Prise en main d'ES6

Aperçu des principales fonctionnalités

Par Adrian Mejia - Jules Renton--Epinette (traducteur)

Date de publication : 27 novembre 2018

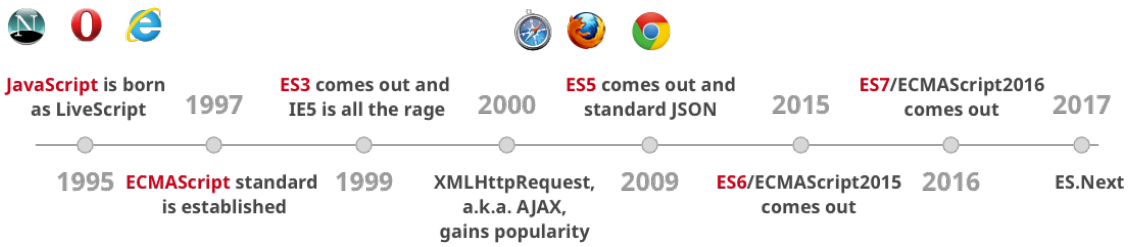
JavaScript a pas mal changé ces dernières années. Ce tutoriel détaille douze nouvelles fonctionnalités que vous pouvez utiliser dès aujourd'hui.

Commentez ♪

I - Historique de JavaScript

Ces nouveaux ajouts au langage sont appelés ECMAScript 6, ou encore ES6, ou ES2015+.

Depuis sa création en 1995, JavaScript a lentement évolué. De nouveaux ajouts ont été faits au fil des années. ECMAScript est arrivé en 1997 pour orienter JavaScript, et a publié des versions telles qu'ES3, ES5, ES6, etc.



Comme vous pouvez le voir, il y a des trous des 10 et 6 ans entre ES3, ES5 et ES6. Le nouveau modèle est de faire de petits ajouts croissants chaque année, au lieu de changements massifs en un coup comme avec ES6.

II - Compatibilité avec les navigateurs

Tous les navigateurs et environnements récents supportent déjà ES6 !

Feature name	Current browser 97%	iOS 10 100%	SF 10 100%	Node 6.5[6] 97%	CH 54, OP 41[1] 97%	XS6 96%	Edge 14[4] 93%	FF 49 92%	FF 45 ESR 86%	Edge 13[4] 83%	Babel + core-js[2] 71%	JXA 59%	Type-Script + core-js 59%	T
Syntax														
default function parameters	7/7	7/7	7/7	7/7	7/7	7/7	7/7	4/7	4/7	0/7	4/7	0/7	5/7	
rest parameters	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	3/5	0/5	4/5	
spread (...) operator	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	13/15	11/15	4/15	
object literal extensions	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	5/6	6/6	
for..of loops	9/9	9/9	9/9	9/9	9/9	9/9	7/9	7/9	7/9	7/9	9/9	8/9	3/9	
octal and binary literals	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	
template literals	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	4/5	5/5	3/5	
RegExp "y" and "u" flags	5/5	5/5	5/5	5/5	5/5	2/5	5/5	5/5	2/5	5/5	3/5	0/5	0/5	
destructuring, declarations	22/22	22/22	22/22	22/22	22/22	21/22	21/22	21/22	19/22	0/22	21/22	19/22	15/22	
destructuring, assignment	24/24	24/24	24/24	24/24	24/24	24/24	23/24	23/24	21/24	0/24	24/24	21/24	19/24	
destructuring, parameters	23/23	23/23	23/23	23/23	23/23	23/23	22/23	19/23	18/23	0/23	20/23	18/23	15/23	
Unicode code point escapes	2/2	2/2	2/2	2/2	2/2	2/2	2/2	1/2	1/2	2/2	1/2	2/2	1/2	
new.target	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	1/2	0/2	0/2	0/2	
Bindings														
const	16/16	16/16	16/16	16/16	16/16	16/16	16/16	12/16	12/16	12/16	14/16	10/16	14/16	
let	12/12	12/12	12/12	12/12	12/12	12/12	12/12	10/12	10/12	10/12	10/12	0/12	10/12	
block-level function declaration[13]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	
Functions														
arrow functions	13/13	13/13	13/13	13/13	13/13	12/13	13/13	13/13	13/13	13/13	9/13	0/13	9/13	
class	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	19/24	18/24	19/24	
super	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	4/8	7/8	7/8	
generators	27/27	27/27	27/27	27/27	27/27	27/27	27/27	25/27	25/27	27/27	24/27	0/27	0/27	

source :

