Documentation du Diagramme de Configuration TypeScript

Introduction

Ce document fournit une explication détaillée des concepts et configurations liés à TypeScript, tels que présentés dans le diagramme fourni. Chaque section abordera un aspect spécifique, en fournissant des informations claires et concises pour une meilleure compréhension.

1. Exclure des fichiers de transpilation

Cette section expliquera comment configurer TypeScript pour exclure certains fichiers ou répertoires du processus de transpilation. Cela est crucial pour optimiser les temps de compilation et éviter la transpilation de fichiers inutiles ou de dépendances.

1.1 Exclure un fichier cible

1.2 Tous les fichiers .dev.ts

1.3 Les .dev.ts de tout répertoire

1.4 Les fichiers de node_modules

2. Opérateur Spread

Cette section décrira l'utilisation de l'opérateur spread en TypeScript, un outil puissant pour la manipulation des tableaux et des objets.

3. Type primitif vs référence

Une explication des différences fondamentales entre les types primitifs et les types de référence en TypeScript, et comment cela affecte la manière dont les données sont stockées et manipulées.

4. Propriétés d'index (signature d'index)

Cette section couvrira les propriétés d'index, également connues sous le nom de signatures d'index, qui permettent de définir des types pour les objets dont les noms de propriétés ne sont pas connus à l'avance.

5. Let - Var - Const

Une comparaison des mots-clés `let`, `var` et `const` pour la déclaration de variables en TypeScript, en soulignant leurs portées et leurs comportements respectifs.

6. Copier un objet de type référence

Cette section expliquera les différentes méthodes pour copier des objets de type référence en TypeScript, en abordant les concepts de copie superficielle et de copie profonde.

7. Nombre de paramètres variable

Comment gérer un nombre variable de paramètres dans les fonctions TypeScript, en utilisant les paramètres rest.

8. Nullish Coalescing

Une explication de l'opérateur `??` (nullish coalescing) en TypeScript, qui fournit une manière concise de gérer les valeurs `null` et `undefined`.

9. Déstructuration

Cette section détaillera la déstructuration en TypeScript, une fonctionnalité qui permet d'extraire des valeurs de tableaux ou d'objets dans des variables distinctes.

9.1 Avec les tableaux

9.2 Avec les objets

10. Les décorateurs

Une introduction aux décorateurs en TypeScript, une fonctionnalité expérimentale qui permet d'ajouter des annotations et une méta-programmation à des classes, des méthodes, des accesseurs, des propriétés ou des paramètres.

10.1 Ajout de décorateurs de classe

11. Map

Cette section expliquera l'objet `Map` en TypeScript, une collection de paires clé-valeur où les clés peuvent être de n'importe quel type.

12. Ajout de décorateurs de propriété

Une explication spécifique sur l'ajout de décorateurs aux propriétés des classes TypeScript.

13. Expliquer les clés du TSCONFIG

Cette section se concentrera sur les clés de configuration importantes dans le fichier `tsconfig.json` de TypeScript.

13.1 Include

13.2 File

13.3 RootDir

13.4 RemoveComments

13.5 NoEmitOnError

1.1 Exclure un fichier cible

Dans un projet TypeScript, il est souvent nécessaire d'exclure des fichiers spécifiques du processus de transpilation. Cela peut être dû à plusieurs raisons : le fichier est généré automatiquement, il contient du code non compatible avec la version cible de TypeScript, ou il s'agit d'un fichier de test qui ne doit pas être inclus dans la version finale de l'application. Pour exclure un fichier cible, vous pouvez utiliser la propriété `exclude` dans votre fichier `tsconfig.json`. Cette propriété accepte un tableau de motifs de fichiers (glob patterns) qui spécifient les fichiers et dossiers à ignorer lors de la compilation. Par exemple, si vous avez un fichier nommé `temp.ts` que vous souhaitez exclure, votre `tsconfig.json` pourrait ressembler à ceci :

```
{
  "compilerOptions": {
    "outDir": "./dist"
  },
  "exclude": [
    "temp.ts"
  ]
}
```

L'utilisation de `exclude` est essentielle pour maintenir un environnement de développement propre et pour s'assurer que seuls les fichiers pertinents sont transpilés, ce qui réduit le temps de compilation et la taille de la sortie finale. Il est important de noter que les chemins spécifiés dans `exclude` sont relatifs au fichier `tsconfig.json`.

