Renvoie true si l'opérande gauche est supérieur (strictement) à l'opérande droit. | <code>var2 > var1</code> <code>"12" > 2</code> |
| Supériorité ou égalité (>=) | Renvoie true si l'opérande gauche est supérieur ou égal à l'opérande droit. | <code>var2 >= var1</code> <code>var1 >= 3</code> |
| Infériorité stricte (<) | Renvoie true si l'opérande gauche est inférieur (strictement) à l'opérande droit. | <code>var1 < var2</code> <code>"2" < "12"</code> |
| Infériorité ou égalité (<=) | Renvoie true si l'opérande gauche est inférieur ou égal à l'opérande droit. | <code>var1 <= var2</code> <code>var2 <= 5</code> |



► Arithmétiques

En plus des opérations arithmétiques standards (+, -, *, /), on trouve en JS :

| Opérateur | Description | Exemple |
|---|--|---|
| Reste (%) | Opérateur binaire. Renvoie le reste entier de la division entre les deux opérandes. | 12 % 5 renvoie 2. |
| Incrément (++) | Opérateur unaire. Ajoute un à son opérande. S'il est utilisé en préfixe (++x), il renvoie la valeur de l'opérande après avoir ajouté un, s'il est utilisé comme opérateur de suffixe (x++), il renvoie la valeur de l'opérande avant d'ajouter un. | Si x vaut 3, ++x incrémente x à 4 et renvoie 4, x++ renvoie 3 et seulement ensuite ajoute un à x. |
| Décrément (--) | Opérateur unaire. Il soustrait un à son opérande. Il fonctionne de manière analogue à l'opérateur d'incrément. | Si x vaut 3, --x décrémente x à 2 puis renvoie 2, x-- renvoie 3 puis décrémente la valeur de x. |
| Négation unaire (-) | Opérateur unaire. Renvoie la valeur opposée de l'opérande. | Si x vaut 3, alors -x renvoie -3. |
| Plus unaire (+) | Opérateur unaire. Si l'opérande n'est pas un nombre, il tente de le convertir en une valeur numérique. | +"3" renvoie 3. +true renvoie 1. |
| Opérateur d'exponentiation (**) (puissance) | Calcule un nombre (base) élevé à une puissance donnée (soit basepuissance) | 2 ** 3 renvoie 8 10 ** -1 renvoie -1 |



► Logiques

| Opérateur | Usage | Description |
|-----------------|----------------|---|
| ET logique (&&) | expr1 && expr2 | Renvoie expr1 s'il peut être converti à false, sinon renvoie expr2. Dans le cas où on utilise des opérandes booléens, && renvoie true si les deux opérandes valent true, false sinon. |
| OU logique () | expr1 expr2 | Renvoie expr1 s'il peut être converti à true, sinon renvoie expr2. Dans le cas où on utilise des opérandes booléens, renvoie true si l'un des opérandes vaut true, si les deux valent false, il renvoie false. |
| NON logique (!) | !expr | Renvoie false si son unique opérande peut être converti en true, sinon il renvoie true. |



▸ Concaténation

```
console.log("ma " + "chaîne"); // affichera "ma chaîne" dans la console
```

▸ Ternaire

```
var statut = (âge >= 18) ? "adulte" : "mineur";
```

▸ Voir aussi

Opérateurs binaires, in, instanceof, delete, typeof...

▸ Attention au '+' qui donne priorité à la concaténation

```
console.log("1" + "1" + "1"); // "111"  
console.log("1" + "1" + 1); // "111"  
console.log("1" + 1 + 1); // "111"  
console.log(1 + 1 + "1"); // "21"
```



► Priorités

| Type d'opérateur | Opérateurs individuels |
|---------------------------|---|
| membre | <code>.</code> <code>[]</code> |
| appel/création d'instance | <code>()</code> <code>new</code> |
| négation/incrémentation | <code>!</code> <code>~</code> <code>-</code> <code>+</code> <code>++</code> <code>--</code> <code>typeof</code> <code>void</code> <code>delete</code> |
| multiplication/division | <code>*</code> <code>/</code> <code>%</code> |
| addition/soustraction | <code>+</code> <code>-</code> |
| décalage binaire | <code><<</code> <code>>></code> <code>>>></code> |
| relationnel | <code><</code> <code><=</code> <code>></code> <code>>=</code> <code>in</code> <code>instanceof</code> |
| égalité | <code>==</code> <code>!=</code> <code>===</code> <code>!==</code> |
| ET binaire | <code>&</code> |
| OU exclusif binaire | <code>^</code> |
| OU binaire | <code> </code> |
| ET logique | <code>&&</code> |
| OU logique | <code> </code> |
| conditionnel | <code>?:</code> |
| assignation | <code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code><<=</code> <code>>>=</code> <code>>>>=</code> <code>&=</code> <code>^=</code> <code> =</code> |
| virgule | <code>,</code> |



▸ Conversions implicites

```
console.log(3 * '3'); // 9  
console.log(3 + '3'); // '33'  
console.log(!'texte'); // false
```

▸ Conversions explicites

```
console.log(parseInt('33.33')); // 33  
console.log(parseFloat('33.33')); // 33.33  
console.log(Number('33.33')); // 33.33  
console.log(Boolean('texte')); // true  
console.log(String(33.33)); // '33.33'
```



- ▶ **Standard Built-in Objects**

Les objets prédéfinies par le langage, voir la doc de Mozilla pour une liste exhaustive

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

- ▶ **Ex : String, Array, Date, Math, RegExp, JSON...**



▸ Structure et API

En JS les tableaux ne sont pas des structures de données mais un type d'objet (une « classe »).

```
var firstNames = ['Romain', 'Eric'];

console.log(firstNames.length); // 2

console.log(firstNames[0]); // Romain
console.log(firstNames[firstNames.length - 1]); // Eric

// boucler sur tous les éléments (ES5)
firstNames.forEach(function(firstName) {
    console.log(firstName); // Romain Eric
});

var newLength = firstNames.push('Jean'); // ajoute Jean à la fin
var last = firstNames.pop(); // retire et retourne le dernier (Jean)
var newLength = firstNames.unshift("Jean") // ajoute Jean au début
var first = firstNames.shift(); // retire et retourne le premier (Jean)

var pos = firstNames.indexOf("Romain"); // indice de l'élément
var removedItem = firstNames.splice(pos, 1); // suppression d'un élément à
partir de l'indice pos
var shallowCopy = firstNames.slice(); // copie d'un tableau
```



▸ if ... else

```
if (typeof console === 'object') {  
    console.log('console est un objet');  
}  
else {  
    // oups  
}
```

▸ switch

```
switch (alea) {  
    case 0:  
        console.log('zéro');  
        break;  
    case 1:  
    case 2:  
    case 3:  
        console.log('un, deux ou trois');  
        break;  
    default:  
        console.log('entre quatre et neuf');  
}
```


JavaScript - Structures de contrôle



▸ while

```
var alea = Math.floor(Math.random() * 10);

while (alea > 0) {
    console.log(alea);
    alea = parseInt(alea / 2);
}
```

▸ do ... while

```
do {
    var alea = Math.floor(Math.random() * 10);
}
while (alea % 2 === 1);

console.log(alea);
```

▸ for

```
for (var i=0; i<10; i++) {
    aleas.push(Math.floor(Math.random() * 10));
}

console.log(aleas.join(', ')); // 6, 6, 7, 0, 5, 1, 2, 8, 9, 7
```



Fonctions en JavaScript

Fonctions en JavaScript - Introduction



- JavaScript est très consommateur de fonctions
 - réutilisation / factorisation
 - récursivité
 - fonction de rappel / écouteur
 - closure
 - module

Fonctions en JavaScript - Syntaxe



▸ Function declaration

```
function addition(nb1, nb2) {  
    return Number(nb1) + Number(nb2);  
}  
  
console.log(addition(2, 3)); // 5
```

▸ Anonymous function expression

```
var addition = function (nb1, nb2) {  
    return Number(nb1) + Number(nb2);  
};  
  
console.log(addition(2, 3)); // 5
```

▸ Named function expression

```
var addition = function addition(nb1, nb2) {  
    return Number(nb1) + Number(nb2);  
};  
  
console.log(addition(2, 3)); // 5
```

