JS rappel des fondamentaux

Sommaire

Introduction & histoire de JS

JS première version 1995, auteur Brendan Eich.

Rappelons que JS est un langage interprété dont le typage est faible. Mais attention, cela ne veut pas dire que JS ne définit pas un type à ses variables.

Un typage faible permet les convertions de type implicite :

```
let foo = 1 + "2";
```

Et JS a un typage dynamique, le type est déterminé à l'exécution :

```
// le type de a sera défini à l'exécution de cette ligne
l       l;
```

Un langage interprété utilise le code source pour le compiler puis l'exécuter, il n'y a pas de création d'exécutable définitif. Pour chaque exécution, JS repartira du code source.

Le moteur JS de compilation peut tagguer du code qui se répète et créer, pour ces parties uniquement, un exécutable "définitif".

JS est un langage de script orienté objet.

JS suit la norme ECMAScript, standard que suivent certains langages de script comme Javascript. Cette norme évolue en permanence. Les principaux navigateurs Web mettent à jour leur moteur d'exécution pour suivre les évolutions de ce langage.

Une version majeure d'ECMAScript est celle qui a été définie en 2015 : ES2015 que l'on appelle ES6. Le nom de la version étant déterminé par la dernière version du standard en cours donc ES6 pour 2015. Aujourd'hui la dernière version officielle est EMACScript 2020.

Notion de type en JS

Bien que JS soit un langage faiblement typé, JS type toutes ses variables.

Le type d'une variable en JS est déterminé lorsqu'on la définit et qu'on lui assigne une valeur particulière. Ce dernier peut changer si on ré-assigne à la variable une valeur d'un autre type.

```
let n = 10;
console.log(typeof n); // number

// ré-assignation
n = "Hello";

console.log(typeof n); // string
```

Dans l'exemple ci-dessus le type de la variable n a changé; il est passé du type number au type string (par ré-assignation).

Notons que lorsque vous définissez une variable sans lui affecter une valeur particulière, celle-ci est de type "undefined" :

```
let username;
console.log(typeof username); // undefined
```

Les différents types en JS

On distingue les types suivants en Javascript. Attention, tous les types primitifs définissent des valeurs non modifiables (immuables).

Types primitifs

- boolean

```
// On ne peut pas modifier un "true" ...
let flag = true;
```

- null

```
// On ne peut pas modifier un "null" ...
let flag = null;
```

- undefined
- number
- bigInt (big integer)

Il faut ajouter l'opérateur n pour définir des BigInt.

```
const x = 2n ** 100n;
console.log(x);
// 1267650600228229401496703205376n
```

- string

```
let message = "Hello World";
```

- Symbole (type introduit à partir de la norme ES6, que nous n'aborderons pas dans ce cours).

Les types Objects

Ils sont mutables, on peut modifier la valeur d'un objet. Un objet est une valeur conservée en mémoire à l'aide d'une référence unique.

Dans la liste des objets vous avez :

- Les objets classiques : les classes et les fonctions.

```
class Model {

  get() {
    return "table";
  }
}

// Création d'un instance (objet)
const myModel = new Model();

function modelFunc(n) {
  let name = n;

  return name;
}
```

- Les objets natifs comme les dates

```
const now = new Date();
```

- Les collections comme les tableaux, Map, Set, ...

Les déclarations suivantes pour un tableau sont identiques :

```
let notes = [1, 2, 3];
let newNotes = new Array(1, 2, 4);
```

Un Map est une simple collection de clés/valeurs.

```
// création d'un Map
const store = new Map();

store.set("A1", 10.6);
store.set("A2", 8.9);

console.log(store);
// {"A1" => 10.6, "A2" => 8.9}

const ensemble = new Set([1, 2, 3, 4, 5, 5]);
console.log(ensemble);
// [1, 2, 3, 4, 5] il n'y a pas de doublon
```

- Les JSON Javascript Object Notation

Ce qui est considéré comme faux en JS

0, NaN, undefined, false, "", '', ``, null

Notez que tout le reste n'est pas considérer comme faux. Tout ce qui a une valeur est donc considérée comme vraie.

Evaluations courcircuit

- Dans le cas où user n'est pas défini avec le connecteur ET

```
false && user
```

- Avec un OU

```
t       ıser
```

Les chaînes de caractères Interpolation

Vous pouvez écrire des chaînes de caractères sur plusieurs lignes et insérer des expressions JS qui seront évaluées à l'aide de backquotes (accent grave).

