formation.tech

Formation JavaScript

Romain Bohdanowicz
Twitter : @bioub - https://github.com/bioub
http://formation.tech/

formation.tech

Introduction

Présentations

- Romain Bohdanowicz
  Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle
- Expérience
  Formateur/Développeur Freelance depuis 2006
  Plus de 9 500 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 ?

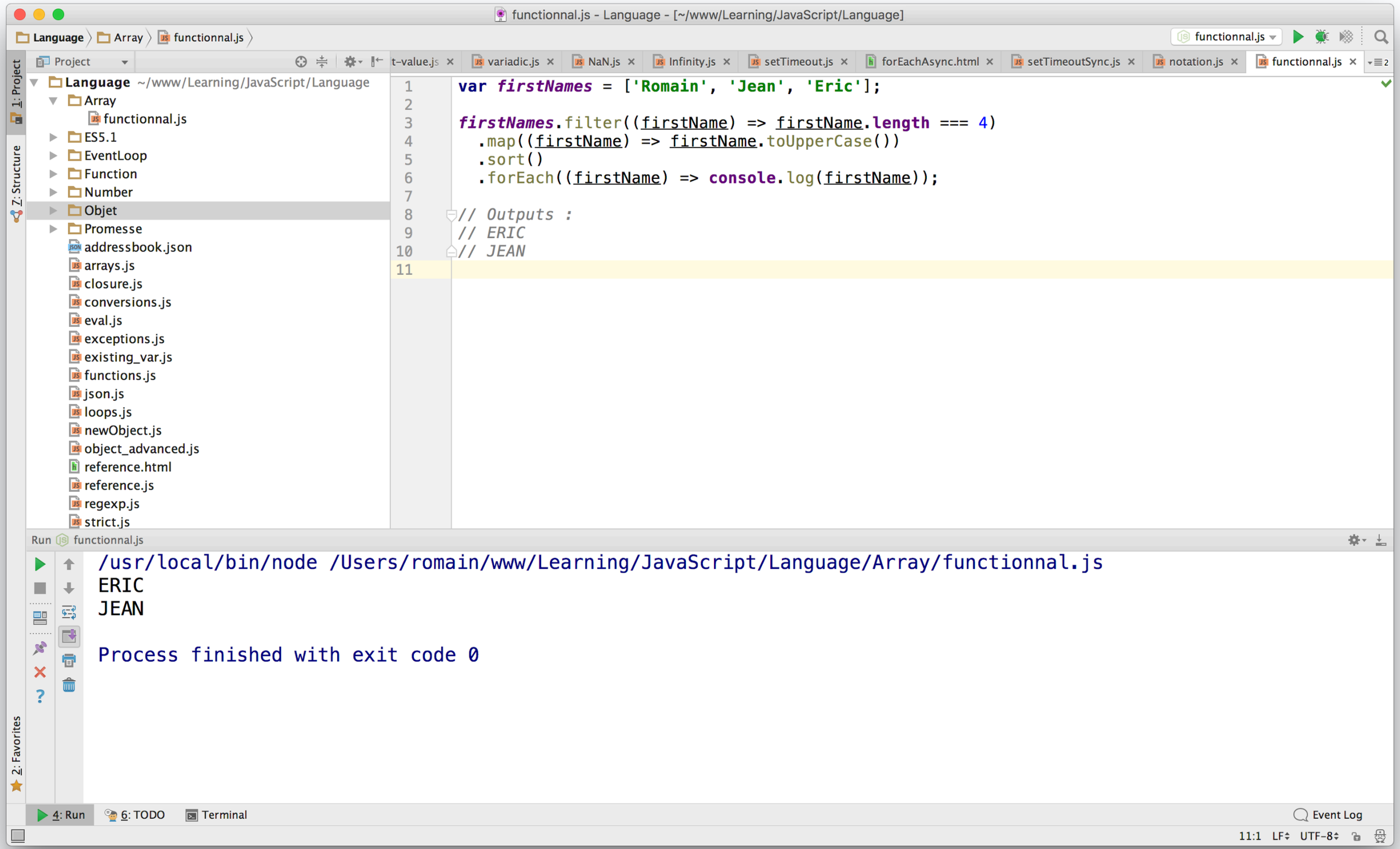
formation.tech

JavaScript IDEs