Fonctions en JavaScript - Function Declaration



- En JavaScript, les fonctions et variables sont hissées (hoisted) au début de la portée dans laquelle elles ont été déclarée.
- Il est donc possible d'appeler une fonction avant sa déclaration
- Pas d'erreur en cas de redéclaration de fonctions, la seconde écrase la première

```
function hello() {  
  return 'Hello 1';  
}  
  
console.log(hello()); // 'Hello 2'  
  
function hello() {  
  return 'Hello 2';  
}
```

Fonctions en JavaScript - Function Expression



- Avec une function expression, la variable est hissée en début de portée
- Mais la fonction est créée au moment où l'expression s'exécute

```
var hello = function () {  
    return 'Hello 1';  
};  
  
console.log(hello()); // 'Hello 1'  
  
var hello = function () {  
    return 'Hello 2';  
};
```

Fonctions en JavaScript - Constantes



- En ES6 on pourrait même empêcher la redéclaration grâce au mot clé `const`

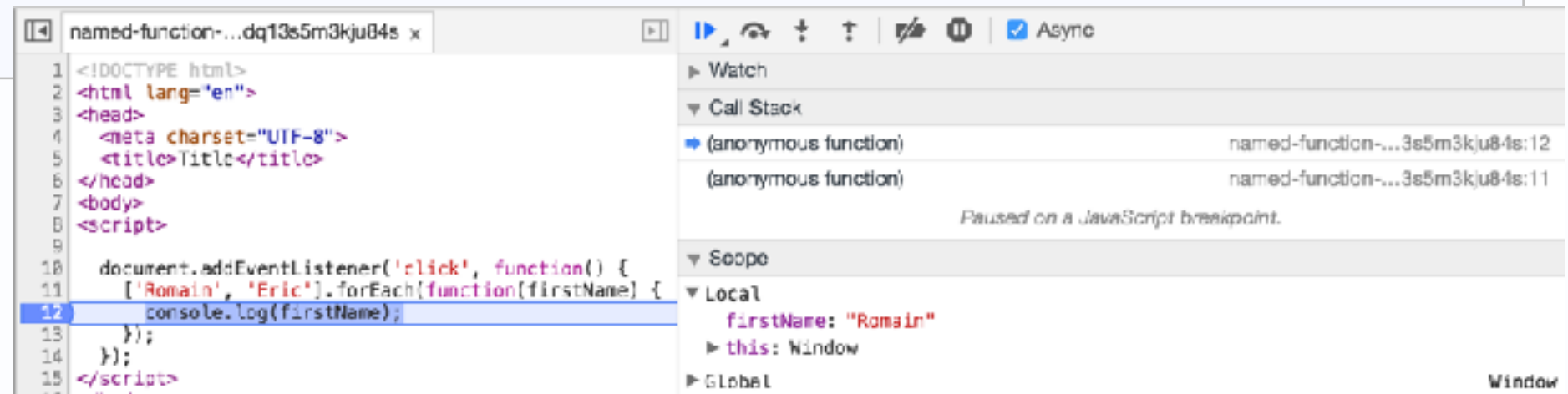
```
const hello = function () {  
  return 'Hello 1';  
};  
  
console.log(hello());  
  
// SyntaxError: Identifier 'hello' has already been declared  
const hello = function () {  
  return 'Hello 2';  
};
```

Fonctions en JavaScript - Named Function Expression

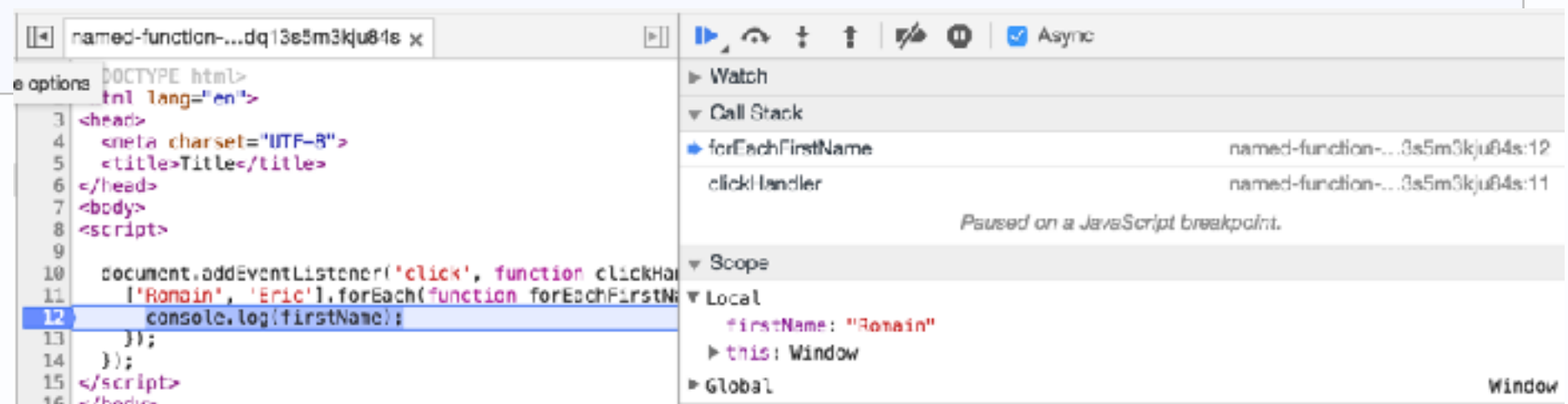


► Anonymous function expression vs Named function expression

```
document.addEventListener('click', function() {  
  ['Romain', 'Eric'].forEach(function(firstName) {  
    console.log(firstName);  
  });  
});
```



```
document.addEventListener('click', function clickHandler() {  
  ['Romain', 'Eric'].forEach(function forEachFirstName(firstName) {  
    console.log(firstName);  
  });  
});
```





► Paramètres

Comme pour les variables, on ne déclare pas les types des paramètres d'entrées et de retours.

Les paramètres ne font pas partie de la signature de la fonction, seul l'identifiant compte, on peut donc appeler une fonction avec plus ou moins de paramètres que prévu.

```
var sum = function(a, b) {  
  return a + b;  
};  
  
console.log(sum(1, 2)); // 3  
console.log(sum('1', '2')); // '12'  
console.log(sum(1, 2, 3)); // 3  
console.log(sum(1)); // NaN
```



▸ Exceptions

En cas d'utilisation anormale d'une fonction, on peut sortir en lançant une exception.

- N'importe quel type peut être envoyé via le mot clé `throw`, mais privilégier les objets de type `Error` et dérivés qui interceptent les fichiers, pile d'appel et numéro de lignes.
- On ne peut pas lancer intercepter une exception avec `try..catch` si elle est lancée dans un callback asynchrone

```
var sum = function(a, b) {  
  if (typeof a !== 'number' && typeof b !== 'number') {  
    throw new Error('sum needs 2 number')  
  }  
  return a + b;  
};  
  
try {  
  sum('1', '2'); // sum needs 2 number  
}  
catch (e) {  
  console.log(e.message);  
}
```

Fonctions en JavaScript - Valeur par défaut



▸ Valeur par défaut

Les paramètres non renseignés lors de l'appel d'une fonction reçoivent la valeur `undefined`.

```
// using undefined
var sum = function(a, b, c) {
  if (c === undefined) {
    c = 0;
  }
  return a + b + c;
};

console.log(sum(1, 2)); // 3

// using or
var sum = function(a, b, c) {
  c = c || 0;
  return a + b + c;
};

console.log(sum(1, 2)); // 3
```



► Fonction Variadique

Pour récupérer les paramètres supplémentaires (non déclarés), on peut utiliser la variable `arguments`. Cette variable n'étant pas un tableau, on ne peut pas utiliser les fonctions du type `Array` (même si des astuces existent).

```
var sum = function(a, b) {  
  var result = a + b;  
  
  for (var i=2; i<arguments.length; i++) {  
    result += arguments[i];  
  }  
  
  return result;  
};  
  
console.log(sum(1, 2, 3, 4)); // 10
```

Fonctions en JavaScript - Imbrication



▸ Fonctions imbriquées

En JavaScript on peut imbriquer les fonctions, la portée d'une fonction étant la fonction qui la contient.

```
var sumArray = function(array) {  
  var sum = function(a, b) {  
    return a + b;  
  };  
  return array.reduce(sum);  
};  
  
console.log(sumArray([1, 2, 3, 4])); // 10  
console.log(typeof sum); // 'undefined'
```