Exemple

```
let a = 51;
let b = 90;
console.log("Somme " + (a + b) + " et\n multiplication " + a * b + ".");
```

Avec les backquotes on aura une expression plus facile à écrire :

```
let a = 51;
let b = 90;
console.log(`Somme : ${a + b} et \n multiplication : ${a * b}.`);
```

Exemple avec une expression JS :

```
let isLoading = true;
const message = `Data is ${isLoading ? 'loading...' : 'done!'}`;
```

Remarque sur la syntaxe ternaire, pour écrire une condition sur une ligne :

```
console.log( true ? 'yes' : 'no'; ); // yes
console.log( false ? 'yes' : 'no'; ); // no
```

Les ternaires sont également très pratiques pour assigner des valeurs avec une condition :

```
logged = true ? 'yes' : 'no'; ; // yes

logeed =  false ? 'yes' : 'no'; ; // no
```

Vous pouvez enchaîner les ternaires mais, attention à la lisibilité.

```
logged = true ? ( true ? 'toujours yes' : 'no' )  : 'no'; ; // toujours yes
```

Portée (ou scope en Anglais) des variables en JS

Définition let : La variable définie avec let a une portée scopée au niveau du bloc dans lequel elle a été déclarée.

Remarque importante : lorsque vous définissez une variable à l'intérieur d'une fonction elle est scopée (portée) dans la fonction elle-même; elle n'a pas d'effet de bord avec le reste du script.

```
function foo() {
  let a = 10;
  console.log(a); // affiche 10
}

foo();

// ReferenceError
console.log(a);
```

Si vous définissez une variable de même nom à l'extérieur de la fonction, alors elle n'aura pas d'effet sur la variable définie à l'intérieur de la fonction foo :

```
let a = 11;

function foo() {
  let a = 10;
  console.log(a);
}

// affiche 10
foo();

// affiche 11
console.log(a);
```

Remonter des scopes

JS cherche la définition de ses variables dans le scope courant et sinon il remonte les scopes. Si la variable n'est définie dans aucun des scopes, alors une erreur ReferenceError est levée.

```
// bloc courant pour b
let b = 11;

function baz() {
  // bloc courant pour c
  let c = 9;

  // JS ne trouve pas b dans le bloc courant => il remonte les scopes
  console.log(b, c);
}

// affiche 11 9
baz();
```

Un autre exemple :

```
// bloc courant
let b = 11;

function baz() {
  console.log(b);

  function foo() {
    console.log("Valeur du symbole b : ", b);
  }

  foo();
}

// affiche 11
baz();
```

01 Exercice scope calcul (sans coder)

Est ce que le code qui suit vous semble correcte ? Répondre sans exécuter le code. Si ce dernier n'est pas valide modifiez-le afin qu'il puisse s'exécuter correctement.

```
let a = 1;

function calcul() {
  let a = 2 + a;

  console.log(a, "calcul");

  function sub_calcul() {
    let b = a + 1;

    console.log(b, "sub_calcul");
  }

  sub_calcul();
}

calcul();
```

02 Exercice TDZ (temporal dead zone) (sans coder)

Est ce que le code qui suit vous semble correcte ? Répondez sans exécuter le code.

```
function tdz() {
  console.log(tdz_val);

  let tdz_val = "Temporal Dead Zone";
}

tdz();
```

03 Exercice for let (sans coder)

Est ce que le code qui suit vous semble correcte ? Répondez sans exécuter le code.

```
let i = 100;

for (let i = 0; i < 10; i++) {
  console.log(i);
}

console.log(i);
```

Est ce que le code qui suit vous semble correcte ? Répondez sans exécuter le code.

Si ce code est valide qu'affichera-t-il ?

```
for (let j = 0; j < 10; j++) {}
console.log(j);
```

Déclaration d'une constante

Définition : La variable définie avec const a une portée scopée au niveau du bloc dans lequel elle a été déclarée.

Le mot réservé du langage JS const permet de définir une constante à assignation unique. Notez que vous êtes obligé de lui donner une valeur lors de sa définition. Une constante ne peut être re-définie.

```
const API_KEY = "ABf#123@";
console.log(API_KEY);
```

Une constante peut contenir tous types de variable. Dans le cas d'un objet comme un tableau par exemple, les valeurs du tableau sont modifiables. En effet, une constante bloque la ré-assignation de la variable, mais ne rend pas l'objet non-modifiable.

```
const STUDENTS = ["Alan", "Bernard", "Jean"];

STUDENTS.push("Sophie");

console.log(STUDENTS);
//["Alan", "Bernard", "Jean", "Sophie"]

STUDENTS.pop();

console.log(STUDENTS);
// ["Alan", "Bernard", "Jean"]
```

Par contre ce qui suit est impossible, l'erreur suivante sera levée : TypeError: Assignment to constant variable.

```
let newStudents = ["Alice"];
// re-assignation impossible
STUDENTS = newStudents;
```

04 Exercice const & for

1. Pouvez-vous utiliser à votre avis le mot réservé const dans la boucle suivante ?

```
for (const j = 0; j < 10; j = j + 1) {}
```

2. Utilisez la syntaxe de boucle for of et const sur l'itérable STUDENTS et affichez (console.log) ses valeurs :

```
const STUDENTS = ["Alan", "Bernard", "Jean"];
```

var définition obsolète !