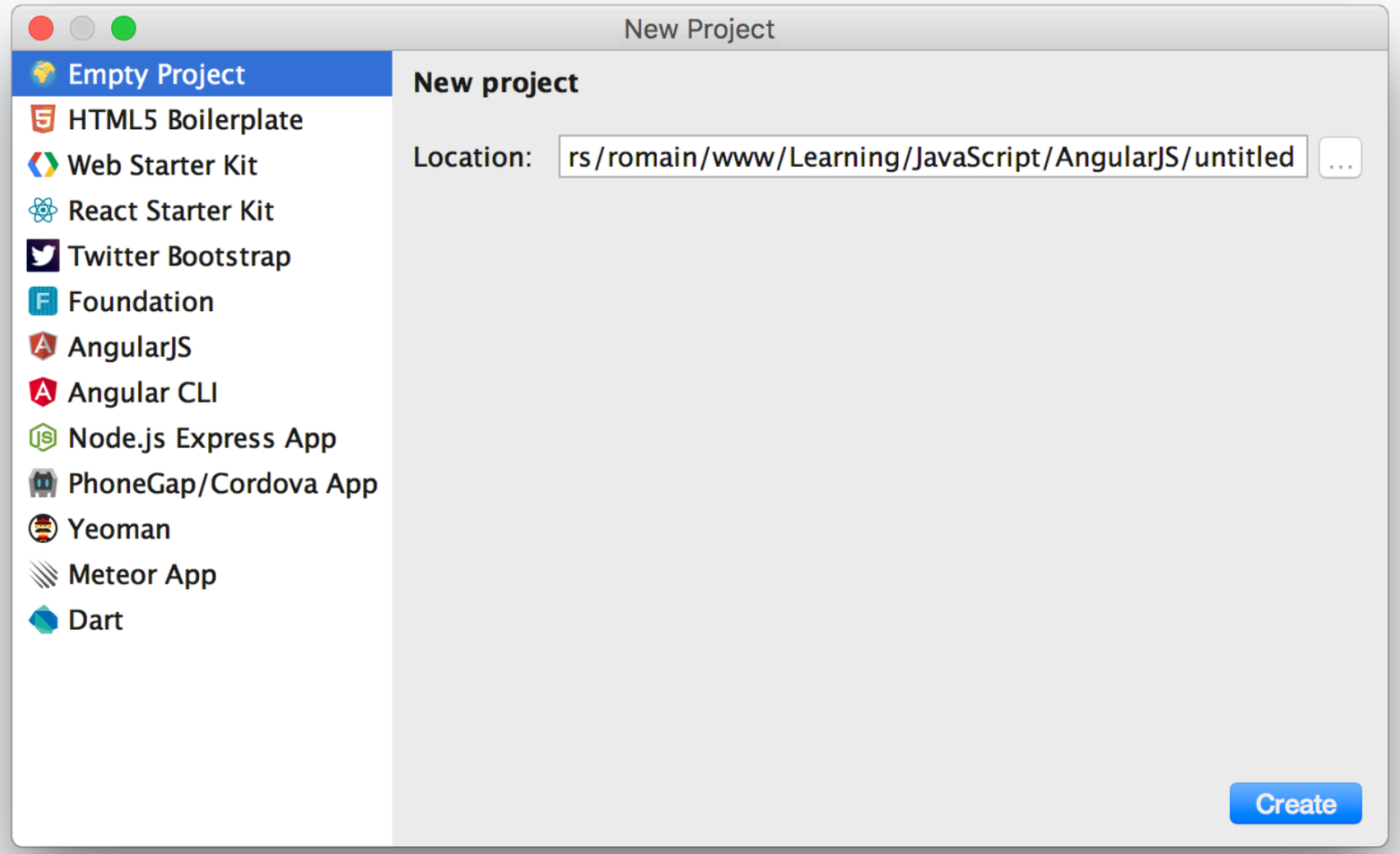
JavaScript IDEs - Webstorm

- 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

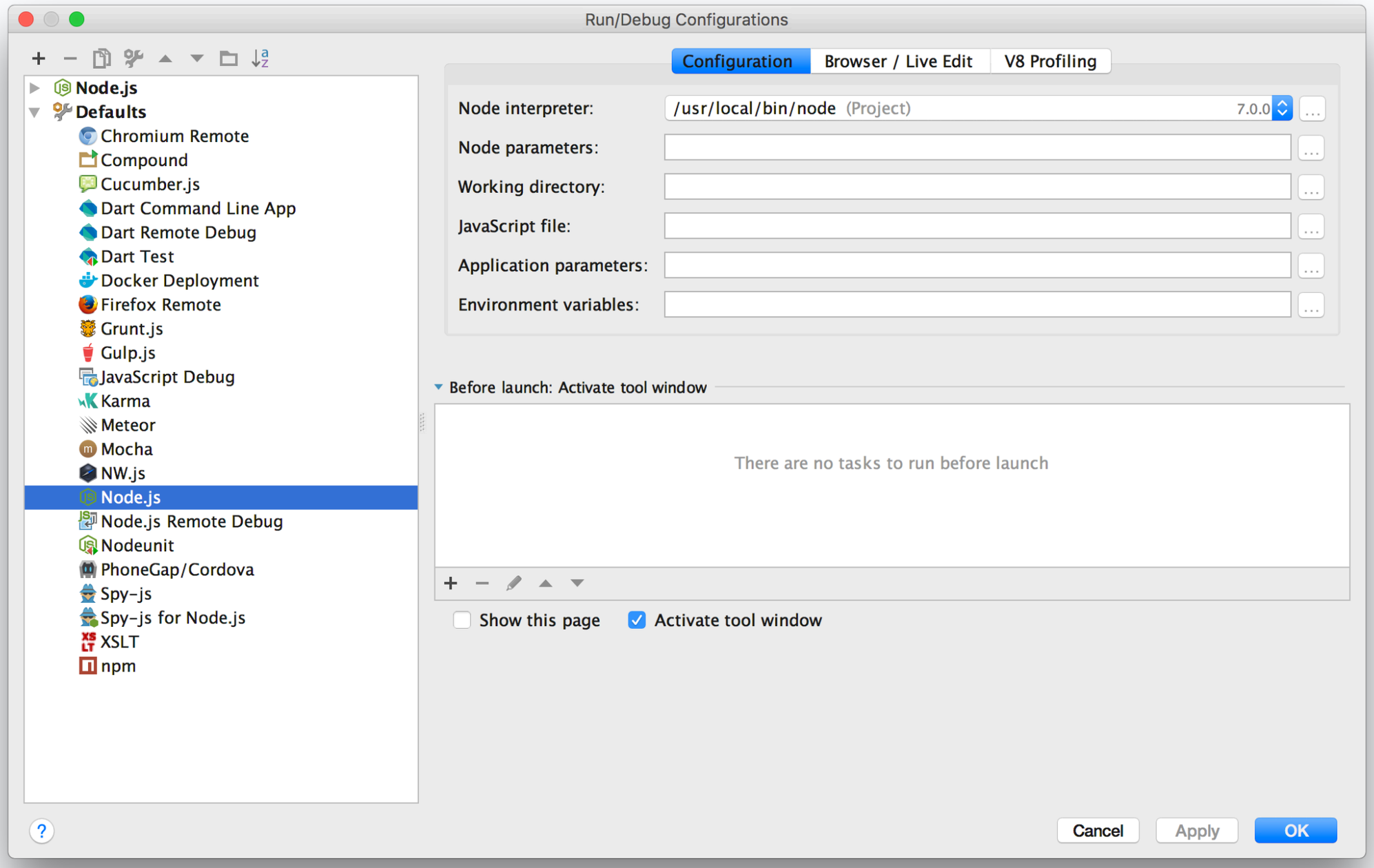
JavaScript IDEs - Webstorm



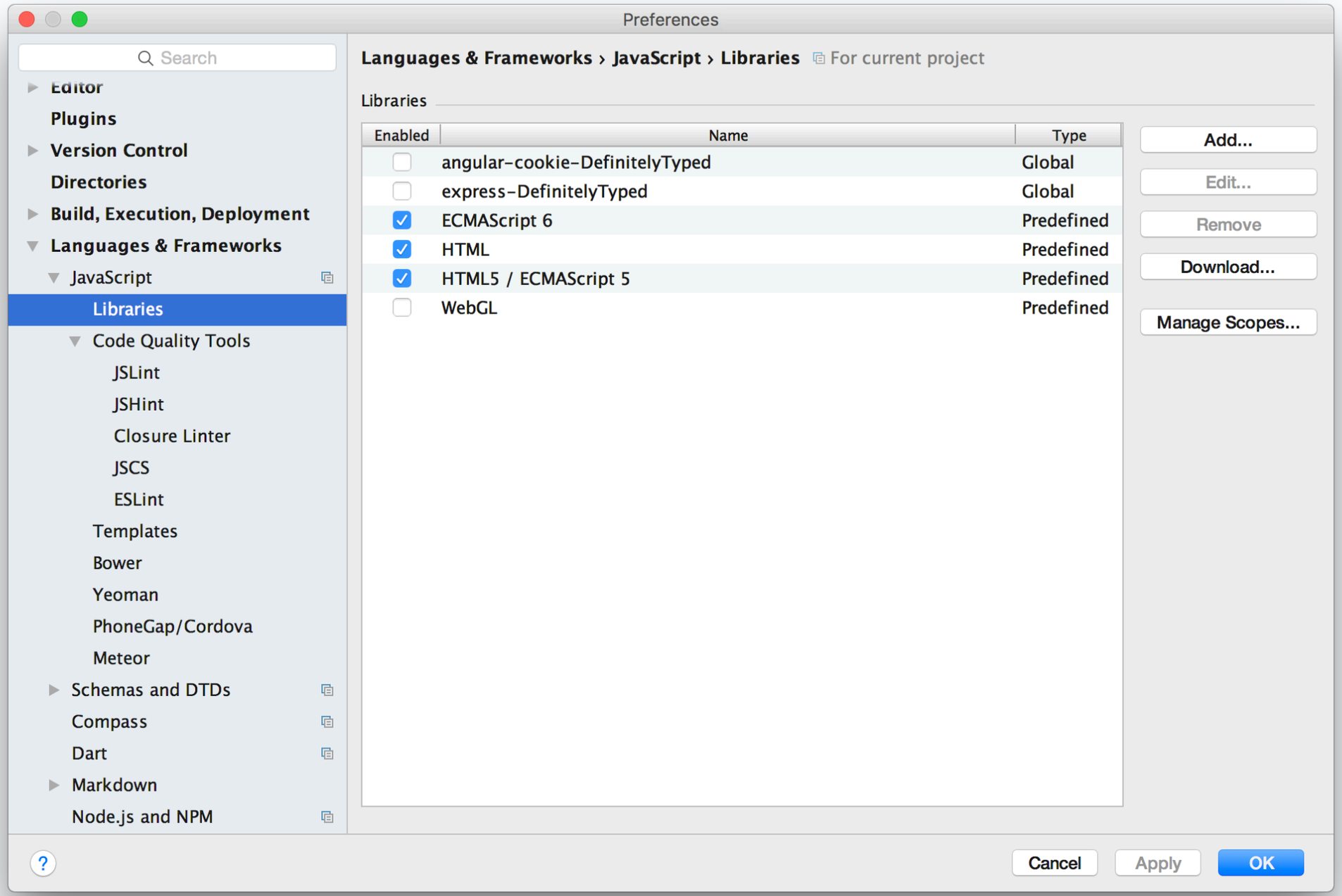
JavaScript IDEs - Webstorm



JavaScript IDEs - Webstorm



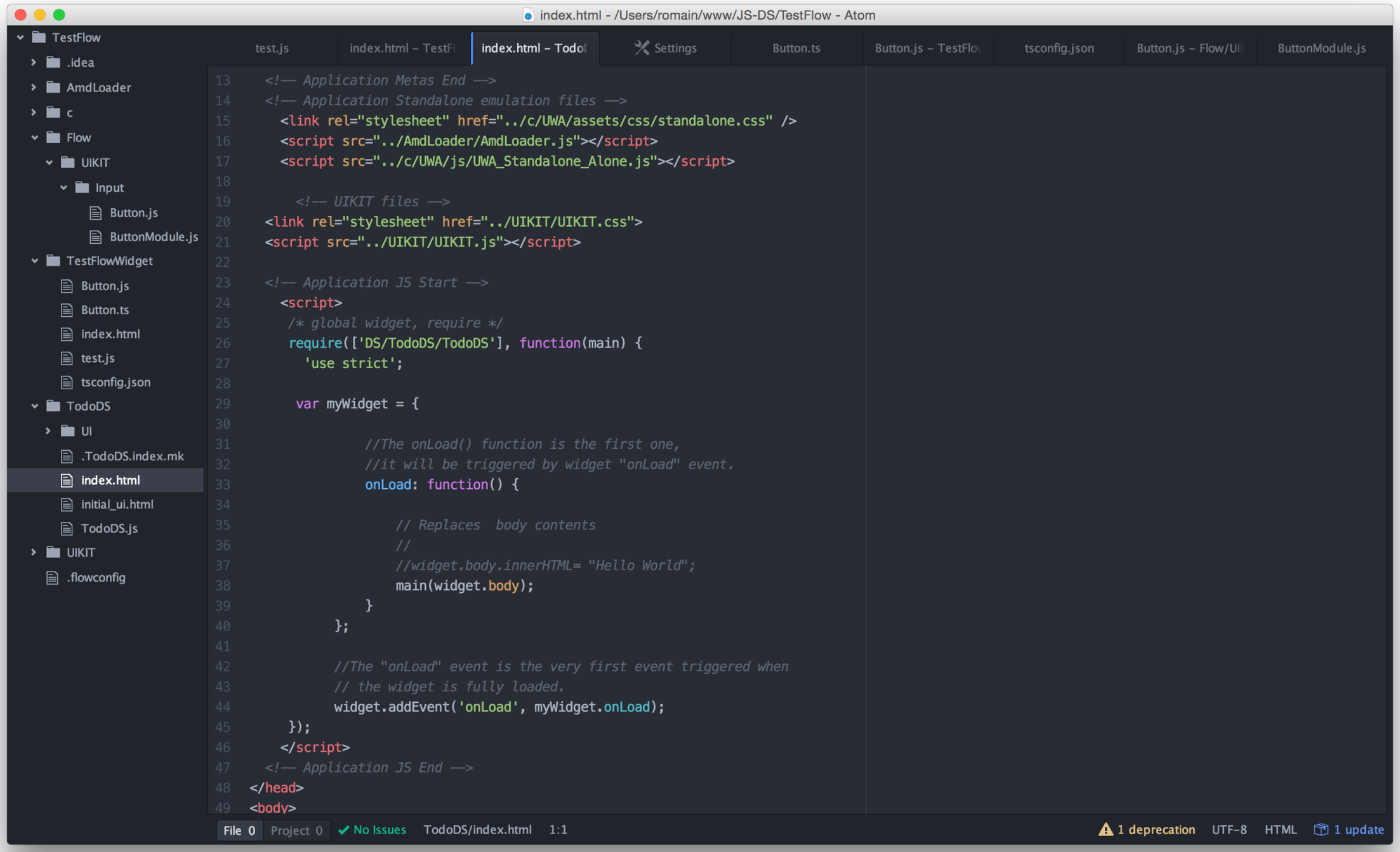
JavaScript IDEs - Webstorm



JavaScript IDEs - Atom

- 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…)