Fonctions en JavaScript - Portées



► Portées

Lorsque l'on imbrique des fonctions, les portées supérieures restent accessibles.

```
var a = function() {  
  var b = function() {  
    var c = function() {  
      console.log(typeof a); // function  
      console.log(typeof b); // function  
      console.log(typeof c); // function  
    };  
    c();  
  };  
  b();  
};  
a();
```

Fonctions en JavaScript - Closure

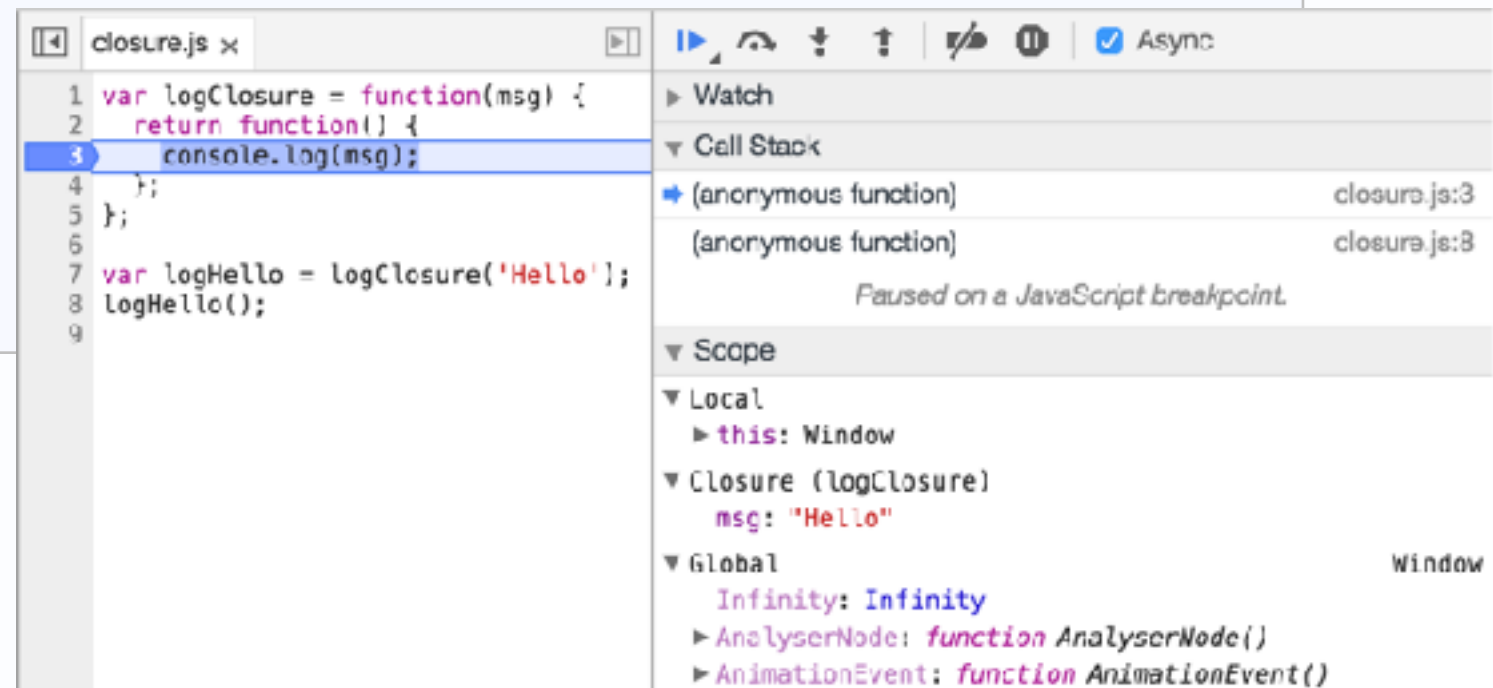


► Closure

Si 2 fonctions sont imbriquées et que la fonction interne est appelée en dehors (par valeur de retour ou asynchronisme), on parle de closure.

La portée des variables au moment du passage dans la fonction externe est sauvegardée.

```
var logClosure = function(msg) {  
  return function() {  
    console.log(msg);  
  };  
};  
  
var logHello = logClosure('Hello');  
logHello();
```



Fonctions en JavaScript - Exemple de Closure



▸ Sans Closure

```
// affiche 4 4 4 dans 1 seconde  
for(var i = 1; i <= 3; i++) {  
    setTimeout(function() {  
        console.log(i);  
    }, 1000);  
}
```

▸ Avec Closure

```
// affiche 1 2 3 dans 1 seconde  
for(var i = 1; i <= 3; i++) {  
    setTimeout(function(rememberI) {  
        return function() {  
            console.log(rememberI);  
        };  
    })(i), 1000);  
}
```


Fonctions en JavaScript - Callbacks



▸ Callback

Lorsqu'une fonction est passée en paramètre d'entrée d'une autre fonction en vue d'être appelée plus tard, on parle de callback.

▸ Callback synchrone / asynchrone

Une fonction recevant un callback peut être synchrone, c'est à dire qu'elle doit s'exécuter entièrement avant d'appeler les instructions suivantes, ou asynchrone ce qui signifie que la fonction sera appelée dans un prochain passage de la « boucle d'événements »

```
var firstNames = ['Romain', 'Eric'];

firstNames.forEach(function(firstName) {
  console.log(firstName);
});

setTimeout(function() {
  console.log('Hello in 100ms');
}, 100);
```



- API recevant un callback synchrone

```
var firstNames = ['Romain', 'Eric'];

var forEachSync = function(array, callback) {
  for (var i=0; i<array.length; i++) {
    callback(array[i], i, array);
  }
};

forEachSync(firstNames, function(firstName) {
  console.log(firstName);
});

console.log('After forEachSync');

// Outputs :
// Romain
// Eric
// After forEachSync
```

Fonctions en JavaScript - Callback Asynchrone



- API recevant un callback asynchrone

```
var firstNames = ['Romain', 'Eric'];

var forEachASync = function(array, callback) {
  for (var i=0; i<array.length; i++) {
    setTimeout(callback, 0, array[i], i, array);
  }
};

forEachASync(firstNames, function(firstName) {
  console.log(firstName);
});

console.log('After forEachASync');

// Outputs :
// After forEachASync
// Romain
// Eric
```

Fonctions en JavaScript - Boucle d'événements



- Les moteurs JS sont par défaut mono-thread et mono-processus, ils ne peuvent donc exécuter qu'une seule tâche à la fois.
- Une boucle d'événements permet de passer d'un callback à l'autre de manière très performante, ex : traiter le clic d'un bouton entre 2 étapes d'une animation
- JavaScript est non-bloquant, il stocke les événements à traiter sous la forme d'une file de message et appellera les callbacks lorsqu'il sera disponible
- Bonne pratique : les callbacks doivent avoir un temps d'exécution court pour ne pas ralentir l'appel des callbacks suivants

```
setTimeout(function() {  
    console.log('1 fois dans 3 secondes');  
}, 3000);  
  
var intervalId = setInterval(function() {  
    console.log('toutes les 2 secondes');  
}, 2000);  
  
setTimeout(function() {  
    console.log('Bye bye');  
    clearInterval(intervalId);  
}, 15000);
```