1.2 Tous les fichiers .dev.ts

Il est courant dans les projets de développement d'avoir des fichiers spécifiques à l'environnement de développement, tels que des scripts de débogage ou des configurations de test, qui se terminent par une extension comme `.dev.ts`. Ces fichiers ne sont généralement pas destinés à être inclus dans la version de production de l'application. Pour exclure tous les fichiers portant cette extension, vous pouvez utiliser un motif générique (wildcard) dans la propriété `exclude` de votre `tsconfig.json`. Le motif `*.dev.ts` permet de cibler tous les fichiers qui se terminent par `.dev.ts` dans le répertoire courant du `tsconfig.json` et ses sous-répertoires. Voici un exemple de configuration :

```
{
  "compilerOptions": {
    "outDir": "./dist"
  },
  "exclude": [
    "*.dev.ts"
  ]
}
```

Cette approche est très efficace pour gérer les fichiers spécifiques au développement, garantissant qu'ils ne sont pas accidentellement inclus dans les builds de production. Cela aide à maintenir la propreté du code base et à éviter les erreurs potentielles liées à l'inclusion de code non pertinent pour l'environnement de production.

1.3 Les .dev.ts de tout répertoire

Pour une exclusion plus globale des fichiers `.dev.ts`, indépendamment de leur emplacement dans la structure du projet, vous pouvez utiliser un motif de répertoire récursif. Le motif `**/*.dev.ts` indique à TypeScript d'exclure tous les fichiers se terminant par `.dev.ts` dans n'importe quel sous-répertoire du répertoire où se trouve le `tsconfig.json`. C'est particulièrement utile dans les grands projets avec des structures de dossiers complexes où les fichiers de développement peuvent être dispersés. Voici comment vous pouvez configurer cela :

```
{
  "compilerOptions": {
    "outDir": "./dist"
  },
  "exclude": [
    "**/*.dev.ts"
  ]
}
```

Cette configuration assure une exclusion complète et robuste des fichiers de développement, peu importe où ils sont placés dans votre arborescence de projet. C'est une pratique recommandée pour les projets de taille moyenne à grande afin de simplifier la gestion des dépendances et des fichiers de build.

1.4 Les fichiers de node_modules

Le répertoire `node_modules` contient toutes les dépendances de votre projet installées via npm ou yarn. Par défaut, TypeScript exclut déjà ce répertoire de la compilation. Cependant, il est bon de comprendre pourquoi et comment cela fonctionne. Inclure `node_modules` dans la compilation ralentirait considérablement le processus et pourrait introduire des erreurs dues à la transpilation de code tiers qui n'est pas destiné à être transpilé par votre configuration. Bien que `node_modules` soit implicitement exclu, vous pouvez l'ajouter explicitement à la propriété `exclude` pour des raisons de clarté ou si vous avez des configurations spécifiques qui pourraient outrepasser ce comportement par défaut. Voici un exemple :

```
{
  "compilerOptions": {
    "outDir": "./dist"
  },
  "exclude": [
    "node_modules"
  ]
}
```

Il est crucial de ne pas transpiler les dépendances installées, car elles sont déjà distribuées dans un format utilisable (souvent JavaScript compilé) et leur recompilation est inutile et potentiellement problématique. L'exclusion de `node_modules` est une pratique standard et essentielle pour la performance et la stabilité des projets TypeScript.

2. Opérateur Spread

L'opérateur spread (`...`) est une fonctionnalité puissante introduite en ECMAScript 2015 (ES6) et pleinement prise en charge par TypeScript. Il permet d'étendre des itérables (comme des tableaux ou des chaînes de caractères) ou des objets dans des endroits où zéro ou plusieurs arguments (pour les appels de fonction) ou éléments (pour les littéraux de tableau) ou paires clé-valeur (pour les littéraux d'objet) sont attendus. Cet opérateur est extrêmement utile pour la manipulation de données de manière concise et lisible, évitant ainsi des boucles ou des méthodes plus verbeuses.

2.1 Utilisation avec les tableaux

L'opérateur spread est couramment utilisé avec les tableaux pour créer de nouveaux tableaux sans modifier les originaux, ou pour combiner plusieurs tableaux. Voici quelques cas d'utilisation :

- **Copie de tableaux** : Il permet de créer une copie superficielle d'un tableau existant. C'est une alternative plus courte et plus lisible à `Array.prototype.slice()` .