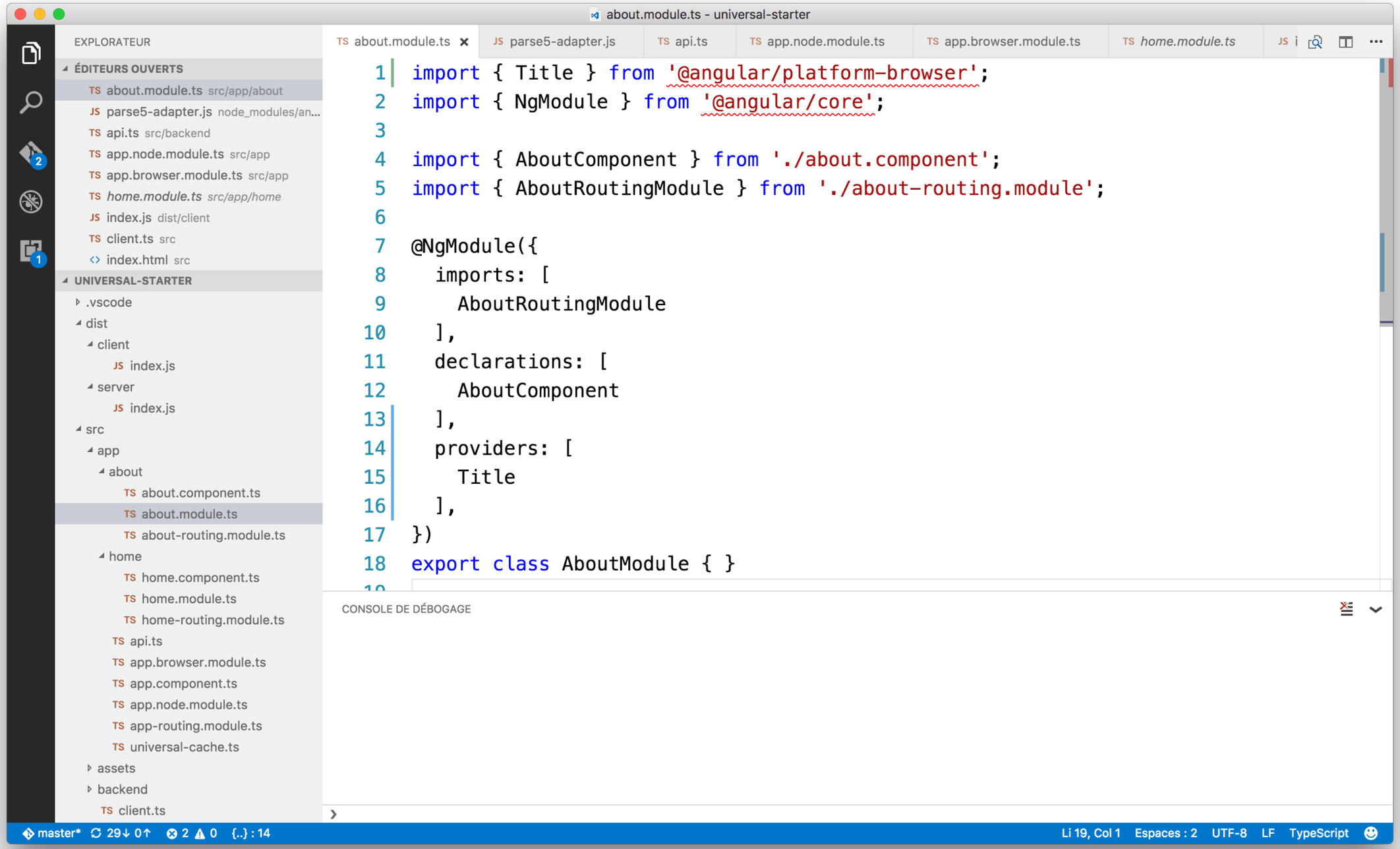
JavaScript IDEs - Atom



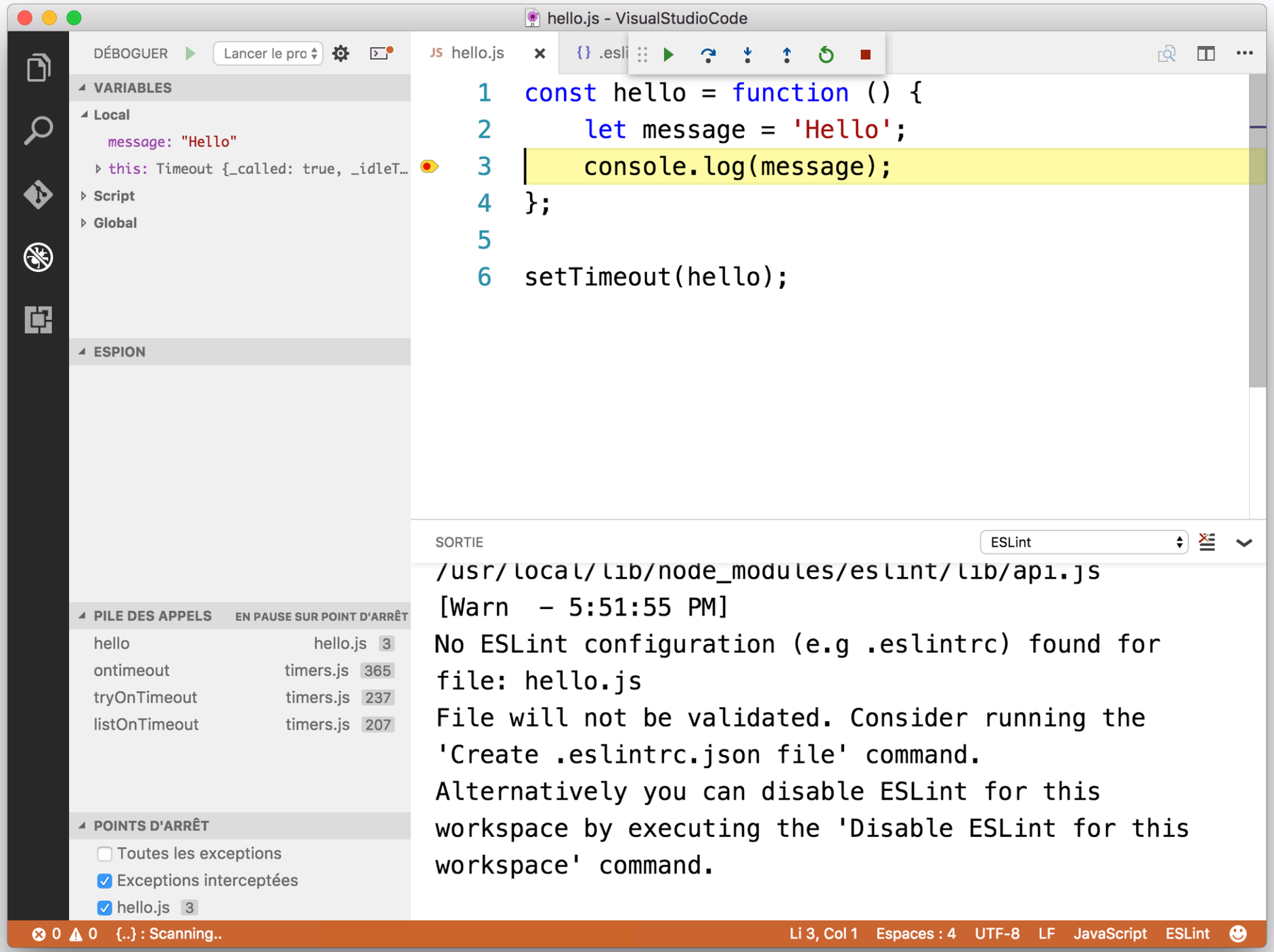
JavaScript IDEs - Visual Studio Code

- 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



JavaScript IDEs - Visual Studio Code



EditorConfig

- 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 (ECMAScript 3)

JavaScript - Introduction

- 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 créé en 1997 : ECMAScript

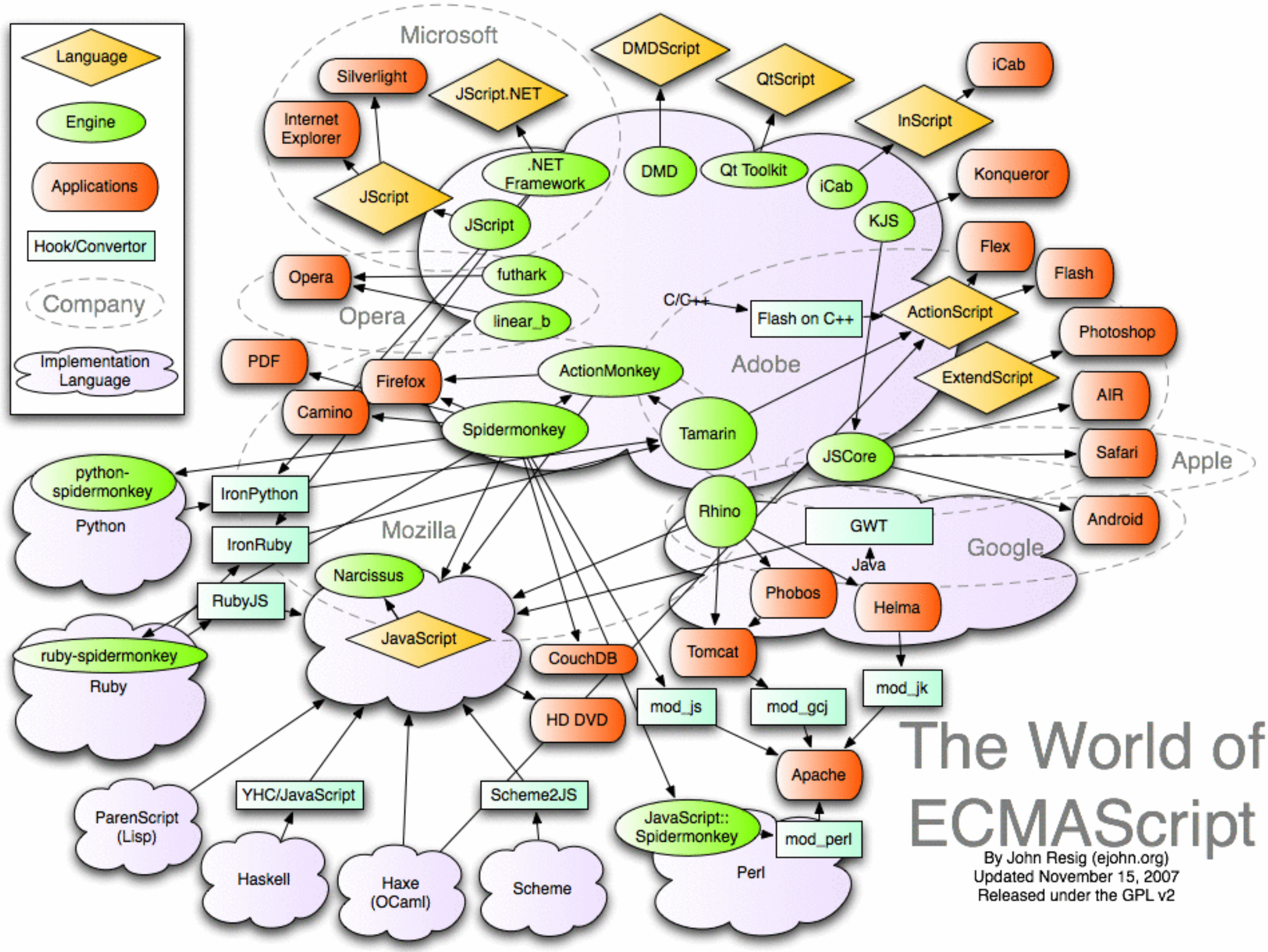
JavaScript - ECMAScript

- JavaScript est une implémentation de la norme ECMAScript 262
- La norme la plus récente est ECMAScript 2018
  Aussi appelée ECMAScript 9 ou ES9 (juin 2018)
  https://www.ecma-international.org/ecma-262/9.0/
- 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

