JavaScript
Closure*

IFNTI Sokodé – L3

04 Mai 2021

1 Rappel : portée des variables et fonctions

Rappelons nous qu'une variable défini par let n'est définie que pour le block (et les blocks enfants) dans lequel elle est définie.

Concernant les fonctions : rappelons qu'il est possible
- d'affecter une fonction à une variable
- qu'une fonction soit définie dans une autre fonction
- qu'une fonction retourne une fonction
- qu'une fonction (interne) définie dans une autre fonction (externe) a accès aux variable définie dans la fonction (externe)

1.1 Fonction définie et retournée par une autre fonction

Imaginons que nous souhaitons une fonction qui, si nous sommes en production affiche des console.log mais si nous debuggons, affiche des alertes.

```
function logging(debug){
if(debug) {
  return alert;
} else {
return consol.log;
}
}

const affiche=logging(true);
affiche("hello");
```

1.2 Utilisation des variables inter Fonctions

```
let prenom = 'Pierre';

//bio() a accès à let prenom (et à let age) mais pas à let hobbie
function bio(){
    let age = 29;
    //hobbies() a accès à let prenom et à let age (et à let hobbie)
    function hobbies(){
        let hobbie =  'Trail';
        return prenom + ', ' + age + ' ans. Je fais du ' + hobbie;
    }
    return hobbies();
}

alert(bio());
```

Quand les fonctions ont finies de s'exécuter, les variables utilisées disparaissent.

*Repris de `https://www.pierre-giraud.com/javascript-apprendre-coder-cours/closure/`

2 Les closures

Que ce passe-t-il si ?

```
function compteur() {
    let count = 0;

    return function() {
          return count++;
    };
}

let plusUn = compteur();
```

Comme vous le voyez, on crée une fonction `compteur()`. Cette fonction initialise une variable count et définit également une fonction anonyme interne qu'elle va retourner. Cette fonction anonyme va elle-même tenter d'incrémenter (ajouter 1) la valeur de let count définie dans sa fonction parente.

Ici, si on appelle notre fonction compteur() directement, le code de notre fonction anonyme est retourné mais n'est pas exécuté puisque la fonction `compteur()` retourne simplement une définition de sa fonction interne.

Pour exécuter notre fonction anonyme, la façon la plus simple est donc ici de stocker le résultat retourné par `compteur()` (notre fonction anonyme donc) dans une variable et d'utiliser ensuite cette variable « comme » une fonction en l'appelant avec un couple de parenthèses. On appelle cette variable `let plusUn`.

A priori, on devrait avoir un problème ici puisque lorsqu'on appelle notre fonction interne via notre variable `plusUn`, la fonction `compteur()` a déjà terminé son exécution et donc la variable `count` ne devrait plus exister ni être accessible.

Pourtant, si on tente d'exécuter code, on se rend compte que tout fonctionne bien.

C'est là tout l'intérêt et la magie des closures : si une fonction interne parvient à exister plus longtemps que la fonction parente dans laquelle elle a été définie, alors les variables de cette fonction parente vont continuer d'exister au travers de la fonction interne qui sert de référence à celles-ci.

Lorsqu'une fonction interne est disponible en dehors d'une fonction parente, on parle alors de closure ou de « fermeture »() en français.

Le code ci-dessus présente deux intérêts majeurs : tout d'abord, notre variable count est protégée de l'extérieur et ne peut être modifiée qu'à partir de notre fonction anonyme. Ensuite, on va pouvoir réutiliser notre fonction `compteur()` pour créer autant de compteurs qu'on le souhaite et qui vont agir indépendamment les uns des autres. Regardez plutôt l'exemple suivant pour vous en convaincre :

```
unction compteur() {
    let count = 0;

    return function() {
        return count++;
    };
}

let plusUn = compteur();
let plusUnBis = compteur();

alert(plusUn());
alert(plusUn());
alert(plusUnBis());
alert(plusUn());
alert(plusUnBis());
```

3 Un autre exemple

Une autre façon n'est pas de retourner la fonction mais de l'enregistrer directement dans window

```
const outer = function() {
  let = "A Local variable"
  let inner = function() {
    alert(a)
  }
  window.fnc = inner
}
outer();
fnc();
```

On peut aller encore plus loin et auto-exécuter la fonction sans même à avoir la nommer `outer`

Cette façon d'écrire n'est pas forcément très clair (d'autant plus si la fonction est un peu longue). Préférez lui la syntaxe suivante :

```
const fnc = function() {
  let = "A Local variable"
  return function() {
    alert(a)
  }
}();
```

Pourquoi ?

4 TD - que font ces bouts de code

1

```
let a = 12;
(function() {
  alert(a);
})();
```

2

```
let a = 5;
(function() {
  let a = 12;
  alert(a);
})();
```

3

```
let a = 10;
let x = (function() {
  let a = 12;
  return (function() {
    alert(a);
  });
})();
x();
```

4

```
let a = 10;
let x = (function() {
  let y = function() {
   let a = 12;
```

```
  };
  return function() {
    alert(a);
  }
})();
x();
```

5

```
let a = 10;
let x = (function() {
  (function() {
    a = 12; // <<< faites attention à cette ligne
  })();
  return (function() {
    alert(a);
  });
})();
x();
```

6

```
let a = 10;
(function() {
  let a = 15;
  window.x = function() {
    alert(a);
  }
})();
x();
```

Executons

Vérifions notre théorie et exécutons ces codes.