Concepteur Développeur en Informatique

Développer des composants d'interface

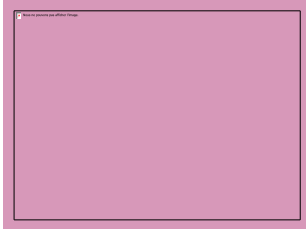
Javascript Closures

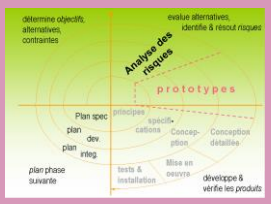
Accueil

Apprentissage

PAE

Evaluation





Localisation : U03-E05-S03

SOMMAIRE

1. Le lexical scope

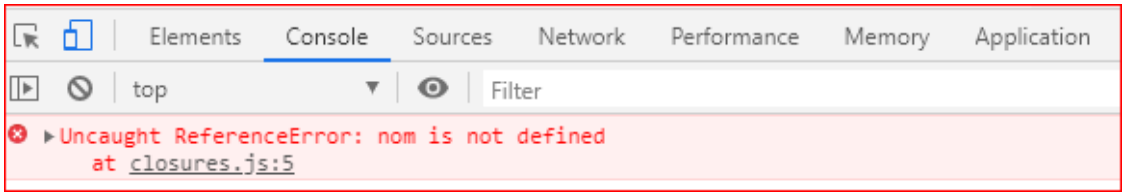
Plus souvent formulées sous le terme anglais Closure, essayons de définir ce qu'elles recouvrent et dans quels contextes y recourir.

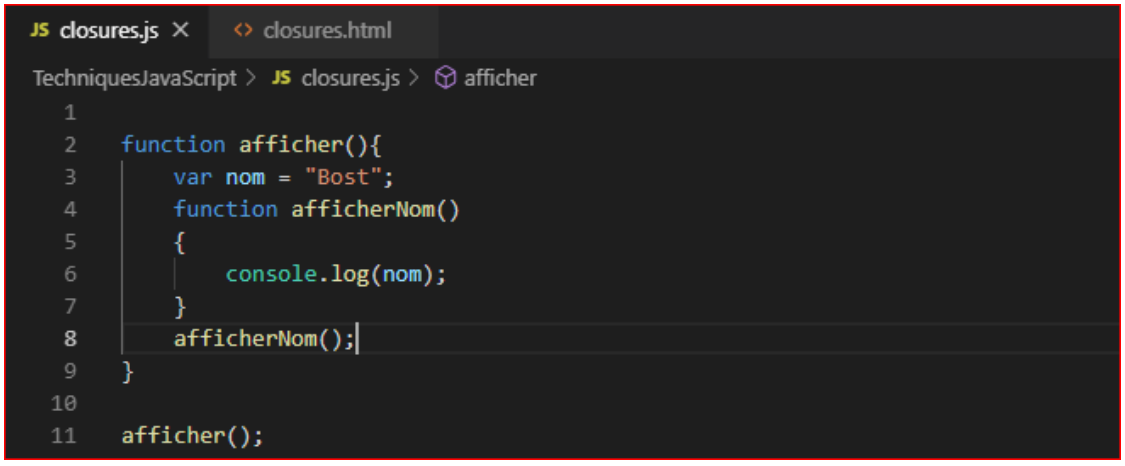
Revenons déjà sur la notion de portée ou scope définie par les fonctions.

Soit le code suivant :

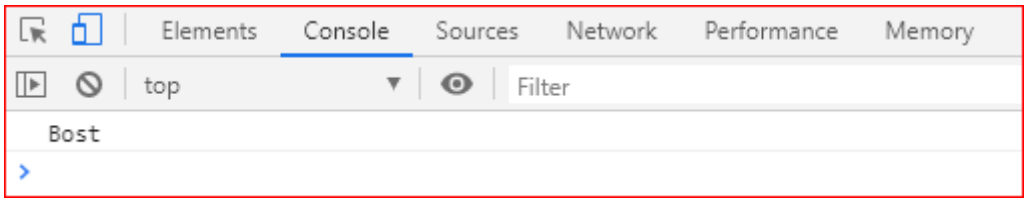
```
function afficher(){
    var nom = "Bost";
}
console.log(nom);
```

L'exécution de ce code provoque une erreur, nom étant hors de portée car local à la fonction.