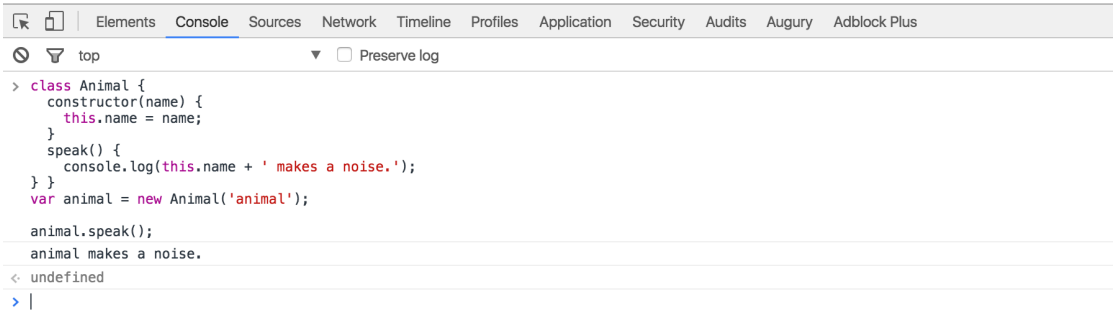
Chrome, Edge, Firefox, Safari, Node et bien d'autres ont déjà intégré la plupart des fonctionnalités de JavaScript ES6. Tout ce que vous aurez donc à faire pour l'utiliser sera de suivre ce tutoriel.

C'est parti pour la prise en main d'ECMAScript 6 !

III - Fonctionnalités principales d'ES6

Vous pouvez tester ces bouts de code dans la console de votre navigateur ! (F12)

Donc ne me croyez pas sur parole et essayez tous les exemples ES5 et ES6. Allons-y !



III-A - Variables à portée de bloc

Avec ES6, nous sommes passé·e·s de la déclaration de variables avec var à l'utilisation de let et const.

Quel était le problème de var ?

Le problème de var est que la variable persiste dans d'autres blocs de code comme des boucles for ou des if.

ES5

```
1. var x = "externe";
2. function test(interne) {
3.   if (interne) {
4.     var x = "interne"; // a la fonction entière comme portée
5.     return x;
6.   }
7.   return x; // est redéfini parce que la définition en ligne 4 est hissée (hoisted)
8. }
9. test(false); // undefined
10. test(true); // "interne"
```

Pour test(false) vous vous attendez à obtenir « externe », mais non, vous avez undefined.

Pourquoi ?

Eh bien, parce que même si le bloc if n'est pas exécuté, l'expression var x en ligne 4 est hissée.

Hissage de variable *(variable hoisting) :*

- *var a la fonction pour portée. La variable est disponible dans toute la fonction, même avant d'être déclarée ;*
- *les déclarations sont hissées, vous permettant d'utiliser une variable avant sa déclaration ;*
- *les initialisations ne sont* ***pas*** *hissées. Si vous utilisez var, déclarez* ***toujours*** *vos variables avant le reste ;*
- *après application des règles de hissage, il est plus facile de comprendre ce qu'il s'est passé :*

ES5
```
1. var x = "externe";
2. function test(interne) {
3.   var x; // DÉCLARATION HISSÉE
4.   if (interne) {
5.     x = "interne"; // INITIALISATION NON HISSÉE
6.     return x;
7.   }
8.   return x;
```

ECMAScript 2015 vient à la rescousse :

ES6
```
1. let x = "externe";
2. function test(interne) {
3.   if (interne) {
4.     let x = "interne";
5.     return x;
6.   }
7.   return x; // on obtient le résultat de la ligne 1, comme attendu
8. }
9. test(false); // "externe"
10. test(true); // "interne"
```

Remplacer var par let fait fonctionner le code comme attendu. Si le bloc if n'est pas appelé, la variable x n'est pas hissée hors du bloc.