Ce mot clé pour définir une variable ne doit plus être utilisé, utilisez let à la place.

Il permet de définir une variable globale ou locale à une fonction sans distinction de bloc :

```
function foo() {
  var x = 10; // portée fonction pas bloc comme let
  if (true) {
    var x = 2;  // c'est la même variable !
    console.log(x);  // 2
  }
  console.log(x);  // 2
}
foo(); // 2 2
```

Par comparaison avec le mot clé let :

```
function bar() {
  let x = 10; // portée fonction pas bloc comme let
  if (true) {
    let x = 2;  // c'est la même variable !
    console.log(x);  // 2
  }
  console.log(x);  // 10
}
bar(); //  2 10
```

Introduction à la notion de fonction

Une fonction en JS est un objet.

Paramètres facultatif

```
function add(a, sup = 1) {
  return a + sup;
}

add(10); // affiche 11

add(10, 0); // affiche 10
```

05 Exercice ttc

1. Créez une fonction qui permet de calculer un prix TTC connaissant un prix HT. Donnez une valeur de 20% par défaut pour la TVA.
2. Vérifiez que le type des variables passées en paramètre ne posent pas de problème. Utilisez parseFloat pour la vérification des types. Affichez les résultats avec au plus 2 chiffres après la virgule.

```


// 1.
ttc(100, 0.2); // 120
ttc(100.50, 0.2); // 144.72

// 2.
// Gestion du type
ttc("hello", 0.2); // Erreur de type
ttc(100.50, "hello"); // Erreur de type
ttc("100", ".3"); // 130
```

Syntaxe par décomposition

Si vous avez une fonction avec de nombreux paramètres ou des paramètres variables, utilisez le spread operator pour passer les valeurs à la fonction :

```
function sum(x, y, z) {
  return x + y + z;
}

let numbers = [1, 2, 3];

sum(...numbers); // sum(1, 2, 3) unpacking
```

06 Exercice fonction ttc spread operator

Ecrivez une fonction sumTTC qui prend 3 nombres arbitraires de prix HT et retourne la somme de ces prix TTC. La TVA est un paramètre facultatif (20%). Vérifiez que le type des variables passées en paramètre ne posent pas de problème, utilisez parseFloat. Affichez les résultats avec au plus 2 chiffres après la virgule.

Les prix HT seront donnés dans un tableau :

```
const priceHT = [100.50, 200.8, 55.7];

console.log(sumTTC(...priceHT));
console.log(sumTTC(...priceHT, .3));

// vérifiez le type des variables
const badPriceHT = [100.50, "hello", 55.7];
console.log(sumTTC(...badPriceHT, .3));
```

littéral pour définir des paramètres

Vous pouvez utiliser la syntaxe suivante pour définir les paramètres d'une fonction. Dans ce cas vous n'avez pas à vous soucier de l'ordre des paramètres passé à la fonction.

```
function baz({ a, b }){
  console.log(a, b )
}

baz({ a: 1, b : 2}); // 1 2
baz({ b: 2, a : 1}); // 1 2
```

this DANS LE CONTEXTE DE L'APPEL d'une fonction sur un objet

Le this d'un objet est déterminé par la manière dont vous allez appeler l'objet "contexte".

L'objet sur lequel vous **appelez** la fonction détermine le this :

objet.my_function()

```
'use strict';

const o1 = {
    f1 : function(){

      return this;
    }
}

console.log(o1.f1()) ; // this de o1

const o2 = {
    f2 : o1.f1
}

console.log(o2.f2()) ; // this de o2

const o3 = o1.f1;

console.log(o3()) ; // undefined car on n'appelle la fonction f1 explicitement
```

De même, faites attention dans les fonctions de callback. Dans l'exemple qui suit setTimeout fera appel à la fonction sans reprendre le context de l'objet lui-même, this sera, en mode strict, undefined :

```
'use strict';

setTimeout(o1.f1, 1000); // ici setTimeout appel la fonction f1.
```

Pour corriger ce problème il faut écrire :

```
setTimeout(() => o1.f1() , 1000); // ici setTimeout appel la fonction f1.
```

Déclaration de fonction

En JS vous avez des fonctions déclarées et des expressions de fonction.