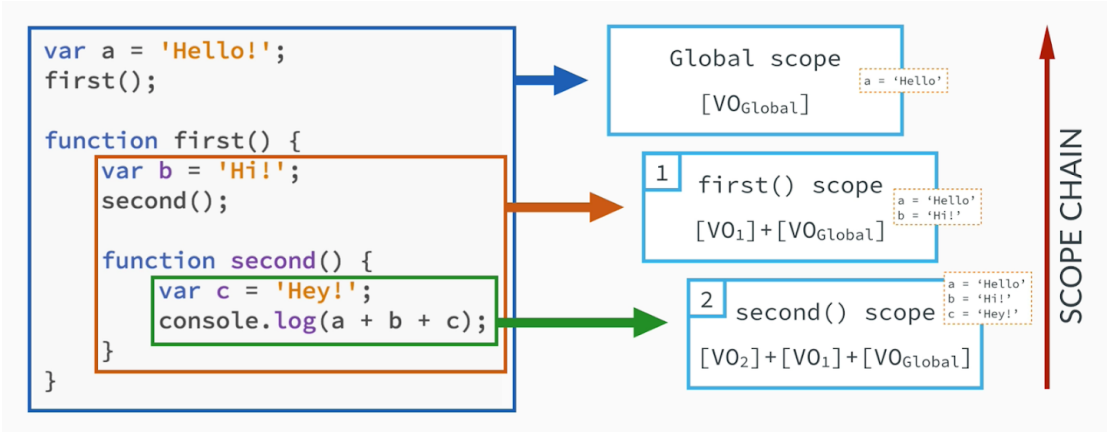




Cette deuxième version du code permet d'obtenir le résultat escompté, à savoir l'affichage à la console du contenu de la variable nom.



Pourquoi ? C'est le principe du lexical scope. Une variable connue dans une fonction parente sera aussi connue dans une variable interne (enfant).
Plus généralement nous avons :



La variable a est globale et sera accessible partout. Attention aux effets de bord.
La variable b est accessible dans first et second.
La variable c n'est accessible que dans la fonction enfant second.

2. Le principe des closures

Les closures sont des fonctions qui exploitent des variables définies en leur sein.
Voyons un exemple plus précis.

```
function increment() {
    var i = 0;
    console.log("Entrée dans la fonction")
    return function() {
        return i++;
    }
}
var closureIncrement = increment();
console.log(closureIncrement());
console.log(closureIncrement());
console.log(closureIncrement());
```

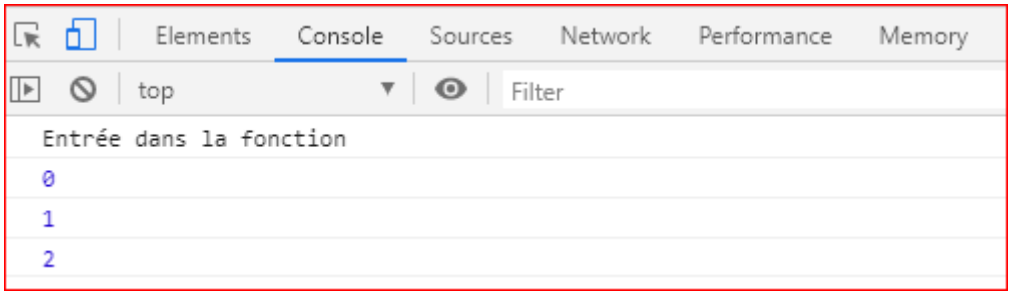
Définition de la closure sur invocation de la fonction f()

closureIncrement = increment() ;

On entre dans la fonction, une variable i est déclarée et une méthode anonyme est retournée.

Premier passage : la méthode anonyme exploite la variable i qui vaut 0, retourne 0 et incrémente i. La variable est connu grâce au principe du lexical scope.
Deuxième passage : la méthode anonyme exploite la variable i qui vaut 1, retourne 1 et incrémente i.
Troisième passage : la méthode anonyme exploite la variable i qui vaut 2, retourne 2 et incrémente i.

La preuve en image :



Intérêt ?
La variable i ne peut être modifiée en dehors de la fonction. Elle est ainsi protégée.

Je peux créer une nouvelle closure et l'exploiter. Les variables associées à son contexte seront spécifiques. Ainsi, pour exemple et dans la poursuite de la première version.

```
var closureIncrement = increment();
console.log(closureIncrement());
console.log(closureIncrement());
console.log(closureIncrement());
var closureIncrement2 = increment();
console.log("closure 2 : " + closureIncrement2());
console.log("closure 2 : " + closureIncrement2());
```

Résultat :

```
Entrée dans la fonction
0
1
2
Entrée dans la fonction
closure 2 : 0
closure 2 : 1
```

3. Problèmes résolus par les closures

3.1. Les champs privés

Vous souhaitez créer des « classes » pour définir des modèles objets en restreignant l'accès à certains champs ? Les closures peuvent répondre à votre problématique et imposer le recours aux accesseurs en lecture et écriture pour maintenir l'état de vos champs privés. Un exemple ici avec une classe Personne muni d'un constructeur d'initialisation, de deux attributs publics nom et prénom et d'un champ privé DateDeNaissance.

```
function Personne(nomx, prenomx, dateNaissancex) { // constructeur d'initialisation
  this.nom = nomx ; // attribut exposé publiquement
  this.prenom = prenomx ; // attribut exposé publiquement
  var dateNaissance = dateNaissancex; // attribut privé

  // méthode calcul de l'âge
  this.age = function(){
    return new Number((Date.now - dateNaissance.getTime())
    / 31536000000).toFixed(2);;
  } ;

  // setter date de naissance avec contrôle
  this.setDateNaissance = function(dateNaissancex) {
    dateNaissance = dateNaissancex;}

  // getter date de naissance
  this.getDateNaissance = function()
  {return dateNaissance ;}

} ;
```

Lors du test, nous pouvons vérifier le fonctionnement correct de l'ensemble.

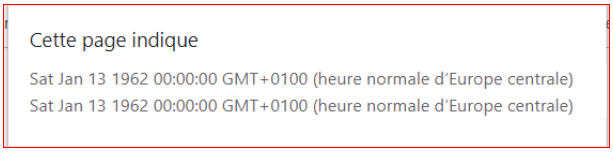
Vous noterez quelques points particuliers au sujet du type date Javascript qui est complexe et d'une manipulation qui nécessite une certaine vigilance.

Un tableau des options de date, pour représentation textuelles (formatage) de ces dernières.

```
// Déclaration des options de date pour présentation textuelle
var options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
```

L'instanciation d'un objet date avec les arguments numériques Year,Month,Day. Le mois est un indice positionnel. (0 pour janvier). Les deux objets de type date instanciés de la manière qui suivante prendront la même valeur :

```
date1 = new Date(1962,0,13);
date2 = new Date(1961,12,13);
```



3.2. Traitements asynchrones

Les closures peuvent être particulièrement utiles pour les traitements asynchrones.
Exemple de la fonction SetTimeout qui permet d'exécuter un traitement différé fonction d'un délai d'attente défini.

```
var nombre = 0;

window.setTimeout(function() {
    window.alert("valeur du nombre : " + nombre);
}, 100);

nombre++;
```

Nous pourrions nous attendre à ce que la valeur affichée soit 0. Et bien non.
La valeur affichée sera celle de la variable i au moment de l'exécution comme le montre le résultat de l'exécution.

Nous devons donc prévoir un mécanisme qui nous permettra de préserver la valeur de la variable au moment de la préparation et non de l'exécution.
La fonction de callback (ici anonyme) sera préparée puis exécutée lors de l'expiration du délai d'attente.Nous pouvons résoudre ce problème grâce au procédé des closures en modifiant légèrement notre code.

```
var nombre = 0;

function afficherNombre (e){
    return function(){ window.setTimeout(function() {
        window.alert("valeur du nombre : " + e);
    }, 2000);
    };
}
var closureNombre = afficherNombre(nombre);
closureNombre();
nombre++;
```

Résultat :

Dans ce contexte, je peux aussi, pour définir ma closure, utiliser une fonction anonyme auto-exécutée Je n'invoquerai jamais ma closure par référence à son nom.
Je peux alors écrire :

```
var nombre = 0;

(function afficherNombre (e){
    window.setTimeout(function() {
        window.alert("valeur du nombre : " + e);
    }, 2000);

})(nombre);

nombre++;
```

Résultat :

Les closures sont souvent invoquées au sein de fonctions auto-invoquées.

