`var`, `let` et `const` : Comprendre la Déclaration de Variables en JavaScript/TypeScript

Introduction

Avant ES2015 (ES6), `var` était la seule façon de déclarer une variable en JavaScript. Avec l'arrivée d'ES6, `let` et `const` ont été introduits pour résoudre certains des problèmes et des comportements inattendus de `var`, offrant un meilleur contrôle sur la portée et la mutabilité des variables.

Comprendre leurs différences est fondamental pour écrire du code JavaScript/TypeScript moderne, propre et sans bugs.

1. `var` : L'Ancien Monde

Portée (Scope) : Fonctionnelle (ou Globale)

- Les variables déclarées avec `var` sont de **portée fonctionnelle**. Cela signifie qu'elles ne sont accessibles qu'à l'intérieur de la fonction où elles sont déclarées.
- Si `var` est déclarée en dehors de toute fonction, elle a une **portée globale** (elle devient une propriété de l'objet `window` dans les navigateurs, ou de l'objet `global` dans Node.js).
- **Problème :** `var` n'est *pas* de portée de bloc (block-scoped). Cela signifie qu'une variable déclarée à l'intérieur d'un bloc `if`, `for`, `while` ou `try/catch` sera accessible en dehors de ce bloc, ce qui peut entraîner des confusions.

Hoisting (Remontée)

- Les déclarations de variables `var` sont "remontées" (hoisted) en haut de leur portée (fonction ou globale). Cela signifie que la variable est *déclarée* avant que le code ne soit exécuté, mais elle est *initialisée* à `undefined`.
- **Problème :** Vous pouvez accéder à une variable `var` avant sa déclaration, elle aura juste la valeur `undefined`. Cela peut masquer des erreurs de logique.

Réaffectation et Redéclaration

- `var` permet la **réaffectation** de sa valeur.
- `var` permet la **redéclaration** de la variable dans la même portée, sans erreur.

Exemple avec `var`

```
// Exemple de Hoisting avec var
console.log(nom); // Output: undefined (pas d'erreur, car 'nom' est hoisted)
var nom = "Alice";
console.log(nom); // Output: Alice

// Exemple de portée fonctionnelle vs. bloc
function saluer() {
    var message = "Bonjour !";
    if (true) {
        var heure = "matin"; // 'heure' n'est PAS bloquée par le 'if'
        console.log(message); // Output: Bonjour !
    }
    console.log(heure); // Output: matin (accessible en dehors du bloc 'if')
}
saluer();
// console.log(message); // Erreur: message is not defined (car 'message' est dans la

// Exemple de réaffectation et redéclaration
var x = 10;
console.log(x); // Output: 10
var x = 20;     // Redéclaration permise, pas d'erreur !
console.log(x); // Output: 20
x = 30;         // Réaffectation permise
console.log(x); // Output: 30
```

Problèmes de `var` (Pourquoi l'éviter)

- **Fuite de portée (Scope Leaking):** Le fait qu'il ne soit pas de portée de bloc peut entraîner des conflits de noms de variables inattendus, surtout dans les boucles.
- **Confusion due au Hoisting:** Le comportement `undefined` avant la déclaration peut rendre le code difficile à déboguer.
- **Manque de clarté:** Ne pas savoir si une variable peut être redéclarée ou réaffectée sans analyser le code.

2. `let` : Le Nouveau Standard pour les Variables

`let` a été introduit pour remplacer `var` et résoudre ses lacunes.

Portée (Scope) : De Bloc

- Les variables déclarées avec `let` sont de **portée de bloc**. Cela signifie qu'elles ne sont accessibles qu'à l'intérieur du bloc de code `{}` où elles sont déclarées (fonctions, boucles, conditions, etc.).

- C'est le comportement le plus intuitif et le plus sûr.

Hoisting : La "Zone Morte Temporelle" (Temporal Dead Zone - TDZ)

- Les déclarations de variables `let` sont également remontées (hoisted) en haut de leur bloc, mais elles ne sont **pas initialisées**.
- Accéder à une variable `let` avant sa déclaration (dans sa TDZ) entraînera une erreur de référence (`ReferenceError`). C'est un comportement plus sûr qui aide à attraper les erreurs.

Réaffectation et Redéclaration

- `let` permet la **réaffectation** de sa valeur.
- `let` **n'autorise PAS la redéclaration** de la variable dans la même portée. Cela évite les bugs où vous déclarez accidentellement une variable existante.