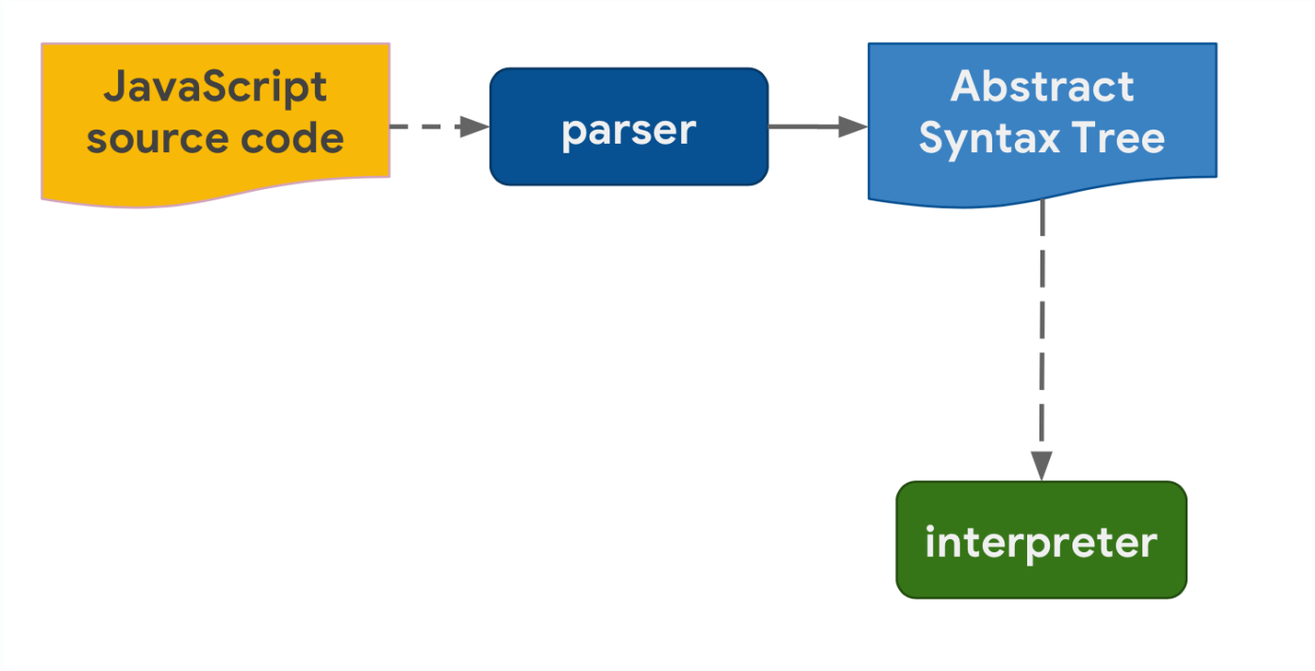
JavaScript - Documentation

- La norme manque d'exemples et d'information sur les implémentations : http://www.ecma-international.org/ecma-262/9.0/
- 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/

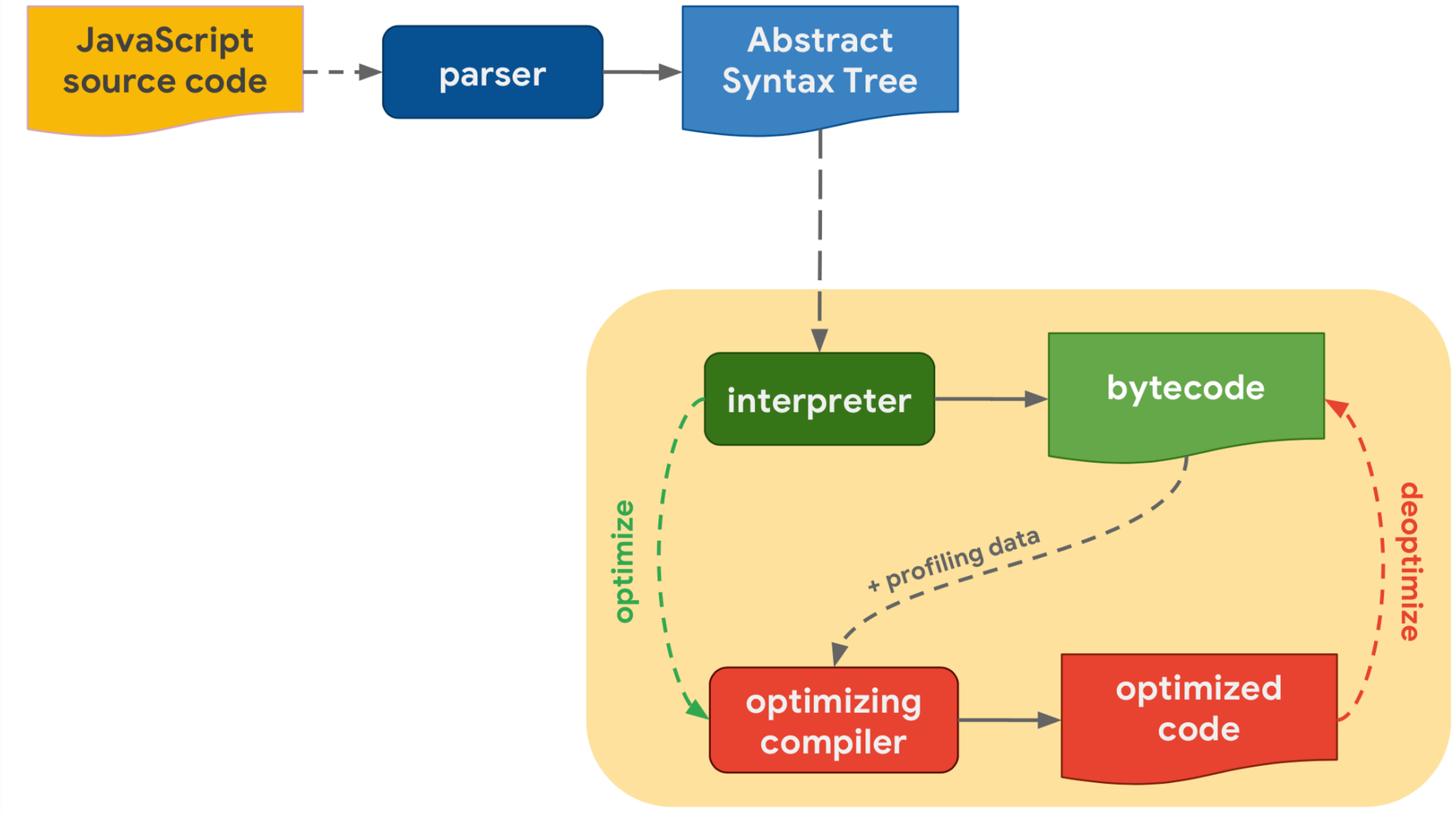
JavaScript - ECMAScript



JavaScript - Interprétation



JavaScript - Compilation JIT



- https://slidr.io/mathiasbynens/javascript-engines-the-good-parts

JavaScript - Syntaxe

- 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 -> http://usejsdoc.org/ */

JavaScript - Identifiants

- 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 ou se terminent par des $ ou _ sont utilisées par certaines conventions
- Exemples :
  - Valides
    i, *maVariable*, *$div*, *v1*, *prénom*
  - Invalides
    ~~*1var*~~, ~~*ma-variable*~~

JavaScript - Mots clés

- Mots clés (ES9) :
  await, 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
- Réservés pour une utilisation future (mode strict) :
  implements, *interface*, *package*, *private*, *protected*, *public*

JavaScript - Types

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

JavaScript - Types

- 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.concat('Romain'); // immuable
  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' ]
```

JavaScript - Number

- 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'
```

JavaScript - NaN

- 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
```

JavaScript - Infinity

- 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 (pas de typage statique)

```
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)

JavaScript - Undefined

- 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'
```

JavaScript - Null

- On utilise généralement null pour déréférencer un objet et ainsi permettre au garbage collector de libérer la mémoire associé
- Dans certaines API, null est souvent utilisé en valeur de retour lorsqu'un objet est attendu mais qu'aucun objet ne convient.

```
var contacts = [{
  firstName: 'Romain'
}, {
  firstName: 'Steven'
}];

function findContact(firstName, contacts) {
  for (var i=0; i<contacts.length; i++) {
    if (contacts[i].firstName === firstName) {
      return contacts[i];
    }
  }

  return null;
}

console.log(findContact('Jean', contacts)); // null;

contacts = null; // dereference
```

JavaScript - Opérateurs

- Affectation

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

JavaScript - Opérateurs

- Comparaison

Opérateur	Description	Exemples qui renvoient true

Égalité (==)	Renvoie true si les opérandes sont égaux après conversion en valeurs de mêmes types.	`3 == var1`
		`"3" == var1`
		`3 == '3'`

Inégalité (!=)	Renvoie true si les opérandes sont différents.	`var1 != 4` `var2 != "3"`
Égalité stricte (===)	Renvoie true si les opérandes sont égaux et de même type. Voir Object.is() et égalité de type en JavaScript.	`3 === var1`
Inégalité stricte (!==)	Renvoie true si les opérandes ne sont pas égaux ou s'ils ne sont pas de même type.	`var1 !== "3"` `3 !== '3'`
Supériorité stricte (>)	Renvoie true si l'opérande gauche est supérieur (strictement) à l'opérande droit.	`var2 > var1` `"12" > 2`
Supériorité ou égalité (>=)	Renvoie true si l'opérande gauche est supérieur ou égal à l'opérande droit.	`var2 >= var1` `var1 >= 3`
Infériorité stricte (<)	Renvoie true si l'opérande gauche est inférieur (strictement) à l'opérande droit.	`var1 < var2` `"2" < "12"`
Infériorité ou égalité (<=)	Renvoie true si l'opérande gauche est inférieur ou égal à l'opérande droit.	`var1 <= var2` `var2 <= 5`

JavaScript - Opérateurs

- 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 renvoie2, 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) (ES7)	2 ** 3 renvoie 8 10 ** -1 renvoie -1

JavaScript - Opérateurs

- 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.

JavaScript - Opérateurs

- 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'
```

JavaScript - Opérateurs

- Priorités

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

JavaScript - Conversions

- 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'
```

- Les conversions de types
  https://www.youtube.com/watch?v=cueiAe7JlfY

JavaScript - API

- Standard Built-in Objects
  Les objets prédéfinies par le langages, 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…

JavaScript - Tableaux

- 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
```

JavaScript - Structures de contrôle

- 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 (ECMAScript 3)

Fonctions en JavaScript - Introduction

- JavaScript est très consommateur de fonctions
  - réutilisation / factorisation
  - récursivité
  - fonction de rappel (callback) / écouteur (listener)
  - 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 grace 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);
  });
});
```

named-function-…dq13s5m3kju84s ×

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <title>Title</title>
6  </head>
7  <body>
8  <script>
9
10   document.addEventListener('click', function() {
11     ['Romain', 'Eric'].forEach(function(firstName) {
12       console.log(firstName);
13     });
14   });
15 </script>
```

Async

Watch

Call Stack

(anonymous function) named-function-…3s5m3kju84s:12

(anonymous function) named-function-…3s5m3kju84s:11

Paused on a JavaScript breakpoint.

Scope

Local

firstName: "Romain"

this: Window

Global Window

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

named-function-…dq13s5m3kju84s ×

e options

```
DOCTYPE html>
tml lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<script>

  document.addEventListener('click', function clickHa
    ['Romain', 'Eric'].forEach(function forEachFirstNa
      console.log(firstName);
    });
  });
</script>
```

Async

Watch

Call Stack

forEachFirstName named-function-…3s5m3kju84s:12

clickHandler named-function-…3s5m3kju84s:11

Paused on a JavaScript breakpoint.