Pattern IIFE : Immediate Invocation Function Expression

Ce principe de fonctions auto-invoquées est identifié sous le terme IIFE pour Immediate Invocation Function Expression et est un patron de conception très usité en Javascript pour encapsuler des comportements.

Il se présente sous la forme :

(function() { })()

En javascript nous pouvons référencer les fonctions dans des variables et ensuite invoquer la variable.
var maFonction = function(){} ;
puis maFonction() ou (maFonction)() les parenthèses (représentant ici les séparateurs d'expression.

D'où au final, l'écriture simplifiée suivante si je ne dois pas conserver la référence à la fonction.
Je supprime alors la référence à la variable (function() { })()

3.2.1. Autre exemple de traitement asynchrone

Pour illustrer de nouveau l'utilisation de closures avec des fonctions asynchrones, j'ai ici préparer l'exécution de méthodes asynchrones au sein d'une boucle. Il s'agit d'un cas d'école que vous retrouverez quelle que soit l'architecture logicielle utilisée.

Je souhaite que les cellules des lignes d'un tableau soient coloriées en respectant un délai d'une seconde entre le traitement de deux lignes. Ceci afin d'apporter un plus visuel à ma page.

Tout naturellement nous aurions tendance à réaliser un cycle pour i allant de 0 à n-1 lignes, faire asynchrone wait 1 peindre(ligne[i]). Mais la valeur de i serait alors celle de fin de cycle.

Pour résoudre ce problème nous allons réaliser des closures.

Ici, la fonction de base pour colorier les lignes :

```
for(let i = 0;i<tableau.tBodies[0].rows.length;i++)
{
  tableau.tBodies[0].rows[i].style.backgroundColor = "yellow";
};
```

La deuxième version pour colorier au rythme de 1 seconde chaque ligne.

```
for (let i = 0; i < tableau.tBodies[0].rows.length; i++) {
    window.setTimeout(
        /* Argument 1 : Fonction à lancer après le délai */
        (function (indice) {
            // "indice" prendra la valeur de "i" lors de l'exécution du cycle
            // Fonction à exécuter à la fin du délai défini dans
            // "setTimeout"
            return function () {
                // Lorsque cette fonction s'exécutera, indice contiendra
                // toujours la valeur qui lui a été transmise à la préparation
                tableau.tBodies[0].rows[indice].style.backgroundColor = "yellow";
            };
        })(i), // on passe en argument le compteur de la boucle
        1000 * (i + 1) // on calcul le délai
    );
}
```

Au niveau du débogueur, nous pouvons constater que nous avons 4 closures avec un indice représentant la position de la ligne dans le tableau. Extrait

```
closure fonction function () {
                    tableau.tBodies[0].rows[indice].style.backgroundColor = "yellow";
                } indice 0
closure fonction function () {
                    tableau.tBodies[0].rows[indice].style.backgroundColor = "yellow";
                } indice 1
closure fonction function () {
                    tableau.tBodies[0].rows[indice].style.backgroundColor = "yellow";
                } indice 2
```

3.3. Variable locale statique

La technique de fermeture permet aussi de conserver des valeurs de variables locales entre 2 invocations d'une fonction.
Cette notion a été supprimée des langages de programmation objet tels que Java ou C# mais demeure dans d'autres langages.

Soit une fonction qui accumule la somme de 2 nombres.

```
function additionner(a){
    return function(y) {
      return  a = a + y;
      };
};
```

Illustration par l'exemple :

```
var closure = additionner(5);
console.log(closure);
console.log(closure(3));
console.log(closure(10));
```

Résultat :

```
ƒ (y) {
      return  a = a + y;
      }
8
18
```

Vous trouverez les exemples sur le site closures.zip.