



ECMAScript 2015 ES6

Sommaire

- A. Introduction d3 à d6
- B. Les variables et constantes
- C. Les types
 - A. Primitif, Boolean, Number, String
- D. Les tableaux : Array
- E. Les objets
- F. Les collections (Map et Set)
- G. Les littérateurs (Iterator)
- H. Les fonctions générateurs (function*)
- I. Les promesses (promise)
- J. Les classes (class)

Introduction

les versions de l'ECMAScript

- JavaScript est depuis plusieurs années recommandé par le W3C sous le nom d'ECMAScript
- Il est souvent appelé ES suivi d'un numéro correspondant à sa version
- ES5, ECMAScript 5.1, publié en 2011
- ES6, ECMAScript 6, également appelé Harmony, publié en 2015
- ES7 (2016) et ES8 (2017), releases 'mineures'
- Tout les ans une nouvelle version sort

Introduction

les versions de l'ECMAScript

- Nous utiliserons ES6 car les autres versions sont mineures.
- La compatibilité pour ES6 des moteurs JS est visible sur :
<https://kangax.github.io/compat-table/es6/>
- Au jour d'aujourd'hui tous les navigateurs supportent ES6.

Les variables et constantes

les identificateurs

- Les identificateurs :
 - sont sensibles à la casse
 - utilisent par convention le style camel case, et non le snake case :
sommeNotes et non *somme_notes*
- Remarques :
 - Les identificateurs peuvent aussi commencer par 1 ou 2 underscores ('_').
 - C'est une convention pour des variables spéciales ou internes.

Les variables et constantes

les mots clés

- Les mots clés à utiliser sont :
 - `let` pour déclarer une variable
 - `const` pour déclarer une constante

Exemple :

```
let somme;  
let moyenneDesNotes = 0; // en camel case  
  
const NBR_NOTE = 10; // en majuscule
```

- Remarque :
 - Eviter l'utilisation de `var` pour déclarer des variables

Les variables et constantes portée de bloc (1)

- Contrairement à « var », « let » et « const » définissent des variables et constantes qui sont locales au bloc où elles apparaissent.

Exemple :

```
{  
    let somme;  
    let moyenneDesNotes = 0;  
}  
console.log(somme);    // erreur
```

- De même les paramètres de fonction sont aussi locaux à la fonction

Exemple :

```
function double(x) {  
    return x * 2;  
}  
console.log(x);    // erreur
```


Les variables et constantes

portée de bloc (2)

- Une variable définie dans un bloc sera accessible dans les autres blocs interne

Exemple :

```
let var1 = 5; // variable de portée globale
let var2 = 3;
if (true) {
  console.log(var1); // erreur : inaccessible avant
                    // déclaration dans ce bloc
  console.log(var2); // affiche var2 globale : 3
  let var1 = 8; // déclar. locale : masque var1 globale
  if (true) {
    console.log(var1); // visible dans le 2ème bloc if : 8
    console.log(var2); // 3
  }
  console.log(var1); // affiche var1 locale au 1er if : 8
}
console.log(var1); // affiche var1 globale : 5
```

13/02/2023

- En js seulement 3 types existent :
 - boolean : true ou false
 - number : entier ou réel
 - string : chaine de caractères
- On peut ajouter ces 3 autres types spéciaux
 - undefined : valeur prise pour une variable non initialisée
 - null : utilisée pour un objet ayant aucune valeur
 - symbole : représente une donnée unique pour un objet

- Pour tous ces types primitifs hors mis le type undefined et null, il existe des objets globaux appelés « wrapper » qui peuvent les contenir :
 - Boolean : pour boolean
 - Number : pour number
 - String : pour string
 - Symbole : pour symbole
- Ces objets apportent des méthodes utiles pour manipuler ces types

- Deux valeurs possibles : true et false
- Toute valeur peut être évaluée comme un booléen :
 - undefined, null, 0, NaN, "" : évalué à false
 - tout objet (non null), tableau, chaîne non vide (longueur > 0) : évalué à true