Hissage de let et « zone morte temporaire » :

- *avec ES6, let hisse la variable en haut du bloc au lieu du haut de la fonction comme ES5 ;*
- *cependant, utiliser la variable dans ledit bloc avant sa déclaration donnera une ReferenceError ;*
- *let a le bloc pour portée. On ne peut pas l'utiliser avant sa déclaration ;*
- *la « zone morte temporaire » est la zone entre le début du bloc et la déclaration de la variable.*

III-A-1 - IIFE

Prenons un exemple avant d'expliquer les IIFE (*Immediately-Invoked Function Expression*, ou fonction immédiatement appelée) :

ES5
```
1. {
2.   var privee = 1;
3. }
4.
5. console.log(privee); // 1
```

Comme vous le voyez, privee persiste hors du bloc. Il faut utiliser une IIFE (Immediately-Invoked Function Expression) pour le contenir :

ES5
```
1. (function() {
2.   var privee = 1;
3. })();
4.
5. console.log(privee); // ReferenceError
```

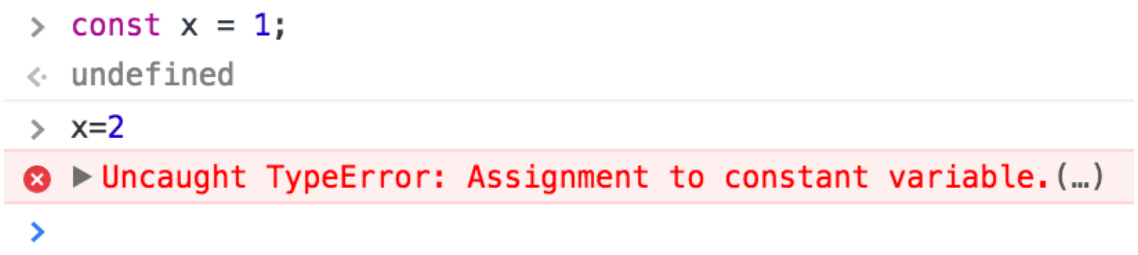
Si vous regardez jQuery, Lodash ou d'autres projets open source vous remarquerez leur utilisation d'IIFE pour éviter de polluer l'environnement global, et ne définir que des globales comme _, $ ou jQuery.

Avec ES6 c'est bien plus propre, plus besoin d'IIFE quand on peut juste utiliser un bloc avec let :

ES6
```
1. {
2.   let privee = 1;
3. }
4.
5. console.log(privee); // ReferenceError
```

III-A-2 - const

Vous pouvez aussi utiliser const si vous ne voulez pas changer une variable (et ainsi en faire une constante).



En bref, laissez tomber var au profit de let et const.

- *Utilisez const pour toutes vos références ; évitez var.*
- *Si vous voulez pouvoir modifier les références, utilisez let au lieu de const.*

III-B - Modèles de libellés (template literals)

Plus besoin de concaténation avec les modèles de libellés. Voyez plutôt :

ES5
```
1. var prenom = "Adrian",
2.     nom = "Mejia";
3. console.log("Vous vous appelez " + prenom + " " + nom + ".");
```

On peut désormais utiliser l'accent grave (`) (se trouve sur la touche 7 sur les claviers AZERTY) avec des interpolations :

ES6
```
1. const prenom = "Adrian",
2.       nom = "Mejia";
3. console.log(`Vous vous appelez ${prenom} ${nom}.`);
```

***NDT** L'espace réservé par ${} n'est pas limité aux références, il peut contenir n'importe quel code JS, y compris des opérations, des appels de fonctions, et même d'autres modèles de libellés.*

III-C - Chaînes de caractères multilignes

Plus besoin de concaténer des chaînes avec \n comme ceci :

ES5
```
var modele = "<li *ngFor='let tache of tachesAFaire' [ngClass]='{completed: tache.estFaite}'
 >\n" +
"  <div class='vue'>\n" +
"    <input class='bascule' type='checkbox' [checked]='tache.estFaite'>\n" +
"    <label></label>\n" +
"    <button class='detruire'></button>\n" +
"  </div>\n" +
"  <input class='modifier' value=''>\n" +
"</li>";
console.log(modele);
```

Les modèles de libellé d'ES6 résolvent une fois de plus le problème :

ES6
```
const modele = `<li *ngFor="let tache of tachesAFaire" [ngClass]="{completed: tache.estFaite}"
 >
   <div class="vue">
     <input class="bascule" type="checkbox" [checked]="tache.estFaite">
     <label></label>
     <button class="detruire"></button>
   </div>
   <input class="modifier" value="">
</li>`;
console.log(modele);
```

Ces deux morceaux de code font exactement la même chose.

