Les spécifications ES6

• EcmaScript 6, la spécification JavaScript en cours de support par les navigateurs, apporte un ensemble d'éléments : mot-clé let, template de chaîne de caractères, paramètres par défaut, gestion des modules, etc.

• Angular a basé son architecture sur ces spécifications. C'est notamment le cas pour l'utilisation des modules ES6, à ne pas confondre avec les modules Angular, qui permettent de déclarer des éléments puis de les importer.

ECMASCRIPT 6

La déclaration de variable avec var peut être délicate. Dans à peu près tous les autres langages, une variable existe à partir de la ligne contenant la déclaration de cette variable. Mais en JS, il y a un concept nommé *hoisting* ("remontée") qui déclare la variable au tout début de la fonction, même si on l'a écrite plus loin.

Ainsi déclarer une variable name dans le bloc if :

```
function getPonyFullName(pony) {
    if (pony.isChampion) {
        var name = 'Champion ' + pony.name;
        return name;
    }
    return pony.name;
}
```

est équivalent à la déclarer tout en haut de la fonction :

```
function getPonyFullName(pony) {
    var name;
    if (pony.isChampion) {
        name = 'Champion ' + pony.name;
        return name;
    }
    // name est encore accessible ici
    return pony.name;
}
```

Let

ES6 introduit un nouveau mot-clé pour la déclaration de variable, let, qui se comporte enfin comme on pourrait s'y attendre :

```
function getPonyFullName(pony) {
    if (pony.isChampion) {
        let name = 'Champion ' + pony.name;
        return name;
    }

    // name n'est pas accessible ici
    return pony.name;
}
```

L'accès à la variable name est maintenant restreint à son bloc, let a été pensé pour remplacer définitivement var.

Constante

ES6 introduit aussi **const** pour déclarer des constantes. Si on déclare une variable avec **const**, elle doit obligatoirement être initialisée, et on ne pourra plus lui affecter de nouvelle valeur par la suite.

```
const poniesInRace = 6;
```

```
poniesInRace = 7; // SyntaxError
```

Comme pour les variables déclarées avec **let**, les constantes ne sont pas **hoisted** (***remontées***) et sont bien déclarées dans leur bloc.

Constante

Par contre on peut initialiser une constante avec un objet et modifier par la suite le contenu de l'objet :

```
const PONY = {};
PONY.color = 'blue'; // Ok
```

Mais on ne peut pas assigner la constante à un nouvel objet :

```
const PONY = {};
PONY = {color: 'blue'}; // SyntaxError
```

Même chose avec les tableaux :

```
const PONIES = [];
PONIES.push({ color: 'blue' }); // Ok
PONIES = []; // SyntaxError
```

Création d'objets

Il y a un nouveau raccourci pour créer des objets, quand la propriété de l'objet qu'on veut créer a le même nom que la variable utilisée comme valeur pour l'attribut.

Par exemple :

```
function createPony() {
    const name = 'Rainbow Dash';
    const color = 'blue';
    return { name: name, color: color };
}
```

Peut être simplifié en :

```
function createPony() {
    const name = 'Rainbow Dash';
    const color = 'blue';
    return { name, color };
}
```

Affectation déstructurées

Il y a maintenant un raccourci pour affecter des variables à partir d'objets ou de tableaux.

En ES5 :

```
var httpOptions = { timeout: 2000, isCache: true };

// Plus tard
var httpTimeout = httpOptions.timeout;
var httpCache = httpOptions.isCache;
```

En ES6 :

```
const httpOptions = { timeout: 2000, isCache: true };

// Plus tard
const { timeout: httpTimeout, isCache: httpCache } = httpOptions;
```

Affectation déstructurées

La clé est la propriété à lire dans l'objet, et la valeur est la variable à affecter. Et comme précédemment : si la variable qu'on veut affecter a le même nom que la propriété de l'objet à lire, on peut simplement écrire :

```
const httpOptions = { timeout: 2000, isCache: true };

// Plus tard
const { timeout, isCache } = httpOptions;

// on a maintenant une variable nommée 'timeout' et une variable
nommée 'isCache' avec les valeurs correctes
```

