ECMAScript 2015

Evolutions de la syntaxe de base

Block scopping

Le scopping en JavaScript

Le scopping = Où vous pourrez utiliser les variables en fonction de où vous les déclarez

Le scopping en JavaScript

- Il existe 2 scopes en JavaScript :
  - Global : visible partout dans le programme/la page
  - Function : visible partout dans la fonction

Exemple 1

```
var firstname = "Mathieu";
function showName() {
  var lastname = "Parisot";

  console.log(firstname + " " + lastname);
}
```

Exemple 2

```
function showName() {
   firstname = "Mathieu";
   var lastname = "Parisot";

   console.log(firstname + " " + lastname);
}
```

Exemple 3

```
var firstname = "Mathieu";
function showName() {
  if(true) {
    var lastname = "Parisot";
  }

  console.log(firstname + " " + lastname);
}
```

hoisting

- Toutes les déclarations de variables et de fonctions dont regroupées au début du scope par l'interpreteur javaScript
- C'est ce qu'on appel l'hoisting
- C'est une source de bug potentiels
- L'instanciation se fait bien au bon endroit

Exemple 1

```
var firstname = "Mathieu";
function showName() {
  var lastname;
  if(true) {
    lastname = "Parisot";
  }

  console.log(firstname + " " + lastname);
}
```

Bugs potentiels avec l'hositing

```
var firstname = "Mathieu";
function showName() {
  console.log(firstname + " " + lastname);
  if(true) {
    var firstname = "Thomas";
    var lastname = "Parisot";
  }

  console.log(firstname + " " + lastname);
}
```

Nouveau scope en ES2015

- En ES2015, il est possible de déclarer des variables dont la visibilité est le bloc courant
- Un bloc est défini par des accolades :
  - Fonctions
  - Tests if/else
  - Boucles
  - Bloc autonome

- Pour définir une variable de scope bloc, on utilise le mot clef **let** à la place de **var**

Exemple

```
var firstname = "Mathieu";
function showName() {
  if(true) {
    let lastname = "Parisot";
  }

  console.log(firstname + " " + lastname);
}
```

Uncaught ReferenceError: lastname is not defined

Constantes

Définir une constante

- Il est désormais possible de définir des constantes avec **const**
- Il ne sera pas possible de redéfinir la valeur
- Attention : les constantes ne sont pas immutables
- Les constantes sont bloc scoppées

```
const aConstant = "test";
aConstant = "test2"; //invalid

const aConstObj = {};
aConstObj.prop = "test"; // valid
```

Spreading

Le spreading

- Permet d'éclater un tableau en une liste d'arguments avec l'opérateur '…' :

```
var myArray = ['val1','val2','val3'];

function myFunction(val1,val2,val3) { ... }

myFunction(myArray); // val1=['val1','val2','val3'],
val2=undefined, val3=undefined

myFunction(...myArray); // val1='val1', val2='val2',
val3='val3'
```

Destructuring

Destructuring de tableau

- Le destructuring permet d’assigner un tableau à plusieurs variables directement

Avant / Après

```
var myArray = ['val1','val2','val3'];
```

Le code

```
var val1 = myArray[0];
var val2 = myArray[1];
var val3 = myArray[2];
```

devient

```
var [val1, val2, val3] = myArray;
```

Récupérer le reste du tableau

- En utilisant l'operateur '…' il est possible de récupérer les autres valeurs du tableau dans un autre tableau :

```
var myArray = ['val1','val2','val3','val4','val5'];

var [val1, val2,val3, ...values] = myArray;
```

Destructuring d’objets

- Cela fonctionne aussi avec les objets en utilisant {} à la place de []

Avant / après

```
var myObject = {
  prop1: 'value1',
  prop2: 'value2',
  prop3: 5,
  prop4: 'ignored'
};
```

Le code

```
var prop1 = myObject.prop1;
var prop2 = myObject.prop2;
var prop3 = myObject.prop3;
```

devient

```
var {prop1, prop2, prop3} = myObject;
```

Changer le nom de la variable

- Il est possible de changer le nom de la variable pour qu'elle ne match pas la propriété

```
var {prop1:val1, prop2:val2, prop3:val3} = myObject;
```

