JavaScript, syntaxe ES6 (ES2015)

Alvin Berthelot

Version 1.0.0

Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons.

Attribution - Partage dans les Mêmes Conditions 3.0 non transposé.

La licence, ses explications ainsi que les moyens de contribution et réappropriation sont détaillés à la fin.

Utiliser ES6

Compatibilité des navigateurs

ES6 (ou ES2015) est désormais un standard, pour autant cela ne veut pas dire qu'il est disponible dans les anciens navigateurs.

Cependant il est fortement conseillé de l'utiliser dès à présent pour plusieurs raisons :

- Bon nombre d'implémentation de fonctionnalités ont été réalisées avant la sortie officielle.
- Il existe de nombreux polyfills pour palier les manques des navigateurs les plus anciens.
- Les nouvelles fonctionnalités de la nouvelle syntaxe sont principalement du sucre syntaxique, c'est à dire que dans le fond le fonctionnement est resté le même mais avec une syntaxe beaucoup plus facile.

Les transpileurs

L'intérêt des transpileurs est de pouvoir écrire du code JavaScript selon les dernières syntaxes tout en assurant la compatibilité avec les anciennes.

Cela permet d'utiliser tout de suite la syntaxe plus légère, élégante et robuste puisqu'elle est en accord avec les standards actuels ou futurs.

Les transpileurs les plus connus :

- Babel
- TypeScript

La portée des variables

`let`

Rappel

Contrairement à d'autres langages, JavaScript n'a pas de "scope" au niveau des blocs (c'est à dire entre {}) mais a **un scope au niveau des fonctions**.

La portée des variables par fonction et non par bloc induit en erreur.

Il sera possible de définir une portée par bloc (plus limitée) sur les variables, il suffit de remplacer la déclaration `var` par `let`.

```
function substract(a, b) {
  if(true) {
    let baseSubstract = 100;
  }
  return baseSubstract - a - b;
};

console.log(substract(1, 2)); // ERROR
```

const

Il est possible de définir des constantes dans le code, il suffit de remplacer la déclaration `var` par `const`.

Une nouvelle valorisation n'est pas possible.

```
const magicNumber = 42;
magicNumber++; // ERROR
```

L'initialisation d'une constante est possible pour un objet, mais elle ne garantie pas ses propriétés et ses méthodes.

```
const animal = {
  name: 'Grumpy',
  age: 3,
  male: true
};
animal.name = 'Droopy';
console.log(animal.name); // Droopy
```

Les template de string

Nouvelle déclaration

Les `string` peuvent être déclarées avec des "backticks" `` ` `` au lieu des apostrophes ou guillemets habituels.

Cela a 2 avantages :

- On peut utiliser des variables avec `${variable}` sans utiliser l'opérateur de concaténation `+`.
- On peut aller à la ligne.

```
var cat = {
  name: 'Grumpy'
};
console.log(`
Hello ${cat.name}
Would you like a cup of tea ?
`);
```

Les "arrow functions"

Nouvelle syntaxe pour les fonctions

Il est désormais très simple d'écrire des fonctions sans écrire les mots `function` et `return`. Pour indiquer la déclaration d'une fonction on utilise l'opérateur ⇒ nommé "fat arrow".

```
var sayGoodbye = function(name) {
  console.log('Goodbye ' + name);
};
// ES2015 syntaxe
var sayGoodbye = (name) => {
  console.log('Goodbye ' + name);
};
```

A noter que si il n'y a pas de paramètre on utilise des parenthèses vides et que si l'instruction `return` est impliciteil n'y a pas de bloc.

```
var sayHello = function() { return 'Hello'; };
// ES2015 syntaxe
var sayHello = () => 'Hello';
```

La perte du `this`

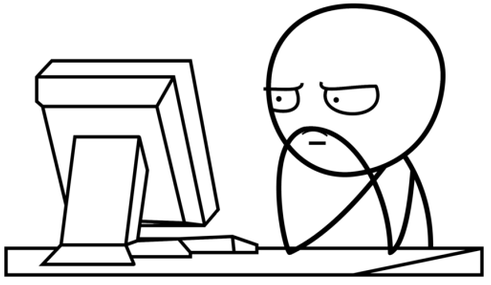
Une grande classique en JavaScript est de perdre la référence du `this`.

```
var runningMan = {
  countRunning: 0,
  run: function() {
    this.countRunning++;
    console.log(this.countRunning);
    setTimeout(function() {
      console.log('Run ' + this.countRunning + ' finished');
    }, this.countRunning * 1000);
  }
};
// workaround
var runningMan = {
  countRunning: 0,
  run: function() {
    this.countRunning++;
    console.log(this.countRunning);
    var that = this;
    setTimeout(function() {
      console.log('Run ' + that.countRunning + ' finished');
    }, this.countRunning * 1000);
  }
};
```

Le "bind" du `this`

Les "fat arrow" réalisent le "bind" de manière automatique.