Exemple :

```
{  
  let somme; // undefined  
  
  if (somme) { // évalué à false  
    ....  
  }  
}
```

- Deux possibilités pour convertir une valeur en type boolean :
- La double négation : « !! »
- La fonction « Boolean() »

Exemple :

```
let somme = 0;
```

```
let bool1 = !!somme; // bool1 boolean false
```

```
let bool2 = Boolean(somme); // idem
```

- Représente les valeurs entières ou réelles
- Possibilité d'écrire une valeur numérique dans différentes bases :

Exemple :

```
let somme = 15;           // 15 base 10
let somme = 017;          // 15 base 8 : octal
let somme = 0b00001111;   // 15 base 2 : binaire
let somme = 0xF;          // 15 base 16 : hexadécimal
let somme = 15.5;         // 15,5 base 10
let somme = 1.55e1;        // 15,5 notation scientifique
let somme = 155e-2;        // idem
```

- L'objet Number possède des valeurs particulières :
 - Number.MIN_VALUE, Number.MAX_VALUE
 - Number.NEGATIVE_INFINITY, POSITIVE_INFINITY

- Trois fonctions existent :
- `Number()`
- `parseInt()`
- `parseFloat()`

Exemple :

```
let num1 = Number("bonjour"); // NaN
let num2 = Number("0010"); // 10
let num3 = Number(true); // 1
let num4 = Number(false); // 0
let num5 = parseInt(""); // NaN
let num6 = parseInt(10.5); // 10
let num7 = parseFloat("150.25"); // 150.25
```

Les types méthodes de l'objet Number

- `Number.isNaN()`
 - Revoie « true » seulement si le paramètre est un nombre qui vaut « NaN ».

Exemple :

- `Number.isNaN(NaN);` // *true*
- `Number.isNaN(0 / 0);` // *true*
- `Number.isNaN(undefined);` // *false*
- `Number.isNaN({});` // *false*
- `Number.isNaN(37);` // *false*

- `Number.isInteger()`
 - Permet de tester si une valeur donnée est un entier

- Définie par des quotes simples (apostrophes) ou doubles (guillemets).
- La propriété « length » donne la longueur de la chaine (nombre de caractères).
- Il faut échapper les caractères « ' ou " » par « \' et \" » lorsque veut les utiliser come simple caractère.

Exemple :

```
let message = " bonjour";  
let message = " bonjour \" Paul \" ";  
let message = ' bonjour bonjour \' Paul \' ';
```

- En utilisant les backticks « ` » (altgr 7 sur pc), depuis ES6, à la place des quotes (simples ou doubles), il est possible d'injecter la valeur d'une variable ou expression numérique dans une string.
- Utiliser la syntaxe : `${nom_variable}`

Exemple :

```
let nom = "Auchon";  
let prenom = "Paul";  
let message = `bonjour ${prenom} ${nom}`;  
let message = ' bonjour ' + prenom + ' ' + nom;
```

- Ceci évite les concaténations souvent un peu compliquées.

Les types méthodes de l'objet String

- Il existe un grand nombre de méthodes :

Exemple :

```
let message = "bonjour"  
message.length           // 7  
message.charAt(2)         // « n »  
message.slice(3)          // « jour »  
message.indexOf("j")      // 3  
message.toUpperCase()     // « BONJOUR »  
message.includes("jour")  // true
```

- Remarque :
 - Techniquement, message est de type primitif et est convertie en un objet String le temps de l'exécution de la méthode

Les types nouvelles méthodes de String

- Avec ES6 de nouvelles méthodes ont vu le jour :
 - Démarrant par :
String.startsWith(chaineRecherche [, position])
 - Finissant par :
String.endsWith(chaineRecherche [, position])
 - Incluant :
String.include(chaineRecherche [, position])
 - Répéter la chaine
String.repeat(nombreDeFois)