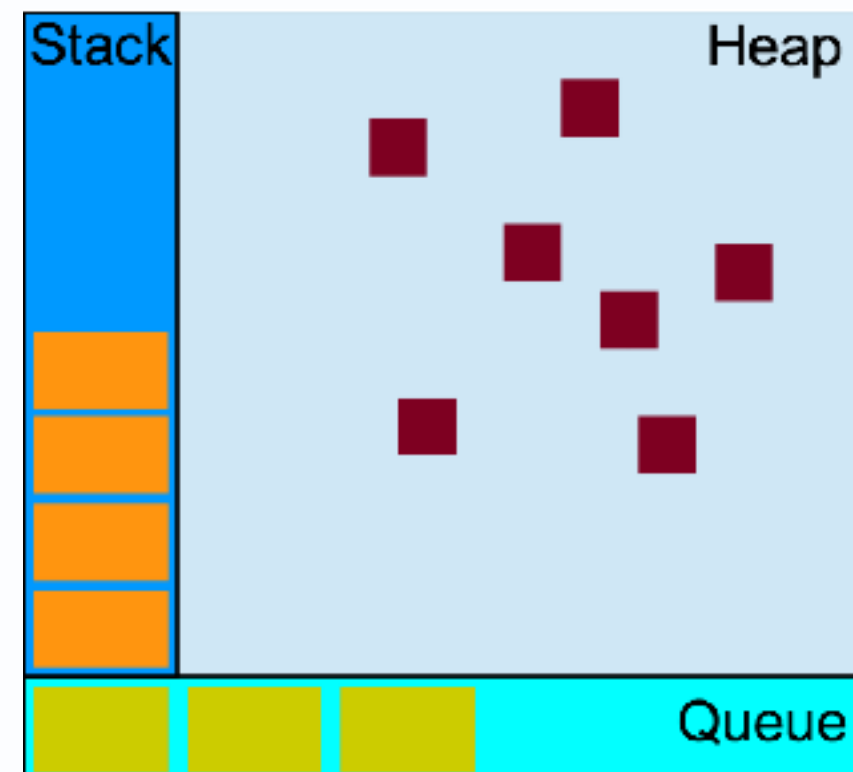
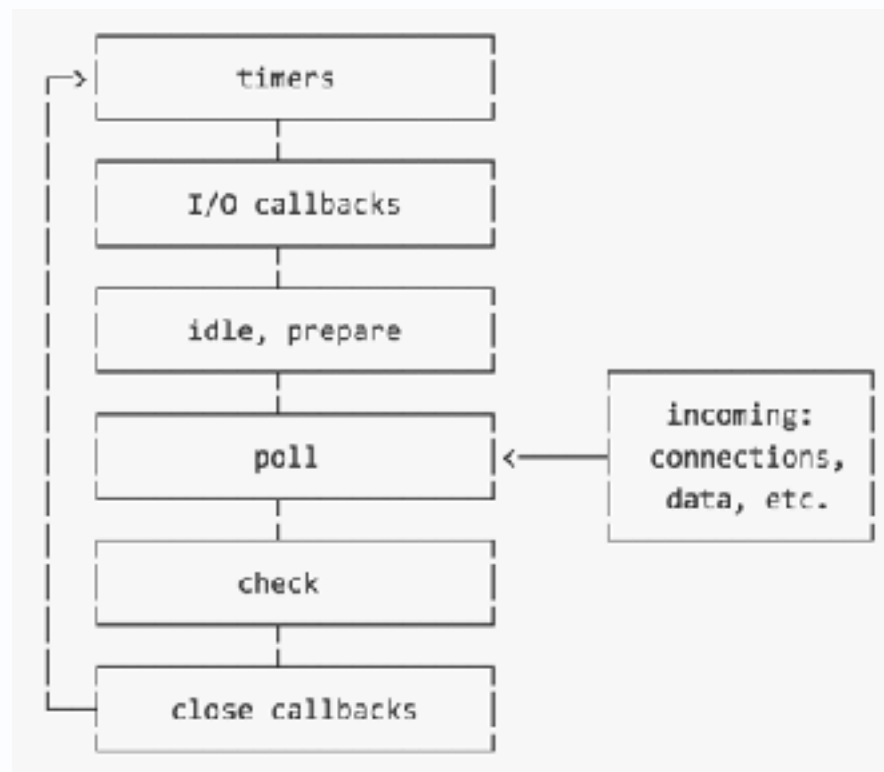
Fonctions en JavaScript - Boucle d'événements



▸ Boucle d'événements

Lorsqu'un programme JS est démarré, il tourne dans une boucle d'événements. Tant qu'il y a des appels en cours dans la pile d'appels, où des callbacks en attente dans la file de callback, on ne passe pas à la prochaine itération. Dans le navigateur, un seul thread est en charge du JavaScript et du rendu, pour un rendu à 60FPS il faut qu'un passage dans la boucle JS + rendu ne dépasse pas 16,67ms.

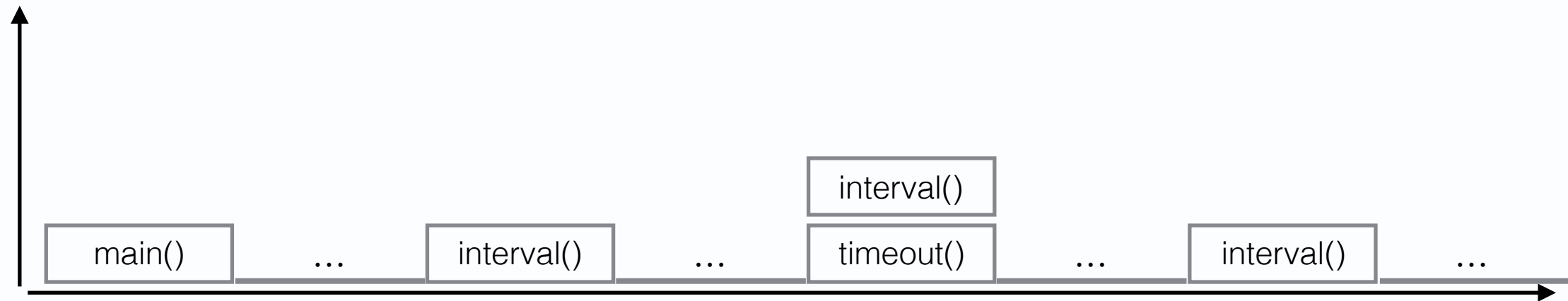
- What the heck is the event loop anyway? | JSConf EU 2014
<https://www.youtube.com/watch?v=8aGhZQkoFbQ>



Fonctions en JavaScript - Boucle d'événements



▸ Boucle d'événements



```
setInterval(function interval() {  
  console.log('interval 1ms')  
}, 1000);  
  
setTimeout(function timeout() {  
  console.log('timeout 2ms')  
}, 2000);
```



► Object function

```
var contact = {  
  prenom: 'Romain',  
  nom: 'Bohdanowicz'  
};  
  
function saluer(prenom) {  
  return 'Bonjour ' + prenom + ' je suis ' + this.prenom;  
}  
  
console.log(saluer('Eric')); // Bonjour Eric je suis undefined  
console.log(saluer.call(contact, 'Eric')); // Bonjour Eric je suis Romain  
console.log(saluer.apply(contact, ['Eric'])); // Bonjour Eric je suis Romain
```

Fonctions en JavaScript - Modules



▸ Module

Contrairement à Node.js, il n'y a pas de portée de fichier dans le navigateur, pour éviter les conflits de nom, on utilise généralement des fonctions anonymes pour créer une portée de fichier, c'est la notion de Module.

▸ Immediately Invoked Function Expression (IIFE)

Lorsque

```
(function($, global) {  
  'use strict';  
  
  function MonBouton(options) {  
    this.options = options || {};  
    this.value = options.value || 'Valider';  
  }  
  
  MonBouton.prototype.creer = function(container) {  
    $(container).append('<button>'+this.value+'</button>')  
  };  
  
  global.MonBouton = MonBouton;  
})(jQuery, window));
```




- Jeu du plus ou moins
 - Générer un entier aléatoire entre 0 et 100 (API Math sur MDN)
 - Demander et récupérer la saisie, afficher si le nombre est plus grand, plus petit ou trouvé (API Readline sur Node.js)
 - Pouvoir trouver en plusieurs tentative (problème d'asynchronisme)
 - Stocker les essais dans un tableau et les réafficher entre chaque tour (API Array sur MDN)
 - Afficher une erreur si la saisie n'est pas un nombre (API Number sur MDN)



JavaScript Orienté Objet



- **Paradigme**

Par opposition à un modèle objet orienté classe, le modèle objet de JavaScript est orienté prototype

- **Classe**

La notion de classe ou d'interface n'existe pas (seulement dans les docs où sous la forme de sucre syntaxique)

- **Modèle statique vs Modèle dynamique**

Il n'y a pas de définition statique du type d'un objet, l'ajout de propriété ou de méthode se fait dynamiquement à la création de l'objet

