en JavaScript (et donc en TypeScript) qui est essentiel pour comprendre comment vos données sont manipulées en mémoire.

Types Primitifs vs. Types de Référence en JavaScript/TypeScript

Introduction

En JavaScript, la façon dont une valeur est stockée en mémoire et comment elle se comporte lors de l'assignation ou du passage à une fonction dépend de son **type**. Il existe deux grandes catégories : les **types primitifs** (ou "par valeur") et les **types de référence** (ou "par référence"). Comprendre cette distinction est crucial pour éviter les bugs inattendus et écrire un code robuste.

1. Les Types Primitifs (Passage par Valeur)

Définition

Un type primitif est une donnée qui n'est pas un objet et n'a pas de méthodes. Les valeurs primitives sont stockées **directement dans l'emplacement mémoire de la variable**. Lorsque vous manipulez une variable de type primitif, vous manipulez sa valeur directement.

Caractéristiques

- **Stockage :** La valeur est stockée directement dans la variable.
- **Affectation/Copie :** Lorsqu'une variable primitive est assignée à une autre, la **valeur est copiée**. Les deux variables sont alors indépendantes.
- **Comparaison :** Les types primitifs sont comparés **par leur valeur**.

Liste des Types Primitifs en JavaScript/TypeScript

1. **`string`** : Séquence de caractères (ex: `"bonjour"`)
2. **`number`** : Nombres entiers ou décimaux (ex: `10`, `3.14`)
3. **`boolean`** : Valeurs vraies ou fausses (ex: `true`, `false`)
4. **`null`** : Représente l'absence intentionnelle de toute valeur objet.
5. **`undefined`** : Représente une variable qui a été déclarée mais n'a pas encore été assignée d'une valeur.
6. **`symbol`** (ES6) : Valeur unique et immuable, souvent utilisée comme clé de propriété d'objet.
7. **`bigint`** (ES2020) : Permet de représenter des nombres entiers de taille arbitraire.

Exemples avec les Types Primitifs

Exemple 1 : Affectation et Indépendance

```
let ageOriginal: number = 25;
let ageCopie: number = ageOriginal; // La valeur 25 est copiée

console.log(`Age original: ${ageOriginal}`); // Output: Age original: 25
console.log(`Age copie: ${ageCopie}`);     // Output: Age copie: 25

ageCopie = 30; // Modifie la valeur de ageCopie uniquement

console.log(`Age original après modification: ${ageOriginal}`); // Output: Age origina
console.log(`Age copie après modification: ${ageCopie}`);     // Output: Age copie apr
```

Comme on le voit, `ageOriginal` reste `25` car `ageCopie` a reçu une *copie de la valeur*, et non une référence au même emplacement mémoire.

Exemple 2 : Passage à une Fonction

```
function incrementer(valeur: number): void {
  valeur = valeur + 1;
  console.log(`À l'intérieur de la fonction (valeur): ${valeur}`);
}

let monNombre: number = 5;
incrementer(monNombre); // La valeur 5 est copiée dans le paramètre 'valeur' de la fon

console.log(`À l'extérieur de la fonction (monNombre): ${monNombre}`); // Output: À l'
```

La variable `monNombre` n'est pas modifiée par la fonction `incrementer` car c'est une copie de sa valeur qui est passée.

2. Les Types de Référence (Passage par Référence / Partage)

Définition

Les types de référence sont des **objets**. Contrairement aux types primitifs, les variables de type de référence ne stockent pas directement la valeur de l'objet. Elles stockent une **référence (une adresse mémoire)** à l'endroit où l'objet est réellement stocké dans la mémoire (le "tas" ou "heap").

Caractéristiques

- **Stockage :** La variable contient une référence (un pointeur) vers l'objet réel en mémoire.

- **Affectation/Copie :** Lorsqu'une variable de référence est assignée à une autre, la **référence (l'adresse mémoire) est copiée**, pas l'objet lui-même. Les deux variables pointent alors vers le même objet en mémoire.
- **Comparaison :** Les types de référence sont comparés **par leur référence** (est-ce la même adresse mémoire ?), pas par la valeur de leurs propriétés.

Liste des Types de Référence Courants

En JavaScript, **tout ce qui n'est pas un type primitif est un objet** et est donc un type de référence.

- **`Object`** (objets simples `{}`, y compris les classes instanciées)
- **`Array`** (tableaux `[]`)
- **`Function`**
- **`Date`**
- **`RegExp`**
- Et tous les objets créés par l'utilisateur ou par des APIs du navigateur/Node.js.

Exemples avec les Types de Référence

Exemple 1 : Affectation et Effets de Bord

```
interface Personne {
  nom: string;
  age: number;
}

let personneOriginale: Personne = { nom: 'Alice', age: 30 };
let personneCopie: Personne = personneOriginale; // La RÉFÉRENCE est copiée, pas l'obj

console.log('Avant modification:');
console.log(personneOriginale); // Output: { nom: 'Alice', age: 30 }
console.log(personneCopie);     // Output: { nom: 'Alice', age: 30 }

personneCopie.age = 31; // Modifie l'objet auquel les deux variables font référence

console.log('Après modification:');
console.log(personneOriginale); // Output: { nom: 'Alice', age: 31 } (MODIFIÉ !)
console.log(personneCopie);     // Output: { nom: 'Alice', age: 31 }
```

Ici, `personneOriginale` est modifiée lorsque `personneCopie` est modifiée car les deux variables pointent vers le *même objet* en mémoire. Il n'y a qu'un seul objet sous-jacent.

Exemple 2 : Passage à une Fonction et Modification de l'Objet

```
interface Produit {
  nom: string;
  stock: number;
}

function reduireStock(item: Produit, quantite: number): void {
  item.stock -= quantite; // Modifie directement la propriété de l'objet référencé
  console.log(`À l'intérieur de la fonction (stock): ${item.stock}`);
}

let monProduit: Produit = { nom: 'Livre', stock: 100 };
reduireStock(monProduit, 10); // La RÉFÉRENCE à monProduit est copiée dans le paramètr

console.log(`À l'extérieur de la fonction (stock): ${monProduit.stock}`); // Output: À
```

La fonction `reduireStock` a pu modifier l'objet `monProduit` car elle a reçu une copie de la *référence* à cet objet, lui permettant d'accéder et de modifier l'objet original en mémoire.

3. Comparaison des Types

Égalité stricte (===)

- **Types Primitifs :** L'opérateur `===` compare les **valeurs**.

```
console.log(5 === 5);         // true
console.log('hello' === 'hello'); // true
console.log(null === null);     // true
console.log(undefined === undefined); // true
console.log(NaN === NaN);     // false (exception pour NaN, il n'est jamais égal à
```

- **Types de Référence :** L'opérateur `===` compare les **références (adresses mémoire)**. Deux objets ne sont strictement égaux que s'ils pointent vers la *même instance* en mémoire.

```
const obj1 = { a: 1 };
const obj2 = { a: 1 };
const obj3 = obj1; // obj3 pointe vers la même instance que obj1

console.log(obj1 === obj2); // false (deux objets différents en mémoire, même si l
console.log(obj1 === obj3); // true (pointent vers la même instance)

const arr1 = [1, 2];
const arr2 = [1, 2];
console.log(arr1 === arr2); // false
```

4. Implications et Bonnes Pratiques

A. Immutabilité

Comprendre la distinction est essentiel pour l'**immutabilité**, une pratique de programmation où les données, une fois créées, ne sont jamais modifiées. Au lieu de modifier un objet ou un tableau existant, vous créez une *nouvelle* instance avec les modifications souhaitées. C'est la base de nombreux frameworks modernes (comme React avec ses états, Redux, Vuex) qui s'appuient sur la détection des changements par comparaison de références pour optimiser les rendus.

Exemple d'immutabilité avec l'opérateur de propagation :

```
interface Task {
  id: number;
  description: string;
  completed: boolean;
}

const tasks: Task[] = [
  { id: 1, description: 'Faire les courses', completed: false },
  { id: 2, description: 'Coder la fonctionnalité', completed: false }
];

// Mettre à jour une tâche de manière immuable
const updatedTasks: Task[] = tasks.map(task =>
  task.id === 1
    ? { ...task, completed: true } // Crée une nouvelle tâche avec 'completed' à true
    : task                         // Conserve les autres tâches telles quelles
);

console.log('Tâches originales:', tasks);
// Output: [ { id: 1, description: 'Faire les courses', completed: false }, { id: 2, d

console.log('Tâches mises à jour (nouvel array):', updatedTasks);
// Output: [ { id: 1, description: 'Faire les courses', completed: true }, { id: 2, de

console.log(tasks === updatedTasks);       // false (ce sont deux tableaux différents)
console.log(tasks[0] === updatedTasks[0]); // false (l'objet tâche a été recréé)
console.log(tasks[1] === updatedTasks[1]); // true (cet objet tâche n'a pas été modifi
```

B. Copie Superficielle vs. Copie Profonde

- **Copie superficielle (Shallow Copy) :** C'est ce que l'opérateur de propagation `...` fait pour les objets et les tableaux. Il copie les propriétés de l'objet/tableau de niveau supérieur. Si l'objet/tableau contient des objets imbriqués, seules les *références* à ces objets imbriqués sont copiées, pas les objets imbriqués eux-mêmes. Cela peut entraîner des effets de bord si vous modifiez un objet imbriqué.

```
const utilisateurProfil = {
  nom: 'Marc',
  infos: {
    email: 'marc@example.com',
    adresse: 'Rue A'
  }
};

const profilCopieSuperficielle = { ...utilisateurProfil };
profilCopieSuperficielle.infos.adresse = 'Nouvelle Rue B'; // Modifie l'objet imbr

console.log(utilisateurProfil.infos.adresse); // Output: Nouvelle Rue B
```

- **Copie Profonde (Deep Copy) :** Pour copier un objet et tous ses objets imbriqués de manière totalement indépendante, vous avez besoin d'une copie profonde. Il n'y a pas de méthode native simple en JavaScript pour une copie profonde généralisée.
  - Pour les objets simples sans fonctions ou `Date` / `RegExp`, `JSON.parse(JSON.stringify(objet))` est une astuce courante mais a ses limites.
  - Pour des cas plus complexes, des bibliothèques comme Lodash (`_.cloneDeep()`) sont souvent utilisées.