  `typescript const arr1 = [1, 2, 3]; const arr2 = [...arr1]; // arr2 est [1, 2, 3], une nouvelle instance de tableau console.log(arr1 === arr2); // false`

- **Concaténation de tableaux** : Il simplifie la fusion de plusieurs tableaux en un seul.

  `typescript const arrA = [1, 2]; const arrB = [3, 4]; const combinedArr = [...arrA, ...arrB]; // combinedArr est [1, 2, 3, 4]`

- **Ajout d'éléments à un tableau** : Il facilite l'insertion d'éléments au début, au milieu ou à la fin d'un tableau sans utiliser `push` , `unshift` ou `splice` .

  `typescript const originalArr = [2, 3]; const newArrStart = [1, ...originalArr]; // newArrStart est [1, 2, 3] const newArrEnd = [...originalArr, 4]; // newArrEnd est [2, 3, 4] const newArrMiddle = [1, ...originalArr, 4]; // newArrMiddle est [1, 2, 3, 4]`

2.2 Utilisation avec les objets

Avec les objets, l'opérateur spread permet de copier les propriétés énumérables d'un objet vers un nouvel objet. C'est particulièrement utile pour fusionner des objets ou pour créer de nouvelles instances d'objets avec des propriétés modifiées sans altérer l'objet original.

- **Copie d'objets** : Il crée une copie superficielle d'un objet. Si l'objet contient des objets imbriqués, ces derniers seront copiés par référence.

  `typescript const obj1 = { a: 1, b: 2 }; const obj2 = { ...obj1 }; // obj2 est { a: 1, b: 2 }, une nouvelle instance d'objet console.log(obj1 === obj2); // false`

- **Fusion d'objets** : Il permet de combiner les propriétés de plusieurs objets en un seul. Si des propriétés ont le même nom, la valeur de la dernière propriété rencontrée sera utilisée.

  ```typescript const objA = { a: 1, b: 2 }; const objB = { c: 3, d: 4 }; const combinedObj = { ...objA, ...objB }; // combinedObj est { a: 1, b: 2, c: 3, d: 4 }

  const objC = { a: 1, b: 2 }; const objD = { b: 3, c: 4 }; const mergedObj = { ...objC, ...objD }; // mergedObj est { a: 1, b: 3, c: 4 } (b est écrasé) ```

- **Mise à jour d'objets de manière immuable** : C'est une technique très courante en React et Redux pour mettre à jour l'état sans le muter directement.

  `typescript const user = { name: 'Alice', age: 30, city: 'Paris' }; const updatedUser = { ...user, age: 31 }; // updatedUser est { name: 'Alice', age: 31, city: 'Paris' } console.log(user === updatedUser); // false`

## 2.3 Utilisation avec les arguments de fonction (paramètres rest)

Bien que le diagramme ne le mentionne pas explicitement sous
  ```

3. Type primitif vs référence

En TypeScript (et JavaScript), il est fondamental de comprendre la distinction entre les types primitifs et les types de référence, car cela affecte la manière dont les valeurs sont stockées en mémoire, copiées et comparées. Cette distinction est cruciale pour éviter des comportements inattendus dans votre code, notamment lors de la manipulation de données.

3.1 Types Primitifs

Les types primitifs représentent des valeurs simples et immuables. Lorsque vous travaillez avec des types primitifs, la variable contient directement la valeur. En TypeScript, les types primitifs incluent :

- **`number`** : Pour les nombres entiers et à virgule flottante (par exemple, `10`, `3.14`).
- **`string`** : Pour les séquences de caractères (par exemple, `"bonjour"`, `"TypeScript"`).
- **`boolean`** : Pour les valeurs logiques `true` ou `false`.
- **`null`** : Représente l'absence intentionnelle de toute valeur d'objet.
- **`undefined`** : Représente une variable qui n'a pas encore été assignée d'une valeur.
- **`symbol`** (ES6) : Pour les valeurs uniques et immuables, souvent utilisées comme clés de propriété d'objet.
- **`bigint`** (ES2020) : Pour les nombres entiers arbitrairement grands.

Caractéristiques des types primitifs :

1. **Stockage par valeur** : Lorsque vous assignez une variable primitive à une autre, la valeur est copiée. Les deux variables sont indépendantes l'une de l'autre.