Scope

Local

firstName: "Romain"

this: Window

Global Window

Fonctions en JavaScript - Paramètres

- 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
```

Fonctions en JavaScript - Exceptions

- 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 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 (err) {
  console.log(err.message);
}
```

Fonctions en JavaScript - Valeur par défaut

- Valeur par défaut
  Les paramètres non renseignées 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 (si la valeur par défaut est falsy uniquement)
var sum = function(a, b, c) {
  c = c || 0;
  return a + b + c;
};

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

Fonctions en JavaScript - Paramètres non déclarés

- 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.

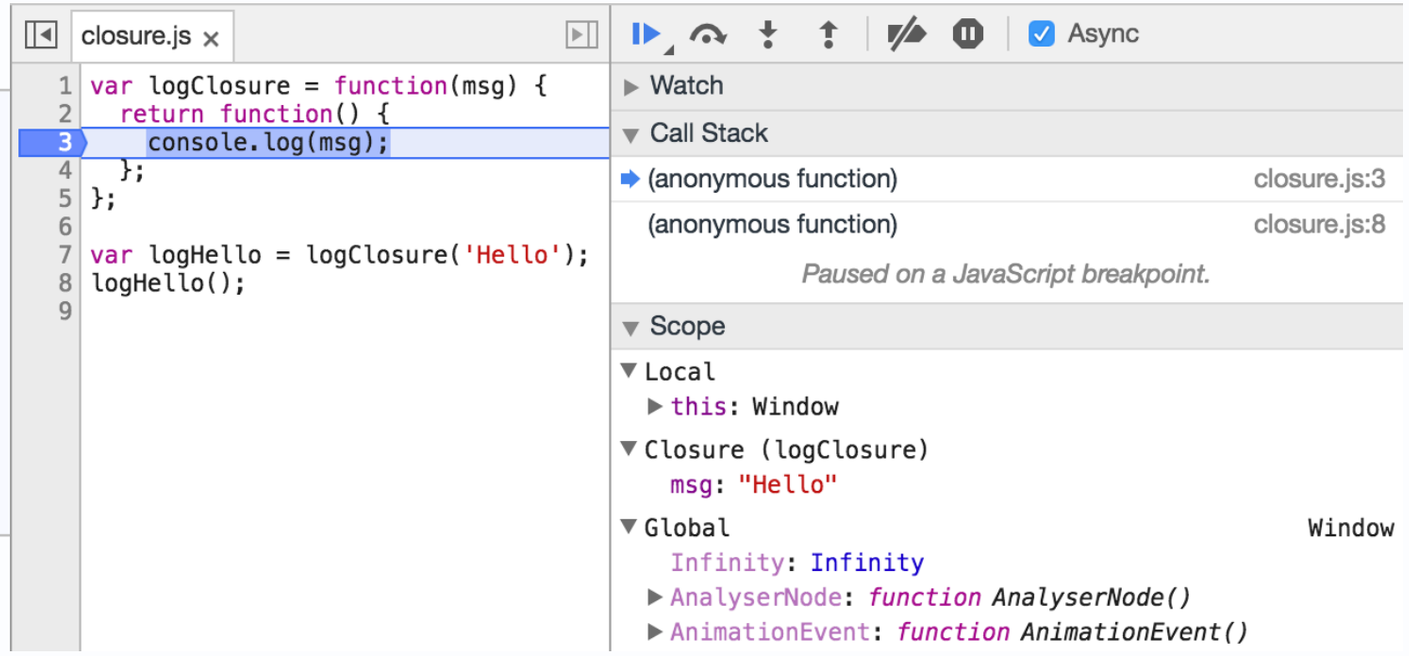
```
var a = function() {
  var b = function() {
    var c = function() {
      console.log(typeof a); // function
      console.log(typeof b); // function
      console.log(typeof c); // function
    };
    c();
  };
  b();
};
a();
```

- Pas besoin de repasser les variables en paramètres si les fonctions sont imbriquées

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(); // Hello
```



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'un 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);
```

Fonctions en JavaScript - Callback Synchrone

- 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

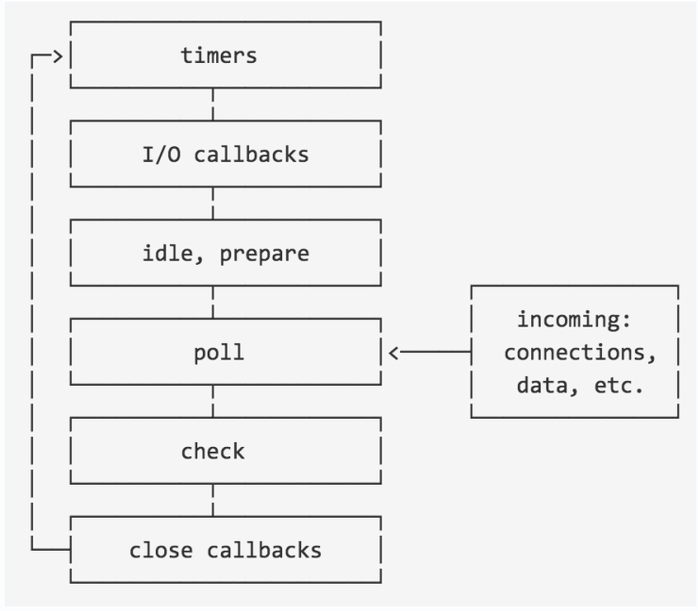
```
setTimeout(function() {
    console.log('1 fois dans 3 secondes');
}, 3000);

var intervalId = setInterval(function() {
    console.log('toutes les 2 secondes');
}, 2000);

setTimeout(function() {
    console.log('Bye bye');
    clearInterval(intervalId);
}, 15000);
```

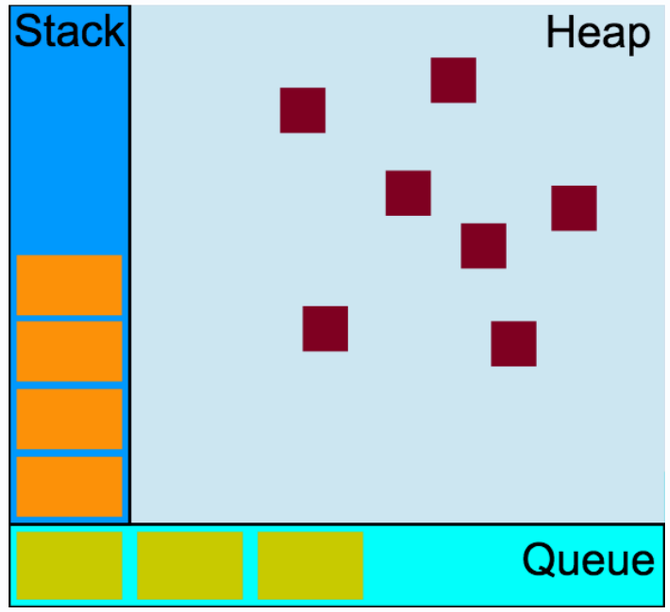
Fonctions en JavaScript - Boucle d’événements

- 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 prochain 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

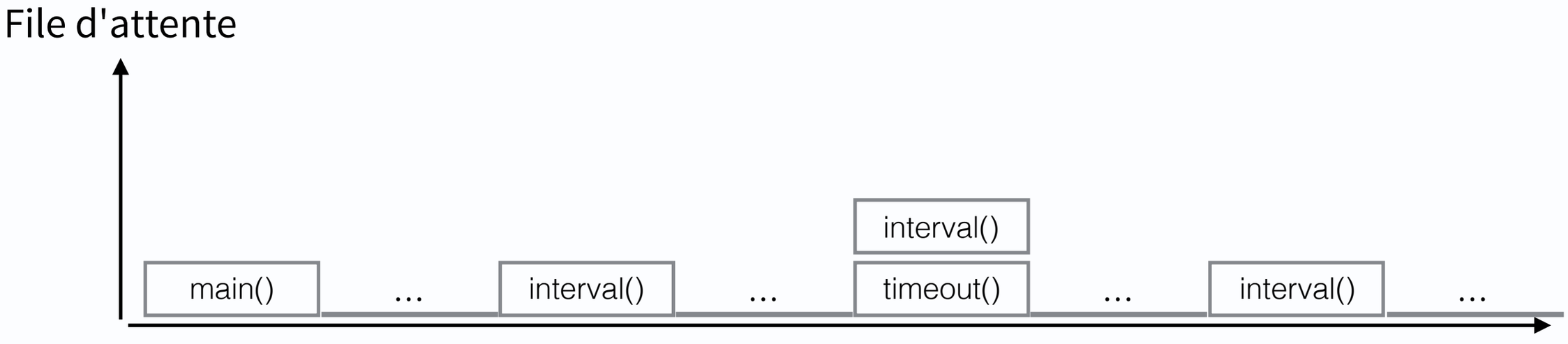
- Jake Archibald: In The Loop - JSConf.Asia 2018
  https://www.youtube.com/watch?v=cCOL7MC4Pl0





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);
```

Fonctions en JavaScript - API Function

- 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)

```
(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));
```

Fonctions en JavaScript - Exercice

- Jeu du plus ou moins

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

- Attention, le callback de question est toujours appelé avec un type String, à convertir si besoin.

JavaScript Orienté Objet (ECMAScript 3)

JavaScript Orienté Objet - Introduction

- JavaScript a un modèle objet orienté prototype
- Modèle statique vs Modèle dynamique
  Il n'y a pas de définition statique du type ou du contenu d'un objet, l'ajout de propriété où de méthode se fait dynamiquement
- Classe vs dictionnaires
  La notion de classe ou d'interface n'existe pas (seulement dans les docs où sous la forme de sucre syntaxique) à la place l'objet JavaScript est un dictionnaire tel que :
  - PHP : tableaux associatifs
  - Java, C++ : Map, HashTable
  - C : Struct
  - C# : Dictionary
  - Python : dictionary
  - Ruby : Hash

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)
```

JavaScript Orienté Objet - Extensibilité

- 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 (mauvaise pratique) :

```
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
```

JavaScript Orienté Objet - Opérateurs

- 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
```

JavaScript Orienté Objet - Fonction Constructeur

- En utilisant une fonction constructeur (avec closure, mauvaise pratique) :

```
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
  }
}
```

JavaScript Orienté Objet - Fonction Constructeur

- 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
  }
}
```

JavaScript Orienté Objet - Héritage

- 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
  }
}
```

JavaScript Orienté Objet - Prototype

- 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 ;

JavaScript Orienté Objet - Prototype

- En ECMAScript/JavaScript, l'écriture foo.bar s'interprète de la façon suivante :
    1. Le nom foo est recherché dans la liste des identifieurs 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, object, 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 Orienté Objet - JSON

- 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

```
const random = {
  ...
};

function Jeu(options) { ... };

// ....

const game = new Jeu({
  min: 0,
  max: 100
});

game.jouer();
```

formation.tech

ECMAScript 5.1

ECMAScript 5.1 - Introduction

- 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++) {}
}());
```

