signatures d'index (Index Signatures) en TypeScript. C'est une fonctionnalité très puissante pour décrire des objets dont vous ne connaissez pas toutes les propriétés à l'avance, mais dont vous connaissez le **type de la clé** et le **type de la valeur** correspondante.

Les Signatures d'Index (Index Signatures) en TypeScript

Introduction

En TypeScript, une **signature d'index** est une manière de spécifier le type des propriétés d'un objet lorsque vous ne connaissez pas les noms exacts de ces propriétés à l'avance, mais que vous savez que toutes les propriétés (ou un sous-ensemble d'entre elles) respecteront un certain modèle de type pour leurs clés et leurs valeurs.

Imaginez que vous ayez un "dictionnaire" ou une "carte" où les clés sont des mots (chaînes de caractères) et les valeurs sont leurs définitions (chaînes de caractères). Vous ne pouvez pas lister tous les mots possibles dans l'interface, mais vous savez que chaque "mot" sera une `string` et chaque "définition" sera une `string`. C'est exactement le cas d'utilisation d'une signature d'index.

Elles sont particulièrement utiles pour :

- Les objets qui servent de **dictionnaires** ou de **tables de hachage**.
- Les données provenant d'API où les noms des propriétés sont dynamiques.
- La modélisation de types de données flexibles.

1. Syntaxe de Base

La syntaxe d'une signature d'index est la suivante :

```
interface MonType {
  [nomDeLaClé: TypeDeLaClé]: TypeDeLaValeur;
}
```

- **`nomDeLaClé`** : C'est juste un nom de paramètre (comme `key`, `index`, `propName`) qui sera utilisé pour représenter la clé dans cette signature. Ce nom n'a pas d'importance pour l'utilisation externe de l'interface.
- **`TypeDeLaClé`** : Le type des clés des propriétés. En TypeScript, `TypeDeLaClé` peut être seulement :
  - `string`
  - `number`

    - `symbol` (moins courant pour les index signatures)
    - Une union de `string` et `number` (voir les considérations).
- **`TypeDeLaValeur`** : Le type des valeurs associées à ces clés. Il peut s'agir de n'importe quel type valide en TypeScript (primitif, objet, union, etc.).

2. Cas d'Utilisation et Exemples

2.1. Objet de Type "Dictionnaire" (Clés de type `string`)

C'est l'utilisation la plus courante. Imaginez un objet où chaque clé est le nom d'un pays et chaque valeur est sa capitale.

```
interface CapitalesDuMonde {
  [pays: string]: string; // Clés de type 'string', Valeurs de type 'string'
}

const mesCapitales: CapitalesDuMonde = {
  France: 'Paris',
  Allemagne: 'Berlin',
  Espagne: 'Madrid',
  // 'Italie': 123 // Erreur TypeScript: Type 'number' n'est pas assignable au type 's
};

console.log(mesCapitales.France);      // Output: Paris
console.log(mesCapitales['Allemagne']); // Output: Berlin

// Vous pouvez ajouter de nouvelles propriétés dynamiquement
mesCapitales.Japon = 'Tokyo';
console.log(mesCapitales.Japon);       // Output: Tokyo

// Tentative d'accès à une propriété inexistante (ne provoque pas d'erreur de compilat
console.log(mesCapitales.Canada); // Output: undefined (comportement normal de JS pour
```

2.2. Objets Similaires aux Tableaux (Clés de type `number`)

Bien que les tableaux soient généralement typés avec `Array<T>` ou `T[]`, vous pouvez utiliser une signature d'index numérique pour un objet qui agit comme un tableau, par exemple un objet avec des clés numériques représentant des positions.

```
interface CollectionNumerique {
  [index: number]: string; // Clés de type 'number', Valeurs de type 'string'
}

const maCollection: CollectionNumerique = {
  0: 'Premier',
  1: 'Deuxième',
```

```
  5: 'Sixième',
  // 'trois': 'Troisième' // Erreur TypeScript: Le type de propriété 'trois' n'est pas
};

console.log(maCollection[0]); // Output: Premier
console.log(maCollection[5]); // Output: Sixième
```