   `typescript let a = 10; let b = a; // La valeur de 'a' (10) est copiée dans 'b' b = 20; // Modifier 'b' n'affecte pas 'a' console.log(a); // 10 console.log(b); // 20`

2. **Immuabilité** : Les valeurs primitives ne peuvent pas être modifiées. Lorsque vous effectuez une opération qui semble modifier une primitive (par exemple,

`str.toUpperCase()`), une nouvelle valeur primitive est en fait créée et renvoyée, tandis que l'originale reste inchangée.

```typescript let greeting = "Hello"; greeting.toUpperCase(); // Renvoie "HELLO", mais 'greeting' reste "Hello" console.log(greeting); // "Hello"```

3. **Comparaison par valeur** : Deux primitives sont considérées comme égales si elles ont la même valeur.

```typescript console.log(10 === 10); // true console.log("abc" === "abc"); // true console.log(null === null); // true```

3.2 Types de Référence

Les types de référence, en revanche, ne contiennent pas directement la valeur. Au lieu de cela, la variable contient une référence (une adresse mémoire) à l'emplacement où l'objet est stocké en mémoire. En TypeScript, les types de référence incluent :

- **`object`** : Le type de base pour tous les objets non primitifs.
- **`Array`** : Les tableaux sont des objets.
- **`Function`** : Les fonctions sont des objets.
- **`Date`** : Les objets Date.
- **`RegExp`** : Les expressions régulières.
- Les instances de classes personnalisées.

Caractéristiques des types de référence :

1. **Stockage par référence** : Lorsque vous assignez une variable de référence à une autre, la référence (l'adresse mémoire) est copiée, et non la valeur de l'objet lui-même. Les deux variables pointent alors vers le même objet en mémoire.

```typescript let obj1 = { name: "Alice" }; let obj2 = obj1; // 'obj2' pointe vers le même objet que 'obj1' obj2.name = "Bob"; // Modifier l'objet via 'obj2' affecte aussi 'obj1' console.log(obj1.name); // "Bob" console.log(obj2.name); // "Bob"```
```

2. **Mutabilité** : Les objets référencés peuvent être modifiés après leur création. Vous pouvez ajouter, supprimer ou modifier des propriétés de l'objet via n'importe quelle variable qui y fait référence.

   `typescript const myArray = [1, 2, 3]; myArray.push(4); // Le tableau est modifié console.log(myArray); // [1, 2, 3, 4]`

3. **Comparaison par référence** : Deux variables de référence sont considérées comme égales uniquement si elles pointent vers le *même* objet en mémoire, et non si elles ont des valeurs structurellement identiques.

   ```typescript const objA = { value: 1 }; const objB = { value: 1 }; const objC = objA;

   console.log(objA === objB); // false (objA et objB sont des objets différents en mémoire) console.log(objA === objC); // true (objA et objC pointent vers le même objet) ```

## 3.3 Implications

Comprendre cette distinction est vital pour :

- **Éviter les effets de bord inattendus** : Si vous modifiez un objet via une référence, toutes les autres références à cet objet verront la modification.
- **Effectuer des copies correctes** : Pour créer une copie indépendante d'un objet de référence, vous devez effectuer une copie superficielle ( `{...obj}` ou `Object.assign()` ) ou une copie profonde (pour les objets imbriqués, nécessitant souvent une bibliothèque ou une implémentation manuelle).
- **Optimisation des performances** : La copie de primitives est rapide, tandis que la copie d'objets peut être coûteuse en fonction de leur complexité.

En résumé, les primitives sont des valeurs directes et immuables, copiées par valeur, tandis que les types de référence sont des objets complexes et mutables, copiés par référence. Maîtriser cette différence est une étape clé pour écrire du code TypeScript et JavaScript robuste et prévisible.
   ```

4. Propriétés d'index (signature d'index)

Les propriétés d'index, ou signatures d'index, en TypeScript sont un moyen puissant de définir des types pour les objets dont les noms de propriétés ne sont pas connus à l'avance, mais dont les valeurs ont un type cohérent. Cela est particulièrement utile lorsque vous travaillez avec des structures de données dynamiques, comme des dictionnaires, des hachages ou des objets qui peuvent être indexés de manière similaire à des tableaux.

4.1 Définition d'une signature d'index

Une signature d'index spécifie le type des clés et le type des valeurs pour les propriétés d'un objet. Il existe deux types de clés possibles pour une signature d'index : `string` ou `number`. TypeScript ne prend en charge que ces deux types pour les clés d'index, car JavaScript convertit toutes les clés d'objet en chaînes de caractères en interne. Si vous utilisez un `number` comme type de clé, TypeScript le convertira en `string` en arrière-plan.

