Découverte des nouveautés de ES6

1] Préparation

T1.1 | A l'aide d'un IDE pour JavaScript, créez un projet vide et tapez le code d'une page `testJS.html` :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>Tests JS</title>
  </head>
  <body>
    <h1>Tests ES6 !</h1>
    <script src="./prems.js"></script>
  </body>
</html>
```

Note : *placer des scripts au bas de l'élément `<body>` améliore la vitesse d'affichage de la page, car même si le langage JavaScript est un langage interprété ligne par ligne, les navigateurs modernes utilisent la technologie* ***Just-In-Time*** *(JIT), qui compile le code JavaScript en code `bytec` au moment de son exécution. On réserve le placement des scripts dans l'élément `<head>` dans le cas d'un traitement à effectuer au moment du chargement de la page.*

T1.2 | Créez aussi le fichier JavaScript (vide) demandé dans le code de la page.

2] Préparation

- ***Hoisting***

En français « hissage », ce terme indique que le code JavaScript est lu une première fois et « hisse » les variables et les fonctions en haut du script. JavaScript met donc en mémoire les déclarations des variables et des fonctions avant d'exécuter un quelconque segment de code.

Tout d'abord, regardons le code suivant :

```
var a = 10;
var b;
console.log(a);      // affiche 10
console.log(b);      // affiche undefined (ce qui n'est pas en soi une erreur)
```

Le `hoisting` fait que ceci est valide, la variable `c` est déclarée mais initialisée après :

```
console.log(c);      // affiche undefined (ce qui n'est toujours pas en soi une erreur)
var c = 20;
```

T2.2 | Mettez les lignes de code dans `prems.js` et vérifiez le retour avec les outils de développement en ouvrant la page HTML par Chrome.

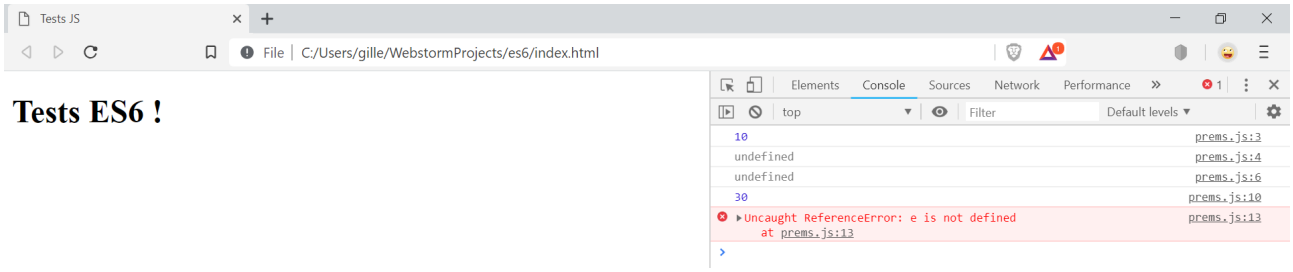
- ***Scope***

Le `scope` désigne la portée locale des variables uniquement entre des accolades, mais pour `var` cela s'applique uniquement dans une fonction.

```
if (true) { var d = 30; }          // exemple 1
console.log(d);                    // affiche 30

function test() { var e = 40; }    // exemple 2
console.log(e);              // affiche une erreur référence inconnue
```

Ceci doit donner :



- ***Fonction anonyme***

Une fonction anonyme se définit évidement comme une fonction sans nom… On l'utilise en général en affectation à une variable ou en paramètre d'une autre fonction, ce qui oblige bien sûr à ce que la fonction anonyme retourne obligatoirement un résultat.

Une fonction anonyme et auto-exécutable (ou IIFE, *Immediately-Invoked Function Expression*) est exécutée dès sa définition (voir https://developer.mozilla.org/fr/docs/Glossaire/IIFE).

En voici une représentation :

```
(function() { ici du code... })();
```

La bibliothèque **JQuery** par exemple l'utilise beaucoup.

- ***Fonction fléchée***

Une fonction fléchée est une nouveauté ES6 et allège le code quand les instructions ne nécessitent pas beaucoup de traitement.

Elles aussi sont anonymes et sont plus destinées à faciliter l'appel aux *callbacks* (c'est-à-dire des fonctions passées en paramètres d'autres fonctions et plutôt destinées au retour d'un appel asynchrone), même si on peut les affecter à une variable.

Ces fonctions renvoient toujours un résultat, mais le `return` est implicite.

Exemple sans fonction fléchée :

```
var maFonction = function(x) { return x + 1; };
```

Exemple en ES6 avec fonction fléchée :

```
const maFonction = x => { return x + 1; };
```

T2.3 | Mettez les lignes de code ci-dessous dans `testJS.html` après la balise `<h1>` :

```
<input id="nombre" type="text" value="0" />
<input id="leBouton" type="button" value="Click !" />
```

T2.4 | Voici l'exemple dans le cadre d'un ajout d'évènement de type `onClick` à mettre cette fois dans `prems.js` en lieu et place de ce qui y avait avant :

```
window.onLoad = function() {
 let b = document.getElementById("leBouton");
 // retourne dans b l'élément identifié
 b.addEventListener('click', () => {
  // ajoute l'écouteur d'évènements click au bouton
  let i = document.getElementsByTagName("input")[0];
  // retourne dans i le premier élément de input
  i.value = Number(i.value) + 1;
  // ajoute 1 à la valeur du contenu
```