2.3. Combinaison avec des Propriétés Nommées

Vous pouvez avoir des propriétés nommées (fixes) et une signature d'index dans la même interface. **La règle clé ici est que le type de la propriété nommée doit être assignable au type de la valeur de la signature d'index.**

```
interface UtilisateurAvecMeta {
  id: number;
  nom: string;
  [cleMeta: string]: string | number; // Toutes les autres propriétés string doivent ê
}

const user: UtilisateurAvecMeta = {
  id: 123,
  nom: 'Alice',
  ville: 'Paris',      // Ok, string est assignable à string | number
  codePostal: 75001,   // Ok, number est assignable à string | number
  // isAdmin: true // Erreur TypeScript: Type 'boolean' n'est pas assignable au type '
};

console.log(user.id);          // Output: 123
console.log(user.ville);       // Output: Paris
console.log(user['codePostal']); // Output: 75001
```

Pourquoi cette règle ? Si une propriété nommée comme `nom` n'était pas assignable à `string | number` (par exemple, si `nom` était `boolean`), alors `user.nom` serait de type `boolean` mais `user['nom']` (qui utilise la signature d'index) serait de type `string | number`, créant une incohérence. TypeScript force cette compatibilité pour garantir la sécurité des types.

2.4. Signature d'Index en Lecture Seule (`readonly`)

Vous pouvez rendre les valeurs accessibles via une signature d'index en lecture seule. Cela empêche la modification des propriétés une fois qu'elles sont initialisées.

```
interface ConfigConstante {
  readonly [key: string]: string;
}

const maConfig: ConfigConstante = {
  API_KEY: 'abc123xyz',
```

```
  VERSION: '1.0.0',
};

// maConfig.API_KEY = 'nouvelle_cle'; // Erreur TypeScript: Impossible d'assigner à 'A
console.log(maConfig.VERSION); // Output: 1.0.0
```

3. Considérations Importantes et Pièges Fréquents

3.1. Une Seule Signature d'Index par Type (pour `string` / `number`)

Vous ne pouvez pas avoir plusieurs signatures d'index de type `string` ou `number` dans la même interface avec des types de valeur différents.

```
interface ErreurExemple {
  [key: string]: string;
  // [anotherKey: string]: number; // Erreur TypeScript: Une interface ne peut déclare
}
```

Si vous avez besoin de valeurs de types différents pour différentes clés, vous devez soit utiliser une union de types pour la valeur de la signature d'index, soit des propriétés nommées spécifiques.

3.2. Relation entre les Signatures d'Index `number` et `string`

En JavaScript, toutes les clés d'objets sont finalement converties en chaînes de caractères. Par exemple, `obj[0]` est équivalent à `obj['0']`. TypeScript prend cela en compte.

Si vous avez une signature d'index `number` ET une signature d'index `string`, le type de la valeur de la signature `number` **doit être assignable** au type de la valeur de la signature `string`.

```
interface MixedIndex {
  [key: number]: number;      // Les clés numériques donnent des 'number'
  [key: string]: number | string; // Les clés string (y compris les numériques convert
}

const data: MixedIndex = {
  0: 10,       // ok, 10 est number
  'name': 'Alice', // ok, 'Alice' est string
  'age': 30,     // ok, 30 est number
  // 'isValid': true // Erreur: boolean n'est pas assignable à number | string
};

console.log(data[0]);      // Output: 10 (type inferred as number)
console.log(data['name']); // Output: Alice (type inferred as string)
console.log(data['0']);    // Output: 10 (type inferred as number | string, mais en ru
```

La valeur `number` est assignable à `number` | `string`. Si la signature `[key: string]` avait été `boolean`, il y aurait eu une erreur, car `number` (de `[key: number]`) n'est pas assignable à `boolean`.

3.3. Clés de type `symbol`

Bien que `symbol` soit un type de clé valide, il est moins couramment utilisé pour les signatures d'index génériques car les symboles sont uniques et ne sont généralement pas utilisés pour un accès dynamique de masse comme les chaînes ou les nombres.

```
const MY_SYMBOL = Symbol('myKey');

interface SymbolIndexed {
  [key: symbol]: string;
}

const myMap: SymbolIndexed = {
  [MY_SYMBOL]: 'valeur symbolique'
};

console.log(myMap[MY_SYMBOL]); // Output: valeur symbolique
```

Conclusion

Les signatures d'index sont un outil puissant en TypeScript pour modéliser des objets à structure dynamique. Elles permettent de maintenir une forte sécurité des types même lorsque les noms exacts des propriétés ne sont pas connus à l'avance, en spécifiant les types attendus pour les clés et les valeurs. Une bonne compréhension de leurs règles et de leurs interactions avec les propriétés nommées est essentielle pour les utiliser efficacement.