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

```
interface MyDictionary {
  [key: string]: string; // Clés de type string, valeurs de type string
}

interface MyNumberDictionary {
  [index: number]: string; // Clés de type number, valeurs de type string
}
```

Dans l'exemple ci-dessus, `MyDictionary` est une interface qui peut avoir n'importe quel nombre de propriétés, tant que leurs clés sont des chaînes de caractères et que leurs valeurs sont également des chaînes de caractères. De même, `MyNumberDictionary` permet des clés numériques avec des valeurs de chaîne.

4.2 Cas d'utilisation

Les signatures d'index sont extrêmement utiles dans plusieurs scénarios :

- **Objets de type dictionnaire** : Lorsque vous attendez un objet où les clés sont arbitraires (par exemple, des IDs, des noms d'utilisateur) et que les valeurs sont d'un type spécifique.

```typescript interface UserScores {

}

const scores: UserScores = { "Alice": 150, "Bob": 200, "Charlie": 120 };

console.log(scores["Alice"]); // 150 // scores["David"] = "abc"; // Erreur: Type 'string' n'est pas assignable au type 'number'. ```

- **Objets avec des propriétés connues et inconnues** : Vous pouvez combiner des propriétés nommées explicitement avec une signature d'index. Cependant, toutes les propriétés nommées doivent être compatibles avec le type de la signature d'index.

  ```typescript interface ProductInfo { id: number; name: string; [key: string]: string | number; // Toutes les propriétés doivent être string ou number }

  const product: ProductInfo = { id: 123, name: "Laptop", "color": "Silver", "weight": 2.5 };

  // product.description = true; // Erreur: Type 'boolean' n'est pas assignable au type 'string | number'. ```

- **Tableaux comme objets** : Bien que les tableaux aient déjà des signatures d'index numériques implicites, vous pouvez les utiliser pour des cas plus spécifiques.

  ```typescript interface StringArray {

  }

  const myArray: StringArray = ["hello", "world"]; // myArray[2] = 123; // Erreur: Type 'number' n'est pas assignable au type 'string'. ```

## 4.3 Restrictions et considérations

- **Un seul type de signature d'index** : Une interface ou un type ne peut avoir qu'une seule signature d'index de type `string` et une seule de type `number`. Si les deux sont présentes, la signature numérique doit être un sous-type de la signature de chaîne, car JavaScript convertit les clés numériques en chaînes.

  ```typescript interface MixedIndex {

index: number: string; // Erreur: La signature d'index numérique doit être un sous-type de la signature d'index de chaîne. }

interface CorrectMixedIndex { key: string: string | number; index: number: string; // OK, car string est un sous-type de string | number } ```

- **Compatibilité des propriétés connues** : Toutes les propriétés explicitement définies dans une interface doivent être compatibles avec le type de retour de la signature d'index. Si une propriété connue a un type qui n'est pas compatible avec la signature d'index, TypeScript signalera une erreur.

Les signatures d'index sont un outil essentiel pour modéliser des données flexibles et dynamiques en TypeScript, offrant une grande flexibilité tout en maintenant la sécurité des types.

## 5. Let - Var - Const

En JavaScript (et par extension en TypeScript), la déclaration de variables a évolué avec l'introduction de `let` et `const` en ECMAScript 2015 (ES6), offrant des alternatives plus robustes et prévisibles à l'ancien mot-clé `var`. Comprendre les différences entre ces trois mots-clés est fondamental pour écrire du code moderne, propre et sans erreurs, notamment en ce qui concerne la portée (scope) et la mutabilité des variables.

### 5.1 `var` : L'ancienne approche

Avant ES6, `var` était le seul moyen de déclarer des variables en JavaScript. Il présente des caractéristiques qui peuvent mener à des comportements inattendus et à des bugs, principalement en raison de sa portée.

- **Portée de fonction (Function Scope)** : Les variables déclarées avec `var` sont limitées à la fonction dans laquelle elles sont déclarées, et non au bloc de code (comme un `if` ou un `for`) dans lequel elles se trouvent. Si `var` est déclaré en dehors de toute fonction, il a une portée globale.

