JAVASCRIPT ES6

J.-M. Pecatte
Dept MMI – IUT Castres
jean-marie.pecatte@iut-tlse3.fr

Javascript aujourd'hui ?

- Un des langages les plus utilisées dans le monde ; dans le web essentiellement pour le front-end,
- Mais évolution importante car, avec l'arrivée de NodeJS, il est aussi possible de développer le back-end
  => Développeur full-stack en JS
- Cette évolution est due aussi à l'apparition de nombreux frameworks comme Angular (Google) ou React (Facebook)

- Etat de l'art d'utilisation du langage JS et frameworks associés
  https://2020.stateofjs.com/en-US/

Javascript et ECMAScript

- **ECMAScript** : normalisation des langages de script (JavaScript, Jscript, ActionScript)
  - ES4 (2007), ES5 (2009) -> bien intégré dans les navigateurs
  - ES6 (2015) -> en cours de prise en charge

  Javascript est une implémentation de ECMAScript utilisé principalement par les navigateurs et NodeJS

Les plus de la version ES6

ES6 : variables

- déclaration de variables : utilisation de let

Syntaxiquement let s'utilise comme var mais son usage est recommandé car la portée des variables est mieux définie.

➔Une variable let n'est visible que dans le bloc où elle est définie

➔Une variable ne peut pas être utilisée avant sa déclaration

➔Une variable ne peut être déclarée qu'une seule fois

- déclaration de constantes : utilisation de const

ES6 : les types

- booléen : true / false
- nombre : entier et réel
- string : entre " ", ou entre ' ' ou entre ` ` *(backquote/backtick)*
  les `` permettent d'avoir une chaine sur plusieurs lignes et avec des expressions de la forme ${ expr }

```
let prenom = 'Jean';
let age = 37;
let message = `Bonjour, mon nom est ${ prenom }.
                j'aurai ${ age + 1 } ans le mois prochain `;
equivalent de :
let message = "Bonjour, mon nom est " + prenom + "\n" +
                "j'aurai " + (age + 1) + " ans le mois prochain";
```

Interpolation de chaine ou template chaine

ES6 : les types

- tableau :

avec les [ ]

```
let liste = [1, 2, 3];
```

rappel :

liste.length → longueur (nb d'éléments)
liste.push(elt) → ajout d'un élement à la fin
liste.pop() → retrait du dernier élément
liste.indexOf(elt) → renvoi l'indice de l'élément, -1 sinon
liste.splice(indice, nb) → supprime nb elts depuis indice
liste.unshift(elt1, elt2, …) → ajoute les elts au début
liste.concat(liste2) → ajoute les elts de la liste2 à la fin

Attention : let tab = liste -> ne crée pas une copie du tableau

ES6 : objet littéral

- Objet littéral :

avec les { }, ensemble de couple propriété **:** valeur

```
let personne = {
        nom : "Dupont",
        prenom : "Jean",
        age : 45
}
```

→ accès à une propriété : personne.nom
→ une propriété peut elle-même être un objet littéral ou un tableau

ES6 : déstructuration / propagation

- déstructurer un tableau ou un objet : permet d'extraire des éléments de tableau ou d'objets dans plusieurs variables :

→ stocker les 2 valeurs d'un tableau dans 2 variables

```
let [un, deux] = [1, 2];    // un = 1    deux =2
```

→ stocker les 3 propriétés d'un objet dans trois variables en une seule instruction

```
let { vnom, vprenom, vage } = personne;
// vnom = "Dupont"    vprenom = "Jean"    vage= 45
```


ES6 : déstructuration / propagation

Mais il est aussi possible de :

→ échanger les valeurs de 2 variables

```
[un, deux] = [deux, un];
```

→ créer une variable avec le reste du tableau

```
let [un, …suite] = [1, 2, 3, 4];
console.log(un) ; // 1
console.log(suite) ; // [2, 3, 4]
```

→ Créer des variables que pour certains éléments :

```
let [un] = [1,2,3,4];
let [ , deux, , quatre ] = [1,2,3,4]
```


ES6 : déstructuration / propagation

- propager plusieurs bouts de tableau en un seul :

```
let un = [1,2];
let deux = [3, 4];
let trois = [0, …un, …deux, 5]; // [0,1,2,3,4,5]
```

- créer une copie d'un tableau

```
let tab1 = [3, 4];
Let tab2 = […tab1]
```


ES6 : boucles

- Ajout de for of

```
let couleurs = [ "bleu", "jaune", "rouge"];
// boucle sur les index des cases du tableau
for (let cle in couleurs) {
        console.log(cle); // --> 0, 1, 2
}
// boucle sur les valeurs des cases du tableau
for (let val of couleurs) {
        console.log(val) // --> "bleu", "jaune", "rouge"];
}
```

