La portée (scope) en JavaScript

> La portée, ou *scope* en anglais, détermine où et comment les variables, fonctions et objets sont accessibles dans votre code.
>
> Comprendre la portée est essentiel pour éviter les erreurs inattendues et écrire un code efficace et maintenable.

1. Qu'est-ce que la portée ?

> La portée est l'espace dans lequel une variable ou une fonction est définie et peut être utilisée.
>
> En JavaScript, cet espace peut être global, local ou lié à un bloc.
>
> Les variables en dehors de leur portée ne sont pas accessibles.

Exemple :

```javascript
function saluer() {
  const message = "Bonjour !";
  console.log(message);
}

// console.log(message); // Erreur : message n'est pas défini en dehors de la fonction.
saluer(); // Affiche : Bonjour !
```

2. Les types de portée

2.1. 1. Portée globale

> Les variables définies en dehors de toute fonction ou bloc ont une portée globale.
>
> Elles sont accessibles partout dans le programme.

Exemple :

```javascript
const nom = "Alice";

function afficherNom() {
  console.log(nom); // Utilise la variable globale
}

afficherNom(); // Affiche : Alice
```

2.2. 2. Portée locale (portée de fonction)

> Les variables définies à l'intérieur d'une fonction sont limitées à cette fonction.
>
> Elles ne peuvent pas être utilisées en dehors.

Exemple :

```javascript
function additionner() {
  const a = 5, b = 3;
  console.log(a + b); // Affiche : 8
}

// console.log(a); // Erreur : a n'est pas défini en dehors de la fonction.
```

2.3. 3. Portée de bloc

> Avec `let` et `const`, les variables définies dans un bloc de code (comme les accolades `{}`) sont limitées à ce bloc.

Exemple :

```javascript
{
  const x = 10;
  console.log(x); // 10
}

// console.log(x); // Erreur : x n'est pas défini en dehors du bloc.
```

2.4. Différence clé entre *var*, *let* et *const* :

- `var` : possède une portée globale ou de fonction, mais ignore la portée de bloc.
- `let` et `const` : respectent la portée de bloc. `const` ne permet pas de réaffectation.

Exemple :

```javascript
for (var i = 0; i < 3; i++) {
  console.log(i); // Affiche : 0, 1, 2
}
console.log(i); // Accessible en dehors de la boucle (i vaut 3 avec var)

for (let j = 0; j < 3; j++) {
  console.log(j); // Affiche : 0, 1, 2
}
// console.log(j); // Erreur : j n'est pas accessible en dehors de la boucle
```

3. Lexical Scope

> JavaScript utilise une portée lexicale.
>
> Cela signifie que la portée est déterminée lors de l'écriture du code, et non lors de son exécution.

Exemple :

```javascript
function externe() {
  const externeVar = "Je viens de externe";

  function interne() {
    console.log(externeVar); // Accessibilité grâce à la portée lexicale
  }

  interne();
}

externe();
// Affiche : Je viens de externe
```

Même si `interne` est exécutée après la définition de `externe`, elle peut toujours accéder aux variables définies dans `externe` grâce à la portée lexicale.

4. Closures (fermetures)

> Une `closure` est une fonction qui se souvient de son environnement lexical, même lorsqu'elle est exécutée hors de cet environnement.

Exemple :

```javascript
function createCounter() {
  let compteur = 0;

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

const compteur1 = createCounter();
console.log(compteur1()); // 1
console.log(compteur1()); // 2

const compteur2 = createCounter();
console.log(compteur2()); // 1
```

Ici, chaque instance de `createCounter` retourne une fonction qui conserve l'accès à la variable `compteur` de son environnement initial.

5. Pièges courants liés à la portée

5.1. 1. Problèmes avec *var* dans les boucles

```javascript
for (var i = 1; i ⩽ 3; i++) {
    console.log("i dans la boucle =", i);
}

console.log("i après la boucle =", i); // i après la boucle = 4
```

Pourquoi ? Les variables définies avec `var` ont une **portée globale** ou de fonction, donc la sont elles sont accessibles en dehors du bloc `for` .

Meilleure pratique : utiliser let afin d'éviter que la variable `j` soit accessible après la boucle.

```javascript
for (let j = 1; j ⩽ 3; j++) {
    console.log("j dans la boucle =", j);
}

console.log("j après la boucle =", j); // Erreur : j n'est pas défini en dehors de la boucle
```

5.2. 2. Redéfinition indésirable

Avec `var` , une variable peut être redéclarée accidentellement.

```javascript
var x = 42;
var x = 100; // Pas d'erreur
console.log(x); // 100
```

Cela ne se produit pas avec `let` ou `const` .

6. À RETENIR

- **La portée** détermine où une variable est accessible dans votre code.
- Les types de portée sont : **globale**, **locale** et **de bloc**.
- Utilisez `var` avec précaution, car il **ignore les blocs** et peut causer des comportements inattendus.
- Les `**closures**` permettent à une fonction de se souvenir de son environnement lexical.
- Préférez `let` et `const` pour éviter les erreurs liées à la redéclaration ou à la portée.

Maîtriser la portée en JavaScript est crucial pour comprendre comment vos variables et fonctions interagissent dans le programme.

Une bonne pratique est de limiter la portée des variables autant que possible pour éviter des bugs subtils.