JavaScript Orienté Objet - Objets préinstanciés



- Il y a un certain nombre d'objet définis au niveau du langage

```
Math.random();  
JSON.stringify({});  
console.log(typeof Math); // object  
console.log(typeof JSON); // object
```

- D'autres par l'environnement d'exécution (Node.js, Navigateur, Mobile...)

```
console.log(typeof console); // object (dans le navigateur et Node.js)  
console.log(typeof document); // object (dans le navigateur)
```



▸ Extensibilité

On peut étendre (sauf verrou), n'importe quel objet. Etendre les objets standards est cependant considéré comme une mauvaise pratique (sauf polyfill). Attention à la casse lorsque vous modifiez une propriété.

```
Math.sum = function(a, b) {  
  return a + b;  
};  
console.log(Math.sum(1, 2)); // 3
```

▸ On peut également modifier ou supprimer des propriétés

```
var randomBackup = Math.random;  
Math.random = function() {  
  return 0.5;  
};  
  
console.log(Math.random()); // 0.5  
Math.random = randomBackup;  
console.log(Math.random()); // quelque chose aléatoire comme 0.24554522  
  
delete Math.sum;  
console.log(Math.sum); // undefined
```

JavaScript Orienté Objet - Objets ponctuels



- Création d'un objet avec l'objet global Object :

```
var instructor = new Object();
instructor.firstName = 'Romain';
instructor.hello = function() {
    return 'Hello my name is ' + this.firstName;
};

console.log(instructor.hello()); // Hello my name is Romain
```

- Création d'un objet avec la syntaxe Object Literal (recommandé) :

```
var instructor = {
    firstName: 'Romain',
    hello: function() {
        return 'Hello my name is ' + this.firstName;
    }
};

console.log(instructor.hello()); // Hello my name is Romain
```



- Accès aux objets possible :

- Avec l'opérateur .
- Avec des crochets

```
var instructor = {  
  firstName: 'Romain',  
  hello: function() {  
    return 'Hello my name is ' + this.firstName;  
  }  
};  
  
instructor.firstName = 'Jean';  
console.log(instructor.hello()); // Hello my name is Jean  
  
instructor['firstName'] = 'Eric';  
console.log(instructor['hello']()); // Hello my name is Eric
```



- En utilisant une fonction constructeur (avec closure) :

```
var Person = function (firstName) {  
    this.firstName = firstName;  
  
    this.hello = function () {  
        // firstName existe aussi grâce à la closure  
        return 'Hello my name is ' + this.firstName;  
    };  
};  
  
var instructor = new Person('Romain');  
  
console.log(instructor.hello()); // Hello my name is Romain  
console.log(typeof instructor); // object  
console.log(instructor instanceof Object); // true  
console.log(instructor instanceof Person); // true  
  
for (var prop in instructor) {  
    if (instructor.hasOwnProperty(prop)) {  
        console.log(prop); // firstName puis hello  
    }  
}
```




- En utilisant une fonction constructeur + son prototype :

```
var Person = function(firstName) {  
    this.firstName = firstName;  
};  
  
Person.prototype.hello = function () {  
    return 'Hello my name is ' + this.firstName;  
};  
  
var instructor = new Person('Romain');  
  
console.log(instructor.hello()); // Hello my name is Romain  
console.log(typeof instructor); // object  
console.log(instructor instanceof Object); // true  
console.log(instructor instanceof Person); // true  
  
for (var prop in instructor) {  
    if (instructor.hasOwnProperty(prop)) {  
        console.log(prop); // firstName  
    }  
}
```



- En utilisant une fonction constructeur + son prototype :

```
var Instructor = function(firstName, speciality) {
    Person.apply(this, arguments); // héritage des propriétés de l'objet (recopie
    // dynamique)
    this.speciality = speciality;
}

Instructor.prototype = new Person; // héritage du type

// Redéfinition de méthode
Instructor.prototype.hello = function() {
    // Appel de la méthode parent
    return Person.prototype.hello.call(this) + ', my speciality is ' + this.speciality;
};

var instructor = new Instructor('Romain', 'JavaScript');

console.log(instructor.hello()); // Hello my name is Romain
console.log(typeof instructor); // object
console.log(instructor instanceof Object); // true
console.log(instructor instanceof Person); // true
console.log(instructor instanceof Instructor); // true

for (var prop in instructor) {
    if (instructor.hasOwnProperty(prop)) {
        console.log(prop); // firstName, speciality
    }
}
```



- Définition Wikipedia :

La programmation orientée prototype est une forme de programmation orientée objet sans classe, basée sur la notion de prototype. Un prototype est un objet à partir duquel on crée de nouveaux objets.

- Comparaison des modèles à classes et à prototypes

- Objets à classes :

- Une classe définie par son code source est statique ;
 - Elle représente une définition abstraite de l'objet ;
 - Tout objet est instance d'une classe ;
 - L'héritage se situe au niveau des classes.

- Objets à prototypes :

- Un prototype défini par son code source est mutable ;
 - Il est lui-même un objet au même titre que les autres ;
 - Il a donc une existence physique en mémoire ;
 - Il peut être modifié, appelé ;
 - Il est obligatoirement nommé ;
 - Un prototype peut être vu comme un exemplaire modèle d'une famille d'objet ;
 - Un objet hérite des propriétés (valeurs et méthodes) de son prototype ;



- En ECMAScript/JavaScript, l'écriture `foo.bar` s'interprète de la façon suivante :
 1. Le nom `foo` est recherché dans la liste des identificateurs déclarés dans le contexte d'appel de fonction courant (déclarés par `var`, ou comme paramètre de la fonction) ;
 2. S'il n'est pas trouvé :
 - Continuer la recherche (retour à l'étape 1) dans le contexte de niveau supérieur (s'il existe),
 - Sinon, le contexte global est atteint, et la recherche se termine par une erreur de référence.
 3. Si la valeur associée à `foo` n'est pas un objet, il n'a pas de propriétés ; la recherche se termine par une erreur de référence.
 4. La propriété `bar` est d'abord recherchée dans l'objet lui-même ;
 5. Si la propriété ne s'y trouve pas :
 - Continuer la recherche (retour à l'étape 4) dans le prototype de cet objet (s'il existe) ;
 - Si l'objet n'a pas de prototype associé, la valeur indéfinie (`undefined`) est retournée ;
 6. Sinon, la propriété a été trouvée et sa référence est retournée.

JavaScript Orienté Objet - JSON



- JSON, JavaScript Object Notation est la sérialisation d'un objet JavaScript
- Seuls les types string, number, boolean, array et regexp sont sérialisable, les fonctions et prototype sont perdus
- On se sert de ce format pour échanger des données entre 2 programmes ou pour créer de la config
- Le format résultant est proche de Object Literal, les clés sont obligatoirement entre guillemets "", un code JSON est une syntaxe Object Literal valide

```
{  
  "name": "My Address Book",  
  "contacts": [  
    {  
      "firstName": "Bill",  
      "lastName": "Gates"  
    },  
    {  
      "firstName": "Steve",  
      "lastName": "Jobs"  
    }  
  ]  
}
```



- JavaScript depuis ECMAScript 5 fourni l'objet global JSON qui contient 2 méthodes, `parse` (désérialiser) et `stringify` (sérialiser)

```
var contact = {  
  prenom: 'Romain',  
  nom: 'Bohdanowicz'  
};  
  
var json = JSON.stringify(contact);  
console.log(json); // {"prenom":"Romain","nom":"Bohdanowicz"}  
  
var object = JSON.parse(json);  
console.log(object.prenom); // Romain
```

JavaScript Orienté Objet - Exercice



- Reprendre le jeu du plus ou moins
- Créer un objet Random avec la syntaxe Object Literal et y regrouper les fonctions aléatoires
- Créer une fonction constructeur Jeu recevant un objet en paramètres d'entrée
- Créer une méthode jouer() tel que le code suivant soit fonctionnel
- Prévoir des valeurs par défaut pour min et max

```
'use strict';  
  
const Random = ...;  
  
const Jeu = ...;  
  
const jeu = new Jeu({  
  min: 0,  
  max: 100  
});  
  
jeu.jouer();
```