Assigner des valeurs par défaut

- Il est possible d'assigner des valeurs par défauts lors de l'éclatement :

```
var {prop1, prop2, prop3, prop4="val4"} = myObject;
```

String templates

Les types de chaines en JavaScript

- Les chaines de caractères peuvent être entourées de simples ou doubles quotes en JavaScript
- La seule différence entre les deux est qu'il faut echappé les simples et doubles quotes dans la chaines :

```
"all is fine"
"it's still fine"
"He said \"fine!\""

'all is fine'
'it\'s not so fine'
'He said "fine!"'
```

Les string templates

- Un nouveau type de chaine apparait en ES2015 : les chaines interpolées
- Elles sont entourées de backquotes ``
- Il est possible d'utiliser du javascript à l'intérieur en l'entourant de ${}
- Les string templates peuvent aussi être multilignes (attention les espaces sont conservés)

Avant / après

Le code

```
var test = "Hello Mr" + name + ", how are
you?\nGreat he said!";
```

Devient

```
var test = `Hello Mr ${name}, how are you?
            Great he said!`;
```

Un exemple plus concret ?

```
var url = 'http://' + host + ':' + port + '/' +
context;
```

Devient

```
var url = `http://${host}:${port}/${context}`;
```

Les tags

- Il est possible d'accoler des « tags » aux string templates
- Ces tags sont des fonctions qui vont modifier le template

```
var test = myTag`The template ${value1} and
${value2}`;
```

Un tag se définit avec la signature suivante :

```
function myTag(strings, value1, value2, ..., valueN)
{
    // strings = ['The template',' and ']
}
```

Boucle for…of

Anciens types de boucle

```
for(var i = 0; i < length; i++) {}
```

et

```
for(var prop in obj) {}
```

Nouvelle boucle for…of

- Permet de boucler sur tous les itérables avec la syntaxe :

```
for(let val of iterable) {}
```

Nous verrons les iterables en détail plus tard mais les chaines, Maps, Sets, Arrays sont itérables et fonctionnent avec la nouvelle boucle.

Nouveaux types

Map

- Il est maintenant possible de créer des Maps de clef/valeur :

```
let map = new Map();

map.set('key','value');
map.get('key');
map.delete('key');
map.has('key');
```

La clef peut être ce que vous voulez : une chaine, un objet, une fonction, un symbol, etc.

Map initialisation rapide

```
const map = new Map([
   [ 1, 'one' ],
   [ 2, 'two' ],
   [ 3, 'three' ]
]);
```

ou

```
const map = new Map()
   .set(1, 'one')
   .set(2, 'two')
   .set(3, 'three');
```

Manipulations globales

- Il est possible de connaître le nombre d'éléments avec :

```
map.size
```

- De vider une map avec :

```
map.clear();
```

- Et de récupérer les clefs, valeurs et couples clefs/valeurs avec :

```
map.keys();

map.values();

map.entries();
```

Convertir une Map en tableau

- Grâce au destructuring il est possible de convertir une Map en tableau :

```
var myArray = [...map];
```

Cela permet d'utiliser les fonctions comme map, filter, find, etc.

TP Map

- Créer une map contenant ??
- Filter la map pour obtenir une autre map avec uniquement ??
- Filter la map pour obtenir une autre map avec uniquement ??
- Combiner ces deux maps ensemble

WeakMap

- Il existe également des weak maps
- Les clefs sont forcément des objets
- Il n'est pas possible d'itérer sur une weak map, seul get(key) fonctionne pour avoir la valeur
- Il n'est pas possible d'utiliser clear()
- Si seule la WeakMap possède une référence sur la clef, elle sera Garbage collectée

```
var wk = new WeakMap();
```

Les sets

- Un Set contient une liste de valeurs arbitraires et permet de les manipuler rapidement :

```
let set = new Set();

set.add('val1');

set.has('val1');

set.delete('val1');
```

Initialisation rapide

```
let set = new Set(['val1','val2']);
```

ou

```
let set = new Set().add('val1').add('val2');
```

Manipulations globales

- Il est possible de connaître le nombre d'éléments avec :

```
set.size
```

- Et de vider une map avec :

```
set.clear();
```

Convertir un Set en tableau

- Grâce au destructuring il est possible de convertir un Set en tableau :

```
var myArray = [...set];
```

Cela permet d'utiliser les fonctions comme map, filter, find, etc.