Exemple avec `let`

```
// Exemple de Zone Morte Temporelle (TDZ) avec let
// console.log(prenom); // Erreur: Cannot access 'prenom' before initialization (Refer
let prenom = "Bob";
console.log(prenom); // Output: Bob

// Exemple de portée de bloc
function calculerPrix() {
    let prixBase = 100;
    if (true) {
        let taxe = 10; // 'taxe' est bloquée par le 'if'
        console.log(prixBase + taxe); // Output: 110
    }
    // console.log(taxe); // Erreur: taxe is not defined (accessible seulement dans le
    console.log(prixBase); // Output: 100
}
calculerPrix();

// Exemple de réaffectation et redéclaration
let y = 10;
console.log(y); // Output: 10
// let y = 20;     // Erreur: Cannot redeclare block-scoped variable 'y'.
y = 30;         // Réaffectation permise
console.log(y); // Output: 30
```

3. `const` : La Constante (ou presque)

`const` est également introduit avec ES6 et est conçu pour les variables dont la valeur ne doit pas changer après leur initialisation.

Portée (Scope) : De Bloc

- Comme `let`, `const` est de **portée de bloc**.

Hoisting : La "Zone Morte Temporelle" (TDZ)

- Comme `let`, les déclarations `const` sont remontées mais ne sont pas initialisées. Accéder avant la déclaration entraîne une `ReferenceError`.
- **Important :** Une variable `const` doit être **initialisée au moment de sa déclaration**.

Réaffectation et Redéclaration

- `const` **n'autorise PAS la réaffectation** de sa valeur. Une fois assignée, la valeur ne peut pas être changée.
- `const` **n'autorise PAS la redéclaration** dans la même portée.

Immutabilité (Nuance Importante)

- Pour les **types primitifs**, `const` garantit une véritable immutabilité : la valeur de la variable ne peut pas être modifiée.
- Pour les **types de référence (objets et tableaux)**, `const` garantit que la *référence* de la variable ne peut pas être modifiée. Cependant, les *propriétés internes* de l'objet ou les *éléments* du tableau peuvent toujours être modifiés. C'est ce qu'on appelle une **"immutabilité superficielle"**.

Exemple avec `const`

```
 de déclaration et TDZ avec const
.log(PI); // Erreur: Cannot access 'PI' before initialization
 3.14159;
g(PI); // Output: 3.14159

14; // Erreur: Cannot assign to 'PI' because it is a constant or a read-only property.

 de portée de bloc avec const
{
couleur = "bleu";
e.log(couleur); // Output: bleu

.log(couleur); // Erreur: couleur is not defined

 d'immutabilité superficielle avec les objets
isateur = {
```

```
Charlie",
5


g(utilisateur); // Output: { nom: 'Charlie', age: 25 }

r.age = 26; // CECI EST PERMIS ! On modifie une propriété de l'objet, pas la référence de
g(utilisateur); // Output: { nom: 'Charlie', age: 26 }

teur = { nom: "David", age: 30 }; // CECI EST INTERDIT ! On essaie de réaffecter la varia
```

Tableau Récapitulatif

Caractéristique	`var`	`let`	`const`
Portée (Scope)	Fonctionnelle/Globale	De Bloc	De Bloc
Hoisting	Oui (initialisé `undefined`)	Oui (TDZ)	Oui (TDZ)
Réaffectation	Oui	Oui	Non
Redéclaration	Oui	Non	Non
Initialisation	Optionnelle	Optionnelle	**Obligatoire**
Utilisation typique	(À éviter en code moderne)	Variables qui changent de valeur	Variables constantes ou références d'objets/tableaux

Bonnes Pratiques

En JavaScript/TypeScript moderne, la règle générale est la suivante :

1. **Préférez toujours `const`** : C'est le choix le plus sûr et le plus explicite. Utilisez-le par défaut pour toutes les variables dont la valeur (ou la référence pour les objets) ne changera pas. Cela rend votre code plus prévisible et réduit les risques d'erreurs.
2. **Utilisez `let` si et seulement si vous avez besoin de réaffecter la variable** : Si la valeur de la variable doit changer au cours de l'exécution du code (par exemple, dans une boucle, un compteur, ou une variable qui est réassignée dans un bloc conditionnel), alors `let` est le bon choix.
3. **Évitez `var`** : Il est rare d'avoir une bonne raison d'utiliser `var` dans le nouveau code. Sa portée confuse et son comportement de hoisting peuvent entraîner des bugs difficiles à identifier.

En suivant ces règles, vous écrirez un code plus robuste, plus clair et plus facile à maintenir.