`typescript function exampleVarScope() { if (true) { var x = 10; // x est accessible ici } console.log(x); // 10, x est accessible en`

`dehors du bloc if } exampleVarScope(); // console.log(x); // Erreur: x n'est pas défini en dehors de la fonction`

- **Hoisting** : Les déclarations `var` sont

remontées (hoisted) en haut de leur portée de fonction ou globale. Cela signifie que vous pouvez utiliser une variable `var` avant de la déclarer, mais sa valeur sera `undefined`.

```
 ```typescript
console.log(y); // undefined
var y = 5;
console.log(y); // 5
```
```

- **Redéclaration** : `var` permet la redéclaration de variables dans la même portée sans erreur, ce qui peut entraîner des confusions et des écrasements de valeurs inattendus.

  `typescript var z = 1; var z = 2; // Aucune erreur, z est maintenant 2 console.log(z); // 2`

En raison de ces comportements, l'utilisation de `var` est généralement déconseillée dans le code moderne au profit de `let` et `const`.

5.2 `let` : La portée de bloc

Introduit en ES6, `let` résout de nombreux problèmes associés à `var` en introduisant la portée de bloc.

- **Portée de bloc (Block Scope)** : Les variables déclarées avec `let` sont limitées au bloc de code (délimité par des accolades `{}`) dans lequel elles sont définies. Cela inclut les boucles `for`, les blocs `if`, etc.

  `typescript function exampleLetScope() { if (true) { let a = 10; console.log(a); // 10 } // console.log(a); // Erreur: a n'est pas défini en dehors du bloc if } exampleLetScope();`

- **Hoisting (avec zone morte temporelle)** : Les déclarations `let` sont également remontées, mais elles sont placées dans une

zone morte temporelle (Temporal Dead Zone - TDZ). Cela signifie que vous ne pouvez pas accéder à une variable `let` avant sa déclaration, même si elle est remontée. Tenter de le faire entraînera une erreur de référence.

```typescript
// console.log(b); // Erreur de référence: Cannot access 'b' before initialization
let b = 5;
console.log(b); // 5
```

- **Pas de redéclaration** : `let` n'autorise pas la redéclaration d'une variable dans la même portée, ce qui aide à prévenir les erreurs de programmation.

  `typescript let c = 1; // let c = 2; // Erreur: Cannot redeclare block-scoped variable 'c'.`

`let` est le choix préféré pour la déclaration de variables mutables dans le code moderne.

5.3 `const` : Les constantes immuables

`const` est également introduit en ES6 et est utilisé pour déclarer des constantes. Comme `let`, `const` a une portée de bloc et est sujet à la zone morte temporelle.

- **Portée de bloc (Block Scope)** : Similaire à `let`, les variables `const` sont limitées au bloc de code dans lequel elles sont définies.

  `typescript function exampleConstScope() { if (true) { const d = 10; console.log(d); // 10 } // console.log(d); // Erreur: d n'est pas défini en dehors du bloc if } exampleConstScope();`

- **Hoisting (avec zone morte temporelle)** : Comme `let`, `const` est remonté mais inaccessible avant sa déclaration.

  `typescript // console.log(e); // Erreur de référence: Cannot access 'e' before initialization const e = 5; console.log(e); // 5`

- **Pas de redéclaration** : `const` n'autorise pas la redéclaration d'une variable dans la même portée.

`typescript const f = 1; // const f = 2; // Erreur: Cannot redeclare block-scoped variable 'f'.`

- **Immuabilité de la liaison** : La principale caractéristique de `const` est qu'une fois qu'une valeur est assignée à une variable `const`, cette liaison ne peut pas être réassignée. Cela ne signifie pas que la valeur elle-même est immuable si c'est un objet ou un tableau.

  ```typescript const PI = 3.14159; // PI = 3.0; // Erreur: Assignment to constant variable.

  const myObject = { name: "Alice" }; myObject.name = "Bob"; // Ceci est autorisé, car l'objet lui-même est mutable console.log(myObject); // { name: "Bob" }

  // myObject = { name: "Charlie" }; // Erreur: Assignment to constant variable. ```

### 5.4 Quand utiliser quoi ?

- **`const`** : Utilisez `const` par défaut. Si vous n'avez pas l'intention de réassigner une variable, `const` est le choix le plus sûr et le plus clair. Cela aide à prévenir les bugs liés aux réassignations accidentelles et rend le code plus facile à raisonner.
- **`let`** : Utilisez `let` lorsque vous savez que la valeur de la variable devra être réassignée à un moment donné (par exemple, dans une boucle, ou une variable qui change d'état).
- **`var`** : Évitez d'utiliser `var` dans le nouveau code. Il est principalement pertinent pour la compatibilité avec l'ancien code JavaScript.

En résumé, `let` et `const` offrent un contrôle plus précis sur la portée et la mutabilité des variables, ce qui conduit à un code plus robuste et plus facile à maintenir en TypeScript et JavaScript moderne.

## 6. Copier un objet de type référence

En TypeScript (et JavaScript), la copie d'objets de type référence est un concept crucial en raison de la manière dont les objets sont stockés et manipulés en mémoire. Contrairement aux types primitifs qui sont copiés par valeur, les objets sont copiés par référence. Cela signifie que lorsque vous assignez un objet à une nouvelle variable, vous ne créez pas une nouvelle copie de l'objet, mais plutôt une nouvelle référence au
  ```

