Introduction à ES6

Quelques nouveautés de ES6

* Déclaration des variables avec let plutôt que var
* Déclaration des constantes avec const
* Les fonctions fléchées =>
* Définition d’un objet avec Class plutôt qu’avec prototype
* Intégrer plusieurs fichiers JavaScript avec les modules et import/export

Un bloc

En JavaScript, un bloc est créé par une paire d’accolades ; on va donc avoir un bloc à chaque fois qu’on utilise l’une des syntaxes suivantes (liste non exhaustive) :

// Test

if (...) {

// bloc

}

// Boucle

for (...) {

// bloc

}

// Une classe

class Foo {

// bloc

}

// Fonction

function foo() {

// bloc

}

let

* Let permet de déclarer une variable dont la portée sera limitée au bloc qui contient cette déclaration. Le bloc est défini par un niveau d’accolade. Si le bloc contient d’autres blocs la portée de la variable inclura tous les autres blocs inclus. En déclarant avec ***let*** il n’est pas obligatoire d’initialiser la nouvelle variable avec une valeur.

let k ; // pas nécessaire d’initialiser la variable

let k = 0 ;

var

* ***var*** C’est la plus connue, celle qui existe depuis Javascript 1.0.
* En résumé on peut dire :
* que ***var*** est similaire à **let**, mais avec une **portée de fonction** (là **ou let a une portée de bloc**).

Différences entre let et var

/\* on ajoute un écouteur sur le chargement d'une fenêtre \*/

window.addEventListener('load',*function*(){

// la différence entre let et var

*let* k = 1;

*console*.log('let ', k) // let 1

{

*let* k=5

*console*.log('let ', k) // let 5

}

*console*.log('let ', k) // let 1

//////////////////////////////////////////////////

*var* j = 1;

*console*.log('var ', j) // var 1

{

*var* j = 5

*console*.log('var ', j) // var 5

}

*console*.log('var ', j) // var 5

Const pour les variables non réassignables

***Const*** permet de déclarer une constante disponible uniquement en lecture. La portée de ***const*** est celle du bloc, comme la déclaration ***let***. Pour la rendre globale, il faut simplement la définir hors de toute fonction.

const a = 'foo';

a = 'bar'; // Erreur, car `a` ne peut pas être réassignée

***const***permet de déclarer une variable qui doit avoir une valeur initiale et ne peut pas être réassignée.

ATTENTION. Cela ne veut pas dire qu’une variable déclarée avec const est une constante.

Par exemple, si la variable contient un objet, on peut modifier les propriétés de cet objet. Si la variable contient un tableau, on peut ajouter ou retirer des membres du tableau. Autrement dit, la valeur d’une const n’est PAS immutable.

Utilisez «const» dans 90% des cas, let le reste du temps

En fréquence, on utilise const quasiment tout le temps, car

1. on a rarement besoin de réassigner une variable ;
2. on manipule souvent des objets et des tableaux plutôt que des primitives,
3. const autorise les modifications dans un objet ou dans un tableau.

Certains IDE comme WebStorm vous indiqueront quand vous devez utiliser let ou const basé sur le fait que vous réassignez vos variables ou pas.

Références :

<https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/const>

<https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/let>

<https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/var>

Les fonctions fléchées de ES6

Const maFonction = () => `Vive les fonctions fléchées`

Les fonctions fléchées (arrow functions) ont été introduites avec ES6 (ES2015), c'est aujourd'hui la fonctionnalité de ES6 la plus utilisée, c'est dire combien elle a apporté aux développeurs Javascript.

**Syntaxe standard**

**function([param] [, param]){  
instructions  
}**

**Syntaxe fléché**

**([param] [, param]) => {  
 instructions  
}**

**(param1, param2, …, param2) => expression  
// équivalent à**

**(param1, param2, …, param2) => {  
return expression;  
}**// Parenthèses non nécessaires // quand il n'y a qu'un seul argument

**param => expression**// Une fonction sans paramètre peut s'écrire   
// avec un couple de parenthèses

**() => { instructions}**

///// Syntaxe d'une fonction ///////////////////////////////

*console*.log('Différentes façons d\'écrire une fonction ')

*function* auCarre1(*x*)

{

return x \* x;

}

*let* auCarre2 = *function*(*x*){

return x \* x;

}

*const* auCarre3 = *function*(*x*){

return x \* x;

}

*let* auCarre4 = (*x*)*=>* {

return x \* x;

}

*const* auCarre5 = (*x*)*=>* {

return x \* x;

}

// une fonction expression (pas de return)

*const* auCarre6 = (*x*) *=>* x \* x

// et encore plus simple

/\*

Les accolades ne sont même pas obligatoire ici et le mot clé return aussi,

parce que l'on renvoie une seule instruction. Les fonctions fléchées permettent ce qu'on appelle implicit return (retour implicite), c'est à dire pas besoin du mot clé return dans le cas ou nous n'avons qu'une seule instruction comme ici par exemple, on peut donc écrire notre fonction comme ça:

\*/

*const* auCarre7 = *x* *=>* x \* x

*console*.log('aucarre1(3) = ' , auCarre1(3))

*console*.log('aucarre2(3) = ', auCarre2(3))

*console*.log('aucarre3(3) = ', auCarre3(3))

*console*.log('aucarre4(3) = ', auCarre4(3))

*console*.log('aucarre5(3) = ', auCarre5(3))

*console*.log('aucarre6(3) = ', auCarre6(3))

*console*.log('aucarre7(3) = ', auCarre7(3))

//https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Fonctions/Fonctions\_fl%C3%A9ch%C3%A9es

Templates

Les « **templates** » fournissent une manière plus simple de construire les strings (avec le caractère accent grave **`**) en permettant de l’intégration de variables avec **${mavariable}.**

let fruit = {nom: 'tomate', couleur: 'rouge'};

let description= `La ${fruit.nom} est ${fruit.couleur}`;

console.log(description); // Log 'La tomate est rouge'

Aussi, ` permet d’écrire des strings multilignes.

/\* Un autre exemple avec les chaînes de caractères \*/

// ES5

*const* entete = *function* (*monTitre*) {

return `Collège de Maisonneuve: ${monTitre}`

}

*console*.log(entete('Programmation d\'animation')) //

// ES6

*const* enteteES6 = *monTitre* *=>* ` Collège de Maisonneuve: ${monTitre}`

*console*.log(enteteES6('Programmation d\'animation')) //

La boucle for of

L'**instruction for...of** permet de créer une boucle [Array](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array" \o "L'objet global Array est utilisé pour créer des tableaux. Les tableaux sont des objets de haut-niveau (en termes de complexité homme-machine) semblables à des listes.) qui parcourt un [objet itérable](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Les_protocoles_iteration#Le_protocole_.C2.AB_it.C3.A9rable_.C2.BB) (ce qui inclut les objets [Array](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array" \o "L'objet global Array est utilisé pour créer des tableaux. Les tableaux sont des objets de haut-niveau (en termes de complexité homme-machine) semblables à des listes.), [Map](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Map" \o "L'objet Map représente un dictionnaire, autrement dit une carte de clés/valeurs. N'importe quelle valeur valable en JavaScript (que ce soit les objets ou les valeurs de types primitifs) peut être utilisée comme clé ou comme valeur.), [Set](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Set), [String](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/String), [TypedArray](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/TypedArray" \o "Un objet TypedArray décrit une vue organisée à la façon d'un tableau pour manipuler un tampon (buffer) de données binaires. TypedArray n'est pas une propriété globale, il n'existe pas non plus de constructeur TypedArray.  En revanche, plusieurs propriétés globales existent et leurs valeurs permettent de construire des tableaux typés (typed arrays) avec différents types de données. Ceux-ci sont listés ci-après. Les pages suivantes permettent de décrire les propriétés et méthodes qui peuvent être utilisées sur les différents tableaux typés.), l'objet [arguments](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Fonctions/arguments), etc.) et qui permet d'exécuter une ou plusieurs instructions pour la valeur de chaque propriété.

for (*variable* of iterable)

*instruction*

**variable**

À chaque itération, la valeur d'une propriété différente est affectée à *variable* (cette variable peut être déclarée avec const, let ou var).

**iterable**

L'objet dont on parcourt les propriétés énumérables.

**instruction**

Une instruction à exécuter pour chaque propriété, cette instruction peut être composée de plusieurs instructions en utilisant un [bloc](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/bloc) d'instructions.

Exemples d’utilisations

let tableauItérable = [1, 2, 3];

for (let valeur of tableauItérable) {

console.log(valeur);

}

// 1

// 2

// 3

let tableauItérable = [1, 2, 3];

for (const valeur of tableauItérable) {

console.log(valeur);

}

// 1

// 2

// 3

let iterable = 'pixel';

for (let valeur of iterable) {

console.log(valeur);

}

// p

// i

// x

// e

// l

Parcourir des collections DOM[Section](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/for...of#Parcourir_des_collections_DOM)

Il est possible de parcourir des collections DOM telles que [NodeList](https://developer.mozilla.org/fr/docs/Web/API/NodeList" \o "Les objets NodeList sont des collections de nœuds comme celles retournées par Node.childNodes et la méthode document.querySelectorAll().). Dans cet exemple, on ajoute une classe read aux paragraphes qui sont des descendants directs d'un article :

// Note : Cela ne fonctionnera que pour les plates-formes

// qui implémentent NodeList.prototype[Symbol.iterator]

let articleParagraphs = document.querySelectorAll("article > p");

for (let paragraph of articleParagraphs) {

paragraph.classList.add("read");

}

Objet littéral

objet littéral. Un littéral d'objet JavaScript est une liste de paires nom-valeur séparées par des virgules, entourées d'accolades. Les littéraux d'objet encapsulent des données, les enfermant dans un paquet ordonné. Cela minimise l'utilisation de variables globales qui peuvent causer des problèmes lors de la combinaison de code.