Ça marche aussi avec des objets imbriqués :

```
const httpOptions = { timeout: 2000, cache: { age: 2 } };

// later
const { cache: { age } } = httpOptions;

// on a maintenant une variable nommée "age" avec la valeur 2
```

Affectation déstructurée

Et la même chose est possible avec les tableaux :

```
const timeouts = [1000, 2000, 3000];

// later
const [shortTimeout, mediumTimeout] = timeouts;

// on a maintenant une variable appelée ‘shortTimeout' avec
la valeur 1000 et une variable nommée ‘mediumTimeout' avec la
valeur 2000
```

Bien entendu, cela fonctionne avec des tableaux de tableaux, des tableaux dans des objets, etc… Un cas d’usage intéressant de cette fonctionnalité est la possibilité de retourner des valeurs multiples. Imaginons une fonction randomPonyInRace qui retourne un poney et sa position dans la course :

```
function randomPonyInRace() {
    const pony = { name: 'Rainbow Dash' };
    const position = 2;
    // ...
    return { pony, position };
}

const { pony, position } = randomPonyInRace();
```

Affectation déstructurée

Cette nouvelle fonctionnalité de déstructuration assigne la position retournée par la méthode à la variable position, et le poney à la variable pony.

Si on n'a pas usage de la position, on peut écrire :

```
function randomPonyInRace() {
    const pony = { name: 'Rainbow Dash' };
    const position = 2;
    // ...
    return { pony, position };
}

const { pony } = randomPonyInRace();
```

Et on aura seulement une variable pony.

Paramètres optionnels et valeur par défaut

JS a la particularité de permettre aux développeurs d'appeler une fonction avec un nombre d'arguments variable :

- si on passe plus d'arguments que déclarés par la fonction, les arguments supplémentaires sont tout simplement ignorés (on peut tout de même les utiliser dans la fonction avec la variable spéciale arguments).
- si on passe moins d'arguments que déclarés par la fonction, les paramètres manquants auront la valeur **undefined**.

Ce dernier cas est celui qui nous intéresse. Souvent, on passe moins d'arguments quand les paramètres sont optionnels, comme dans l'exemple suivant :

```
function getPonies(size, page) {
    size = size || 10;
    page = page || 1;
    // ...

    server.get(size, page);
}
```

Paramètres optionnels et valeur par défaut

Les paramètres optionnels ont la plupart du temps une valeur par défaut. L'opérateur OR (||) va retourner l'opérande de droite si celui de gauche est undefined, comme cela serait le cas si le paramètre n'avait pas été fourni par l'appelant (pour être précis, si l'opérande de gauche est falsy, c'est-à-dire undefined, 0, false, "", etc...).

Avec cette astuce, la fonction getPonies peut ainsi être invoquée :

```
getPonies(20, 2);
getPonies(); // équivalent à getPonies(10, 1);
getPonies(15); // équivalent à getPonies(15, 1);
```

Paramètres optionnels et valeur par défaut

Cela fonctionnait, mais ce n'était pas évident de savoir que les paramètres étaient optionnels, sauf à lire le corps de la fonction.

ES6 offre désormais une façon plus formelle de déclarer des paramètres optionnels, dès la déclaration de la fonction :

```
function getPonies(size = 10, page = 1) {
    // ...

    server.get(size, page);
}
```

Il y a cependant une subtile différence, car maintenant 0 ou "" sont des valeurs valides, et ne seront pas remplacées par les valeurs par défaut, comme **size = size || 10** l'aurait fait.

C'est donc plutôt équivalent à **size = size === undefined ? 10: size;**

La valeur par défaut peut aussi être un appel de fonction :

```
function getPonies(size = defaultSize(), page = 1){
    // la méthode defaultSize sera appelée si size n'est pas fournie
    // …
    server.get(size, page);
}
```

Paramètres optionnels et valeur par défaut

Ou même d'autres variables, d'autres variables globales, ou d'autres paramètres de la même fonction :

```
function getPonies(size = defaultSize(), page = size - 1){
    // si page n'est pas renseigné, elle sera mise à la
    // valeur du paramètre de taille moins un.
    // ...

    server.get(size, page);
}
```

Note que si on essaie d'utiliser des paramètres sur la droite, leur valeur sera toujours undefined :

```
function getPonies(size = page, page = 1) {
    // la taille sera toujours indéfinie, car le
    // paramètre page se trouve à sa droite.

    server.get(size, page);
}
```

Ce mécanisme de valeur par défaut ne s'applique pas qu'aux paramètres de fonction, mais aussi aux valeurs de variables, par exemple dans le cas d'une affectation déstructurée :

```
const { timeout = 1000 } = httpOptions;
// on a maintenant une variable nommée 'timeout',
// avec la valeur 'httpOptions.timeout' si elle
// existe ou 1000 si ce n'est pas le cas
```

Rest operator

ES6 introduit aussi une nouvelle syntaxe pour déclarer un nombre variable de paramètres dans une fonction.

Comme on l'a vu précédemment, on peut toujours passer des arguments supplémentaires à un appel de fonction, et y accéder avec la variable spéciale **arguments**.

par exemple :

```
function addPonies(ponies) {
    for (var i = 0; i < arguments.length; i++) {
        poniesInRace.push(arguments[i]);
    }
}

addPonies('Rainbow Dash', 'Pinkie Pie');
```

Ce n'est ni élégant, ni évident : le paramètres n'est jamais utilisé, et rien n'indique que l'on peut fournir plusieurs poneys.

Rest operator

ES6 propose une syntaxe bien meilleure, grâce au rest operator … (opérateur de reste) :

```
function addPonies(...ponies) {
    for (let pony of ponies {
        poniesInRace.push(pony);
    }
}
```

ponies est désormais un véritable tableau, sur lequel on peut itérer

La boucle **for ... of** utilisée pour l'itération est aussi une nouveauté d'ES6

Elle permet d'être sûr de n'itérer que sur les valeurs de la collection, et non pas sur ses propriétés comme **for ... in**. Le **rest operator** peut aussi fonctionner avec des affectations déstructurées :

```
const [winner, ...losers] = poniesInRace;
// 'poneysInRace' est un tableau de poneys
// 'winner' aura le premier poney et
// 'losers' sera un tableau des autres poneys
```

rest operator ne doit pas être confondu avec **spread operator** (opérateur d'étalement) qui est son opposé.

Il prend un tableau, et l'étale **en arguments variables**.

Un exemple avec la fonction **Min** :

```
const ponyPrices = [12, 3, 4];
const minPrice = Math.min(...ponyPrices);
```

Classes

Il s'agit d'une des fonctionnalités les plus emblématiques, et qui va largement être utilisée dans l'écriture d'applications Angular.

ES6 introduit les **classes** en JavaScript. On peut désormais facilement faire de l'héritage de classes en JavaScript.

Par exemple :

```
class Pony {
      constructor(color) {
            this.color = color;
      }
      toString() {
            return `${this.color} pony`;
      }
}
const bluePony = new Pony('blue');
console.log(bluePony.toString()); // blue pony
```

Les déclarations de classes, contrairement aux déclarations de fonctions, ne sont pas **hoisted** (***remontées***).

On doit donc déclarer une classe avant de l'utiliser. La fonction spéciale **constructor** est le constructeur, c'est à dire la fonction appelée à la création d'un nouvel objet avec le mot-clé **new**.

Classes

Dans l'exemple nous créons une nouvelle instance de la classe **Pony** avec la couleur ***blue***.

Une classe peut aussi avoir des méthodes, ***appelables*** sur une instance, comme la méthode ***toString*** dans l'exemple précédemment.

Une classe peut aussi avoir des attributs et des méthodes statiques :

```
class Pony {
    static defaultSpeed() {
        return 10;
    }
}
```

Ces méthodes statiques ne peuvent être appelées que sur la classe discrètement :

```
const speed = Pony.defaultSpeed();
```

Classes

Une classe peut avoir des accesseurs (*getters, setters*), si on veut implémenter du code sur ces opérations :

```
class Pony {
     get color() {
          console.log('get color');
          return this._color;
     }
     set color(newColor) {
          console.log(`set color ${newColor}`);
          this._color = newColor;
     }
}
const pony = new Pony();
pony.color = 'red'; // set color red
console.log(pony.color); // get color // red
```

Classes

L'héritage est possible en ES6 :

```
class Animal {
     speed() {
          return 10;
     }
}
class Pony extends Animal {
}

const pony = new Pony();
console.log(pony.speed()); // 10
// Pony hérite de la méthode de son parent Animal
```

Animal est appelée la *classe de base*, et Pony *la classe dérivée*. La classe dérivée possède toutes les méthodes de la classe de base.

Mais elle peut aussi les redéfinir :

Comme on peut le voir, le mot-clé super permet d'invoquer la méthode de la classe de base.

Dans cet exemple super.speed() redéfinit la méthode speed() de Animal.

```
class Animal {
     speed() {
          return 10;
     }
}

class Pony extends Animal {
     speed() {
          return super.speed() + 10;
     }
}
const pony = new Pony();
console.log(pony.speed()); // 20
// Pony surcharge la méthode de son parent
```

Classes

Ce mot-clé super peut aussi être utilisé dans les constructeurs, pour invoquer le constructeur de la classe de base :

```
class Animal {
    constructor(speed) {
        this.speed = speed;
    }
}

class Pony extends Animal {
    constructor(speed, color) {
        super(speed);
        this.color = color;
    }
}
const pony = new Pony(20, 'blue');
console.log(pony.speed); // 20
```

Promises

L'objectif des promises est de simplifier la programmation asynchrone.

Notre code JS est plein *d'asynchronisme*, comme des requêtes AJAX, et en général on utilise des callbacks pour gérer le résultat et l'erreur.

Mais le code devient vite confus, avec des callbacks dans des callbacks, qui le rendent illisible et peu maintenable.

Les promises sont plus pratiques que les callbacks, parce qu'elles permettent d'écrire du code à plat, et le rendent ainsi plus simple à comprendre.

Promises

Prenons un cas d'utilisation simple, où on doit récupérer un utilisateur, puis ses droits, puis mettre à jour un menu quand on a tout récupéré.

Avec des *callbacks* :

```
getUser(login, function (user) {
    getRights(user, function (rights) {
            updateMenu(rights);
    });
});
```

Avec des *promises* :

```
getUser(login)
.then(function (user) {
    return getRights(user);
})
.then(function (rights) {
    updateMenu(rights);
})
```

Promises

Les promises s'exécutent comme elles se lisent :

- je veux récupérer un utilisateur
- puis ses droits
- puis mettre à jour le menu

Une promise est un objet **thenable**, ce qui signifie simplement qu'il a une méthode **then**.

Cette méthode prend deux arguments :

- un callback de succès
- un callback d'erreur

Une promise a trois états :

- pending ("en cours") : tant que la promise n'est pas réalisée, par exemple quand l'appel serveur n'est pas encore terminé
- fulfilled ("réalisée") : quand la promise s'est réalisée avec succès, par exemple quand l'appel HTTP serveur a retourné un status 200-OK
- rejected ("rejetée") : quand la promise a échoué, par exemple si l'appel HTTP serveur a retourné un status 404-NotFound

Quand la promesse est réalisée (fulfilled), alors le callback de succès est invoqué, avec le résultat en argument.

Si la promesse est rejetée (rejected), alors le callback d'erreur est invoqué, avec la valeur rejetée ou une erreur en argument.

Promises

Alors, comment crée-t-on une promise ? C'est simple, il y a une nouvelle classe **Promise**, dont le constructeur attend une fonction avec deux paramètres, **resolve** et **reject**.

```
const getUser = function (login) {
    return new Promise(function (resolve, reject) {
      // des choses asynchrones, comme aller chercher
         des utilisateurs sur le serveur, renvoyer une
         réponse
         if (response.status === 200 {
              resolve(response.data);
         } else {
              reject('No user');
         }
    });
};
```

Une fois la promise créée, on peut enregistrer des **callbacks**, via la méthode **then**

Cette méthode peut recevoir deux arguments, les deux callbacks qu'on invoque en cas de succès (resolve) ou en cas d'échec (catch).

Promises

Dans l'exemple suivant, on passe simplement un seul callback de succès, ignorant ainsi une erreur potentielle :

```
getUser(login)
.then(function (user) {
    console.log(user);
});
```

Quand la promesse sera réalisée, le callback de succès sera invoqué Le code peut aussi s'écrire à plat (chainé).

Si par exemple le callback de succès retourne lui aussi une promise :

```
getUser(login)
.then(function (user) {
    return getRights(user)
    // getRights retourne aussi une promise
.then(function (rights) {
    return updateMenu(rights); });
});
```

Promises

On peut définir une gestion d'erreur par promise, ou globale à toute la chaîne.

Une gestion d'erreur par promise :

```
getUser(login)
.then(function (user) {
     return getRights(user);
}, function (error) {
     console.log(error); // sera appelé si getUser échoue
     return Promise.reject(error);
})
.then(function (rights) {
     return updateMenu(rights);
}, function (error) {
     console.log(error); // sera appelé si getRights échoue
     return Promise.reject(error);
});
```

Une gestion d'erreur globale pour toute la chaîne :

```
getUser(login)
.then(function (user) {
    return getRights(user);
})
.then(function (rights) {
    return updateMenu(rights);
})
.catch(function (error) {
     console.log(error);
    // sera appelé si getUser ou getRights échoue
});
```

Il faut s'intéresser aux promises, parce que ça va devenir la nouvelle façon d'écrire des APIs, et toutes les bibliothèques vont les utiliser.

Arrow functions

ES6 introduit la nouvelle syntaxe arrow function (*fonction flèchée*), utilisant l'opérateur fat arrow (*grosse flèche*) : ⇒

C'est très utile pour les callbacks et les fonctions anonymes.

Prenons notre exemple précédent avec des promises :

```
getUser(login)
.then(function (user) {
      return getRights(user); // getRights retourne une promise
})
.then(function (rights) {
      return updateMenu(rights);
})
```

Il peut être réécrit avec des arrow functions comme ceci :

```
getUser(login)
  .then(user => getRights(user))
  .then(rights => updateMenu(rights))
```

Arrow functions

Notez que le **return** est implicite *s'il n'y a pas de bloc* :

pas besoin d'écrire **user ⇒ return getRights(user).**

Mais si nous avions un **bloc**, nous aurions besoin d'un **return explicite** :

```
getUser(login)
.then(user => {
    console.log(user);
    return getRights(user);
})
.then(rights => updateMenu(rights))
```

Et les **arrow functions** ont une particularité bien agréable que n'ont pas les fonctions normales : le **this** reste attaché lexicalement, ce qui signifie que ces **arrow functions** n'ont pas un nouveau **this** comme les fonctions normales.

Arrow functions

Prenons un exemple où on itère sur un tableau avec la fonction map pour y trouver le maximum

Comparons le code en ES5 et en ES6 :

En ES5 :

```
var maxFinder = {
      max: 0,
      find: function (numbers) { // itération
            numbers.forEach(
                  function(element) {
                        if (element > this.max) {
                              this.max = element; }
                        }
                  );
            }
      };

maxFinder.find([2, 3, 4]); // log le résultat
console.log(maxFinder.max);
```

Arrow functions

Ça semble correct, mais en fait ça ne marche pas.

Le **forEach** dans la fonction ***find*** utilise **this**, mais ce **this** n'est lié à aucun objet. Donc ***this.max*** n'est en fait pas le ***max*** de l'objet ***maxFinder***.

On pourrait corriger ça facilement avec un alias :

```
var maxFinder = {
      max: 0,
      find: function (numbers) {
            var self = this;
            numbers.forEach(
                  function (element) {
                        if (element > self.max) {
                              self.max = element;
                        }
                  }
            );
      }
};
```

Arrow functions

Ou en *bindant* le this :

```
var maxFinder = {
     max: 0,
     find: function (numbers) {
          numbers.forEach(
               function (element) {
                    if (element > this.max) {
                         this.max = element; }
                    }.bind(this)
          );
     }
};
maxFinder.find([2, 3, 4]); // log le résultat
console.log(maxFinder.max);
```

ou en le passant en second paramètres de la fonction forEach :

```
var maxFinder = {
     max: 0,
     find: function (numbers) {
          numbers.forEach(
               function (element) {
                    if (element > this.max) {
                         this.max = element;
                    }
               }, this
          );
     }
};
maxFinder.find([2, 3, 4]); // log le résultat
console.log(maxFinder.max);
```