ECMAScript 5.1



- ▶ Après ECMAScript 3, le groupe ECMAScript avance sur une nouvelle version, ECMAScript 4 qui inclut notamment les classes et les types.
- ▶ ES4 sera supporté par ActionScript (AS3) mais jamais par les navigateurs qui travaillent à une version 3.1 qui s'appellera 5 puis 5.1 après corrections pour ne pas prêter à confusion.
- ▶ **Compatibilité**
CH13+, FF4+, SF5.1+, OP11.6+, IE9+ (10+ pour le mode strict, 8+ pour l'objet global JSON)
<http://kangax.github.io/compat-table/es5/>
- ▶ **Aperçu des nouvelles fonctionnalités**
<https://dev.opera.com/articles/introducing-ecmascript-5-1/>

ECMAScript 5.1 - Mode Strict



- Le mode strict est un mode d'exécution apparu en ECMAScript 5.1 qui vient limiter un certain nombre de mauvaises pratiques ou de problèmes de sécurité.
- Par opposition au mode strict (strict mode), on parle parfois de sloppy mode

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

ECMAScript 5.1 - Mode Strict



▸ Activer le mode strict

▸ Globalement

```
'use strict';  
// ... code strict...
```

▸ A partir d'une ligne

```
// ... code sloppy ...  
'use strict';  
// ... code strict...
```

▸ Dans une fonction

```
(function () {  
  'use strict';  
  // ... code strict ...  
})();
```

ECMAScript 5.1 - Mode Strict



▸ Mots clés réservés

▸ Sloppy Mode

```
var let = 'Hello';  
console.log(let);
```

▸ Strict Mode

```
'use strict';  
  
var let = 'Hello'; // SyntaxError: Unexpected strict mode reserved word  
console.log(let);
```

ECMAScript 5.1 - Mode Strict



▸ Oubli du mot clé var

▸ Sloppy Mode

```
(function() {  
  // firstName est globale  
  firstName = 'Romain';  
})();  
  
console.log(firstName); // Romain
```

▸ Strict Mode

```
(function() {  
  'use strict';  
  // ReferenceError: firstName is not defined  
  firstName = 'Romain';  
  
  // ReferenceError: i is not defined  
  for (i=0; i<10; i++) {}  
})();
```



▸ Désactivation de with

▸ Sloppy Mode

```
var int, floor = function(n) {  
    return parseInt(String(n));  
};  
  
with (Math) {  
    int = floor(random() * 101); // floor global ? Math.floor ?  
}  
  
console.log(int); // 42
```

▸ Strict Mode

```
'use strict';  
  
var entier, floor = function(n) {  
    return parseInt(String(n));  
};  
  
with (Math) { // SyntaxError: Strict mode code may not include a with statement  
    entier = floor(random() * 101);  
}  
  
console.log(entier); // 42
```

ECMAScript 5.1 - Mode Strict



- Pas d'identifiant dans eval

- Sloppy Mode

```
eval('var sum = 1 + 2');  
console.log(sum); // 3
```

- Strict Mode

```
'use strict';  
eval('var sum = 1 + 2');  
console.log(sum); // ReferenceError: sum is not defined
```



▸ Supprimer des variables

▸ Sloppy Mode

```
var firstName = 'Romain';
var contact = {
  firstName: 'Romain'
};

delete contact.firstName;
console.log(contact.firstName); // undefined

delete firstName;
console.log(firstName); // Romain
```

▸ Strict Mode

```
'use strict';

var firstName = 'Romain';
var contact = {
  firstName: 'Romain'
};

delete contact.firstName;
console.log(contact.firstName); // undefined

delete firstName; // SyntaxError: Delete of an unqualified identifier in strict mode.
console.log(firstName); // Romain
```




▸ Utilisation de this

▸ Sloppy Mode

```
var Contact = function(firstName) {  
    this.firstName = firstName;  
};  
  
var contact = Contact('Romain');  
  
console.log(global.firstName); // Romain (Node.js)  
console.log(window.firstName); // Romain (Browser)
```

▸ Strict Mode

```
'use strict';  
  
var Contact = function(firstName) {  
    this.firstName = firstName; // TypeError: Cannot set property 'firstName' of  
    // undefined  
};  
  
var contact = Contact('Romain');  
  
console.log(global.firstName); // undefined  
console.log(window.firstName); // undefined
```

ECMAScript 5.1 - Immutable globals



- Nouvelles variables globales non modifiables

```
console.log(undefined);  
console.log(NaN);  
console.log(Infinity);
```



▸ Programmation fonctionnelle

Paradigme de programmation dans lequel les fonctions ont un rôle central et viennent remplacer les concepts de programmation impérative comme les variables, boucles, etc...

▸ Tableaux

Le type Array contient depuis ES5 quelques fonction qui permettent ce type de programmation (filter, map, sort, reverse, reduce, forEach...)

```
var firstNames = ['Eric', 'Romain', 'Jean', 'Eric', 'Jean'];

firstNames
  .filter(firstName => firstName.length === 4) // filtre ceux de 4 lettres
  .map(firstName => firstName.toUpperCase()) // transforme en majuscule
  .sort() // trie croissant
  .reverse() // inverse l'ordre
  .reduce((firstNames, firstName) => { // dédoublone
    if (!firstNames.includes(firstName)) {
      firstNames.push(firstName)
    }
    return firstNames;
  }, [])
  .forEach(firstName => console.log(firstName)); // affiche
```

```
// Outputs :
// JEAN
// ERIC
```

ECMAScript 5.1 - Function.prototype.bind



- La méthode bind d'une fonction retourne une nouvelle fonction sur laquelle sera liée une nouvelle valeur this

```
var contact = {  
  firstName: 'Romain'  
};  
  
var hello = function() {  
  return 'Hello my name is ' + this.firstName;  
};  
  
console.log(hello()); // Hello my name is undefined  
var helloContact = hello.bind(contact);  
console.log(helloContact()); // Hello my name is Romain
```



- JavaScript depuis ECMAScript 5 fourni l'objet global JSON qui contient 2 méthodes, `parse` (désérialiser) et `stringify` (sérialiser)

```
var contact = {  
    prenom: 'Romain',  
    nom: 'Bohdanowicz'  
};  
  
var json = JSON.stringify(contact);  
console.log(json); // {"prenom":"Romain","nom":"Bohdanowicz"}  
  
var object = JSON.parse(json);  
console.log(object.prenom); // Romain
```



- On peut masquer une méthode derrière une propriété en lecture

```
var contact = {  
  firstName: 'Romain',  
  lastName: 'Bohdanowicz',  
  get fullName() {  
    return this.firstName + ' ' + this.lastName;  
  }  
};  
  
console.log(contact.fullName); // Romain Bohdanowicz
```



- On peut également masquer une méthode derrière l'écriture d'une propriété

```
var contact = {  
  firstName: 'John',  
  lastName: 'Doe',  
  set fullName(fullName) {  
    var parts = fullName.split(' ');  
    this.firstName = parts[0];  
    this.lastName = parts[1];  
  }  
};  
  
contact.fullName = 'Romain Bohdanowicz';  
console.log(contact.firstName); // Romain  
console.log(contact.lastName); // Bohdanowicz
```

ECMAScript 5.1 - Object.getPrototypeOf



- Object.getPrototypeOf permet de retrouver le prototype d'un objet déjà instancié

```
var Person = function (firstName) {  
    this.firstName = firstName;  
};  
  
Person.prototype.hello = function () {  
    return 'Hello my name is ' + this.firstName;  
};  
  
var instructor = new Person('Romain');  
console.log(Object.getPrototypeOf(instructor)); // Person { hello: [Function] }  
console.log(Person.prototype); // Person { hello: [Function] }
```


ECMAScript 5.1 - Object.defineProperty



- ▶ Permet une définition plus fine d'une propriété

```
var contact = { firstName: 'Romain' };

Object.defineProperty(contact, 'lastName', {
  value: 'Bohdanowicz',
  writable: false,
  enumerable: false,
  configurable: false
});

// writable: false
contact.lastName = 'Doe';
console.log(contact.lastName); // Bohdanowicz

// enumerable: false
for (var prop in contact) {
  console.log(prop); // firstName
}

// enumerable: false
console.log(JSON.stringify(contact)); // {"firstName":"Romain"}

// configurable: false
try {
  Object.defineProperty(contact, 'lastName', { value: 'Doe' });
}
catch (e) {
  console.log(e.message); // Cannot redefine property: lastName
}
```