Syntaxe

Syntaxe littérale d'objet Les littéraux d'objet sont définis à l'aide des règles de syntaxe suivantes: Deux points séparent le nom de la propriété et de la valeur. Une virgule sépare chaque paire nom-valeur de la suivante. Il ne devrait pas y avoir de virgule après la dernière paire nom-valeur. Si l'une des règles de syntaxe est violée, telle qu'une virgule, deux points ou une accolade manquante, une erreur JavaScript sera déclenchée.

Référence :

<https://www.w3schools.com/js/js_objects.asp>

<https://www.dyn-web.com/tutorials/object-literal/>

///// Objet litéral ///////////////////////////////

*const* profil = {

nom : "Martin",

prenom : "Eddy",

adresse : "3800 Sherbrooke Est",

Ville : "Montréal",

Province : "Québec",

Pays : "Canada"

}

// on peut faire afficher l'objet dans la console

*console*.log(profil)

// on peut lire / modifier une valeur d'une propriété

*console*.log(profil.nom)

profil.nom = "Lafleur"

*console*.log(profil.nom)

// On peut aussi accéder un objet javaScript

// comme si on accédait un tableau associatif PHP

profil['nom'] = "EncoreMartin"

*console*.log(profil.nom)

*console*.log(profil['nom'])

La boucle for … in

L'**instruction for...in** permet d'itérer sur les [propriétés énumérables](https://developer.mozilla.org/fr/docs/Web/JavaScript/Caract%C3%A8re_%C3%A9num%C3%A9rable_des_propri%C3%A9t%C3%A9s_et_rattachement) d'un objet qui ne sont pas [des symboles](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Symbol). Pour chaque propriété obtenue, on exécute une instruction (ou plusieurs grâce à un [bloc](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/bloc) d'instructions).

// On peut parcourir l'ensemble des valeurs des propriétés de l'objet

for (*const* propriete in profil){

*console*.log(propriete ,' = ', profil[propriete])

}

}

Tableau d’objet littéral

/\* Tableau d'objet litteral \*/

*const* tabDeProfil = [{

nom: "Martin",

prenom: "Eddy",

adresse: "3800 Sherbrooke Est",

Ville: "Montréal",

Province: "Québec",

Pays: "Canada"

},

{

nom: "Marchand",

prenom: "Xavier",

adresse: "2030 boul Pie-IX",

Ville: "Montréal",

Province: "Québec",

Pays: "Canada"

},

{

nom: "Masse",

prenom: "Johanne",

adresse: "23 Sherbrooke Est",

Ville: "Montréal",

Province: "Québec",

Pays: "Canada"

}

]

Parcourir un tableau d’objet littéral avec une boucle (for … of)

*console*.log(tabDeProfil)

for(*const* obj of tabDeProfil)

{

*console*.log(obj.prenom, obj.nom)

}

Parcourir avec une boucle (for … of) et une boucle (for … in)

for (*const* obj of tabDeProfil) {

for (propriete in obj){

*console*.log(propriete, ' = ', profil[propriete])

}

}

Format JSON

Un fichier JSON est un fichier qui stocke des structures de données simples et des objets au format JavaScript Object Notation (JSON), qui est un format d'échange de données standard. Il est principalement utilisé pour la transmission de données entre une application Web et un serveur. Les fichiers JSON sont légers, basés sur le texte, lisibles par l'homme et peuvent être édités à l'aide d'un éditeur de texte.

Le format JSON était à l'origine basé sur un sous-ensemble de JavaScript, mais est considéré comme un format indépendant du langage, pris en charge par de nombreuses API de programmation différentes. JSON est couramment utilisé dans la programmation d'applications Web Ajax. Il devient de plus en plus populaire comme alternative au XML.

Exemple d’un contenu JSON

*const* objJSON = {

"squadName": "Super hero squad",

"homeTown": "Metro City",

"formed": 2016,

"secretBase": "Super tower",

"active": true,

"members": [

{

"name": "Molecule Man",

"age": 29,

"secretIdentity": "Dan Jukes",

"powers": [

"Radiation resistance",

"Turning tiny",

"Radiation blast"

]

},

{

"name": "Madame Uppercut",

"age": 39,

"secretIdentity": "Jane Wilson",

"powers": [

"Million tonne punch",

"Damage resistance",

"Superhuman reflexes"

]

},

{

"name": "Eternal Flame",

"age": 1000000,

"secretIdentity": "Unknown",

"powers": [

"Immortality",

"Heat Immunity",

"Inferno",

"Teleportation",

"Interdimensional travel"

]

}

]

}

for (*const* propriete in objJSON) {

*console*.log(propriete, ' = ', objJSON[propriete])

}

*console*.log(objJSON["members"][0].powers[0])

// Radiation resistance

Références

<https://jsonformatter.curiousconcept.com/>

<https://www.w3schools.com/js/js_json_intro.asp>

<https://www.w3schools.com/js/js_json_syntax.asp>

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON>