6 ERREURS

A EVITER EN JS

BRYAN PARASOTE & AXEL PARIS

Sommaire

1. Mauvais scope de variable

Selon vous, que va afficher le deuxième console.log ?

```
for (var i = 0; i < 5; i++) {
   console.log("Inside : " + i)
}
console.log("Outside : " + i)
```

Si vous avez répondu “Outside : undefined”, alors vous n’avez pas encore bien compris le scope en JavaScript : la bonne réponse étant “Outside : 5”.

Rappelons d’abord ce qu’est un scope : le scope correspond à une portée, cela peut être la portée d’une variable ou bien celle d’une fonction. Cette portée permet de savoir où est-ce que notre variable (ou fonction) est utilisable dans notre code, et le scope est défini principalement par l’emplacement où l’on définit notre objet (sauf dans notre cas).

Ici, le problème vient donc de la définition de la variable, nous l’avons défini avec le mot-clé “var” et celui-ci met automatique en place un scope globale pour la variable créée, c’est-à-dire que notre variable **i** sera disponible partout, quelque soit l’emplacement de sa définition. On pourra donc toujours y accéder après le block for, d’où le “Outside : 10”.

Mais alors comment faire en sorte que le scope de la variable **i** soit défini uniquement dans le for ? Ce n’est pas si compliqué, et c’est grâce à l’ES6 que nous allons pouvoir faire ça. Il existe maintenant un nouveau mot-clé pour la définition de variable : ***“let”***.
Let permet de scoper une variable uniquement dans le block (et les sous-blocks) où elle a été définie, nous allons donc modifier notre code et voir que le second console.log n’affiche plus la même chose :

```
for (let i = 0; i < 5; i++) {
   console.log("Inside : " + i)
}
console.log("Outside : " + i)
```

Il n'affiche même plus rien, car il renvoie une erreur du type *"Uncaught ReferenceError: i is not defined"*. On comprend donc que la variable **i** n'est pas définie, son scope est donc dans la boucle for, et non en dehors de celle-ci.

2. Référence incorrecte à “this”

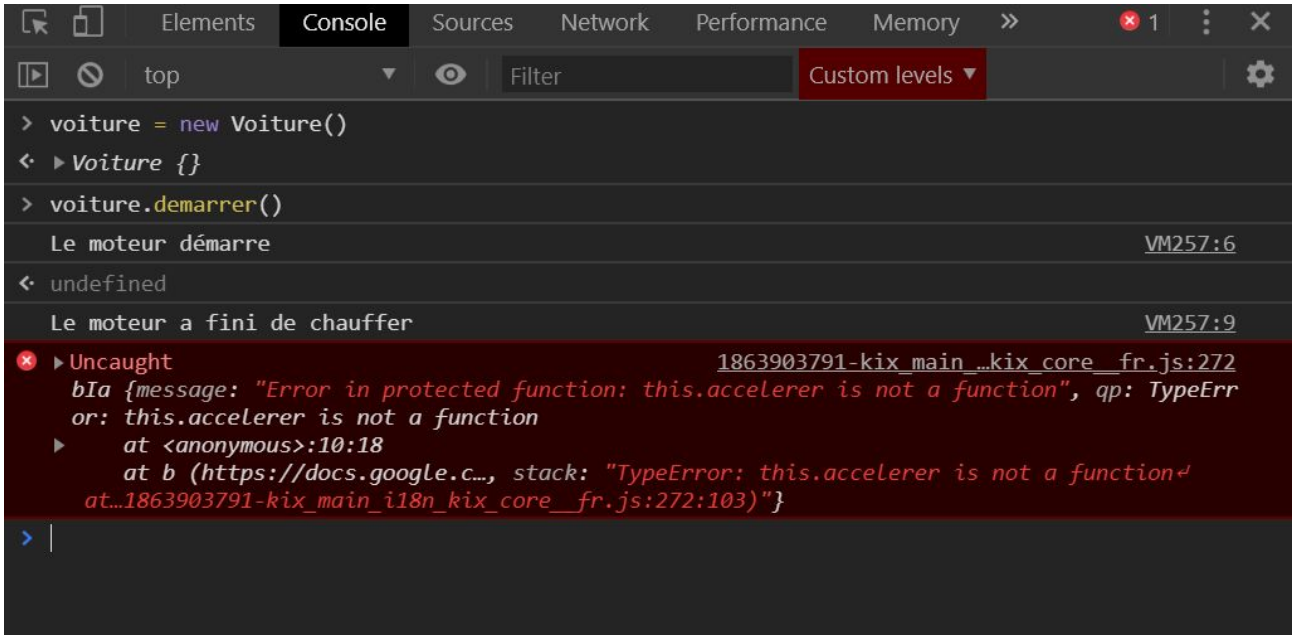
Vous voyez ce ***this*** ? Ce fameux ***this*** dont on va parler ici...
Est-ce que ça vous est déjà arrivé d'utiliser une méthode ou une propriété sur l'élément this et d'avoir une erreur alors que vous êtes certain qu'elle existe belle et bien ?

Prenons l'exemple suivant (ES6) :

```
class Voiture {
    accelerer(){
        console.log("Vroum !")
    }
    demarrer(){
        console.log("Le moteur démarre")

        setTimeout(function() {
            console.log("Le moteur a fini de chauffer")
            this.accelerer()
        }, 2000)
    }
}
```

Voici le résultat que j'ai en exécutant la méthode demarrer()



Vous comprenez pourquoi j'ai cette erreur ?
Cela vient du fait que this ne fait plus référence à mon objet Voiture ! (dans cet exemple précis, this fait référence à la fonction passée en paramètre au setTimeOut()).
Et vous devez vous dire ? "*Hm Okay, mais comment je fais maintenant pour régler ce problème ?*"

Solution 1 : Variable self

La 1ère méthode pour résoudre ce problème consiste à déclarer une nouvelle variable qui sera égale à notre ***this*** (faisant référence à notre objet) et ensuite on pourra l'utiliser !

Exemple :

```
// ...
demarrer() {
   console.log("Le moteur démarre")

   let self = this

   setTimeout(function () {
       console.log("Le moteur a fini de chauffer")
       self.accelerer() // On utilise self !
       // this peut être utilisé si on veut faire référence à notre
fonction
   }, 2000)
}
//...
```

Solution 2 : Fonction fléchée

C'est ma solution préférée ! (car je la trouve plus "propre")
On va utiliser les fonctions fléchées (arrow functions), c'est une syntaxe un peu bizarre si vous ne l'avez jamais utilisé.
En savoir plus sur les fonctions fléchées >

Et l'avantage de cette nouvelle syntaxe, c'est que **dans une fonction fléchée, il n'y a pas de this !**

Kesako ? ça veut dire que notre this précédemment utilisé ne va pas être “écrasé” par la création d’un nouveau this.

Exemple :

```
// ...
demarrer() {
   console.log("Le moteur démarre")

   setTimeout( () => {
       console.log("Le moteur a fini de chauffer")
       this.accelerer() // this fait référence à l'objet !
   }, 2000)
}
//...
```

3. Confusion dans les égalités

Il existe deux types d'égalité en Javascript (et également dans d'autres langages), et cela peut souvent amener à de mauvaises conditions, voici donc plus de détails concernant ces égalités :

- L'égalité simple ou == : Cette égalité converti les deux comparés dans le même type, et effectue ensuite une égalité stricte
- L'égalité stricte ou === : Cette égalité renvoie true uniquement si les deux comparés sont strictement égale (donc même type et même valeur).

Voici quelques comportements de JavaScript à connaître :

```
// Toutes ces conditions renvoient 'true'
console.log(false == '0');
console.log(null == undefined);
console.log(" \t\r\n" == 0);
console.log('' == 0);

// Et celles-ci aussi !
if ({}) // ...
if ([]) // ...
```

Concernant les deux derniers, on pourrait penser que cela renvoie 'false' car ces deux objets sont vides, cependant ce n'est pas le cas: tous les objets renverront 'true' dans une condition simplement parce que ce sont des objets, c'est une des particularités du JavaScript.

Pour finir sur les égalités parlons de 'NaN', il est important de savoir que comparer NaN à n'importes quelles autres variables (y compris NaN lui-même) renverra 'false'. Le seul moyen d'inclure NaN à une condition est d'utiliser la fonction globale isNaN() :

```
console.log(NaN == NaN);    // false
console.log(NaN === NaN);   // false
console.log(isNaN(NaN));    // true
```

4. Définition de fonctions dans une boucle

Il vous est peut-être déjà arrivé de créer des fonctions à l'aide d'une boucle et d'un index (par exemple pour un event au click sur un groupe d'éléments). Et vous pourriez remarquer un problème, regardez ce code et demandez-vous ce que va afficher chaque console.log :

```
for (var i = 0; i < 3; i++) {
   setTimeout(function () {
       console.log('Valeur de i : ' + i)
   }, 1000)
}
```

Si vous pensez qu'il va y avoir une succession de 0, 1 puis 2 alors vous avez tort. Il va y avoir le même texte à chaque fois : "Valeur de i : 3".
Mais alors pourquoi ? Car i est une variable globale (déclarée avec 'var'), donc à l'instant où il y a une itération de la boucle, elle s'incrémente. Chaque itération consiste à exécuter un console.log 1 seconde plus tard, et c'est en moins d'une seconde que la boucle s'arrête et que i vaut 3, c'est pourquoi dès la première exécution du console.log, i vaut déjà 3 et cela affiche donc "Valeur de i : 3".
On peut comprendre qu'il n'y a qu'une seule variable i et que sa valeur n'est pas "bloquée" pour le setTimeout.

