Les fermetures (closures)

Cette leçon est également très importante car nous allons étudier un concept propre au `JavaScript` qui est également très utilisé et très puissant : les fermetures (`closures`).

Qu'est-ce qu'une fermeture ?

Une fermeture (ou `closure`) est une fonction qui utilise des identifiants de la portée parente, et ce même si la fonction parente n'existe plus.

Prenons un exemple simple pour comprendre :

```
function fonction1() {
  const prenom = "Jean";
  return () => console.log(prenom);
}

const fonction2 = fonction1();
fonction2(); // "Jean"
```

Comment est-ce possible ? Normalement lorsque nous exécutons `fonction2()`, `fonction1()` n'est plus sur la `stack` !

Si fonction1() n'est plus sur la `stack` comment `fonction2()` peut-elle accèder à la variable `prenom` lors de son exécution ?

C'est justement grâce à la fermeture : lors de la création de la `fonction2()`, elle a capturé son environnement.

Nous allons voir que c'est lié à la chaîne des portées. Prenons l'environnement lexical de la `fonction1()` :

```
environnementLexicalFonction1 = {
  prenom = "Jean"
  environnementParent: <reférence à l'env lexical global>
}
```

Et celui de la fonction fléchée anonyme imbriquée :

```
environnementLexicalFonctionAnonyme = {
  environnementParent: <reférence à environnementLexicalFonction1>
}
```

En fait, la fonction imbriquée a une photographie de son environnement lors de sa création. Cela veut dire que même après la fin de l'exécution de la `fonction1()` et sa disparition de la `stack`, la fonction imbriquée peut accèder à l'environnement lexical parent qui est gardé en mémoire.

Second exemple

Nous allons prendre un deuxième exemple qui montre encore mieux qu'il s'agit vraiment d'une photographie de l'environnement lexical parent :

```
function créerCompteur() {
  let compteur = 0;
  return () => ++compteur;
}

const compteur1 = créerCompteur();
const compteur2 = créerCompteur();

compteur1(); // 1
compteur1(); // 2
compteur2(); // 1
```

Une remarque avant d'étudier l'exemple : `++compteur` et `compteur++` n'ont pas le même résultat. Dans le premier cas, la valeur est incrémentée puis retournée. Dans le second cas, la valeur est retournée puis incrémentée.

Remarquez que chaque compteur est totalement indépendant. Ils ont chacun une variable interne `compteur` indépendante.

Cela signifie que la fermeture est **véritablement une photographie de la chaîne des environnements lexicaux lors de l'exécution de la fonction**.

Bref aperçu de la gestion mémoire en `JavaScript`

Le mécanisme est en fait que `JavaScript` ne va pas supprimer un environnement lexical tant qu'une fonction l'utilise, et ce même si la fonction est imbriquée sur plusieurs niveaux :

```
function créerCompteur() {
  let compteur = 0;
  return () => () => () => ++compteur; // trois niveaux d'imbrication
}

const compteur1 = créerCompteur()()();
```

```
compteur1(); // 1
```

Peu importe que la fonction soit imbriquée sur plusieurs niveaux, grâce à la chaîne de portée et à la sauvegarde des références encore utilisées, la variable `compteur` reste accessible malgré le fait qu'il faille remontée de plusieurs niveaux et que les fonctions aient terminées leurs exécution.

C'est lié à ce qu'on appelle la **gestion de la mémoire**, et plus précisément aux algorithmes ramasse-miettes (ou `garbage collection`).

Ces algorithmes sont chargées de vider la mémoire des valeurs et objets qui ne sont plus accessibles ou n'ont plus de référence.

L'interpréteur `JavaScript` sait que pour les fonctions, si une fonction imbriquée utilise des variables sur la chaîne de ses environnement lexicaux parents, il ne faut pas supprimer ces environnements avant que la fonction imbriquée ait terminé son exécution.

Vous pouvez facilement raisonner en vous disant que tant qu'une référence est utilisée lors de l'exécution elle ne sera pas supprimée de la mémoire et donc restera accessible.