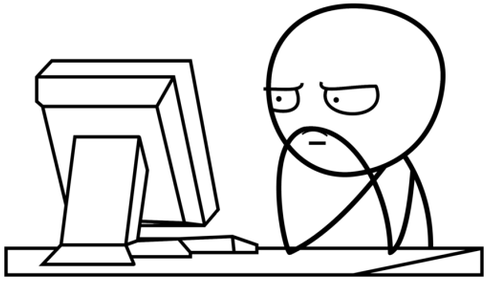
```
var runningMan = {
  countRunning: 0,
  run: function() {
    this.countRunning++;
    console.log(this.countRunning);
    setTimeout(() => {
      console.log('Run ' + this.countRunning + ' finished');
    }, this.countRunning * 1000);
  }
};
```



Exercices

On va s'appuyer sur la plateforme NodeSchool.io, pour cela vous devez installer le TP "Tower of babel".

⇒ BABEL SETUP



Exercices

On va s'appuyer sur la plateforme NodeSchool.io, pour cela vous devez installer le TP "Count to 6".

⇒ HELLO ES6
⇒ TEMPLATE STRINGS
⇒ ARROW FUNCTIONS, Part 1
⇒ ARROW FUNCTIONS AND THIS

Affectations déstructurées

Affectations déstructurées (objets)

Il s'agit d'une nouvelle syntaxe pour affecter des variables via des objets;

```
let cat = { name: 'Grumpy', age: 3, male: true };
// syntaxe classique
let famousCat = cat.name;
// syntaxe déstructurée
let { name: famousCat } = cat;
```

L'ordre des clés et des valeurs est inversée.

Si la variable à déclarer a le même nom que la propriété à lire elle peut être omise.

```
let { name } = cat;
console.log(name); // Grumpy
```

Affectations déstructurées (tableaux)

Cette nouvelle syntaxe s'applique également aux tableaux.

```
let animals = ['Grumpy', 'Droopy', 'Nemo'];

// syntaxe classique
let famousDog = animals[1];

// syntaxe déstructurée
let [, famousDog,] = animals;
```

Syntaxe de décomposition

Une syntaxe, 2 opérateurs

La nouvelle syntaxe de décomposition est utilisée avec `...` mais celle-ci peut avoir une signification différente car elle est utilisée avec 2 opérateurs :

- L'opérateur "spread".
- L'opérateur "rest".

L'opérateur *spread*

La syntaxe de décomposition "spread operator" est utilisée pour déstructurer un tableau sans forcément connaître le nombre d'éléments.

```
let ages = [5, 13, 7, 32, 21];
let max = Math.max(...ages);
```

Pour rappel `Math.max()` prend une liste d'arguments et non un tableau (Cf. MDN).

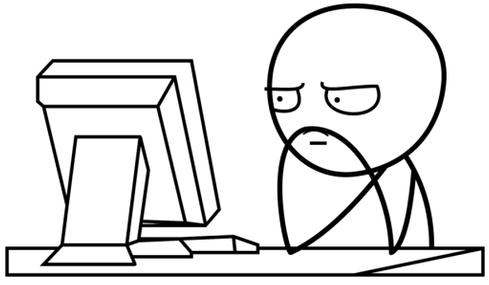
L'opérateur *rest*

La syntaxe de décomposition "rest operator" est utilisée pour construire implicitement un tableau avec des arguments restants.

```
function countAnimals(...args) {
   return args.length;
}
countAnimals('Grumpy', 'Droopy', 'Nemo');
```

Quizz spread Vs rest

```
let [first, ...others] = ['Grumpy', 'Droopy', 'Nemo']

console.log(first); // QUIZZ 1 ?

console.log(others); // QUIZZ 2 ?

// QUIZZ 3, spread or rest ?

let animals = [...others, first];

console.log(animals); // QUIZZ 4 ?

// QUIZZ 5, spread or rest ?
```



Exercices

On va s'appuyer sur la plateforme NodeSchool.io, pour cela vous devez installer le TP "Count to 6".

⇒ DESTRUCTURING
⇒ SPREAD
⇒ REST

POO avec ES2015

Héritage prototypal

Précédemment nous avons vu comment fonctionnait l'héritage prototypal en JavaScript. Cela était rendu possible en appliquant un certains nombres de concepts, ce n'était pas toujours trivial de mettre ça en oeuvre et au mieux c'était assez verbeux.

Désormais JavaScript supporte le mot clé `class` qui va permettre de faire la même chose mais de manière beaucoup plus simple !

```
class MyObject {
  constructor(myProperty) {
    this.myProperty = myProperty;
  }
  myFunction() {
    return `My property is ${this.myProperty}`;
  }
}

let anotherObject = new MyObject('great');
console.log(anotherObject.myFunction());
```

Utilisation de `class`

L'utilisation de `class` est plus simple, il y a tout de même des notions à connaître :

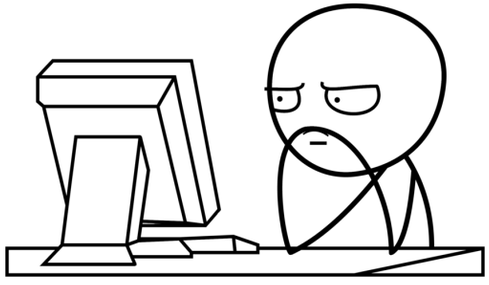
- Une `class` dispose généralement d'une fonction `constructor` qui est la fonction permettant d'instancier un objet. Elle ne peut en avoir qu'une seule.
- On déclare les méthodes d'une `class` directement dans celle-ci et une méthode peut être déclarée comme statique avec le mot clé `static`.
- Il est possible de définir des méthodes `get` et `set` sur une propriété pour forcer un comportement lorsque l'on travaille directement sur la propriété.
- Contrairement à du code JavaScript classique, une classe doit obligatoirement être déclarée avant d'être utilisée.
- L'héritage est rendu possible avec le mot clé `extends` et on peut faire appel à une class parente avec le mot clé `super`.

Rappel, héritage au final en ES5

```
function Animal(name, age, male) {
  this.name = name;
  this.age = age;
  this.male = male;
};

function Cat(name, age, male, color) {
  Animal.apply(this, [name, age, male]);
  this.color = color;
};
Cat.prototype = Object.create(Animal.prototype, {
  constructor: {value: Cat}
});
Cat.prototype.action = function() {
  return 'meaows';
};

var cat = new Cat('Grumpy', 3, true, 'grey');

console.log('My ' + cat.color + ' cat ' + cat.name + ' ' + cat.action());
```

Héritage au final en ES6

```
class Animal {
  constructor(name, age, male) {
    this.name = name;
    this.age = age;
    this.male = male;
  }
}

class Cat extends Animal {
  constructor(name, age, male, color) {
    super(name, age, male);
    this.color = color;
  }
  action() {
    return 'meaows';
  }
}

var cat = new Cat('Grumpy', 3, true, 'grey');

console.log(`My {cat.color} cat ${cat.name} ${cat.action()}`);
```



Exercices

On va s'appuyer sur la plateforme NodeSchool.io, pour cela vous devez installer le TP "Tower of babel".

⇒ CLASS
⇒ CLASS EXTEND

Asynchronisme

Les callbacks

En JavaScript il est tout à fait possible de **passer une fonction en paramètre d'une autre fonction** et/ou de retourner une fonction dans une fonction.

```
setTimeout(
  // premier paramètre
  function() { console.log('Sorry I\'m late !'); },
  // deuxième paramètre
  4000
);
```

On appelle callback un appel à une fonction par une autre fonction à laquelle il a été passé en paramètre d'appel une référence à la première.

Il s'agit juste d'une convention de nommage sur l'appel de fonctions car rien de particulier n'est effectué.

Asynchronisme en JavaScript

Les callbacks sont très utilisés pour les traitements asynchrones et pour déclencher un traitement sur la réception d'un événement.

```
// Asynchronous with jQuery style
let showAnimals = function() {
  $.get('/api/animals', function(response) {
    response.map(animal => console.log(animal));
  });
}

// Trigger event in Vanilla style
let myButton = document.getElementById('buttonSearch');
myButton.addEventListener('click', showAnimals);
```

On associe un callback nécessairement à un traitement asynchrone, ce n'est pas forcément vrai.

Le callback hell

Si ce mode de fonctionnement est puissant, il peut aussi vite devenir problématique pour la lisibilité et maintenabilité du code.

```
async1(function(){
  async2(function(){
    async3(function(){
      async4(function(){
        ...
      });
    });
  });
});
```

On appelle cela le callback hell, une solution avec ES2015 est d'utiliser les `promises`.

Les `promises`

Les `promises` (promesses en français) permettent de simplifier la programmation asynchrone en affichant le code "à plat".

```
let task1 = async1();
let task2 = task1.then(async2);
let task3 = task2.then(async3);
let task4 = task3.then(async4);
...
```

Il est même possible de chaîner les `promises` pour gagner encore en lisibilité.

```
async1()
  .then(async2)
  .then(async3)
  .then(async4)
  ...
```

Fonctionnement d'une "promise"

Une "promise" est un objet qui possède une méthode `then` prenant 2 paramètres, un callback de succès et un callback d'erreur. Les callbacks sont déterminés en fonction des 3 états d'une "promise" :

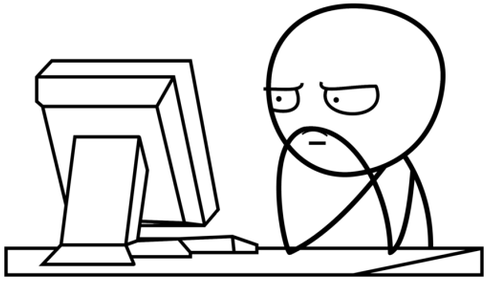
- *pending* lorsque la promise est en cours d'exécution.
- *fulfilled* lorsque la promise s'est exécutée avec succés, on appelle alors le callback de succès.
- *rejected* lorsque la promise n'a pas pu s'exécuter correctement, on appelle alors le callback d'erreur.

```
async1()
  .then(async2, console.error)
  .then(async3, console.error)
  .then(async4, console.error)
  ...
```

Créer sa `promise`

Si dans de nombreux cas on utilisera des "promises" déjà fournies dans des librairies, rien ne vous empêche de créer la votre avec la `class Promise`.

```
let stupidPromise = new Promise(function (resolve, reject) {
  if(true) {
    resolve('OF COURSE');
  } else {
    reject('WTF !!!');
  }
});

stupidPromise.then(console.log, console.log);
```



Exercices

On va s'appuyer sur la plateforme NodeSchool.io, pour cela vous devez installer le TP "Promise It Won't Hurt".

⇒ Warm up
⇒ Fulfill a promise
⇒ Reject a promise
⇒ To reject or not to reject
⇒ Always asynchronous
⇒ Shortcuts
⇒ Promise after promise

Licence

CC BY-SA 3.0

Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons. Attribution - Partage dans les Mêmes Conditions 3.0 non transposé.

Copyright © 2017 Alvin Berthelot.

Pour toutes questions, réclamations ou remarques, merci d'envoyer un message à alvin.berthelot@webyousoon.com.

Explications licence CC BY-SA 3.0

Cette licence permet aux autres de remixer, arranger, et adapter votre œuvre, même à des fins commerciales, tant qu'on vous accorde le mérite en citant votre nom et qu'on diffuse les nouvelles créations selon des conditions identiques.

Cette licence est souvent comparée aux licences de logiciels libres, "open source" ou "copyleft".

Toutes les nouvelles œuvres basées sur les vôtres auront la même licence, et toute œuvre dérivée pourra être utilisée même à des fins commerciales.

C'est la licence utilisée par Wikipédia ; elle est recommandée pour des œuvres qui pourraient bénéficier de l'incorporation de contenu depuis Wikipédia et d'autres projets sous licence similaire.

Contribution et réappropriation

Ce fichier PDF est généré avec Asciidoctor à partir d'un dépôt Git se trouvant sous GitHub.

https://github.com/alvinberthelot/slides-js

Cela signifie que vous n'avez pas besoin de vous battre avec un fichier binaire (le PDF) pour **contribuer**, **vous réapproprier le contenu** ou **modifier le thème** de présentation.

Contribution

Vous voulez **contribuer au contenu** car :

- Il y a une erreur (ça arrive à tout le monde), de typographie, de compréhension, ou tout autre chose.
- Vous souhaitez apporter une précision.

Il vous suffit de contribuer au projet via Git par le moyen d'une "pull request" sur le dépôt Git.

Réappropriation

N'oubliez pas les conditions de la licence.

Vous voulez vous **réapproprier le contenu** car :

- Vous souhaitez donner un style différent.
- Vous souhaitez enlever/ajouter/modifier des sections dans votre contexte.

Il vous suffit de "forker" le dépôt Git et d'y apporter vos propres modifications, puis de générer par vous même le nouveau PDF.