Functions in JS

Définition d'une fonction récursive

Une fonction récursive

Une fonction qui va, durant son traitement, faire appel a elle même

- Il est parfois plus facile de résoudre un problème avec des fonctions récursives

Exemple d'une fonction récursive

- Puissance(x,n)
- ```
  function puissance(x, n) {
    if (n == 1) {
      return x;
    } else {
      return x * puissance(x, n - 1);
    }
  }
  ```
  ```

Profondeur de la récursion

Recursion depth:

est le nombre d'appels récursifs imbriqués

- Il existe une limite dans la profondeur pour les appels récursifs
- En JS c'est la MV qui détermine cette limite: 10000 - 100000

Comprendre les appels de fonction

- Pour exécuter une fonction nous avons besoin d'une structure pour ranger toutes les variables locales et le retour
- Cette structure est l'environnement d'exécution
- Pour chaque appel de fonction nous allons créer exactement un environnement d'exécution
- Quand une fonction fait appel à une autre fonction:
  - Elle se met en pause
  - Son environnement est sauvegardé dans une pile
  - la nouvelle fonction est lancée
  - Au retour, l'ancien environnement est retiré de la pile et considéré comme l'environnement courant
  - La fonction appelante retrouve son exécution

Exemple

- Dans le cas de notre fonction puissance, un environnement va ranger les variables locales et la ligne courante d'exécution
  - `{x:2,n:3,line:5}`
- Pour faire appel une la fonction interne puissance(2,2)
  - Nous allons ranger le contexte dans la pile d appels
  - Nous creeons un nouveau contexte pour cet appel
  - Quand le sous appel se termine, nous remettons l'ancien contexte
- La pile de contexte va etre par exemple
  - `Context : {x:2,n:1,line:1}`
    `Context : {x:2,n:2,line:5}`
    `Context : {x:2,n:3,line:5}`
- Après chaque exit/return d'une fonction, l'environnement est dépilé et la fonction appelante retrouve le résultat et continue son exécution

Une structure de données récursive

- Une structure de données est récursive si un de ses sous composants est du même type
- Par exemple, une liste
  - `let list = {value:10}`
    `list = {value:11, next:list}`

Introduction

- En JS, une fonction peut déclarer un nombre indéterminé de paramètres
  - `Math.max(a,b,c,...)`
- Pour déclarer une fonction qui va prendre une liste de paramètres il faut utiliser '...'
  - ```
    function somme(...args){
    for (let r of args) {}

    }
    ```
    ```

Parametres

- Nous pouvons aussi spécifier les premiers paramètres et laisser les autres en 'reste'
  - `function (a,b,...args){}`

Arguments

- Il existe une autre solution avec l'acces a l'attribut : arguments
  - C'est une solution ancienne qu'il faudrait éviter
  - Mais pour une fonction vous avez l'attribut arguments qui vous la liste des ses arguments

Opérateur Spread

- L'opérateur de spread va transformer une liste en une séquence de valeurs
  - `Math.max([1,2,3])`
    `Math.max(...[1,2,3])`

Comprendre le problème

- En JS, nous pouvons déclarer des fonctions à l'intérieur d'un bloc
  - Bloc simple
  - Corps d'une fonction englobante
  - Condition
  - Boucle
- Vous pouvez accéder à une variable extérieure depuis une fonction
- La question qui se pose, qu'elle la valeur pour une variable extérieure
  - La valeur lors de la définition de la fonction ?
  - La valeur lors de l'appel de la fonction ?

Comprendre le problème

- Exemple 1
  - ```
    let n = "X"
    function sayHello(){
    console.log(n)
    }
    n = "Y"
    sayHello()
    ```
    ```

Comprendre le problème

- Exemple 2
  - ```
    function makeFunction(){
    let n = "X"
    return function(){
    console.log(n)
    }

    }
    let n = "Y"
    let f = makeFunction()
    f()
    ```
    ```

Environnement lexical

- Un environnement lexical est simplement un tableau qui associe les noms de variables à leurs valeurs
- Nous allons avoir un environnement lexical pour:
  - Le script lui meme
  - Chaque bloc {}
  - Pour chaque appel de fonction

Inclusion des environnements

- Chaque environnement lexical pointe sur un seul environnement qui le contient
- La relation d'inclusion entre les environnements lexicaux
  - Chaque environnement va référencer son contenant
  - Sauf l'environnement global qui ne référence aucun autre environnement
  - La résolution des noms de variables sera toujours de l environnement actuel vers l'environnement contenant
  - Dans ce cas, nous pouvons comprendre pourquoi une fonction va pouvoir accéder aux variables extérieures avec leurs valeurs actuelles

Les appels de fonctions

- Il est important de se rappeler que, à chaque appel d'une fonction, on fait correspondre un nouvel environnement lexical
- Si une fonction est appelée à plusieurs reprises, alors il y aura autant d'environnements créés

Fonction internes

- il est possible de créer des fonctions internes dans au sein d'une autre fonction en JS
- Une fonction interne peut aussi être retournée comme un résultat
-
```
function hi(n,p){

function sayIt(){
console.log("Hi "+n+p)
}
sayIt()
}
```

L'environnement lexical d'une fonction

- Au début de l'interprétation d'un script la VM crée un seul environnement lexical celui du script
  - Cet environnement ne contient pas de variables
  - A chaque 'let' la variable est introduite
- Les fonctions sont traitées différemment:
  - Au départ, l'environnement lexical va contenir que les noms des fonctions
  - Autrement, nous ne pourrions pas utiliser une fonction avant sa déclaration
- Cycle de vie d'un environnement lexical
  - Le cycle de vie d'un environnement lexical est semblable à celui d'un objet
  - Nous n'effaçons un environnement lexical que si aucun autre environnement lexical ne fait référence sur lui !
  - Cela veut dire que les fonctions internes vont toujours maintenir leurs environnement de création en vie tant qu'elles sont en vie.

Environnement de lexical pour une fonction

- L'environnement de création
  - En réalité, nous avons deux types d'environnement
  - Un environnement de création: c'est une référence vers l'environnement lexicale de création de la fonction
    - Une fonction lors de sa création garde toujours une trace de son environnement lexical englobant et le range dans une propriété interne
  - Un environnement d'appel
    - Un environnement spécifique est créé pour cet appel
    - Il contient toutes les variables locales
    - Il fait référence vers l'environnement de création comme environnement contenant !

Environnement lexical : Code Block

- Code blocks
  - Nous avons vu qu'un environnement lexical est créé pour chaque appel d'une fonction
  - Il en va de même pour chaque bloc de contrôle {}

Environnement lexical : If

- Pour les conditions if...else
  - Nous allons avoir un premier bloc pour le 'if'
  - Un deuxieme bloc pour le 'else'
  - Pour chaque bloc un environnement lexical est créé

Environnement lexical : Boucles

- Boucles
  - il faut faire attention au cas des boucles 'for', 'while'
  - Il est important de noter qu'un nouveau bloc est créé pour chaque itération !

Exemple 1/2

```
function Compteur(){
let c = 0;
return function(){
return c++;
}



}
let counter = Compteur()
counter()
counter()
counter()
```

Exemple 2/2

- Etudions cet exemple
- Il y a trois niveaux d'environnements
  - Global
  - Compteur
  - Fonction interne
- La variable 'c' est dans le niveau 2
- Est-ce que nous pouvons modifier cette variable depuis l'extérieur ? Non
- Si nous appelons la fonction Compteur plusieurs fois, est-ce les variables 'c' seront indépendantes ? Oui

IIFE 1/2

- Historiquement, JS n'offrait pas une gestion des environnements lexicaux
- Une astuce de programmation permettait de créer des variables locales: IIFE
- IIFE: veut dire Immediatly Invoked Functional Expression
- C'est une valeur fonctionnelle qui est créée et utilisée sur place

IIFE 2/2

- Comme nous l'avons vu, l'environnement d'une fonction est créé pour chaque appel.
- Créer une fonction et faire un appel revient donc à créer un environnement tout en accédant aux variables extérieures : une fermerture (closure)
- Remarque: pour éviter les problèmes de syntaxe nous allons entourer la déclaration de la fonction par des parenthèses.

Exemple

- ```
  (function() {
            let counter = 0;
  return function(){ return counter++;}
  } )();
  ```
  ```

Une fonction: objet actif

- En JS une fonction est une valeur comme les autres
- Mais une fonction va aussi avoir le comportement d'un objet
- On parlera alors d'un objet actif

Propriétés objets

- Nous pouvons par exemple demander une propriété d'une fonction
- La propriété .name va donner le nom lexical associé à la fonction
- Nous pouvons aussi demander le nombre d'arguments qu'elle prend avec .length
- Il est possible de creer ses propres propriete sur une fonction !
- Mais attention ceci n'est pas a confondre avec les variables locales de la fonction qui vivent dans l'environnement lexical

Exemple

- NFE: Named Functional Expression
- ```
  let hello = function(n) {
  if(n){
  console.log(n)
  }else{
  hello("invite")
  }
  }
  ```
- C'est une fonction récursive; mais que se passe-t-il si la variable 'hello' est modifiée ?
  ```

Solution: NFE

- Pour éviter ce problème nous pouvons donner un nom interne à l'expression fonctionnelle et l'utiliser dans la récursion
- ```
  let hello = function func(n) {
  if(n){
  console.log(n)
  }else{
  func("invite")
  }
  }
  ```
- hello' existe toujours comme fonction; et func est connu juste en interne dans la définition de la fonction
  ```

Création avec Function

- Voici une autre méthode pour créer une fonction dynamiquement
- `let f = new Function( [arg1,arg2], bodystring)`

```
let sun = new Function('a','b','return a+b')
```

Avantages

- Cette méthode de créer les fonctions peut être très utile pour créer des fonctions en récupérant leurs codes sur un serveur par exemple.

Différences

- Il existe une différence entre la création classique et new Function
- L'environnement lexical de création de la fonction n'est plus l'environnement lexical courant mais toujours l'environnement global
- Conséquence: la fonction ne peut accéder qu'aux variables globales

Contexte d'appel

- Let hello = function() console.log(this.name); p = name:"John"; q = name:"Alex"; hello.call(p) hello.call(q)
- Nous pouvons configurer le contexte d'appel d'une fonction en utilisant call et apply
- Avec call et apply nous pouvons renseigner l'objet contexte sur lequel va pointer 'this'

Perte de contexte

- Nous pouvons definir une fonction dans un objet JS
- Cependant, la fonction interne va perdre le lien avec son objet de création si elle est utilisée dans un autre contexte

Exemple

```
let p = {
name: "John",
hello(){
log(this.name);
}
}
setTimeout(p.hello,1000)
```

Explication

- La fonction setTimeout a reçu une référence vers une fonction sans le contexte/objet associe a la fonction
- La fonction setTimeout aura dans son propre contexte 'this' qui pointe vers l'objet global; sauf que hello cherche une propriété 'name' dans 'this'

Solution

- Pour resoudre ce probleme nous allons voir les solutions suivantes
  - Utiser une fonction wrapper
  - Utiliser un bind

Solution 1: Wrapper

- Une fonction wapper c est une fonction qui va entourer l'utilisation d'une autre fonction
- ```
  setTimeout(   function() {
  user.hello();
  } , 1000 )
  ```
- A la création de la fonction wrapper l'environnement lexical va contenir la référence vers 'user', qui sera utilisé correctement comme contexte pour l'appel
  ```

Solution 2: Bind

- Nous pouvons aussi demander à lier le contexte à une fonction avec 'bind'
- `let bhello = user.hello.bind(user)`
- bhello est une fonction où le contexte est lié à 'user'
- Vous pouvez généraliser et également faire un bind sur des valeurs de certains paramètres

Déclaration

- Vous pouvez déclarer une fonction avec la notation arrow
- `()=>{}`
- La notation arrow n'est pas une abbréviation pour la création classique de fonctions
- Il existe des différences avec la méthode classique de création de fonction

Différences

- this
  - Une fonction arrow n'a pas de 'this'
  - Si this apparait dans le corps de la fonction arrow il est recupere de l'environnement exterieur
- new
  - Les arrow fonctions ne peuvent pas être utilisées comme constructeurs
  - En effet, elles ne peuvent pas accéder à this !
- arguments
  - Une fonction arrow n'a pas de propriété arguments