Les fonctions

ES6 : fonctions *(ES6)*

- Rappels fonctions :
  - avec des paramètres (non typés et obligatoires)
  - une valeur de retour non typée
  - l'instruction return définie la valeur de retour

```
function nomcomplet(prenom, nom) {
   return `${prenom}  ${nom}`;
}

let result1 = nomcomplet("Jean");                  // erreur
let result2 = nomcomplet("Jean", undefined);  // erreur
let result3 = nomcomplet("Jean", "Dupont", "Dr.");  // erreur
let result4 = nomcomplet("Jean", "Dupont");         // ok Jean Dupont
```


ES6 : fonctions *(ES6)*

- paramètre avec valeur pas défaut :

```
function nomcomplet(prenom, nom = "Durand") {
   return `${prenom}  ${nom}`;
}

let result1 = nomcomplet("Jean");                  // ok Jean Durand
let result2 = nomcomplet("Jean", undefined);  // ok Jean Durand
let result3 = nomcomplet("Jean", "Dupont", "Dr.");  // erreur
let result4 = nomcomplet("Jean", "Dupont");         // ok Jean Dupont
```


ES6 : fonctions flèches *(ES6)*

- Une expression de fonction flèche permet d'obtenir un code plus synthétique (plus court) qu'une fonction
- Une fonction flèche n'a pas de **this** spécifique, c'est-à-dire pas de contexte spécifique ; si le mot clé this est utilisé, il fera référence au contexte parent
- Une fonction flèche est souvent anonyme
- Etant anonymes, les fonctions flèches sont :
  - stockées dans des variables
  - utilisées en paramètre de fonctions

ES6 : fonctions flèches *(ES6)*

- Syntaxe d'une fonction flèche :

(param1, param 2) => { instructions }

```
(a, b) => {
    let c = a + b
    return c
}
```

(param1, param 2) => { return expression }

```
(a, b) => { return a + b }
```

et peut être abrégé en (param1, param 2) => expression

```
(a, b) => a + b
```

(param1) => { corps de la fonction } // un seul paramètre

peut être abrégé en param1 => { corps de la fonction }

exemple :

ES6 : fonctions flèches *(ES6)*

- Exemple 1 : stocker une fonction flèche dans une variable

```
const somme = (a, b) => { return a + b }
// utilisation comme une fonction « classique »
let resultat = somme(5, 10)
```

- Exemple 2 : utiliser une fonction flèche en paramètre
  on suppose que la méthode truc accepte une fonction en paramètre qui compare deux nombres et renvoie vrai si le 1er est plus grand que le 2ème , faux sinon

```
liste.truc ( function (n1, n2 ) {
 if ( n1 > n2 ) {
   return true
 } else {
   return false
 } } )
```

```
liste.truc ( (n1, n2 ) => {
    return n1 > n2
}}
```


Tableau et fonctions

- forEach : applique une fonction sur chaque elt du tableau ; le tableau n'est pas modifié

```
let T = [8, 3, 12];
T.forEach( (v) => console.log(v) )
```

équivalent à

```
for (let i=0; i<T.length ; i++) {
        console.log( T[i] );
}
```

équivalent à

```
for v of T {
        console.log( v );
}
```


Tableau et fonctions

- sort : tri les éléments du tableau

```
let T = [8, 3, 12];
// si pas de paramètre → tri croissant (mais sous forme de string)
T.sort( ) → [12, 3, 8] mais pas [3, 8, 12]
// avec en paramètre une fonction → tri personnalisé
// la fonction doit avoir 2 params v1 et v2
//       = les 2 vals à comparer
// elle doit renvoyer une valeur < 0 si v1 avant v2
//  > 0 si v1 après v2, 0 si inchangé
T.sort( (v1,v2) => v1 < v2 ? 1 : -1)
→ [12, 8, 3] → tri décroissant
```

Tableau et fonctions

- map: applique une fonction sur chaque elt du tableau et stocke le résultat dans le tableau renvoyé ;
- Le tableau résultat a la même taille que le tableau T
- exemple

```
let T = [8, 3, 12];
let T2 = T.map( (v) => v - 3 )  → [5, 0, 9]
```

équivalent à

```
let T2 = [ ];
for (let i=0; i < T.length ; i++) {
        T2[i] = T[i] - 3 ;
}
```


Tableau et fonctions

- reduce : applique la fonction sur chaque elt pour réduire le tableau à une seule valeur
- exemple

```
let T = [8, 3, 12];
let somme = T.reduce( (res, v) => res + v,  0)  → 23
```

équivalent à

```
let somme = 0;
for (let i=0; i < T.length ; i++) {
        somme = somme + T[i];
}
```