- Le type « Symbol » est un nouveau type (ES6) permettant de représenter des « tokens » uniques.
- Les valeurs sont créées en utilisant la fonction « Symbol() » (pas de new)
- Possibilité de donner une description au symbol

Exemple :

```
const ERR1 = Symbol("Erreur de syntaxe")
```

```
const ERR2 = Symbol("Erreur de syntaxe")
```

```
ERR1 === ERR2    // false Tout symbol est unique
```

- 2 possibilités pour déclarer un tableau :
 - Notation littérale

Exemple :

```
let nom_tab = []  
let nom_tab = [val1, val2, val3]
```

- En Objet : appel au constructeur de la classe « Array »

Exemple :

```
let nom_tab = new Array()  
let nom_tab = new Array(4) // dimensionne le tableau  
let nom_tab = new Array(val1, val2, val3)
```


- De nombreuses méthodes existent :
 - Ajouter / supprimer un élément :
 - en queue de tableau : push / pop
 - en tête de tableau : unshift / shift
 - Ajouter / supprimer des éléments à n'importe quelle position : splice
 - Ajouter des éléments en queue de tableau : concat
 - Extraire un sous-tableau : slice
 - Remplir un tableau : fill
 - Inverser et trier un tableau : reverse et sort
 - copy and replace au sein d'un tableau : copyWithin
 - trouver le premier ou dernier indice d'une valeur : indexOf et lastIndexOf

- D'autres méthodes utilisent des fonctions de « callback » :

- `some()` : renvoi true si au moins 1 élément du tableau satisfaisant la condition

```
let test = nom_tab.some(function(valeur) {  
    return valeur == 12;  
}); // vrai si au moins 1 élément vaut 12
```

- `Every()` : renvoi true si tous les éléments du tableau satisfaisant la condition

```
let test = nom_tab.every(function(valeur) {  
    return valeur > 12;  
}); // vrai si tous les éléments sont supérieur à 12
```

- `Find()` : renvoi la valeur du 1er élément du tableau satisfaisant la condition sinon undefined

```
let val = nom_tab.find(function(valeur) {  
    return valeur >= 12;  
}); // 13 1er élément >= 12
```

- Cette méthode permet de créer un nouveau tableau.
- Elle prend en paramètre une fonction de callback qui va être appelée pour chaque élément du tableau et qui renvoi un élément du nouveau tableau

Exemple :

```
let test = ["Créteil", "Paris", "Nantes"]
```

```
let list = test.map(  
  (ville) => return ` <li>${ville}</li> `
```

```
)
```

Les tableaux

l'opérateur spread « ... »

- Cet opérateur transforme un tableau en une liste de valeurs (valeurs de ce tableau).
 - Ceci permet de faire des copies de tableau
 - D'utiliser un tableau comme liste d'argument pour une fonction
 - De concaténer 2 tableaux pour en créer un troisième

Exemple :

let tab1 = [1, 2, 3]

Copie : *let* tab2 = [...tab1]

Concaténation : *let* tab3 = [...tab1, ...tab2]

Argument de fonction :

function somme(val1, val2, val3) { suite... }

appel : *let* som = somme(...tab1)

Les tableaux

la boucle for(élément of tableau)

- La boucle « for ... of ... » permet d'itérer sur l'ensemble des valeurs d'un tableau.

Exemple :

```
let tab = [1, 2, 3]
```

```
for( const element of tab) {  
    console.log(element)  
}
```

=>

1
2
3

Les objets

la boucle `for(propriété in objet)`

- La boucle « `for ... in ...` » permet d'itérer sur l'ensemble des propriétés d'un objet.

Exemple :

```
let obj = {a: 1, b: 2, c: 3}
```

```
for(const prop in obj){  
    console.log(` ${prop}: ${obj[prop]} `)  
}
```

=>

```
a: 1  
b: 2  
c: 3
```

- L'objet Map permet de créer une collection de paires « clé / valeur »
- La clé et la valeur peuvent être de type primitif ou objet.

Exemple :

```
let maMap = new Map()  
maMap.set("nom", "Paul")
```

```
console.log(maMap.size) => 1
```

```
console.log(maMap.get("nom")) => Paul
```

- Autres méthodes :
clear, delete, keys, values

- L'objet Set est identique à un Array mais ne peut pas contenir deux fois la même valeur.

Exemple :

```
let maSet = new Set()  
maSet.add("Paul")  
maSet.add("Eric")  
maSet.add("Paul")
```

refuse : existe déjà

```
console.log(maSet.size)           => 2  
console.log(maSet.has("Eric"))    => true
```

- Autres méthodes :
 - clear, has, delete, keys, values

- Un itérateur est un objet sachant comment accéder aux éléments d'une collection un par un.
- Un objet est itérable si il implémente la fonction `iterator()`.
- Les String, Array, Map, Set sont itérables et implémente la méthode `Symbol.iterator`.
- Un objet Iterator possède une méthode « `next()` » qui renvoi pour chaque élément de la collection un objet ayant 2 propriétés :
 - `value` : valeur de l'élément
 - `done` : booléen vrai si `next()` ne trouve plus d'élément

- Utilisation de l'iterator d'un Array

Exemple :

```
let monTab = [10, 20, 30]
```

```
let iter = monTab[Symbol.iterator]
```

```
console.log(iter.next())    => {value: 10, done: false}  
console.log(iter.next())    => {value: 20, done: false}  
console.log(iter.next())    => {value: 30, done: false}  
console.log(iter.next())    => {value: undefined,  
                                done: true}
```

- Remarque : la boucle « for ... of ... » sur un Array utilise l'iterator

Les générateurs

la fonction « `function*` »

- Un générateur est un type de fonction spécial qui fonctionne comme une fabrique (*factory*) d'itérateurs.
- elle contient une ou plusieurs expressions « `yield` ».
- Elle utilise la syntaxe :

```
function* nomFonction(params) {  
    yield val1  
    yield val2  
}
```
- L'appel de la fonction renvoi un iterator.
- Ce type de fonction garde son état.

- Générateur renvoyant le double d'une valeur passée en paramètre

Exemple :

```
function* genDouble(val) {  
    var valeur = val  
    while(true){  
        yield valeur *= 2  
    }  
}
```

```
let iterGenDouble = genDouble(25)
```

```
console.log(iterGenDouble.next())    => 50  
console.log(iterGenDouble.next())    => 100
```

- Une promesse en JavaScript est un objet qui représente l'état d'une opération asynchrone.
- L'opération asynchrone peut être dans l'un des états suivants :
 - opération en cours (non terminée) ;
 - opération terminée avec succès (promesse résolue) ;
 - opération terminée ou stoppée après un échec (promesse rejetée).
- nous allons pouvoir créer nos propres promesses ou manipuler des promesses déjà créées par des API.

- Syntaxe de base d'un objet « Promise »

```
const maPromesse = new Promise(  
  (resolve, reject) => {  
    // tâche asynchrone  
    // appel de resolve() si ok  
    // appel de reject() si erreur  
  }  
)
```

- Généralement on crée une fonction qui retourne une promesse.

```
function maFonctionAsynch(param) {  
  return new Promise( (resolve, reject) => { } )  
}
```

```
const maPromesse = maFonctionAsynch()
```


- L'exploitation des résultats d'une promesse se fait par les méthodes :
 - `then()` quand la requête est un succès
 - `catch()` quand il y a erreur

```
function maFonctionAsynch(param) {  
    return new Promise( (resolve, reject) => { } )  
}
```

```
const maPromesse = maFonctionAsynch()
```

```
maPromesse  
    .then(result => fonctionDeTraitement(result))  
    .catch(result => fonctionErreur(result))
```


- Extrait de « developer.mozilla.org »

```
function faireQqc() {  
  return new Promise(  
    (successCallback, failureCallback) => {  
      console.log("C'est fait");  
      // réussir une fois sur deux  
      if (Math.random() > .5) {  
        successCallback("Réussite");  
      } else {  
        failureCallback("Échec");  
      }  
    })  
  )  
}
```

```
const promise = faireQqc();  
promise.then(alert).catch(alert);
```

- L'API Fetch va nous permettre de faire des requêtes asynchrones.
- Elle fournit trois interfaces :
 - Request, Response, Headers
- Et un mixin Body qui permet d'exploiter la réponse en fonction de son type.
- La méthode « fetch() » prend 2 arguments :
 - l'URL de la ressource que l'on veut
 - des options pour la requête
- Cette méthode renvoie une promesse (un objet Promise) que l'on pourra exploiter avec les méthodes « then() » et « catch() »

Les promesses

L'API fetch : exemple

- Appel de l'api rest de citation

```
fetch("https://api.quotable.io/random")  
  .then(response => response.json())  
  .then(response => alert(JSON.stringify(response)))  
  .catch(error => alert("Erreur : " + error));
```

- L'objet « Reject » a une propriété
« message »

```
.catch(error => alert("Erreur : " + error.message));
```

Les promesses

l'API fetch : exemple

- Attention : dans le cas d'une réponse 404 ou 500, la promesse est tenue.
 - Il faut tester la propriété « ok » de l'objet « Response »

```
fetch("https://api.quotable.io/random")  
  .then(function(response) {  
    if(response.ok)  
      response.json().then(function(objJson) {  
        alert(JSON.stringify(objJson)) })  
    else  
      alert('Erreur du serveur')  
  })  
  .catch(error => alert("Erreur : " + error.message));
```

Les promesses

L'API « Fetch » complément

- Le 2^{ème} argument de la méthode « fetch() » permet de donner des options :
 - method, headers, cache, mode, ...

```
let myInit = { method: 'GET',  
                headers: myHeaders,  
                mode: 'cors',  
                cache: 'default' };
```

```
fetch('url',myInit)
```

```
...
```

- symboles « { » et « } » encadrant la définition de l'objet
- introduction de champs (propriétés) et fonctions (méthodes)
- propriété : constituée d'un nom suivi de ':' et de la valeur
- symbole « , » comme séparateur entre les différents propriétés/méthodes

```
let person = {  
  nom: 'Alice',  
  age: 20,  
  
  introduction() {  
    return this.nom + ' ' + this.age;  
  }  
};
```


Les Objets

accès aux membres d'un objet

- Pour accéder à tout membre (propriété ou méthode) d'une variable de type Object, on utilise généralement la notation pointée :
nom de la variable suivi de « . » suivi du nom du membre
- Toutefois, il est également possible d'utiliser les crochets pour accéder aux membres.

```
let person = {  
  nom: 'Alice',  
  age: 20,  
  ...  
};
```

```
console.log(person.nom + ' a ' + person.age + ' ans';  
console.log(person['nom'] + ' a ' + person['age'] + ' ans';
```


- « *this* » représente l'objet lui même si on l'utilise dans le code d'une méthode de l'objet

```
let jeu = {  
  fini: false,  
  ...  
  isFini() {  
    return this.fini;  
  },  
};  
  
if (jeu.isFini()) {  
  ...  
}
```

- On peut utiliser « for in », mais il faut généralement prêter attention aux propriétés héritées.

```
let pers = { nom: 'AIMAR', prenom: 'Jean', age: 30};
```

```
for (let prop in pers)  
if (pers.hasOwnProperty(prop))  
console.log(prop + ' : ' + pers[prop]);
```

Resultat :

```
nom : AIMAR  
prenom : Jean  
age : 30
```

- Une classe se déclare en utilisant le mot clé « class » suivi du nom dont on mettra la première lettre en majuscule.
- Une classe, on peut avoir un constructeur (un seul !).
- Les propriétés sont déclarées par le mot clé « this » dans le constructeur.
- Les méthodes sont des fonctions déclarées sans le mot-clé « function »).

```
class Personne {  
    constructor(unNom, unPrenom) {  
        this.nom = unNom;  
        this.prenom = unPrenom;  
    }  
}
```

- Une instance une classe (on créer des objets de cette classe) en utilisant le mot clé « new »
- Suivi du nom de la classe avec les parenthèses « () » et les valeurs des paramètres à l'intérieur.
- Ceci appel le constructeur.

```
class Personne {  
    constructor(unNom, unPrenom) {  
        this.nom = unNom;  
        this.prenom = unPrenom;  
    }  
}
```

```
let pers = new Personne('AIMAR', 'Jean');
```

- Une méthode est une fonction déclarée à l'intérieur de la définition de la classe sans utiliser le mot clé « fonction ».
- Elle est appelée à partir d'une « instance » de la classe.

```
class Personne {  
    constructor(unNom, unPrenom) {  
        this.nom = unNom;  
        this.prenom = unPrenom;  
    }  
    presenteToi() {  
        return 'Je suis ' + this.prenom + ' ' + this.nom;  
    }  
}
```

```
let pers = new Personne('AIMAR', 'Jean');  
Console.log(pers.presenteToi());
```

- Une méthode de class est une fonction déclarée à l'intérieur de la définition de la classe avec le mot clé « static ».
- Elle est appelée à partir de la classe.

```
class Personne {  
  constructor(unNom, sonAge) {  
    this.nom = unNom;  
    this.age = sonAge;  
  }  
  static compare(pers1, pers2){  
    return pers1.age - pers2.age;  
  }  
}  
  
let p1 = new Personne('PARFAIT', 30);  
let p2 = new Personne('AIMAR', 25);  
Console.log(Personne.compare(p1, p2));
```


- Une classe peut hériter des propriétés et méthodes d'une autre classe.
- L'héritage se fait en utilisant le mot clé « extends »

```
class Personne {  
    constructor(unNom, unPrenom) {  
        this.nom = unNom;  
        this.prenom = unPrenom;  
    }  
}
```

```
class Employe extends Personne {  
    constructor(unNom, unPrenom, unMatricule) {  
        super(unNom, unPrenom);  
        this.matricule = unMatricule;  
    }  
}
```

- L'utilisation du constructeur dans une classe dérivée impose d'appeler le constructeur de la classe mère en utilisant « `super()` ».
- Cette méthode doit être appelée en premier dans le constructeur de la classe fille.

```
class Employe extends Personne {  
    constructor(unNom, unPrenom, unMatricule) {  
        super(unNom, unPrenom);  
        this.matricule = unMatricule;  
    }  
}
```

```
let emp = new Employe('PARFAIT', 'Alain', 235);  
console.log( `l'employé ${emp.nom} a le matricule  
    ${emp.matricula}` );
```

- L'utilisation du mot clé « super » permet à partir d'une méthode de la classe fille, d'appeler une méthode de la classe mère.

```
class Personne {  
    constructor(unNom, unPrenom) {  
        this.nom = unNom;  
        this.prenom = unPrenom;  
    }  
  
    presenteToi(){  
        return `Je m'appelle ${this.prenom} ${this.nom}`  
    }  
}
```

- L'utilisation du mot clé « super » permet à partir d'une méthode de la classe fille, d'appeler une méthode de la classe mère.

```
class Employe extends Personne {  
    constructor(unNom, unPrenom, unMatricule) {  
        super(unNom, unPrenom);  
        this.matricule = unMatricule;  
    }  
  
    presenteToi(){  
        return super.presenteToi() + ` te j'ai le matricula  
        ${this.matricula}`;  
    }  
}
```