ECMAScript 5.1 - Object.defineProperty



- ▶ En mode strict, une propriété en lecture seule lance une exception en écriture.

```
'use strict';

var contact = {
  firstName: 'Romain'
};

Object.defineProperty(contact, 'lastName', {
  value: 'Bohdanowicz',
  writable: false,
  enumerable: false,
  configurable: false
});

// writable: false
try {
  contact.lastName = 'Doe';
}
catch (e) {
  console.log(e.message); // Cannot assign to read only property 'lastName' of
  // object '#<Object>'
}
```

ECMAScript 5.1 - Object.defineProperty



- On peut masquer des méthodes derrière des propriétés en lecture/écriture

```
var contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz'
};

Object.defineProperty(contact, 'fullName', {
  set: function(fullName) {
    var parts = fullName.split(' ');
    this.firstName = parts[0];
    this.lastName = parts[1];
  },
  get: function() {
    return this.firstName + ' ' + this.lastName;
  }
});

console.log(contact.fullName); // Romain Bohdanowicz

contact.fullName = 'John Doe';
console.log(contact.firstName); // John
console.log(contact.lastName); // Doe
```



- Object.keys permet de lister les propriétés propres et énumérables

```
var Person = function (firstName) {  
    this.firstName = firstName;  
};  
  
Person.prototype.hello = function () {  
    return 'Hello my name is ' + this.firstName;  
};  
  
var instructor = new Person('Romain');  
console.log(Object.keys(instructor)); // [ 'firstName' ]
```

ECMAScript 5.1 - Object.preventExtensions



- Il est possible d'empêcher l'extension d'un objet

```
var contact = {  
  firstName: 'Romain'  
};  
  
Object.preventExtensions(contact);  
console.log(Object.isExtensible(contact)); // false  
  
contact.name = 'Bohdanowicz';  
console.log(contact.name); // undefined
```

ECMAScript 5.1 - Object.preventExtensions



- En mode strict, écrire dans un objet non-extensible provoque une exception

```
'use strict';

var contact = {
  firstName: 'Romain'
};

Object.preventExtensions(contact);
console.log(Object.isExtensible(contact)); // false

contact.name = 'Bohdanowicz';
console.log(contact.name); // TypeError: Can't add property name, object is not
extensible
```



- Résumé des appels aux méthodes `Object.preventExtensions`, `Object.seal` et `Object.freeze`

| Function | L'objet devient non extensible | configurable à false sur chaque propriété | writable à false sur chaque propriété |
|---------------------------------------|--------------------------------|---|---------------------------------------|
| <code>Object.preventExtensions</code> | Oui | Non | Non |
| <code>Object.seal</code> | Oui | Oui | Non |
| <code>Object.freeze</code> | Oui | Oui | Oui |



- Grâce à `Object.create`, l'héritage se fait sans dupliquer les propriétés dans le prototype.

```
var Instructor = function (firstName, speciality) {  
    Person.apply(this, arguments); // héritage des propriétés de l'objet (recopie  
    dynamique)  
    this.speciality = speciality;  
};  
  
Instructor.prototype = Object.create(Person.prototype); // héritage du type et des  
méthodes  
Instructor.prototype.constructor = Instructor;  
  
// Redéfinition de méthode  
Instructor.prototype.hello = function () {  
    // Appel de la méthode parent  
    return Person.prototype.hello.call(this) + ', my speciality is ' +  
    this.speciality;  
};  
  
var instructor = new Instructor('Romain', 'JavaScript');  
  
console.log(instructor.hello()); // Hello my name is Romain  
console.log(typeof instructor); // object  
console.log(instructor instanceof Object); // true  
console.log(instructor instanceof Person); // true  
console.log(instructor instanceof Instructor); // true  
console.log(instructor.constructor);
```




ECMAScript 6



- ▶ ECMAScript 6, aussi connu sous le nom ECMAScript 2015 ou ES6 est la plus grosse évolution du langage depuis sa création (juin 2015)

<http://www.ecma-international.org/ecma-262/6.0/>

- ▶ Le langage est enfin adapté à des application JS complexes (modules, promesses, portées de blocks...)

- ▶ Pour découvrir les nouveautés d'ECMAScript 2015 / ES6

<http://es6-features.org/>



- Compatibilité (novembre 2016) :
 - Dernière version de Chrome/Opera, Edge, Firefox, Safari : ~ 90%
 - Node.js 6 et 7 : ~ 90% d'ES6
 - Internet Explorer 11 : ~ 10% d'ES6
- Pour connaître la compatibilité des moteurs JS :
<http://kangax.github.io/compat-table/>
- Pour développer dès aujourd'hui en ES6 ou ES7 et exécuter le code sur des moteurs plus anciens on peut utiliser des :
 - Compilateurs ou transpileurs : Babel, Traceur, TypeScript... Transforment la syntaxe ES6 en ES5
 - Bibliothèques de polyfills : core-js, es6-shim, es7-shim... Recréent les méthodes manquante en JS



- On peut remplacer le mot-clé `var`, par `let` et obtenir ainsi une portée de bloc

```
for (var i=0; i<3; i++) {}  
console.log(typeof i); // number  
  
for (let j=0; j<3; j++) {}  
console.log(typeof j); // undefined
```



- La portée de bloc s'applique également aux fonction en mode strict

```
'use strict';  
  
if (true) {  
  function test() {}  
  console.log(typeof test); // function  
}  
  
console.log(typeof test); // undefined
```



- A l'instar de arguments, new.target est créé automatiquement lors de l'appel à une fonction fait avec new
- Contient la fonction utilisé ou undefined si pas d'appel avec new

```
var Contact = function() {  
  console.log(new.target);  
};  
  
var c1 = new Contact(); // [Function: Contact]  
var c2 = Contact(); // undefined
```



- Les fonctions fléchées sont plus courtes syntaxiquement

```
var firstNames = ['Eric', 'Romain', 'Jean', 'Eric', 'Jean'];

firstNames.filter(firstName => firstName.length === 4)
  .map(firstName => firstName.toUpperCase())
  .sort()
  .reverse()
  .reduce((firstNames, firstName) => {
    if (!firstNames.includes(firstName)) {
      firstNames.push(firstName)
    }
    return firstNames;
  }, [])
  .forEach(firstName => console.log(firstName));
```



- Les fonctions fléchées ne lient pas les variables `this`, arguments ou `new.target`
- Elles ne doivent pas être utilisées pour déclarer des méthodes !

```
var contact = {  
  firstName: 'Romain',  
  helloAsyncFunctionExpression: function() {  
    setTimeout(function() {  
      console.log('Hello my name is ' + this.firstName);  
    }, 1000);  
  },  
  helloAsyncArrow: function() {  
    setTimeout(() => {  
      console.log('Hello my name is ' + this.firstName);  
    }, 1000);  
  }  
};  
  
contact.helloAsyncFunctionExpression(); // Hello my name is undefined  
contact.helloAsyncArrow(); // Hello my name is Romain
```


ECMAScript 6 - Default Params



- Les paramètres d'entrées peuvent maintenant recevoir une valeur par défaut si rien ne leur est transmis
- La valeur par défaut peut être un appel à une fonction ou une création d'objet

```
var sum = function(a, b, c = 0) {  
    return a + b + c;  
};  
  
console.log(sum(1, 2, 3)); // 6  
console.log(sum(1, 2)); // 3  
  
var frDate = function(date = new Date()) {  
    var day = (date.getDate() < 10) ? '0' + date.getDate() : date.getDate();  
    var month = (date.getMonth() + 1 < 10) ? '0' + (date.getMonth() + 1) :  
date.getMonth() + 1;  
    var year = date.getFullYear();  
    return day + '/' + month + '/' + year;  
};  
  
console.log(frDate(new Date('1985-10-01'))); // 01/10/1985  
console.log(frDate()); // 14/11/2016
```

ECMAScript 6 - Rest Parameters



- ▶ Un Rest parameter permet de récupérer des arguments multiple dans une seule variable de type Array.

```
var sum = function(a, b, ...c) {  
  var result = a + b;  
  
  c.forEach(n => result += n);  
  
  return result;  
};  
  
var sum = function(...n) {  
  return n.reduce((a, b) => a + b);  
};  
  
console.log(sum(1, 2, 3, 4)); // 10
```