Il y a plusieurs solutions à ce problème, la première (et la plus simple), consiste à remplacer le 'var' par un 'let', et dans ce cas la variable i sera bloqué pour chaque setTimeout dans un scope spécifique. Cette solution est très simple, mais elle ne fonctionne qu'au minimum avec l'ES6.

```
for (let i = 0; i < 3; i++) {
   setTimeout(() => {
       console.log('Valeur de i : ' + i)
   }, 1000)
}
```

Il existe également une solution sans l'ES6, qui est plus complexe mais tout aussi fonctionnelle. Le principe est d'encapsuler le contenu de la boucle dans une fonction, que ce soit une fonction nommée ou bien une fonction anonyme :

```
// Fonction nommée (sans ES6)

for (var i = 0; i < 3; i++) {
   setFuncWithIndex(i)
}

function setFuncWithIndex(index) {
   setTimeout(function () {
       console.log('Valeur de i : ' + index)
   }, 1000)
}


// Fonction anonyme (sans ES6)
for (var i = 0; i < 3; i++) {
   (function (index) {
       setTimeout(function () {
           console.log('Valeur de i : ' + index)
       }, 1000)
   })(i)
}
```

Le fait d'encapsuler dans une fonction permet d'utiliser non pas la variable i dans le console.log, mais la variable 'index' qui est l'argument de la nouvelle fonction, c'est donc une nouvelle variable qui ne sera pas atteinte par les itérations de la boucle.

5. Bien utiliser une boucle for

Lorsque l'on utilise une boucle for pour parcourir un tableau, on peut souvent voir ce code :

```
let myArray = ['item1', 'item2', 'item3']

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

Ici nous avons un tableau qui reste fixe, aucun ajout est fait : c'est notre tableau final. On peut se dire qu'il n'y a aucun problème dans ce code, et c'est vrai, il n'y a aucun problème.
Par contre il y a de l'optimisation possible, car en se mettant à la place de la machine, on peut vite comprendre qu'à chaque itération, un calcul est fait, le même calcul est refait : myArray.length. "myArray.length" n'est pas une constante, c'est une méthode qui s'exécute à chaque fois qu'on l'appelle. Cela signifie donc que l'on a le même calcul à chaque itération, d'où l'optimisation possible.

La solution est très simple, mettre ce calcul dans une variable et placer celle-ci dans la condition de la boucle :

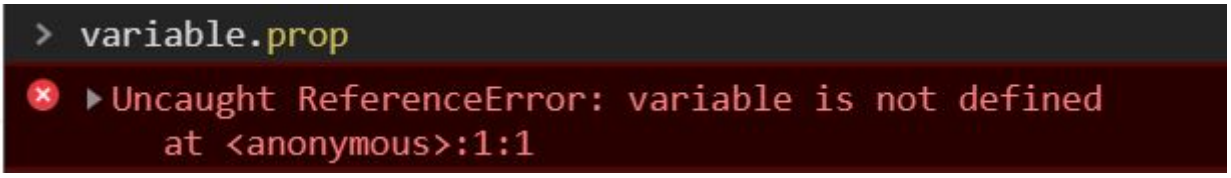
```
let myArray = ['item1', 'item2', 'item3'],
    myArrayLength = myArray.length

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

Pour un tableau à 3 items l'optimisation n'aura pas beaucoup de sens, mais passer plusieurs dizaines d'items, son intérêt est révélé par rapport au temps d'exécution du programme qui se réduit (le but d'une optimisation).

6. Uncaught ReferenceError: X is not defined

Quand on développe en JavaScript, on a tous déjà (et sans exception) eu cette fameuse erreur lorsqu'on essaie d'accéder à une propriété :



Et résoudre ce type d'erreur c'est vraiment facile mais il y a encore trop de monde qui n'y arrive pas...
Il faut juste LIRE l'erreur ! Encore beaucoup trop de débutant se contente de se dire "*Ah zut ça marche pas... je comprends pas pourquoi :/*"
Si on la traduit, voici ce qu'on a ***"La variable variable n'est pas définie"*** c'est aussi simple que ça !
Tout ce qu'on a faire c'est de se demander : "*Mais pourquoi variable n'est pas définie ? Est-ce que j'ai oublié de la déclarée ? Est-ce que je ne confonds pas avec une autre variable ?*"

Une fois que vous avez répondu à ces questions, il ne vous reste plus qu'à résoudre le problème ! Dans le cas de l'exemple, j'ajoute simplement ce code :

```
let variable = {
   prop: 'Je suis une chaine de caractère :)'
}
```

Et voilà le résultat !

Conclusion

Nous venons donc de voir 6 erreurs assez courantes en JavaScript, nous espérons vraiment que cet ebook vous aura été utile !

Si vous voulez continuer votre apprentissage dans JavaScript, nous vous invitons à joindre notre club : **le club JS Master !** Celui-ci vous permettra de faire parti d'une communauté de de développeurs passionnés par JavaScript ! De plus, vous aurez accès à deux projets complet en vidéos par mois (avec le code source fourni) que l'on appelle TP.
Ce club comporte encore plus d'avantages, vous pouvez voir tout cela sur cette page : **Club JS Master**.

Pour finir nous vous souhaitons un bon apprentissage du JavaScript sur le web, et à bientôt avec *DevTheory* !