ECMAScript 5.1 - Mode Strict

- 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
```

ECMAScript 5.1 - Mode Strict

- 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
```

ECMAScript 5.1 - Mode Strict

- 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);
```

ECMAScript 5.1 - Array

- 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
```

ECMAScript 5.1 - JSON

- 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
```

ECMAScript 5.1 - Trailing commas

- Il est désormais possible de placer une virgule après le dernier élément d'un tableau ou d'un objet.

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

var coords = {
  x: 10,
  y: 20,
};
```

ECMAScript 5.1 - get syntax

- 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
```

ECMAScript 5.1 - set syntax

- 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
```

ECMAScript 5.1 - Object.keys

- 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
```

ECMAScript 5.1 - Verrous

- 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
Object.preventExtensions	Oui	Non	Non
Object.seal	Oui	Oui	Non
Object.freeze	Oui	Oui	Oui

ECMAScript 5.1 - Héritage en ES5

- 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 2015

ECMAScript 6 - Introduction

- 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/

ECMAScript 6 - Compatibilité

- 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 et exécuter le code sur des moteurs plus anciens on peut utiliser des :
  - Compilateurs ou transpilateurs : Babel, Traceur, TypeScript… Transforment la syntaxe ES6 en ES5
  - Bibliothèques de polyfills : core-js, es6-shim, es7-shim… Recréent les méthodes manquante en JS

ECMAScript 6 - Portées de bloc

- let
  - On peut remplacer le mot-clé var, par let et obtenir ainsi une portée de bloc
  - La portée de bloc ainsi créée peut devenir une closure

```
for (var globalI=0; globalI<3; globalI++) {}
console.log(typeof globalI); // number

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

// In 1s : 0 1 2
for (let i=0; i<3; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000);
}
```

ECMAScript 6 - Portées de bloc

- Fonction avec une portée de bloc
  - 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
```

ECMAScript 6 - Constantes

- Constantes
  - Il est désormais possible de créer des constantes
  - Comme pour let, les variables déclarées via const on une portée de bloc
  - Bonne pratique, utiliser const ou bien let lorsque ce n’est pas possible (plus jamais var)

```
if (true) {
  const PI = 3.14;
}

console.log(typeof PI); // undefined

const hello = function() {};
// SyntaxError: Identifier 'hello' has already been declared
const hello = function() {};
```

ECMAScript 6 - Template literal

- Template literal / Template string
  - Permet de créer une chaine de caractères à partir de variables ou d'expressions
  - Permet de créer des chaines de caractères multi-lignes
  - Déclarée avec un backquote ` (rarement utilisé dans une chaine)

```
const prenom = 'Romain';
console.log(`Bonjour ${prenom} !`);

// ES5
// console.log('Bonjour ' + prenom + ' !');

const html = `
<table class="table">
  <tr><td>${prenom.toUpperCase()}</td></tr>
</table>
`;
```

ECMAScript 6 - new.target

- new.target
  - Lors de l'appel d'une fonction, les variables this et arguments sont créées
  - En ES6 il y a également super et new.target, qui est une référence vers la fonction appelante si elle est appelée avec l'opérateur new, sinon undefined

```
const Contact = function() {
  if (new.target === undefined) {
    throw new Error('Contact is a constructor');
  }
};

const c1 = new Contact(); // OK
const c2 = Contact(); // Error: Contact is a constructor
```

ECMAScript 6 - Fonctions fléchées

- Arrow Functions
  - Plus courtes à écrire : (params) => retour.
  - Si un seul paramètre, les parenthèses des paramètres sont optionnelles.
  - Si le retour est un objet, les parenthèses du retour sont obligatoires.

```
const sum = (a, b) => a + b;
const hello = name => `Hello ${name}`;
const getCoords = (x, y) => ({x: x, y: y});

// ES5
// var sum = function (a, b) {
//   return a + b;
// };
// var hello = function (name) {
//   return 'Hello ' + name;
// };
// var getCoords = function (x, y) {
//   return {
//     x: x,
//     y: y,
//   };
// };
```

ECMAScript 6 - Fonctions fléchées

- Avec bloc d'instructions
  - Si les fonctions nécessitent plusieurs lignes, on peut utiliser un bloc { }
  - Le mot clé return devient alors obligatoire

```
const isWon = (nbGiven, nbToGuess) => {
  if (nbGiven < nbToGuess) {
    return 'Too low';
  }

  if (nbGiven > nbToGuess) {
    return 'Too high';
  }

  return 'Won !';
};
```

ECMAScript 6 - Fonctions fléchées

- Bonnes pratiques
  - Attention à ne pas utiliser les fonctions fléchées pour déclarer des méthodes !
  - Utiliser les fonctions fléchées pour les callback ou les fonctions hors objets
  - Utiliser les method properties pour les méthodes
  - Utiliser class pour les fonctions constructeurs

```
const globalThis = this;

const contact = {
  firstName: 'Romain',
  method1: () => { // Mauvaise pratique
    console.log(this === globalThis); // true
  },
  method2() { // Bonne pratique
    console.log(this === contact); // true
  }
};

contact.method1();
contact.method2();
```

ECMAScript 6 - Default Params

- Paramètres par défaut
  - Les paramètres d'entrées peuvent maintenant recevoir une valeur par défaut

```
const sum = function(a, b, c = 0) {
  return a + b + c;
};

console.log(sum(1, 2, 3)); // 6
console.log(sum(1, 2)); // 3

// ES5
// var sum = function(a, b, c) {
//   if (c === undefined) {
//     c = 0;
//   }
//   return a + b + c;
// };
```

ECMAScript 6 - Default Params

- Paramètres par défaut
  - Les valeurs par défaut sont calculées au moment de l'appel et peuvent être des appels de fonctions

```
const frDate = function(date = new Date()) {
  return date.toLocaleDateString();
};

console.log(frDate(new Date('1985-10-01'))); // 01/10/1985
console.log(frDate()); // 26/11/2017
```

ECMAScript 6 - Rest Parameters

- Paramètres restants
  - Pour récupérer les valeurs non déclarées d'une fonction on peut utiliser le REST Params
  - Remplace la variable arguments (qui n'existe pas dans une fonction fléchée)
  - La variable créé est un tableau (contrairement à arguments)
  - Bonne pratique : ne plus utiliser arguments

```
const sum = (a, b, ...others) => {
  let result = a + b;

  others.forEach(nb => result += nb);

  return result;
};
console.log(sum(1, 2, 3, 4)); // 10

const sumShort = (...n) => n.reduce((a, b) => a + b);
console.log(sumShort(1, 2, 3, 4)); // 10
```

ECMAScript 6 - Spread Operator

- Spread Operator
  - Le Spread Operator permet de transformer un tableau en une liste de valeurs.

```
const sum = (a, b, c, d) => a + b + c + d;

const nbs = [2, 3, 4, 5];
console.log(sum(...nbs)); // 14
// ES5 :
// console.log(sum(nbs[0], nbs[1], nbs[2], nbs[3]));

const otherNbs = [1, ...nbs, 6];
console.log(otherNbs.join(', ')); // 1, 2, 3, 4, 5, 6
// ES5 :
// const otherNbs = [1, nbs[0], nbs[1], nbs[2], nbs[3], 6];

// Clone an array
const cloned = [...nbs];
```

ECMAScript 6 - Shorthand property

- Shorthand property
  - Lorsque l'on affecte une variable à une propriété (maVar: maVar), il suffit de déclarer la propriété

```
const x = 10;
const y = 20;

const coords = {
  x,
  y,
};

// ES5
// const coords = {
//   x: x,
//   y: y,
// };
```

ECMAScript 6 - Method properties

- Method properties
  - Syntaxe simplifiée pour déclarer des méthodes

```
const maths = {
  sum(a, b) {
    return a + b;
  }
};

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

// ES5
// const maths = {
//   sum: function(a, b) {
//     return a + b;
//   }
// };
```

ECMAScript 6 - Computed Property Names

- Computed Property Names
  Permet d'utiliser une expression en nom de propriété

```
let i = 0;

const users = {
  [`user${++i}`]: { firstName: 'Romain' },
  [`user${++i}`]: { firstName: 'Steven' },
};

console.log(users.user1); // { firstName: 'Romain' }

/* ES5
var i = 0;
var users = {};
users['user ' + (++i)] = { firstName: 'Romain' };
users['user ' + (++i)] = { firstName: 'Steven' };

console.log(users.user1); // { firstName: 'Romain' }
*/
```

ECMAScript 6 - Array Destructuring

- Déstructurer un tableau
  - Permet de déclarer des variables recevant directement une valeur d'un tableau

```
const [one, two, three] = [1, 2, 3];
console.log(one); // 1
console.log(two); // 2
console.log(three); // 3

// ES5
// var tmp = [1, 2, 3];
// var one = tmp[0];
// var two = tmp[1];
// var three = tmp[2];
```

ECMAScript 6 - Array Destructuring

- Déstructurer un tableau
  - Il est possible de ne pas déclarer un variable pour chaque valeur
  - Il est possible d'utiliser une valeur par défaut
  - Il est possible d'utiliser le REST Params

```
const [one, , three = 3] = [1, 2];
console.log(one); // 1
console.log(three); // 3

const [romain, ...others] = ['Romain', 'Jean', 'Eric'];
console.log(romain); // Romain
console.log(others.join(', ')); // Jean, Eric
```

ECMAScript 6 - Object Destructuring

- Déstructurer un object
  - Comme pour les tableaux il est possible de déclarer une variable recevant directement une propriété

```
const {x: varX, y: varY} = {x: 10, y: 20};
console.log(varX); // 10
console.log(varY); // 20
```

ECMAScript 6 - Object Destructuring

- Déstructurer un object
  - Il est possible de nommer sa variable comme la propriété et d'utiliser shorthand property
  - Il est possible d'utiliser une valeur par défaut

```
const {x: x, y, z = 30} = {x: 10, y: 20};
console.log(x); // 10
console.log(y); // 20
console.log(z); // 30
```

ECMAScript 6 - Mot clé class

- Simplifie la déclaration de fonction constructeur
- Les classes n'existent pas pour autant en JavaScript, ce n'est qu'une syntaxe simplifiée (sucre syntaxique)
- Le contenu d'une classe est en mode strict

```
class Person {
  constructor(firstName) {
    this.firstName = firstName;
  }
  hello() {
    return `Hello my name is ${this.firstName}`;
  }
}
const instructor = new Person('Romain');
console.log(instructor.hello()); // Hello my name is Romain

// ES5
// var Person = function(firstName) {
//   this.firstName = firstName;
// };
// Person.prototype.hello = function() {
//   return 'Hello my name is ' + this.firstName;
// };
```