TP Map

- Créer une set contenant ??
- Filter le set pour obtenir une autre map avec uniquement ??
- Filter le set pour obtenir une autre map avec uniquement ??
- Combiner ces deux sets ensemble

WeakSet

- Idem que WeakMap sauf que ça concerne les valeurs du set

Typed Arrays

- Il est possible de créer des tableaux typés
- Ils fonctionnent comme des tableaux classiques mais sont limités à un seul type
- Les types supportés sont : int8, Uint8, Uint8C, int16, Uint16, int32, Uint32, Float32, Float64

```
const uint8 = new Uint8Array(1);
const int8 = new Int8Array(1);
```

Changements des APIs de base

Math 1/2

- Nouveaux litéraux :

```
let two = 0b10; // binaire
let height = 0o10; // octal
```

- Vérifier le signe :

```
Math.sign(-15); // -1
Math.sign(0); // 0
Math.sign(4); // 1
```

- Tronquer les décimales :

```
Math.trunc(4.9); // 4
```

Math 2/2

- Logarithme base 10 :

```
Math.log10(100); // 2
```

- Calcul de l'hypoténuse :

```
Math.hypot(3,4); // 5
```

Number 1/2

- Check si un nombre est fini ou non

```
Number.isFinite(Infinity); // false
Number.isFinite(-Infinity); // false
Number.isFinite(NaN); // false
Number.isFinite(123); // true
```

- Check si un nombre n'est pas un nombre

```
Number.isNaN('???'); // true
Number.isNaN(123); // false
```

- Check si un nombre est entier

```
Number.isInteger(1.3); // false
Number.isInteger(1); // true
```

Number 2/2

- Parsing de nombres

```
Number.parseFloat('2.3');
Number.parseInt('123',10);
```

Chaines de caractères

- Vérifier si une chaine commence par une autre

```
'hello'.startsWith('hell'); // true
```

- Vérifier si une chaine finie par une autre

```
'hello'.endsWith('ello'); // true
```

- Vérifier si une chaine en inclue une autre

```
'hello'.includes('ell'); //true
```

- Répéter une chaine

```
'hello '.repeat(3); // 'hello hello hello '
```

Arrays - conversions

- Convertir un simili tableau (un objet avec une propriété length et indexé) en Array :

```
Array.from(arrayLike);
```

➔ Très pratique pour convertir le résultat de document.getElementById

- Mapping automatique lors de la conversion :

```
Array.from(arrayLike, mapFunction, thisObj);
```

- Convertir une liste de paramètres en tableaux :

```
Array.of(val1,val2,...,valN);
```

➔Utilisez les tableaux littéraux en priorité

Arrays - iterating

- Récupérer les clefs

```
Array.from(['a', 'b'].keys()) // [ 0, 1 ]
```

- Récupérer les valeurs

```
Array.from(['a', 'b'].values()) // [ 'a', 'b' ]
```

- Récupérer les couples clefs/valeurs

```
Array.from(['a', 'b'].entries())
 // [ [ 0, 'a' ],
      [ 1, 'b' ] ]
```

Arrays - searching

- Il est maintenant possible de faire des recherches avancées avec :

```
Array.prototype.find(predicate, thisArg?)
Array.prototype.findIndex(predicate, thisArg?)
```

Le prédicat est une fonction qui renvoit true lorsqu'il trouve l'occurrence cherchée :

```
let myArray = ['val1', 'val2', 'val3'];

myArray.find(function(value) {
  return value == 'val1';
});

myArray.findIndex(function(value) {
  return value == 'val1';
});
```

Arrays - misc

- Copier des éléments du tableau à un autre index :

```
Array.prototype.copyWithin(target, start, end)
```

```
const arr = [0,1,2,3];
arr.copyWithin(2, 0, 2); // [ 0, 1, 0, 1 ]
```

- Remplir un tableau

```
Array.prototype.fill(value, start, end)
```

```
['a', 'b', 'c'].fill(7); // [ 7, 7, 7 ]
['a', 'b', 'c'].fill(7, 1, 2); // [ 'a', 7, 'c' ]
```