III-D - Affectation par décomposition (destructuring assignment)

ES6 est très pratique et concis. Voyez ces exemples :

Obtenir les éléments d'un tableau

ES5
```
var tableau = [1, 2, 3, 4],
    premier = tableau[0],
    troisieme = tableau[2];
console.log(premier, troisieme); // 1 3
```

Avec ES6 :

ES6
```
const tableau = [1, 2, 3, 4],
      [premier, ,troisieme] = tableau;
console.log(premier, troisieme); // 1 3
```

Échanger les valeurs

ES5
```
var a = 1, b = 2;

var tmp = a;
a = b;
b = tmp;

console.log(a, b); // 2 1
```

Avec ES6 :

ES6
```
1. let a = 1, b = 2;
2.
3. [a, b] = [b, a];
4.
5. console.log(a, b); // 2 1
```

Décomposer pour retourner plusieurs valeurs

ES5
```
1. function marges() {
2.   var gauche=1, droite=2, haut=3, bas=4;
3.   return { gauche: gauche, droite: droite, haut: haut, bas: bas };
4. }
5. var data = marges(),
6.     gauche = data.gauche,
7.     bas = data.bas;
8. console.log(gauche, bas); // 1 4
```

Ligne 3, vous pourriez aussi retourner un tableau comme ceci :

```
return [gauche, droite, haut, bas];
```

Mais après, il faut retenir l'ordre des valeurs retournées.

```
var gauche = data[0], bas = data[3];
```

Avec ES6, l'appelant ne prend que les données dont il a besoin :

ES6
```
1. function marges() {
2.   const gauche=1, droite=2, haut=3, bas=4;
3.   return { gauche, droite, haut, bas };
4. }
5. const { gauche, bas } = marges();
6. console.log(gauche, bas); // 1 4
```

Note : ligne 3, nous voyons une autre fonctionnalité d'ES6 : il est possible de compresser { gauche: gauche } en { gauche }. C'est pas formidable ?

Décomposer pour faire correspondre les paramètres

ES5
```
1. var utilisateur = { prenom: "Adrian", nom: "Mejia"};
2.
3. function nomComplet(utilisateur) {
4.   var prenom = utilisateur.prenom,
5.       nom = utilisateur.nom;
6.   return prenom + " " + nom;
7. }
8.
9. console.log(nomComplet(utilisateur)); // Adrian Mejia
```

La même chose, mais plus concise :

ES6
```
1. const utilisateur = { prenom: "Adrian", nom: "Mejia"};
2.
3. function nomComplet({ prenom, nom }) {
```

ES6
```
4.   return `${prenom} ${nom}`;
5. }
6.
7. console.log(nomComplet(utilisateur)); // Adrian Mejia
```

Correspondance en profondeur

ES5
```
1. function reglages() {
2.   return {
3.     affichage: { couleur: 'rouge' },
4.     clavier: { type: 'azerty'} };
5. }
6.
7. var tmp = reglages(),
8.     couleurAffichage = tmp.affichage.couleur,
9.     typeClavier = tmp.clavier.type;
10.
11. console.log(couleurAffichage, typeClavier); // rouge azerty
```

Idem :

ES6
```
1. function reglages() {
2.   return {
3.     affichage: { couleur: 'rouge' },
4.     clavier: { type: 'azerty'} };
5. }
6.
7. const {
8.          affichage: { couleur: couleurAffichage },
9.          clavier: { layout: typeClavier }
10.       } = reglages();
11.
12. console.log(couleurAffichage, typeClavier); // rouge azerty
```

Ce qu'on appelle aussi la décomposition d'objet *(object destructuring)*.

Bonne pratique :

- *utilisez la décomposition de tableau pour en récupérer des éléments ou échanger des variables. Cela vous épargne la création de variables temporaires ;*
- *n'utilisez pas la décomposition de tableaux pour retourner de multiples valeurs, utilisez plutôt lé décomposition d'objet.*

III-E - Classes et objets

Avec ECMAScript 6, nous passons des « fonctions constructrices » aux « classes ».