même objet en mémoire. Toute modification apportée via l'une ou l'autre des références affectera l'objet original.

Pour créer une copie indépendante d'un objet, il est nécessaire d'effectuer une opération de copie explicite. Il existe deux types de copies : la copie superficielle (shallow copy) et la copie profonde (deep copy).

6.1 Copie Superficielle (Shallow Copy)

Une copie superficielle crée un nouvel objet, puis copie les propriétés de l'objet original vers le nouvel objet. Si les propriétés de l'objet original sont des types primitifs, leurs valeurs sont copiées. Cependant, si les propriétés sont elles-mêmes des objets (types de référence), ce sont les *références* à ces objets imbriqués qui sont copiées, et non les objets imbriqués eux-mêmes. Cela signifie que l'objet copié et l'objet original partageront les mêmes objets imbriqués.

Plusieurs méthodes permettent de réaliser une copie superficielle :

- **Opérateur Spread (`...`)** : C'est la méthode la plus moderne et la plus concise pour copier des objets. Elle est largement utilisée pour sa lisibilité.

  ```typescript const originalObject = { a: 1, b: { c: 2 } }; const shallowCopy = { ...originalObject };

  console.log(shallowCopy); // { a: 1, b: { c: 2 } } console.log(originalObject === shallowCopy); // false (ce sont des objets différents) console.log(originalObject.b === shallowCopy.b); // true (ils partagent la même référence pour l'objet imbriqué)

  shallowCopy.a = 10; shallowCopy.b.c = 20; // Ceci affecte aussi originalObject.b.c

  console.log(originalObject); // { a: 1, b: { c: 20 } } console.log(shallowCopy); // { a: 10, b: { c: 20 } } ```

- **`Object.assign()`** : Cette méthode copie toutes les propriétés énumérables et propres d'un ou plusieurs objets sources vers un objet cible. Elle est souvent utilisée pour fusionner des objets ou pour créer une copie superficielle.

  ```typescript const originalObject = { a: 1, b: { c: 2 } }; const shallowCopy = Object.assign({}, originalObject);

console.log(shallowCopy); // { a: 1, b: { c: 2 } } console.log(originalObject === shallowCopy); // false console.log(originalObject.b === shallowCopy.b); // true ```

- **`Array.prototype.slice()` (pour les tableaux)** : Bien que spécifiquement pour les tableaux, `slice()` crée une copie superficielle d'une portion d'un tableau.

  ```typescript const originalArray = [1, { a: 2 }, 3]; const shallowCopyArray = originalArray.slice();

  console.log(shallowCopyArray); // [1, { a: 2 }, 3] console.log(originalArray === shallowCopyArray); // false console.log(originalArray[1] === shallowCopyArray[1]); // true ```

## 6.2 Copie Profonde (Deep Copy)

Une copie profonde crée un nouvel objet et copie récursivement toutes les propriétés de l'objet original, y compris les objets imbriqués. Cela signifie que l'objet copié est entièrement indépendant de l'objet original, et toute modification apportée à l'un n'affectera pas l'autre.

La copie profonde est plus complexe à réaliser en JavaScript/TypeScript pur, surtout pour des objets complexes avec des fonctions, des dates, des expressions régulières, ou des références circulaires. Les méthodes courantes incluent :

- **`JSON.parse(JSON.stringify(object))`** : C'est une méthode simple et courante pour effectuer une copie profonde d'objets simples (sans fonctions, `undefined`, `Date`, `RegExp`, ou références circulaires). Elle fonctionne en convertissant l'objet en une chaîne JSON, puis en re-parcourant cette chaîne pour créer un nouvel objet.

