Contents

1 Programmez en ECMAScript 6 aujourd'hui (Chapitre 1)

Nombreux sont les développeurs qui n'aiment pas le JavaScript (pas typé, pas de classes, modèle objet "bizarre", ...), et pourtant le futur de ce "petit" langage omniprésent est passionnant et prometteur? Et ce futur c'est **ECMAScript 6**, et vous pouvez déjà y avoir accès.

L'objectif de ce premier article est de vous mettre le pied à l'étrier et de vous faire découvrir quelques aspects d'ECMAScript 6 (mais pas tous). Nous aborderons donc les sujets suivants:

- Préparation de votre projet
- Les constantes
- Les nouveautés dans les chaînes de caractères
- Les objets "littéraux"
- Les classes
- Les "Arrow Functions"
- Les promises

Dans un 1er temps, Nous allons voir comment développer en ECMAScript 6 "côté navigateur". Il y a plusieurs façons de s'y prendre mais le principe de base est le suivant:

1. Vous codez avec une syntaxe ES6 *(le petit nom d'ECMAScript 6)*
2. Vous "transpilez" *(une sorte de compilation script à script)* votre code ES6 vers du code JavaScript 5 ou 3 pour qu'il soit compréhensible (exécutable) par votre navigateur

Ensuite vous avez deux sortes de "transpilations":

- la transpilation **"online/in-browser"**: votre navigateur va se charger (grâce à un script de transpilation) de transpiler votre code à la volée (au chargement). C'est la méthode la moins optimisée, mais la plus simple d'un point de vue apprentissage, et donc celle que nous retiendrons.
- la transpilation **"build step/offline"**: avant de publier vos scripts et pages sur le web, vous transpilez "à l'avance" à l'aide d'un "transpileur" vos codes ES6.

Le transpileur ES6 que nous utiliserons est **Traceur**: `https://github.com/google/traceur-compiler`.

Si vous souhaitez l'utiliser en mode "offline", il faudra installer le projet et le builder pour disposer en mode commande du transpileur, ou encore plus simplement, en mode commande, utilisez: `npm install traceur -g` (j'insiste sur l'option **-g** qui permet d'installer **Traceur** de manière globale, donc accessible de n'importe où).

Si vous souhaitez l'utiliser en mode "offline", nous installerons **Traceur** de manière locale à votre répertoire projet avec la commande `npm install traceur` (donc sans l'option **-g**).

Les pré-requis sont donc d'avoir **Nodejs** et **npm** installés. Vous aurez aussi besoin d'un serveur http pour que cela fonctionne (nous verrons plus loin pourquoi). Le plus simple est d'installer l'utilitaire **http-server** `https://github.com/nodeapps/http-server` d'une simple commande `npm install http-server -g`.

1.1 Préparons notre projet

Dans un répertoire de votre choix, à partir d'un terminal (ou une console selon votre OS), tapez donc la commande `npm install traceur`. Le gestionnaire de packages **npm** va aller télécharger les dépendances nécessaires et les installer dans un sous-répertoire de votre projet appelé `node_modules`.

Ensuite à la racine de votre projet, créez un sous-répertoire `js` (vous pouvez tout à fait le nommer autrement), avec à l'intérieur un fichier `main.js` avec par exemple le code suivant:

```
console.log("=== Hello World! ===");
```

C'est dans ce fichier `main.js` que nous saisirons nos codes d'exemples par la suite.

Nous avons ensuite besoin de charger (et transpiler) ce fichier `main.js` à partir d'une page html (que nous appelerons `index.html`). Nous créons donc ce fichier `index.html` avec le code suivant:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>es6-project</title>
</head>

<body>

  <script src="node_modules/traceur/bin/traceur.js"></script>
  <script>
    traceur.options.experimental = true;
  </script>
```

```
  <script>
    System.import('js/main').catch(function(e){
      console.log(e);
    });
  </script>
</body>
</html>
```

Notre page n'affichera rien, elle sert juste à lancer notre script, pour la suite de notre article nous utiliserons la console de notre navigateur pour afficher les éléments de sortie (les Chrome Developper Tools ou la console de FireFox). La commande `System.import('js/main')` permet de charger le fichier `main.js` *(notez que l'on ne précise pas l'extension)*. Ensuite l'utilisation de `catch(...)` permet de "capturer" les erreurs et les afficher.

Si vous vous contentez de double-clicker sur la page pour l'ouvrir (c'est à dire en mode `file:///`), votre console va vous retourner un message d'erreur de ce type:

```
XMLHttpRequest cannot load file:///.../js/main.js.
Cross origin requests are only supported for HTTP.
```

En effet `System.import('js/main')` utilise "ajax" pour charger le fichier `main.js`, vous avez donc besoin de "servir" votre page avec un serveur http pour que cela fonctionne.

Donc depuis votre répertoire, tapez: `http-server` qui va créer une instance de serveur http écoutant sur le port 8080 et ouvrez à nouveau votre page à partir du navigateur mais en utilisant cette url: `http://localhost:8080/` et vous obtiendrez dans la console de votre navigateur un "superbe":

```
=== Hello World! ===
```

Voilà! Vous disposez de tout ce qui est nécessaire pour écrire vos 1ères lignes de code ES6. Nous allons y aller pas à pas.

1.2 Les constantes

Ce n'est pas forcément ce qui m'a le plus manqué dans JavaScript, mais il faut avouer que pouvoir définir des constantes apporte de la lisibilité et de la robustesse au code. Il est donc temps d'aller dans votre fichier `main.js` et d'y saisir quelques lignes:

```
const GEEK_NAME = "k33g";
const GEEK_NUMBER = 42;

function giveMeAGeek() {
  const GEEK_NAME = "geek" + GEEK_NUMBER;
  return GEEK_NAME;
}

console.log(giveMeAGeek());
console.log(GEEK_NAME);
```

Si vous raffraichissez votre page, vous obtiendrez ceci dans votre console:

```
geek42
k33g
```

Vous remarquez que les constantes peuvent aussi être spécifiques à un scope.

Maintenant ajoutez le code ci-dessous à la fin de votre script:

```
GEEK_NAME = "Bob Morane";
```

Puis raffraichissez votre page, vous obtiendrez dans votre console un message d'erreur de ce type:

```
Error {stack: (...), message: "http://.../js/main.js:12:1: GEEK_NAME is read-only"}
```

Je ne vous ai donc pas menti, ce sont bien des constantes, ce qui à l'avenir vous évitera bien des ennuis.

Voyons maintenant quelques nouveautés bien pratiques spécifiques aux chaîne de caractères.

1.3 Les chaînes de caractères

Les chaînes de caractères héritent de méthodes utilitaires telles:

```
"Quarante-deux".contains("-");      // true
"Quarante-deux".startsWith("Q");    // true
"Quarante-deux".endsWith("x");      // true

"Quarante-deux".repeat(3);
// Quarante-deuxQuarante-deuxQuarante-deux
```

Mais la nouveauté que je préfère (pour les chaînes) est sans aucun doute **l'interpolation de chaîne**:

```
var clark = {
    firstName: "Clark"
  , lastName: "Kent"
  , superheroName: "SuperMan"
}

console.log(
  `Je suis ${clark.superheroName} aka "${clark.firstName} ${clark.lastName}"`
);
```

Vous obtiendrez:

```
Je suis SuperMan aka "Clark Kent"
```

Ou en mode multi-lignes:

```
console.log(
‘Je suis ${clark.superheroName},
et mon identité secrète est:
”${clark.firstName} ${clark.lastName}”‘
);
```

Et vous obtiendrez:

```
Je suis SuperMan,
et mon identité secrète est:
”Clark Kent”
```

Notez bien l'utilisation des **backticks (‘)** à la place des double-quotes (“) ou simple-quotes (’). Tout cela est beaucoup plus agréable que les laborieuses concaténation de chaînes, de variables et de sauts de lignes.

Maintenant il est temps d'attaquer la programmation “objet”, en commençant justement par les objets (non, pas les classes, pas tout de suite).

1.4 Les objets JavaScript

On parle aussi “d'objet littéraux”. Ils ont eux aussi gagné quelques améliorations et tout particulièrement des “raccourcis”. Comme la simplification de l'initialisation des propriétés.

1.4.1 Initialisation des propriétés

Avant, par exemple, si vous souhaitiez avoir une fonction qui “fournisse” des personnes, vous écriviez ceci:

```
function getHuman (firstName, lastName) {
  var superheroName = ”?”;
  return {
    firstName: firstName,
    lastName: lastName
  }
}
```

Maintenant, vous pouvez aller plus vite et écrire ceci:

```
function getHuman (firstName, lastName) {
  var superheroName = ”?”;
  return {
    firstName, lastName, superheroName
  }
}

var clark = getHuman(”Clark”, ”Kent”);

console.log(clark);
```

Et vous obtiendrez ceci:

```
Object {firstName: "Clark", lastName: "Kent", superheroName: "?"}
```

Le moteur JavaScript a recherché dans son "scope" courant (le corps de la fonction) des variables avec le même nom que celles de l'objet littéral. Dans le même ordre d'idée, la notation des méthodes d'un objet a elle aussi été simplifiée.

1.4.2 Définition des méthodes de l'objet

Pour ajouter une fonction `hello()` à notre objet, habituellement nous procédons comme ceci:

```
function getSuperhero (
    firstName
  , lastName
  , superheroName) {

  return {
      firstName: firstName
    , lastName: lastName
    , superheroName: superheroName
    , hello: function()  {
      return
        "Je suis "+superheroName+ "aka " +firstName +" " +lastName;
    }
  }
}
```

Avec ES6, nous pouvons nous passer du mot clé `function`, de cette manière:

```
function getSuperhero (
    firstName
  , lastName
  , superheroName) {

  return {
      firstName
    , lastName
    , superheroName
    , hello()  {
      return `Je suis ${superheroName} aka "${firstName} ${lastName}"`;
    }
  }
}

var clarkKent = getSuperhero("Clark", "Kent", "Superman");

console.log(clarkKent.hello())
```

Et vous obtiendrez bien:

```
Je suis Superman aka "Clark Kent"
```

Nous avons donc une notation plus concise qui permet de gagner en lisibilité. Dans le paragraphe suivant je vais vous parler d'une nouvelle méthode "utilitaire" ajoutée à `Object`, la méthode `assign()` qui va vous permettre de donner un côté "fonctionnel" à votre code.

1.4.3 `Object.assign()`

Cette nouvelle méthode, nous permet de faire des "mixins" entre objets, c'est à dire de greffer les propriétés et/ou méthodes d'un objet sur un autre. Voyons tout de suite comment procéder:

```
var tonyStark = {
    firstName: "Tony"
  , lastName: "Stark"
  , strength: 100
  , superheroName: "IronMan"
}

console.log(tonyStark);
```

Vous obtenez:

```
Object {firstName: "Tony", lastName: "Stark", strength: 100, superheroName: "IronMan"}
```

Ensuite je créée un autre objet que je vais "greffer" à `tonyStark` avec `Object.assign(tonyStark, autreObjet)`:

```
var armorAbilities = {
    strength: 1000
  , flight () { console.log("Flying..."); }
}

Object.assign(tonyStark, armorAbilities);

console.log(tonyStark);
```

Et vous obtenez un `tonyStark` amélioré avec de nouvelles capacités:

```
Object {
    firstName: "Tony",
    lastName: "Stark",
    strength: 1000,
    superheroName: "IronMan",
    flight: function () {}
}
```

Par contre toute modification de `armorAbilities` ne sera pas répercutée sur `tonyStark`. Pour cela il faudrait plutôt travailler par référence en créant une propriété `armorAbilities` que l'on "assigne" à `tonyStark`:

```
Object.assign(tonyStark, {armorAbilities});

Object.assign(armorAbilities, {createMagneticFields () { return []}});

console.log(tonyStark);
```

Et nous obtiendrons l'objet suivant:

```
Object {
    armorAbilities: Object {
        createMagneticFields: function () {},
        flight: function () {},
        strength: 1000,
    },
    firstName: "Tony",
    lastName: "Stark",
    strength: 100,
    superheroName: "IronMan"
}
```

Tout dépend de votre besoin à l'exécution. Mais passons à quelque chose d'encore plus "exitant", les classes!

1.5 Les classes et un début de modularisation

Même si on peut très bien s'en passer (étant donné le modèle objet de JavaScript), de nombreuses personnes reprochaient à JavaScript l'absence de classe. Et bien, c'est fait elles sont là! Pour ceux qui connaissent les classes de **CoffeeScript**, le principe est le même. Voire même, elles auraient inspiré celles d'ECMAScript 6 si l'on en croit le blog de **Brendan Heich** (le papa de JavaScript): https://brendaneich.com/tag/javascript-ecmascript-harmony-coffeescript/.

1.5.1 1ère classe

Créeons donc notre première classe. Au même niveau que `main.js`, créez un fichier `Human.js` avec le code suivant:

```
class Human {

  constructor(firstName = "John", lastName = "Doe") {
    /* Et oui les paramètres par défaut existent! */

    /* membres d'instance */
    this.firstName = firstName;
    this.lastName = lastName;
```

```
  }

  hello () {
    console.log(‘Hello, I’m ${this.firstName} ${this.lastName}‘);
  }

}

export default Human;
```

Ensuite pour utiliser notre classe `Human`, modifiez le fichier `main.js` de cette façon:

```
import Human from ’./Human’;

var john = new Human();
john.hello();

var tony = new Human(”Tony”, ”Stark”);
tony.hello();
```

Et si vous raffraichissez votre page, vous obtiendrez ceci dans la console du navigateur:

```
Hello, I’m John Doe
Hello, I’m Tony Stark
```

Les possibilités des classes évolueront sûrement, mais rien que ça est déjà très pratique pour structurer son code. **Notez bien** la façon de déclarer une référence à un fichier: `import Human from ’./Human’;` (toujours sans l'extension `.js`). Mais sachez qu'il existe déjà une autre possibilité déjà fonctionnelle: **l'héritage**!

1.5.2 L'héritage

Même s'il ne faut pas en abuser, l'héritage est utile dans certains design de code (à vous ensuite de faire la part des choses avec des solutions plus fonctionnelles comme avec les “mixins”). L'héritage se fait tout simplement avec le mot clé `extends`. Créez un nouveau fichier `SuperHero.js` (toujours au même endroit) avec le code source suivant:

```
import Human from ’./Human’;

class SuperHero extends Human {

  constructor (firstName, lastName, heroName, ability) {
    super(firstName, lastName);
    this.heroName = heroName;
    this.ability = ability;
  }
  power () {
    console.log(‘I’m ${this.heroName} and I’m ${this.ability}‘);
```

```
  }
}

export default SuperHero;
```

Notez bien l'utilisation du mot-clé `super` pour utiliser le constructeur de la classe parente. Ensuite pour utiliser notre nouvelle classe retournez dans `main.js`:

```
var clark = new SuperHero("Clark", "Kent", "SuperMan", "flying");

clark.hello();
clark.power();
```

... Tout simplement!

Une autre nouveauté est l'apparition des **Arrow Functions**, ou comment nous allons faire moins d'erreurs dans le futur.

1.6 Arrow function

Avant nous faisions ceci:

```
var bob = {
  name: "Bob Morane",
  friends : ["Sam", "Ted", "Paul"],
  helloBuddies: function() {
    this.friends.forEach(function(buddy) {
      console.log(buddy + " is a friend of " + this.name);
    });
  }
}

bob.helloBuddies();
```

Et nous obtenions ceci:

```
Sam is a friend of
Ted is a friend of
Paul is a friend of
```

au lieu de:

```
Sam is a friend of Bob Morane
Ted is a friend of Bob Morane
Paul is a friend of Bob Morane
```

Erreur classique en javascript, le `this` de `this.name` est une référence à `this.friends` (qui n'a pas de propriété `name`) et non pas à `bob`, d'où le comportement inattendu.

Jusqu'ici il fallait pallier cela en posant une référence à l'objet attendu, comme ceci par exemple:

```
var bob = {
  name: "Bob Morane",
  friends : ["Sam", "Ted", "Paul"],
  helloBuddies: function() {
    var that = this; // <- that fait référence à Bob
    this.friends.forEach(function(buddy) {
      console.log(buddy + " is a friend of " + that.name);
    });
  }
}

bob.helloBuddies();
```

Ou, l'on pouvait aussi faire un `bind(this)` sur la fonction:

```
var bob = {
  name: "Bob Morane",
  friends : ["Sam", "Ted", "Paul"],
  helloBuddies: function() {
    this.friends.forEach(function(buddy) {
      console.log(buddy + " is a friend of " + this.name);
    }.bind(this));
  }
}
bob.helloBuddies();
```

En ES6, avec la **Arrow function** ou **fat arrow** (=>), plus de problèmes, vous pourrez écrire ceci:

```
var bob = {
  name: "Bob Morane",
  friends : ["Sam", "Ted", "Paul"],
  helloBuddies () {
    this.friends.forEach((buddy) => {
      console.log(buddy + " is a friend of " + this.name);
    });
  }
}

bob.helloBuddies();
```

ou même plus simplement:

```
var bob = {
  name: "Bob Morane",
  friends : ["Sam", "Ted", "Paul"],
  helloBuddies () {
    this.friends.forEach((buddy) => console.log(buddy + " is a friend of " + this.name));
  }
}

bob.helloBuddies();
```

Vous pouvez considérer les "Arrow Functions" comme des lambda expressions et les utiliser de la sorte:

```
var addition = (a,b) => a+b;

var substraction = (a,b) => { return a - b; }

console.log(addition(5,4));
console.log(substraction(5,4));
```

Non seulement cela va nous éviter pas mal d'erreurs (le syndrome du `this`) et autres effets de bord, mais en plus cette façon de faire optimise le fonctionnement du moteur JavaScript car il travail sur un scope unique et la arraw function n'a pas de constructeur. De plus comme vous pouvez le voir dans l'utilisation du `forEach`, elle est parfaitement adaptée pour l'utilisation des callbacks.

Maintenant que nous connaissons cette nouvelle notation, nous pouvons attaquer quelque chose de "génialement" pratique, tout particulièrement dans un monde asynchrone: les **Promises**.

1.7 Promise

Je vais tenter ma propre définition: la **Promise** c'est ce qui nous permettra de définir **simplement** à la fois:

- ce qu'il va se passer une fois qu'un traitement est terminé
- ce qu'il va se passer si le traitement "fonctionne mal"

Nous allons donc voir comment créer facilement une **Promise** et l'utiliser. Par exemple, je fais un traitement qui divise `42` par un nombre aléatoire compris entre `0` et `42`:

```
var divideFourtyTwoBySomething = () => new Promise((resolve, reject) => {

  var randomnumber = Math.floor(Math.random() * 43);

  if (randomnumber > 0) {
    var result = 42 / randomnumber;
    resolve({randomnumber, result});
  } else {
    reject(new Error("Divided by 0!"));
  }

});
```

Pour créer une **Promise**, j'utilise tout simplement l'objet `Promise` qui prend en paramètre de son constructeur une lambda ayant `2` paramètres correspondants chacun à un "callback" : `resolve(data)`, ce que j'appelle si tout se termine bien et `reject(error)`, ce que j'appelle si tout se passe mal. De cette manière, l'objet `Promise` nous permet d'utiliser `divideFourtyTwoBySomething()` de la façon suivante:

```
/* appeler divideFourtyTwoBySomething toute les secondes */
setInterval(() =>
```

```
  divideFourtyTwoBySomething().then((data) => {
    /* tout va bien */
    console.log(42,"divide by", data.randomnumber, "=", data.result);
  }).catch((err) => {
    /* tout va mal */
    console.log(err);
  })
, 1000);
```

ce qui nous donnera quelque chose comme ceci:

```
42 "divide by" 36 "=" 1.1666666666666667
42 "divide by" 27 "=" 1.5555555555555556
42 "divide by" 25 "=" 1.68
Error {stack: (...), message: "Divided by 0!"}
42 "divide by" 16 "=" 2.625
42 "divide by" 42 "=" 1
```

Donc notre objet `Promise` nous fournit 2 méthodes chaînées:

- `then(successCallback)` *(`successCallback` correspond à notre `resolve`)*
- `catch(errorCallback)` *(`errorCallback` correspond à notre `reject`)*

ECMAScript 6 nous propose là un moyen très simple de travailler en asynchrone et de reproduire en quelques lignes de code ce qui en prenait des dizaines et des dizaines (je pense notamment aux promises jQuery). Il deviendra très facile de créer des utilitaires JavaScript pour faire par exemple des requêtes ajax "user friendly":

```
new Request("/humans/bob")
    .get()
    .then((data) => {})
    .catch((error) => {});
```

Il y a encore beaucoup d'autres choses à découvrir comme les **collections** (Set, Map), les **proxies**, ... mais c'est une autre histoire (préparez vous à la suite bientôt). En attendant, n'hésitez pas à réagir et à contribuer à cette première partie dont vous trouverez une version "github" (avec les exemples de code) par ici: https://github.com/my-tutorials/je.decouvre.es6.