En JavaScript, tout objet a un prototype, qui est un autre objet. Tous les objets JavaScript héritent leurs méthodes et attributs de leur prototype.

En ES5, nous faisions la Programmation Orientée Objet (POO) avec des fonctions constructrices utilisées comme suit :

ES5
```
var Animal = (function () {
  function Constructeur(name) {
    this.name = name;
  }
  Constructeur.prototype.speak = function parler() {
    console.log(this.name + ' makes a noise.');
  };
  return Constructeur;
})();

var animal = new Animal("animal");
animal.parler(); // animal fait du bruit.
```

ES6 nous donne un peu de sucre syntaxique. On peut faire la même chose avec moins d'expressions génériques et de nouveaux mots-clés tels que class et constructor. Remarquez notamment comment nous définissons la méthode parle() :

ES6
```
class Animal {
  constructor(nom) {
    this.name = nom;
  }
  parler() {
    console.log(`${this.name} fait du bruit.`);
  }
}

const animal = new Animal('animal');
animal.parler(); // animal fait du bruit.
```

Comme vous le voyez, les deux styles (ES5/6) donnent le même résultat et sont utilisés de la même manière.

Bonnes pratiques :

- *utilisez toujours la syntaxe avec class et évitez de changer manuellement le prototype. Pourquoi ? Parce qu'ainsi le code est davantage concis et facile à comprendre ;*
- *évitez d'avoir un constructeur vide. Les classes ont un constructeur par défaut si aucun n'est défini.*

III-F - Héritage

Travaillons notre classe Animal. Mettons que nous voulons l'étendre et définir une classe Lion.

En ES5, il faut bidouiller encore plus le prototype :

ES6
```
var Lion = (function () {
  function Constructeur(name){
    Animal.call(this, name);
  }
  // prototypal inheritance
  Constructeur.prototype = Object.create(Animal.prototype);
  Constructeur.prototype.constructor = Animal;
  Constructeur.prototype.parler = function parler() {
    Animal.prototype.parler.call(this);
     console.log(this.name + ' rugit');
   };
   return MyConstructor;
})();

```

ES6
```
15. var lion = new Lion('Simba');
16. lion.parler(); // Simba fait du bruit.
17. // Simba rugit.
```

Pour expliquer un peu :

- ligne 2, nous appelons explicitement le constructeur d'Animal en passant les paramètres ;
- lignes 6-7, nous assignons à Lion le prototype d'Animal ;
- ligne 10, nous appelons la méthode parle de la classe parente Animal.

En ES6, nous avons les mots-clés extends et super.

ES6
```
1. class Lion extends Animal {
2.   parle() {
3.     super.parle();
4.     console.log(`${this.nom} rugit.`);
5.   }
6. }
7.
8. const lion = new Lion('Simba');
9. lion.parle(); // Simba fait du bruit.
10. // Simba rugit.
```

Constatez l'immense gain en lisibilité de ce code ES6 comparé à l'ES5. Yeah !

Bonne pratique :

utilisez la fonctionnalité d'héritage intégrée avec extends.

III-G - Fonctions fléchées

ES6 n'a pas supprimé les expressions de fonctions, mais en a ajouté une nouvelle appelée « fonctions fléchées ».

En ES5, on avait quelques soucis avec this :

ES5
```
1. var _this = this; // besoin de tenir une référence
2.
3. $(".btn").click(function(event) {
4.   _this.sendData(); // fait référence au this externe
5. });
6.
7. $(".input").on("change", function(event){
8.   this.sendData(); // fait référence au this externe
9. }.bind(this)); // on lie le this externe
```

On doit utiliser une référence temporaire à this pour l'utiliser au sein d'une fonction, ou d'utiliser bind.

En ES6, on peut utiliser une fonction fléchée !

ES6
```
1. // this fera référence à celui externe
2. $(".btn").click(event => this.sendData());
3.
4. // retour implicite
5. const ids = [291, 288, 984],
```

ES6
```
6.        messages = ids.map(valeur => `L'ID est ${valeur}`);
```

***NDT** En principe, la syntaxe est (paramètres…) => { code },*

mais il est possible de se passer des parenthèses s'il n'y a qu'un paramètre, et de se passer des accolades s'il n'y a qu'une instruction.