Notez que

- fonction déclarée :

```
function foo(){

}
```

- Expression de fonction

```
setTimeout( function (){

})
```

07 Exercice function & expression

Sc uter le code.

Nommez les types de fonction ci-dessous :

```
const myFunc = function(){

  function bar(){
    // ...
  }
}
```

Les fonctions déclarées sont définies dès le début du script ou de la fonction qui la contient.

Les expressions de fonction sont définies après leur évaluation.

08 Exercice déclaration d'une fonction

Sans exécuter le code.

1. Le code suivant est-il valide ?

```
bar();

function bar(){
  console.log("bar");
}
```

2. Le code suivant est-il valide ?

```
myFunc();

const myFunc = function(){
    console.log("Expression");
}
```

09 Créez une fonction zip

La fonction zip prend en paramètre deux tableaux de même dimension et crée des couples de 2 éléments terme à terme, et retourne un tableau des couples.

```
zip([1,2], [3, 4]);

//[[1,3], [2, 4]]
```

10 Objet add

Créez un objet Add qui additionne soit deux entiers soit une liste de nombre(s), voyez un exemple d'utilisation ci-dessus :

```
Add.a = 10;
Add.b = 20;

Add.sum(); // 30

Add.numbers = [1, 2, 4];

Add.sum(); // 37

Add.reset()

Add.numbers = [1, 2, 4];

Add.sum(); // 7
```

11 Exercice de synthèse corrigé un effet de bord

Comment éviter l'effet de bord sur la propriété this (undefined) dans le code suivant? Proposez une solution.

```
const log = {
    count : 100,
    save: function () {
        'use strict';
        console.log(this.count);
    }
}
setTimeout(log.save, 500);
```

L'objet arguments et paramètres d'une fonction

Vous n'êtes pas obligé de renseigner le nombre d'argument(s) d'une fonction en JS. La fonction possède en interne une propriété **arguments** qui récupère les paramètres de la fonction, attention arguments n'est pas un tableau :

```
function sum(){
  let total = 0;
  for(let i =0; i < arguments.length; ++i ) total += arguments[i];

  return total;
}

console.log(sum(1,2,3,4, 5, 6));
```

L'objet arguments peut-être converti en tableau à l'aide de la méthode from de l'objet Array :

```
const args = Array.from(arguments);
```

On peut par exemple définir la fonction sum en utilisant la méthode from :

```
function sum(){
  const args = Array.from(arguments);

  return args.reduce( (acc, curr) => acc + curr );
}

console.log( sum(1,2,3,5) ); // 11
```

Les fonctions fléchées

Les fonctions fléchées (arrow function) permettent d'avoir une syntaxe plus courte pour définir facilement des fonctions de rappel. On les utilise dans les fonctions JS telles que map, reduce, filter, ...

```
const power2 = (x) => {
  return x ** 2 ;
};
const numbers = [1, 2, 5];
console.log(numbers.map( power2 ));
// [1, 4, 25]
```

Si vous ne retournez qu'une seule valeur, vous pouvez écrire la syntaxe suivante :

```
// syntaxe consise
const sum = (x, y) => x + y ;
```

Dans le cas où vous souhaiteriez retourner un unique littéral, utilisez la syntaxe suivante (expression) :

```
// syntaxe consise
const model = (x, y) => ({ x, y }) ;
console.log(model(1,2)); // retournera { x : 1, y : 2 }

// Une syntaxe plus longue mais équivalente
const model2 = (x, y) => {
    return { x : x, y : y }
}
```

Contrairement aux fonctions classiques, les fonctions fléchées ne re-définissent pas de this. Si vous vous référez dans une fonction fléchée au mot clé this, la fonction fléchée **récupérera le this du contexte** dans lequel elle a été définie.

Testez l'exemple suivant dans un fichier :

```
const School = {
    name: "Alan",
    sayHello() {
        // récupérer le this du context
        const that = this;
        function getName() {
            console.log(that.name); // Alan
            console.log(this.name); // undefined
        }
        getName();
    },

    sayHelloArrowFunc(){
        // La fonction fléchée récupère le context de l'objet courant School
        let func = () => {
            console.log(this.name); // Alan
        }
        func();
    }
}
School.sayHello();
School.sayHelloArrowFunc();
```

Modifié le: vendredi 22 septembre 2023, 13:01