ECMAScript 6 - Mot clé class

- Héritage avec le mot clé class
  - Utilisation du mot clé extends pour l'héritage
  - Utilisation de super pour appeler la fonction constructeur parent et les accès au méthodes parents si redéclarée dans la classe

```
class Instructor extends Person {
  constructor(firstName, speciality) {
    super(firstName);
    this.speciality = speciality;
  }
  hello() {
    return `${super.hello()}, my speciality is ${this.speciality}`;
  }
}

const romain = new Instructor('Romain', 'JavaScript');
console.log(romain.hello()); // Hello my name is Romain, my speciality is
JavaScript
```

ECMAScript 6 - Boucle for .. of

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

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

for (const firstName of firstNames) {
  console.log(firstName);
}
```

ECMAScript 6 - Object.assign

- Object.assign
  - Permet d'affecter toutes les clés d'un objet à un autre objet
  - Utile pour cloner une objet
  - Attention le clone ne concerne pas les sous-objets ou tableaux

```
const obj = {
  firstName: 'Romain',
  address: {
    city: 'Paris'
  }
};

const copy = Object.assign({}, obj);
console.log(copy === obj); // false
console.log(copy.address === obj.address); // true
```

ECMAScript 6 - Object.assign

- Object.assign
  - Pour faire un clone de tous les sous objet on pourrait écrire une fonction récursive
  - Ou alors utiliser la fonction cloneDeep de Lodash

```
const deepClone = function(obj) {
  let clone = Object.assign({}, obj);

  for (let p in obj) {
    if (obj.hasOwnProperty(p) && typeof obj[p] === 'object') {
      clone[p] = deepClone(obj[p]);
    }
  }

  return clone;
};

const deepCopy = deepClone(obj);
console.log(deepCopy.address === obj.address); // false
```

ECMAScript 6 - Générateurs

- Générateurs
  - Fonction déclarée avec *
  - Peut être retourner un résultat et se mettre en pause (yield)

```
function *nbs() {
  let i = 0;
  while (i < 3) {
    yield ++i;
  }
}

for (let i of nbs()) {
  console.log(i); // 1 2 3
}
```

ECMAScript 6 - Symbol

- Symbol est un nouveau type primitif qui n'a pas de syntaxe litéral, 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 = locales.en_US) {
  return translations[locale][key];
};

console.log(translate('hello', locale.fr_FR)); // bonjour
```

ECMAScript 6 - Symbol

- 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
}
```

ECMAScript 6 - Exercice

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

ECMAScript 7 / ECMAScript 2016

ECMAScript 7 - Introduction

- ECMAScript 7 sort en juin 2016 et ne contient que 2 nouveautés
  - 1 nouvelle syntaxe : Opérateur d’exponentiation
  - 1 nouvel API : Array.prototype.includes

ECMAScript 7 - Opérateur d'exponentiation

- Opérateur d'exponentiation
  Renvoie le résultat de l'élévation d'un nombre (premier opérande) à une puissance donnée (deuxième opérande).

  Par exemple : var1 ** var2 sera équivalent à $var1^{var2}$ en notation mathématique.

```
console.log(2 ** 3); // 8

// Equivalent en ES6 à
console.log(Math.pow(2, 3)); // 8
```

- Plugin babel : transform-exponentiation-operator
  https://babeljs.io/docs/plugins/transform-exponentiation-operator/

ECMAScript 7 - Array.prototype.includes

- ## Array.prototype.includes

  L'API Array introduit une nouvelle méthode pour vérifier si un élément est présent dans un tableau.
  Attention pour les objets (sauf string), includes vérifie les références et non le contenu de l'objet.

```
const nbs = [2, 3, 4];

console.log(nbs.includes(2)); // true
console.log(nbs.includes(5)); // false

const contacts = [{prenom: 'Romain'}];
console.log(contacts.includes({prenom: 'Romain'})); // false
```

ECMAScript 8 / ECMAScript 2017

ECMAScript 8 - Introduction

- ECMAScript 8 sort en juin 2017 et contient 4 nouveautés
  - 2 nouvelles syntaxes :
    - Virgules finales sur les fonctions
    - Fonctions async / await
  - 3 nouveaux APIs
    - Object.values / Object.entries
    - String Padding
    - Atomics

ECMAScript 8 - Virgules finales sur les fonctions

- Virgules finales sur les fonctions
  Comme pour object literal et array literal, il est désormais possible que le dernier argument d'une fonction soit suivi d'une virgule (au moment de l'appel ou de la déclaration)

```
console.log(
  'A very very very very very very looooooooooonnnnggggg string',
  'Lorem ipsum dolor sit amet, consectetur adipiscing elit.',
  'New line',
);
```

```
class ContactsComponent {
  constructor(
    contactService,
    logService,
    httpClient,
  ) {
    this.contactService = contactService;
    this.logService = logService;
    this.httpClient = httpClient;
  }
}
```

ECMAScript 8 - Virgules finales sur les fonctions

- Depuis ES5 c'était déjà possible sur object et array literal
  - Array literal

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

  - Object literal

```
const coords = {
  x: 10,
  y: 20,
};
```

ECMAScript 8 - Virgules finales sur les fonctions

- Exemple de diff sans et avec virgule finale

```
$ git diff trailing-commas.js
diff --git a/trailing-commas.js b/trailing-commas.js
index da942a9..9064804 100644
--- a/trailing-commas.js
+++ b/trailing-commas.js
@@ -1,4 +1,5 @@
 console.log(
   'A very very very very very very looooooooooonnnnggggg string',
-  'Lorem ipsum dolor sit amet, consectetur adipiscing elit.'
+  'Lorem ipsum dolor sit amet, consectetur adipiscing elit.',
+  'New line',
 );
```

```
$ git diff trailing-commas.js
diff --git a/trailing-commas.js b/trailing-commas.js
index 32f26a2..6ccd800 100644
--- a/trailing-commas.js
+++ b/trailing-commas.js
@@ -1,4 +1,5 @@
 console.log(
   'A very very very very very very looooooooooonnnnggggg string',
   'Lorem ipsum dolor sit amet, consectetur adipiscing elit.',
+  'New line’,
 );
```

ECMAScript 8 - Fonctions async / await

- async / await
  Permet d'écrire du code basé sur des promesses comme on aurait écrit du code synchrone.

- Soit la fonction timeout suivante (qui retourne une Promesse)

```
const timeout = (delay) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (delay % 1000 !== 0) {
        return reject(new Error('delay must be a multiple of 1000'));
      }
      resolve();
    }, delay);
  });
};
```

ECMAScript 8 - Fonctions async / await

- Sans async / await

```
timeout(1000)
  .then(() => {
    console.log('1s');
    return timeout(1000);
  })
  .then(() => {
    console.log('2s');
    return timeout(500);
  })
  .then(() => console.log('2.5s'))
  .catch(err => console.log(`Error : ${err.message}`));
```

- Avec async / await

```
(async () => {
  try {
    await timeout(1000);
    console.log('1s');
    await timeout(1000);
    console.log('2s');
    await timeout(500);
  }
  catch (err) {
    console.log(`Error : ${err.message}`);
  }
})();
```

ECMAScript 8 - Object.values / Object.entries

- Object.values
  Retourne les valeurs d'un objet sous la forme d'un tableau

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

for (const value of Object.values(contact)) {
  console.log('value', value); // Romain, Bohdanowicz
}
```

- Object.entries
  Retourne les clés/valeurs d'un objet sous la forme d'un tableau de tableau [key, value]

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

for (const [key, value] of Object.entries(contact)) {
  console.log('key', key); // firstName, lastName
  console.log('value', value); // Romain, Bohdanowicz
}
```

ECMAScript 8 - String Padding

- String Padding
  Permet d'obtenir une chaine de caractère sur un nombre de caractères fixes en la complétant au début (padStart) ou à la fin (padEnd) par une chaine de caractère de son choix.

```
console.log('7'.padStart(3, '0')); // 007
console.log('Titre'.padEnd(20, '.')); // Titre.............
console.log('Woohoo'.padEnd(20, 'Oo')); // WoohooOoOoOoOoOoOo
```

ECMAScript 8 - Atomics

- Méthodes statiques de l'objet Atomics
  Lorsque la mémoire est partagée, plusieurs threads peuvent lire et écrire sur les mêmes données en mémoire. Les opérations atomiques permettent de s'assurer que des valeurs prévisibles sont écrites et lues, que les opérations sont finies avant que la prochaine débute et que les opérations ne sont pas interrompues.

```
// create a SharedArrayBuffer with a size in bytes
var buffer = new SharedArrayBuffer(16);
var uint8 = new Uint8Array(buffer);
uint8[0] = 7;

// 7 + 2 = 9
console.log(Atomics.add(uint8, 0, 2));
// expected output: 7

console.log(Atomics.load(uint8, 0));
// expected output: 9
```

ECMAScript 8 - Object.getOwnPropertyDescriptors

- La méthode Object.getOwnPropertyDescriptors() renvoie l'ensemble des descripteurs des propriétés propres d'un objet donné.

```
const contact = {
  firstName: 'Romain',
};

Object.defineProperty(contact, '_visible', {
  value: true,
  enumerable: false,
});

const descriptors = Object.getOwnPropertyDescriptors(contact);
console.log(descriptors.firstName.writable); // true
console.log(descriptors._visible.enumerable); // false
```

ECMAScript 9 / ECMAScript 2018

ECMAScript 9 - Rest/Spread Properties

- Rest/Spread Properties
  Syntaxes inspirée de Rest/Spread sur les tableaux
- Rest
  Permet de récupérer les propriétés restantes
- Spread
  Permet d'affecter l'ensemble des propriétés à un autre objet

```
const coords = {
  x: 10,
  y: 20,
  z: 30,
};

// Rest
const {z, ...coords2d} = coords;
console.log(JSON.stringify(coords2d)); // {"x":10,"y":20}

// Spread
const coords3d = {...coords2d, z};
console.log(JSON.stringify(coords3d)); // {"x":10,"y":20,"z":30}

const clone = {...coords};
```

ECMAScript 9 - Template Literal Revision

- Modifie la spec Template Literal pour qu'elle autorise des séquences commençant par \u, \x, \0 autre que \u00FF or \u{42}, \xFF ou \0100

```
let bad = `bad escape sequence: \unicode`; // throws early error
```

ECMAScript 9 - Regexp DotAll Flag

- Ajoute un flag aux expressions régulières pour que le meta-caractère '.' inclue les retours à la ligne
- Avant ES9 il aura fallu créer un meta-caractère [\s\S] (\s tous les caractères autres que des caractères non-imprimable, \S tous les caractères non imprimables)

```
const htmlStr = `
<body>
  <div class="clock"></div>
  <script src="clock.js"></script>
  <script src="index.js"></script>
</body>
`;

const distTag = '<script src="bundle.js"></script>';
const regex = /<script.*<\/script>/s;
// ES8
// const regex = /<script[\s\S]*<\/script>/;

console.log(htmlStr.replace(regex, distTag));
/*
<body>
  <div class="clock"></div>
  <script src="bundle.js"></script>
</body>
*/
```

ECMAScript 9 - Regexp Capture Group

- Il est désormais possible de nommer les capture groups (entre parenthèse dans une regexp)
- Les parenthèses commencent alors par ?<nom>
- Le retour n'est plus un tableau mais un objet