Exemples
```
() => console.log("Fonction fléchée sans paramètre.");

(chaine) => { console.log(chaine); }
chaine => console.log(chaine) // idem, mais plus concis

(prm1, prm2) => {
  console.log(prm1);
  console.log(prm2);
} // Plusieurs paramètres et instructions, pas le choix que de mettre
 parenthèses et accolades
```

***NDT** Les fonctions fléchées ne redéfinissant pas this, il ne faut pas les utiliser comme méthodes pour les objets.*

Exemple
```
1. const objet = {
2.   a: 5,
3.   methode: () => this.a = 10
4. }
5.
6. objet.methode();
7. // objet.a n'a pas été modifié, mais une variable a globale a été créée !
8. console.log(objet.a); // 5
9. console.log(a); // 10
```

III-H - Promesses natives

Nous sommes passé·e·s de l'enfer des rappels *(callback hell)* aux promesses *(promises)*.

ES5
```
1. function afficheApresDelai(chaine, delai, fait){
2.   setTimeout(function(){
3.     fait(chaine);
4.   }, delai);
5. }
6.
7. afficheApresDelai("Salut ", 2e3, function(resultat){
8.   console.log(resultat);
9.
10.   // rappel imbriqué (nested callback)
11.   afficheApresDelai(resultat + "lecteur·rice", 2e3, function(result){
12.     console.log(result);
13.   });
14. });
```

Nous avons là une fonction qui reçoit un rappel à exécuter une fois que c'est fait. Nous devons l'exécuter deux fois l'une après l'autre. C'est pourquoi nous avons rappelé afficheApresDelai une seconde fois dans la fonction de rappel.

Cela peut vite devenir un sacré bazar si vous avez besoin d'un 3e ou 4e rappel. Voyons un peu comment s'en sortir avec les promesses :

ES6
```
function afficheApresDelai(chaine, delai){
  return new Promise((resout, rejette) => {
    setTimeout(function(){
      resout(chaine);
    }, delai);
  });
}
afficheApresDelai("Salut ", 2e3).then(resultat => {
  console.log(resultat);
   return afficheApresDelai(`${resultat}lecteur·rice`, 2e3);
}).then(resultat => {
   console.log(resultat);
});
```

Comme vous le voyez, avec les promesses nous pouvons utiliser then pour faire quelque chose après qu'une autre fonction est terminée. Plus besoin d'imbriquer les rappels.

III-I - For…of

Nous sommes passé·e·s de forEach à for…of :

ES5
```
// for
var tableau = ["a", "b", "c", "d"];
for (var i = 0; i < tableau.length; i++) {
  var element = tableau[i];
  console.log(element);
}

// forEach
tableau.forEach(function (element) {
   console.log(element);
});
```

Le for…of d'ES6 nous permet de faire des itérations :

ES6
```
// for ...of
const tableau = ["a", "b", "c", "d"];
for (const element of tableau) {
    console.log(element);
}
```

***NDT Conseil** : oubliez forEach. La raison est simple : forEach est une fonction, et appeler une fonction à répétition a un coût. Préférez l'utilisation de for…of pour de meilleures performances.*

III-J - Paramètres par défaut

Alors que nous devions vérifier qu'un paramètre avait bien été défini, nous pouvons désormais lui assigner une valeur par défaut. Avez-vous déjà fait quelque chose de ce genre ?

ES5
```
function point(x, y, estMarque){
```

ES5
```
2.   x = x || 0;
3.   y = typeof(y) === "undefined" ? -1 : y;
4.   isFlag = typeof(estMarque) === 'undefined' ? true : estMarque;
5.   console.log(x,y, estMarque);
6. }
7.
8. point(0, 0) // 0 0 true
9. point(0, 0, false) // 0 0 false
10. point(1) // 1 -1 true
11. point() // 0 -1 true
```

Tous ces typeof(…) === « undefined »… Heureusement, ES6 nous facilite la tâche :

ES6
```
1. function point(x = 0, y = -1, estMarque = true){
2.   console.log(x, y, estMarque);
3. }
4.
5. point(0, 0) // 0 0 true
6. point(0, 0, false) // 0 0 false
7. point(1) // 1 -1 true
8. point() // 0 -1 true
```

III-K - Paramètres du reste (rest parameters)

Des arguments, nous sommes passé·e·s aux paramètres du reste et à la syntaxe de décomposition *(spread operator)*.