Tableau et fonctions

- filter : applique la fonction sur chaque elt pour réduire le tableau à un tableau qui ne contient que les valeurs vérifiant la condition
- exemple

```
let T = [8, 3, 12];
let inf10 = T.filter( (v) => v < 10)  → [8, 3]
```

équivalent à

```
let inf10 = [ ], j = 0 ;
for (let i=0; i < T.length ; i++) {
        if ( T[i] <10 ) { Inf10[j] = T2[i]; j = j +1}
}
```


Les classes

ES6 : classes

- les classes sont introduites pour 'normaliser' l'usage de la programmation objet en javascript (car la programmation par prototype est parfois déroutante) :

```
class Salutation {
   constructor(message) {  // constructeur (facultatif)
      this.texte = message;
   }
   saluer() {  // méthode
      return "Bonjour, " + this.texte;
   }
}
let salut = new Salutation("tout le monde");
salut.saluer();
```


ES6 : classes

Les propriétés : pour le moment, il n'est pas possible de les déclarer, on doit les ajouter à l'aide des méthodes ;
Par exemple, ajout de 2 propriétés avec le constructeur :

```
class Personne {
        constructor( pnom , page ) {
                this.nom = pnom;
                this.age = page;
        };
}
let dupont = new Personne("Dupont",43);
console.log(dupont);
```

```
Personne {
  nom : "Dupont"
  age : 43
}
```


ES6 : classes

- l'héritage est possible :

```
class Animal {
   constructor(theName) { this.name = theName; }
   move(distanceInMeters=0) {
      console.log(`${this.name} moved ${distanceInMeters}m.`);
   }
}
class Serpent extends Animal {
   constructor(name) { super(name); } // surcharge du constructeur
   move(distanceInMeters = 5) {    // surcharge de la méthode move
      console.log("Slithering...");
      super.move(distanceInMeters);  // appel de la méthode de la superclasse
   }
}
```


ES6 : classes

```
class Cheval extends Animal {
   constructor(name: string) { super(name); }
   move(distanceInMeters = 45) {
      console.log("Galloping...");
      super.move(distanceInMeters);
   }
}

let sam = new Serpent("Sammy le Python");
let tom: Animal = new Cheval("Tommy le Palomino");

sam.move(); // Slithering... Sammy le Python moved 5m.
tom.move(34); // Galloping...Tommy le Palomino moved 34m.
```

Modules

ES6 : modules

- Regrouper dans un fichier séparé monmodule.js plusieurs éléments de code : des variables, des fonctions, des classes,… un module => un fichier
- Par défaut tout est local au module donc pas visible de l'extérieur.
- Le mot clé export permet de préciser les éléments que l'on veut rendre visibles ; il y a deux syntaxes :
  - l'export nommé :

```
// soit directement devant l'élément à rendre visible
   export const PI = 3.1415;
// soit en 2 étapes :
   const PI : number = 3.1415;
   export { PI }
```


ES6 : modules

  - l'export par défaut export default permet de définir ce qui est exporté par défaut du module.
- Il peut y avoir plusieurs exports nommés mais un seul export par défaut

ES6 : modules

- Pour faire référence à un module : **import**
- Imports nommés en référence aux exports nommés *(les noms d'imports doivent être les mêmes que les noms d'export)* :

```
// module situé dans le même dossier
import { machin, bidule } from './monmodule'
// module d'une librairie Angular
import  { Component } from '@angular/core'
```

- Import en référence à l'export par défaut :

```
// module situé dans le même dossier
import truc  from './trucmodule'
```

- Import nommés et par défaut *(le par défaut tjs en 1er)* :

```
import truc, { machin, bidule } from './monmodule'
```

Asynchrone & Promesses

Callbacks et promesses

- ES6 propose la notion de promesse pour gérer l'asynchrone en JS
- Une promesse peut avoir plusieurs états :
  - en cours : la valeur n'est pas encore arrivée
  - résolue : la valeur est arrivée, on peut l'utiliser
  - rejetée : une erreur est survenue, on peut y réagir
- Une promesse possède 2 méthodes then et catch chacune avec une fonction callback en paramètre.
- Si then renvoie une promesse, il est alors possible de les chainer : .then().then().then().catch()

https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Utiliser_les_promesses

Enchainements d'appels asynchrones

```
faireQqc( function(result) {
   faireAutreChose( result, function(newResult) {
      faireUnTroisiemeTruc(newResult, function(finalResult) {
          console.log('Résultat final :' + finalResult);
       }, failureCallback);
   }, failureCallback);
}, failureCallback);
-- plus lisible avec des promesses et des fonctions flèches !
faireQqc()
 .then( result => faireAutreChose(result) )
 .then( newResult => faireUnTroisiemeTruc(newResult) )
 .then( finalResult => {
          console.log('Résultat final : ' + finalResult); } )
.catch( failureCallback );
```