```
const regex = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/;
const {groups: {year, month, day}} = regex.exec('1985-10-01');

console.log('year', year); // 1985
console.log('month', month); // 10
console.log('day', day); // 01

// ES8
// const regex = /(\d{4})-(\d{2})-(\d{2})/;
// const [, year, month, day] = regex.exec('1985-10-01');
//
// console.log('year', year); // 1985
// console.log('month', month); // 10
// console.log('day', day); // 01
```

ECMAScript 9 - Regexp Lookbehind

- On pouvait avec des expressions régulières, matcher qui en précède une autre, sans capturer la seconde
- On peut depuis ES9 capturer la suite, sans capturer ce qui précède

```
// Lookbehind ES9
const str = 'It is two past ten';
console.log(/(?<=\spast\s)\w+/.exec(str)[0]); // ten
console.log(/(?<!\spast\s)\w+/.exec(str)[0]); // It

// Lookahead ES8
console.log(/\w+(?=\spast\s)/.exec(str)[0]); // two
console.log(/\w+(?!\spast\s)/.exec(str)[0]); // It
```

ECMAScript 9 - Regexp Unicode property escapes

- ES6 introduit les expressions régulières unicode
- ES9 permet de spécifier des types de caractères unicode
  http://unicode.org/reports/tr18/#Categories

```
const frenchStr = 'Une phrase en français';
console.log(frenchStr.match(/[a-zA-Z]+/gu));
// [ 'Une', 'phrase', 'en', 'fran', 'ais' ]

console.log(frenchStr.match(/\p{Alphabetic}+/gu));
// [ 'Une', 'phrase', 'en', 'français' ]

const emojiStr = 'LOL 😂😅🤣 Awesome !!! 😜';
console.log(emojiStr.match(/\p{Emoji}/gu));
// [ '😂', '😅', '🤣', '😜' ]
```

ECMAScript 9 - Promise.prototype.finally

- Introduction d'une méthode finally qui sera toujours exécutée, erreur ou non

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

let filehandle;

fs.promises.open('./example.txt', 'r')
  .then((fd) => {
    filehandle = fd;
    return filehandle.readFile();
  })
  .then((data) => {
    console.log(data.toString()); // Contenu du fichier
  })
  .finally(() => {
    filehandle.close();
  });
```

ECMAScript 9 - Promise.prototype.finally

- Avec les fonctions asynchrones

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

(async () => {
  let filehandle;
  try {
    filehandle = await fs.promises.open('./example.txt', 'r');
    const data = await filehandle.readFile();
    console.log(data.toString()); // Contenu du fichier
  }
  finally {
    filehandle.close();
  }
})();
```

ECMAScript 9 - Asynchronous iteration

- for - await - of permet de boucler sur des itérateurs ou des générateurs asynchrones

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

async function* readLines(path, bufferSize = 4096, encoding = 'utf-8') {
  let filehandle = await fs.promises.open(path, 'r');

  try {
    let eof = false;
    let strBuffer = '';
    do {
      const { bytesRead, buffer } = await filehandle.read(Buffer.alloc(bufferSize), 0, bufferSize);
      strBuffer += buffer.toString(encoding, 0, bytesRead);
      const lines = strBuffer.split(/\n|\r\n|\r/);

      for (const line of lines.slice(0, lines.length - 1)) {
        yield line;
      }

      strBuffer = lines[lines.length - 1];
      eof = !bytesRead;
    }
    while (!eof);
  } finally {
    await filehandle.close();
  }
}

(async () => {
  for await (const line of readLines('./3-lines.txt')) {
    console.log(line);
  }
})();
```

ECMAScript 10 / ECMAScript 2019

ECMAScript 10 - Optional Catch Binding

- Les parenthèses d'un bloc catch deviennent optionnel lorsque l'erreur n'est pas utilisée

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

try {
  fs.statSync('./does-not-exists');
} catch {
  console.log('An error occured');
}
```

ECMAScript 10 - JSON Superset

- Ajoute le support des caractères Unicode non échappés U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR
- Ces derniers étant présents dans la norme JSON mais indisponibles dans les chaines de caractères JS

ECMAScript Next / ESNext

ECMAScript Next - Introduction

- TC39 : Ecma's Technical Committee 39
  Groupe de travail sur la norme JavaScript
  https://github.com/tc39/ecma262

- Membres
  Bloomberg, Microsoft, AirBnb, Apple, Google, Facebook, Mozilla, Meteor, Salesforce, GoDaddy, JS Foundation…

- 5 étapes :
  - Stage 0: Strawman → Publiée via un formulaire
  - Stage 1: Proposal → A l'étude
  - Stage 2: Draft → Peut encore bouger
  - Stage 3: Candidate → Figée
  - Stage 4: Finished → Sera dans la prochaine norme

ECMAScript Next - global

- Dans le navigateur l'objet global s'appelle window, dans Node.js global, parfois this (en mode sloppy)
- Cette norme prévoit de faire référence à l'objet global via la variable global, quelque soit l'environnement

```
function foo() {
  global.firstName = 'Romain';
}

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

ECMAScript Next - Dynamic Imports

- ES6 introduit les modules ECMAScript (statiques en début de fichiers)
- Node.js et des outils de build comme webpack, browserify ou SystemJS permettent d'utiliser les modules CommonJS (fonctions synchrones)
- ESNext prévoit un import dynamique (fonctions asynchrones retournants une promesse)
- webpack est déjà compatible (le code est mis dans un bundle différent, chargé au moment de l'import)

```
document.addEventListener('click', () => {
  import('./calc').then((module) => {
    console.log(module.add(1, 2));
  });
});
```

ECMAScript Next - BigInt

- ESNext prévoit une classe BigInt et une syntaxe littérale pour les déclarer
- Les opérateurs utilisables sur des nombres le restent sur des BigInt
- La norme prévoit également des tableaux typés BigInt64Array et BigUint64Array

```
const n = Number.MAX_SAFE_INTEGER;
console.log(n); // 9007199254740991, 2^53 - 1
console.log(n + 1); // 9007199254740992, 2^53
console.log(n + 2); // 9007199254740992, 2^53, same as above

const bigint = 9007199254740991n;
console.log(bigint); // 9007199254740991n, 2^53 - 1
console.log(bigint + 1n); // 9007199254740992n, 2^53
console.log(bigint + 2n); // 9007199254740993n, 2^53 + 1
```

ECMAScript Next - Class Properties

- ESNext prévoit que l'on puisse déclarer des propriétés au niveau de la classe

```
class Contact {
  firstName = 'Romain';
  hello() {
    return `Hello my name is ${this.firstName}`;
  }
}

const romain = new Contact();
console.log(romain.hello()); // Hello my name is Romain
```

- Les propriétés préfixées par un dièse seraient privées

```
class Contact {
  #firstName = 'Romain';
  hello() {
    return `Hello my name is ${this.#firstName}`;
  }
}

const romain = new Contact();
console.log(romain.hello()); // Hello my name is Romain
```

- React encourage déjà l'utilisation de cette syntaxe dans ses docs

ECMAScript Next - Array.prototype.flat

- ESNext prévoit une méthode pour aplatir les tableaux https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/flat
- flatMap est une méthode qui revient à appeler map suivi de flat (moins verbeux et plus performant)

```
console.log([[1, 2], [3, 4]].flat());
// [1, 2, 3, 4]

console.log([[[1, 2], [3, 4]], [[5, 6], [7, 8]]].flat(Number.MAX_SAFE_INTEGER));
// [1, 2, 3, 4, 5, 6, 7, 8]

console.log([1, 2, 3, 4].map(x => [x ** 2]).flat());
// [2, 4, 9, 16]

console.log([1, 2, 3, 4].flatMap(x => [x ** 2]));
// [2, 4, 9, 16]
```

ECMAScript Next - Optional Chaining

- L'opérateur ? permet de ne pas générer d'erreur lorsque l'on essaye d'accéder à des propriétés sur null ou undefined (comme dans les templates Angular)

```
const contacts = [{
  firstName: 'Romain',
  address: {
    city: 'Paris',
  },
}, {
  firstName: 'Steven',
}];

for (const contact of contacts) {
  const city = contact.address?.city;
  console.log('city', city); // Paris, undefined
}
```

formation.tech

JavaScript Asynchrone

JavaScript Asynchrone - Introduction

- 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'une code au moment où une interaction se produit (ex : clic, fin de chargement, reception 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
```

JavaScript Asynchrone - Perte de contexte

- Dans l'implémentation setTimeout de Node.js : https://github.com/nodejs/node/blob/v6.x/lib/timers.js#L382

```
function ontimeout(timer) {
  var args = timer._timerArgs;
  var callback = timer._onTimeout;
  if (!args)
    timer._onTimeout();
  else {
    switch (args.length) {
      case 1:
        timer._onTimeout(args[0]);
        break;
      case 2:
        timer._onTimeout(args[0], args[1]);
        break;
      case 3:
        timer._onTimeout(args[0], args[1], args[2]);
        break;
      default:
        Function.prototype.apply.call(callback, timer, args);
    }
  }
  if (timer._repeat)
    rearm(timer);
}
```

JavaScript Asynchrone - Perte de contexte

- 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
```

JavaScript Asynchrone - Perte de contexte

- 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
```

JavaScript Asynchrone - Perte de contexte

- Solution 3 : Arrow Function (ES6)
  Les fonctions fléchées ne lient pas de valeur pour this, ce qui permet au callback de retrouvé la valeur de la fonction parent.

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

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

JavaScript Asynchrone - Callback Hell

- 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.`);
  });
});
```

JavaScript Asynchrone - Async

- 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.`);
});
```

JavaScript Asynchrone - Promesses

- 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 grace 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);
```

JavaScript Asynchrone - Promesses

- 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);
```

JavaScript Asynchrone - Observables

- Promise
  - Utilisable uniquement pour des événements ponctuels (ex: setTimeout)
  - Pas annulable (sauf en utilisant des bibliothèques comme bluebird)
- Observable
  - Utilisable pour des événements ponctuels ou récurrents (ex: setInterval)
  - Annulable
  - Contient des opérateurs pour l'enrichir (map, debounce, flatMap…)
  - Peut se créer et se déclencher en 2 temps

JavaScript Asynchrone - Observables

- Exemple Promise vs Observable

```
const Observable = require('rxjs').Observable;

const timeout = new Promise((resolve) => {
  setTimeout(() => {
    resolve();
  }, 1000);
});

timeout.then(() => {
  console.log('timeout 1s');
});

const interval$ = new Observable((observer) => {
  setInterval(() => {
    observer.next();
  }, 1000);
});

interval$.subscribe(() => {
  console.log('observable 1s');
});
```

JavaScript Asynchrone - Exercice

- Exercice de build
  Ennoncé sur https://github.com/bioub/Exercice-Build
- A faire avec Promise et éventuellement async / await