En ES5, obtenir un nombre arbitraire d'arguments est assez pataud :

ES5
```
1. function printf(format) {
2.   var params = [].slice.call(arguments, 1);
3.   console.log("params: ", params);
4.   console.log("format: ", format);
5. }
6.
7. printf("%s %d %.2f", "adrian", 321, Math.PI);
```

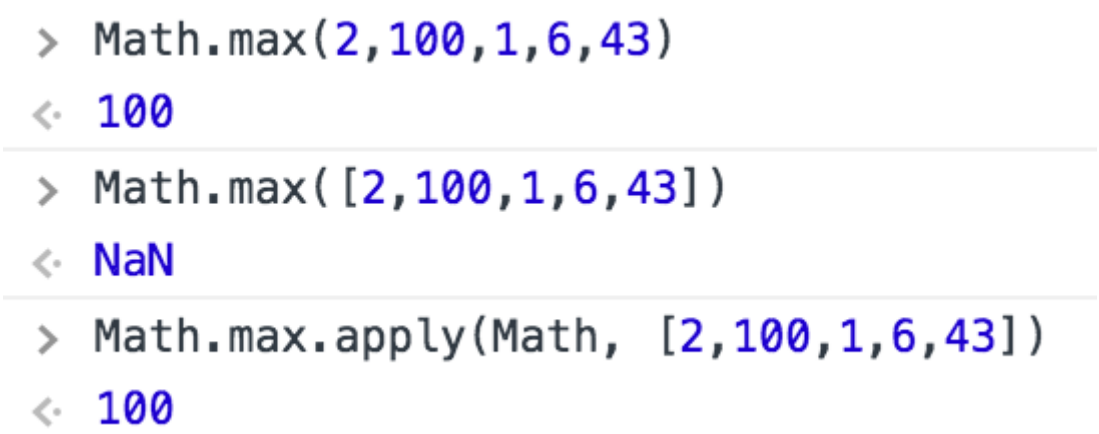
Nous pouvons faire la même chose avec l'opérateur de reste (…) :

ES6
```
1. function printf(format, ...params) {
2.   console.log("params: ", params);
3.   console.log("format: ", format);
4. }
5.
6. printf("%s %d %.2f", "adrian", 321, Math.PI);
```

III-L - Syntaxe de décomposition (spread operator)

Nous sommes passé·e·s d'apply() à la syntaxe de décomposition, avec … à la rescousse :

Rappel : nous utilisons apply() pour convertir un tableau en list d'arguments. Par exemple, Math.max() prend une liste de paramètres, mais si nous avons un tableau nous pouvons utiliser apply() pour que cela fonctionne :



Comme nous venons de le voir, nous pouvons utiliser apply pour passer des tableaux comme des listes d'arguments :

ES5
```
Math.max.apply(Math, [2,100,1,6,43]) // 100
```

En ES6, nous pouvons utiliser la syntaxe de décomposition :

ES6
```
Math.max(...[2,100,1,6,43]) // 100
```

Cela permet également de remplacer concat :

ES5
```
1. var tableau1 = [2, 100, 1, 6, 43],
2.     tableau2 = ["a", "b", "c", "d"],
3.     tableau3 = [false, true, null, undefined];
4.
5. console.log(tableau1.concat(tableau2, tableau3));
```

En ES6 :

ES6
```
1. const tableau1 = [2, 100, 1, 6, 43],
2.       tableau2 = ["a", "b", "c", "d"],
3.       tableau3 = [false, true, null, undefined];
4.
5. console.log([...tableau1, ...tableau2, ...tableau3]);
```


NDT *La syntaxe de décomposition a été ajoutée pour les objets, mais seulement dans le standard ES2018 (aussi appelé ES9), qui est très récent au moment d'écrire cette note et devrait donc être évité pendant encore quelque temps.*

Cherchez Object Rest/Spread Properties pour plus de détails.

IV - Conclusion

JavaScript a subi beaucoup de changements. Ce tutoriel couvre la plupart des principales fonctionnalités que tout·e développeu·r·se JavaScript devrait connaître. Aussi, nous avons vu les bonnes pratiques à adopter pour rendre votre code plus concis et facile à comprendre.