
ECMAScript

Table des matières

Introduction

JavaScript est un langage de programmation utilisé principalement sur le Web, développé par Netscape et repris par Microsoft sous le nom de JScript. ECMAScript est une tentative de normalisation du noyau du langage : sa syntaxe, ses mots-clés et ses composants natifs. La troisième édition du standard ECMA-262 a été publiée en décembre 1999.

Variables locales

Il est maintenant possible de définir des variables locales à un bloc (let, const), ce qui évite des conflits potentiels de variables déclarées avec var.

```
function a() {
  let b = "foo"
}

// b is not defined
```

Arguments par défaut

Les fonctions permettent maintenant de passer des arguments par défaut. Quant au chaines elles supportent l'interpolation.

```
function greetings(recipient = "synbioz") {
 let message = `Hello ${recipient}!`
  console.log(message)
}
```

Déstructuration

```
let [a, b, c] = [1, 2, 3, 4]

// a == 1

// b == 2

// c == 3

let [a, b, ...c] = [1, 2, 3, 4]

// a == 1

// b == 2
```

```
// c == [3, 4]
```

À l'image des splats en coffescript les `...` permettent de récupérer le reste des arguments.

```
// ne stocke pas le 2
function f() { return [1, 2, 3] }
let [a, , b] = f();
// swap
[a, b] = [b, a]
```

Chaînes de caractères sur plusieurs lignes

Les utilisateurs d'azerty risquent de ticker devant le caractère choisi (backtick), mais ce sera une excellente raison de passer au bépo.

```
var str = `hello
 i'm talking
 to you.
Sucre syntaxique
for(x of [3, 8, 12]) // 3, 8, 12
console.log("even") if(x % 2 == 0)
```

Note: `for of` n'a pas vocation à remplacer `for in`.

Proxies

Les proxies offre une interface de méta-programmation. Ils permettent de définir des méthodes et autres propriétés objets à la volée.

Pour rapprocher cela de ruby il faut imaginer des choses comme `Class.new`, `define_method`, `method_missing` …

Quand le proxy va recevoir un appel de méthode, c'est le get du proxy qui sera appelé avec en paramètre l'objet et le nom de la méthode.

Ici `get` va vérifier que le nom de la méthode fait partie des tags et agir en conséquence.

Weak maps

Les weak maps s'apparentent à un hash dont les clés sont des objets.

Les éléments auxquels on n'accède pas depuis longtemps sont automatiquement collectés par le ramasse miette.

```
var wm = WeakMap();


(function () {
   let o = {};
   wm.set(o, "bar");
})();
// Ici o n'existera plus en mémoire.
```

L'intérêt de ce système est d'éviter les fuites mémoires d'objets qui seraient dans un hash classique et qui auraient une référence circulaire (et donc ne serait jamais collectés par le ramasse miette).

Les weak map peuvent aussi permettre d'étendre un objet global dans un contexte défini, sans risque de collision.

```
   // avant: aucune garantie que l'on n'écrase pas une valeur existante.
window.my_object["foo"] = "bar"
// après: plus de risque de collision
function() {
 let wm = new WeakMap()
 wm.set(window.my_object, { 'foo': 'bar' })
}()
```

Générateurs

Il s'agit de fonctions interruptibles, un exemple ici d'implémentation de fibonacci avec:

```
function fibonacci() {
```

```
 var i = 0, j = 1;
 while (true) {
   yield i;
   var t = i;
   i = j;
   j += t;
 }
}
var f = fibonacci();
for (var i = 0; i < 10; i++) {
 console.log(f.next());
 // 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
}
```

Structure de données

ECMAScript 6 offre la possibilité de définir ses propres structures de données, en utilisant des données binaires.

```
const Point2D = new StructType({ x: uint32, y: uint32 });
const Color = new StructType({ r: uint8, g: uint8, b: uint8 });
const Pixel = new StructType({ point: Point2D, color: Color });
```

Color, Pixel et Point2D agissent tous les 3 comme des objets. Ils permettent de définir le cadre des futures instances.

Approche objet

ECMA 6 offre des mécanismes natifs de gestion objet, comme la définition de classe, l'héritage…

Un exemple de définition d'une classe:

```
class Point extends Base {
```

```
  constructor(x,y) {
    super();
    this[px] = x, this[py] = y;
    this.r = function() { return Math.sqrt(x * x + y * y); }
  }
  get x() { return this[px]; }
  get y() { return this[py]; }
}
```

Modules

ECMAScript 6 propose un système de module qui permettra de factoriser le code et importer les éléments définis à divers endroits.

```
module math {
  export function sum(x, y) {
    return x + y;
  }
  export var pi = 3.141593;
}
import {sum, pi} from math;
console.log(sum(pi, 42))
```

Ce système va s'avérer très pratique pour les libs telles que MooTools, jQuery ou autre.

Variable privées

À l'heure actuelle pour définir des variables privées en javascript dans une classe, voilà le code qu'on doit utiliser:

```
function A() {
  // private var
  var _foo = "bar";
```

```
  this.foo = function() {
    return _foo;
  };
}
```

Le problème de cette solution est qu'elle n'est pas efficace car la méthode d'accès (fonction this.foo) sera régénérée au runtime à chaque instance créée, alors que le traitement est toujours le même.

ECMAScript 6 introduit donc un nouveau concept pour la gestion des propriétés privées:

```
var key = Name.create("awesome_key")
function A() { this[key] = 42 }
A.prototype.getKey = function() { return this[key] }
```

Avec ce système on n'a pas de visiblité sur le nom de la propriété en elle même.

Cette présentation d'ECMAScript 6 vous donne un avant goût du potentiel à venir de javascript.

Évidemment il est impossible de prédire quand cette version sera disponible dans nos navigateurs, encore plus quand elle le sera dans tous les navigateurs ; mais cela vous donne un avant goût de ce que nous réserve javascript dans les versions à venir.