ECMAScript 6 - Spread Parameter



- Le Spread parameter permet de renseigner plusieurs arguments avec un seul paramètre de type Array

```
var sum = function(a, b, c, d) {  
  return a + b + c + d;  
};  
  
var nbs = [1, 2, 3, 4];  
  
console.log(sum(...nbs)); // 10
```

ECMAScript 6 - Boucle for..of



- Permet de boucler sur des objets itérables (Array, Map, Set, String, TypedArray, arguments)

```
var firstNames = ['Romain', 'Eric'];  
  
for (let firstName of firstNames) {  
  console.log(firstName);  
}
```



- Symbol est un nouveau type primitif qui n'a pas de syntaxe littéral, seul l'appel à la fonction Symbol est possible
- 2 appel successif à Symbol donneront 2 valeurs uniques

```
var locale = {  
  fr_FR: Symbol(),  
  en_US: Symbol()  
};  
  
var translations = {  
  [locale.fr_FR]: {  
    'hello': 'bonjour',  
    'cat': 'chat'  
  },  
  [locale.en_US]: {  
    'hello': 'hello',  
    'cat': 'cat'  
  }  
};  
  
var translate = function (key, locale = locale.en_US) {  
  return translations[locale][key];  
};  
  
console.log(translate('hello', locale.fr_FR)); // bonjour
```



- Symbol permet également de redéfinir des comportements du langage, comme la boucle `for..of` avec `Symbol.iterator`

```
class Collection {
  constructor() {
    this.list = [];
  }
  add(elt) {
    this.list.push(elt);
    return this;
  }
  *[Symbol.iterator]() {
    for (let elt of this.list) {
      yield elt;
    }
  }
}

let firstNames = new Collection();
firstNames.add('Romain').add('Eric');

for (let firstName of firstNames) {
  console.log(firstName); // Romain Eric
}
```



- Reprendre le jeu du plus ou moins
- Le transformer en utilisant les mots clés class, let et les fonctions fléchées



- Boucle d'événement

Comme vu précédemment, le code JavaScript s'exécute au sein d'une boucle appelée « boucle d'événement ». Ceci permet de différer l'exécution d'une partie d'un code au moment où une interaction se produit (ex : clic, fin de chargement, réception de données, requêtes HTTP, lecture de fichier).

- Avantages

- Gestion de la concurrence simplifiée
- Performance

- Inconvénients

- Perte de contexte (mot clé this)
- Callback Hell

JavaScript Asynchrone - Perte de contexte



► Où est this ?

Dans l'exemple ci-dessous on mélange code objet et programmation asynchrone. Problème, au moment où le callback est appelé (dans un prochain passage de la boucle d'événement), le moteur JavaScript perd la référence sur l'objet `this` qui était attaché à la méthode `helloAsync`.

```
var contact = {  
  firstName: 'Romain',  
  helloAsync: function() {  
    setTimeout(function() {  
      console.log('Hello my name is ' + this.firstName);  
    }, 1000)  
  }  
};  
  
contact.helloAsync(); // Hello my name is undefined
```



► Solution 1 : Sauvegarder this dans la portée de closure

La valeur de this peut être sauvegardée dans la portée de closure, la variable s'appelle généralement that (ou _this, self, me...)

```
var contact = {  
  firstName: 'Romain',  
  helloAsync: function() {  
    var that = this;  
    setTimeout(function() {  
      console.log('Hello my name is ' + that.firstName);  
    }, 1000)  
  }  
};  
  
contact.helloAsync(); // Hello my name is Romain
```



► Solution 2 : Function.bind (ES5)

La méthode bind du type function retourne une fonction dont la valeur de this ne peut être modifiée.

```
var contact = {  
  firstName: 'Romain',  
  helloAsync: function() {  
    setTimeout(function() {  
      console.log('Hello my name is ' + this.firstName);  
    }.bind(this), 1000)  
  }  
};
```

```
contact.helloAsync(); // Hello my name is Romain
```

```
var contact = {  
  firstName: 'Romain',  
  hello: function() {  
    console.log('Hello my name is ' + this.firstName);  
  },  
  helloAsync: function() {  
    setTimeout(this.hello.bind(this), 1000);  
  }  
};
```

```
contact.helloAsync(); // Hello my name is Romain
```



▸ Solution 3 : Arrow Function (ES6)

Les fonctions fléchées ne lient pas de valeur pour this, ce qui permet au callback de retrouver la valeur de la fonction parent.

```
var contact = {  
  firstName: 'Romain',  
  helloAsync: function() {  
    setTimeout(() => {  
      console.log('Hello my name is ' + this.firstName);  
    }, 1000)  
  }  
};  
  
contact.helloAsync(); // Hello my name is Romain
```



▸ Callback Hell

A force le code JavaScript a tendance à s'imbriquer, ici une simple copie de fichier nécessite de lire le fichier de manière asynchrone puis de l'écrire.

```
const fs = require('fs');
const path = require('path');

const file = 'index.html';
const distDirPath = path.join(__dirname, 'dist');
const srcDirPath = path.join(__dirname, 'src');
const srcFilePath = path.join(srcDirPath, file);
const distFilePath = path.join(distDirPath, file);

fs.readFile(srcFilePath, (err, data) => {
  if (err) {
    return console.log(err);
  }
  fs.writeFile(distFilePath, data, (err) => {
    if (err) {
      return console.log(err);
    }
    console.log(`File ${file} copied.`);
  });
});
```



▸ Async

La bibliothèque Async contient un certain nombre de méthodes pour simplifier les problématiques d'asynchronisme, ici waterfall appelle le premier callback, passe le résultat au second puis appelle le dernier callback, ou directement le dernier en cas d'erreur.

```
const fs = require('fs');
const path = require('path');
const async = require('async');

const file = 'index.html';
const distDirPath = path.join(__dirname, 'dist');
const srcDirPath = path.join(__dirname, 'src');
const srcFilePath = path.join(srcDirPath, file);
const distFilePath = path.join(distDirPath, file);

async.waterfall([(callback) => {
  fs.readFile(srcFilePath, callback);
}, (data, callback) => {
  fs.writeFile(distFilePath, data, callback);
}], (err) => {
  if (err) {
    return console.log(err);
  }
  console.log(`File ${file} copied.`);
});
```



▸ Exemple avancé

Les promesses sont un concept pas si nouveau en JavaScript, on les retrouve dans jQuery depuis la version 1.5 (deferred object).

Elle permet de gagner en lisibilité en remettant à plat un code asynchrone, tout en offrant la possibilité à du code asynchrone d'utiliser les exceptions.

On peut les utiliser grâce à des bibliothèques comme bluebird ou q, ou bien nativement depuis ES6.

```
const fsp = require('fs-promise');
const path = require('path');

const file = 'index.html';
const distDirPath = path.join(__dirname, 'dist');
const srcDirPath = path.join(__dirname, 'src');
const srcFilePath = path.join(srcDirPath, file);
const distFilePath = path.join(distDirPath, file);

fsp.readFile(srcFilePath)
  .then(content => fsp.writeFile(distFilePath, content))
  .then(() => console.log(`File ${file} copied.`))
  .catch(console.log);
```



▸ Exemple avancé

5 callbacks imbriqués et une gestion d'erreur intermédiaire puis finale avec les promesses

```
const fsp = require('fs-promise');
const path = require('path');

const file = 'index.html';
const distDirPath = path.join(__dirname, 'dist');
const srcDirPath = path.join(__dirname, 'src');
const srcFilePath = path.join(srcDirPath, file);
const distFilePath = path.join(distDirPath, file);

fsp.stat(distDirPath)
  .catch(err => fsp.mkdir(distDirPath))
  .then(() => fsp.readFile(srcFilePath))
  .then(content => fsp.writeFile(distFilePath, content))
  .then(() => console.log(`File ${file} copied.`))
  .catch(console.log);
```




- Observables

Les promesses ont leur limite, il faut recréer une promesse si elle se répète, il est également impossible de les annuler. Dans ce cas la tendance est aux observables, via la bibliothèque rxJS et bientôt intégrés au langage. On parle de Reactive Programming

- Angular 2 intègre rxJS par défaut



- Remplacer les appels à Mongoose par l'utilisation de Promesse
- Utiliser les méthodes then et catch