```
  });
  // donc vous avez compris qu'il s'agit d'un compteur de clics…
}
```

T2.5 | Testez le script…

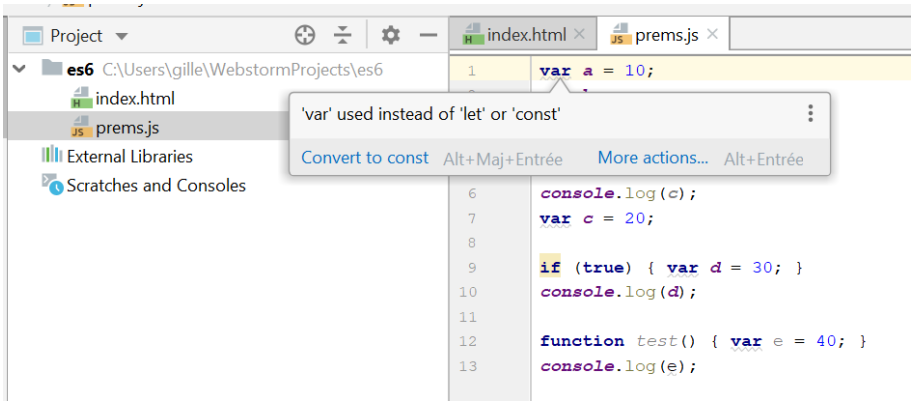
Note : *les fonctions fléchées n'ont pas de `this` ; elles partagent le même `this` lexical que leur scope parent.*

Lien pour une présentation plus complète :
https://maurice-chavelli.developpez.com/tutoriels/nouveautes-es6/variables-et-fonctions/

3] Les bons usages

- ***`let` et `const`***

Il vaut mieux avec ES6 utiliser `let` et `const` que `var`, car celui-ci apporte des comportements particuliers. D'ailleurs WebStorm vous le signale :



Théoriquement, le meilleur usage possible de `var` serait la déclaration d'une variable globale au script et placée au tout début.

Exemples pour `let` :

```
let v = 10;
let v = 5;            // déclenche une erreur, car on ne peut pas réaffecter avec let à
                      // nouveau
```

```
let v = 10;
v = 5;                // C'est possible : v vaut 5 maintenant
console.log(v);
```

```
let g;                // déclaration sans valeur
g = "coucou !";
console.log(g);       // affiche coucou !
```

```
console.log(h);
let h = 10;           // déclenche une erreur à cause du hoisting (moins permissif que var)
```

Note : *un `let` n'est pas accessible en dehors d'un bloc.*

Exemples pour `const` :

```
const x = 10;
const x = 5;   // déclenche une erreur, car on ne peut pas réaffecter avec const à nouveau
x = 5;         // déclenche toujours une erreur, différent de let donc
```

```
const t = [];        // possible car on déclare un tableau vide
t.push(9);
t.push(12);
console.log(t);      // affiche Array[9,12]
```

```
const objet = { nom : "toto"; };
objet.nom = "titi";  // cela fonctionne
```

```
const x;             // erreur car on a l'obligation de mettre une valeur avec const
```

Note : *les règles de* `hoisting` *et de portée de bloc sont identiques à* `let`*.*

- ***Le parcours d'un tableau avec*** `for...of`

Oubliez le `foreach`, maintenant le parcours d'un tableau ce fait comme cela :

```
const tableau = ["a", "b", "c", "d"];

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

- ***Les objets***

« JavaScript peut prêter à confusion lorsqu'on est habitué à manipuler des langages de programmation manipulant les classes (tels que Java ou C++). En effet, JavaScript est un langage dynamique et ne possède pas de concept de classe à part entière (le mot-clé `class` *a certes été ajouté avec ES2015 mais il s'agit uniquement d'une pseudo syntaxe, JavaScript continue de reposer sur l'héritage prototypique).*

En ce qui concerne l'héritage, JavaScript n'utilise qu'une seule structure : les objets. ***Chaque objet possède une propriété privée qui contient un lien vers un autre objet appelé le prototype. Ce prototype possède également son prototype et ainsi de suite, jusqu'à ce qu'un objet ait*** `null` ***comme prototype. Par définition,*** `null` ***ne possède pas de prototype et est ainsi le dernier maillon de la chaîne de prototype.*** *»*

Cette présentation vient de la page :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Héritage_et_chaîne_de_prototypes

L'écriture d'un objet change avec ES6 et est plus proche de la rédaction classique, voici un exemple :

```
// Définition de l'objet Animal
class Animal {

  constructor(nom) {
    this.name = nom;
  }

  parler() {
    console.log(`${this.name} fait du bruit.`);
  }

}

const animal = new Animal('animal');
animal.parler();     // affiche animal fait du bruit.
```

L'objet `Animal` hérite du prototype `Object`, lui-même ayant `null` comme lien d'héritage.

Et un exemple d'héritage :

```
// Définition de l'objet Lion héritant de Animal
class Lion extends Animal {

  parler() {
    super.parler();
    console.log(`${this.name} rugit.`);
  }

}

const lion = new Lion('Simba');
lion.parler();       // affiche 'Simba fait du bruit.' et 'Simba rugit.'
```

T3.1 | Faites en sorte de démontrer les lignes de code ci-dessous.