Arrow functions

Mails il y a maintenant une solution bien plus élégante avec les arrow functions:

```
const maxFinder = {
      max: 0,
      find: function (numbers) {
            numbers.forEach(
                  element => {
                        if (element > this.max) {
                              this.max = element;
                        }
                  }
            );
      }
};
maxFinder.find([2, 3, 4]); // log le résultat
console.log(maxFinder.max);
```

Les arrow functions sont donc idéales pour les fonctions anonymes en callback.

SET et MAP

On a maintenant de vraies *collections* en ES6.

On utilisait jusque-là de *simples objets JavaScript* pour jouer le rôle de map (“dictionnaire"), c’est à dire un objet JS standard, dont les clés étaient nécessairement des chaînes de caractères.

Mais nous pouvons maintenant utiliser la nouvelle classe Map :

```
const cedric = {
     id: 1,
     name: 'Cedric'
};

const users = new Map();
users.set(cedric.id, cedric); // ajoute un user
console.log(users.has(cedric.id)); // true
console.log(users.size); // 1
users.delete(cedric.id); // supprimer le user
```

SET et MAP

On a aussi une classe Set (ensemble) :

```
const cedric = {
     id: 1,
     name: 'Cedric'
};

const users = new Set();
users.add(cedric); // ajoute un user
console.log(users.has(cedric)); // true
console.log(users.size); // 1
users.delete(cedric); // supprimer le user
```

On peut aussi itérer sur une collection, avec la nouvelle syntaxe for … of :

```
for (let user of users) {
     console.log(user.name);
}
```

Template de string

Construire des **strings** a toujours été pénible en JavaScript, où nous devions généralement utiliser des ***concaténations*** :

```
const fullname = 'Miss ' + firstname + ' ' + lastname;
```

Les templates de string sont une nouvelle fonctionnalité mineure mais bien pratique.

On doit utiliser des accents graves (backticks `) au lieu des habituelles apostrophes (*quote* ') ou apostrophes doubles (*double-quotes* «).

Cela fournit un moteur de template basique avec support du multi-ligne :

```
const fullname = `Miss ${firstname} ${lastname}`;
```

Template de string

Le support du *multi-ligne* est bien adapté à l'écriture de morceaux.

d'HTML, comme nous le ferons dans nos composants Angular :

```
const template = `
     <div>
          <h1>Hello</h1>
</div>`;
```

Un peu plus loin en ES6

Types dynamiques et statiques

Les applications Angular peuvent être écrites en ES5, ES6, ou TypeScript

Alors qu'est-ce que TypeScript, et qu'est-ce qu'il apporte de plus ?

JavaScript est dynamiquement typé :

```
// on déclare implicitement un type String
let pony = 'Rainbow Dash';

// maintenant on déclare dynamiquement un type Integer
pony = 2;
```

Cette nature dynamique est formidable, mais elle est aussi un handicap dans certains cas, comparée à *d'autres langages plus fortement typés*.

Le cas le plus évident est quand on doit appeler une fonction inconnue d'une autre API en JS : *on doit lire la documentation* (ou pire le code de la fonction) pour deviner à quoi doivent ressembler les paramètres.

Sans les informations de type, *les IDEs n'ont aucun indice* pour savoir si on écrit quelque chose de faux, et les outils ne peuvent pas nous aider à trouver des bugs dans notre code.

Angular a toujours facilité les tests, mais c'est pratiquement impossible d'avoir une parfaite couverture de tests.

Un peu plus loin en ES6

Cela nous amène au sujet de la **maintenabilité**.

Le code JS peut être difficile à maintenir, malgré les tests et la documentation.

Refactoriser une grosse application JS n'est pas chose aisée, comparativement à ce qui peut être fait dans des langages **statiquement typés**.

La *maintenabilité* est un sujet important, et les *types* aident les outils, ainsi que les développeurs, à éviter les erreurs lors de l'écriture et la modification de code.