  ```typescript const originalObject = { a: 1, b: { c: 2 }, d: new Date() }; const deepCopy = JSON.parse(JSON.stringify(originalObject));

  console.log(deepCopy); // { a: 1, b: { c: 2 }, d: '2025-07-31T...' } (Date est convertie en string) console.log(originalObject === deepCopy); // false console.log(originalObject.b === deepCopy.b); // false console.log(originalObject.d === deepCopy.d); // false

deepCopy.b.c = 20; console.log(originalObject); // { a: 1, b: { c: 2 }, d: Date object } console.log(deepCopy); // { a: 1, b: { c: 20 }, d: '2025-07-31T...' } ```

**Limitations de `JSON.parse(JSON.stringify())` :** * Ne peut pas copier les fonctions, les valeurs `undefined`, les `Symbol`. * Les objets `Date` sont convertis en chaînes de caractères. * Les `RegExp` sont converties en objets vides. * Ne gère pas les références circulaires (cela provoquerait une erreur).

- **Bibliothèques externes** : Pour des copies profondes robustes et complètes, il est souvent recommandé d'utiliser des bibliothèques tierces comme `lodash` avec sa fonction `_.cloneDeep()`. Ces bibliothèques gèrent les cas complexes (fonctions, dates, références circulaires, etc.) de manière plus fiable.

  `typescript // Exemple conceptuel avec lodash (nécessite l'installation de lodash) // import cloneDeep from 'lodash/cloneDeep'; // const originalObject = { a: 1, b: { c: 2 }, func: () => {} }; // const deepCopy = cloneDeep(originalObject); // console.log(deepCopy.func === originalObject.func); // false (la fonction est aussi copiée)`

## 6.3 Choisir la bonne méthode

Le choix entre une copie superficielle et une copie profonde dépend entièrement de vos besoins :

- Utilisez une **copie superficielle** lorsque vous savez que votre objet ne contient pas d'objets imbriqués qui nécessitent une indépendance totale, ou lorsque vous n'avez pas l'intention de modifier ces objets imbriqués.
- Utilisez une **copie profonde** lorsque vous avez besoin d'une indépendance totale entre l'objet original et sa copie, y compris pour tous les objets imbriqués. Soyez conscient des limitations de `JSON.parse(JSON.stringify())` et envisagez une bibliothèque externe pour des scénarios plus complexes.

Comprendre ces mécanismes de copie est essentiel pour la gestion de l'état dans les applications, en particulier dans les frameworks qui favorisent l'immuabilité (comme React avec Redux).

# 7. Nombre de paramètres variable

En TypeScript (et JavaScript), il est souvent nécessaire de créer des fonctions qui peuvent accepter un nombre indéfini d'arguments. Cette flexibilité est gérée par les **paramètres rest** (rest parameters), une fonctionnalité introduite en ECMAScript 2015 (ES6) et pleinement prise en charge par TypeScript. Les paramètres rest permettent de représenter un nombre indéfini d'arguments sous la forme d'un tableau, ce qui simplifie la manipulation de ces arguments au sein de la fonction.

## 7.1 Définition des paramètres rest

Un paramètre rest est désigné par trois points ( `...` ) suivis du nom du paramètre, qui sera un tableau. Il doit toujours être le dernier paramètre dans la liste des arguments d'une fonction.

```
function sum(a: number, b: number, ...restOfNumbers: number[]): number {
  let total = a + b;
  for (let num of restOfNumbers) {
    total += num;
  }
  return total;
}

console.log(sum(1, 2));   // 3
console.log(sum(1, 2, 3));   // 6
console.log(sum(1, 2, 3, 4, 5));   // 15
```

Dans cet exemple, `...restOfNumbers` collecte tous les arguments supplémentaires passés à la fonction `sum` après `a` et `b` dans un tableau nommé `restOfNumbers`. Ce tableau peut ensuite être itéré ou manipulé comme n'importe quel autre tableau.

7.2 Règles et considérations

- **Un seul paramètre rest** : Une fonction ne peut avoir qu'un seul paramètre rest, et il doit être le dernier dans la liste des paramètres. Cela garantit que tous les arguments restants sont correctement collectés.

  `typescript // function example(...args: string[], param2: number) { } // Erreur: Un paramètre rest doit être le dernier paramètre.`

- **Type du paramètre rest** : Le paramètre rest est toujours un tableau. Vous devez spécifier le type des éléments de ce tableau. Par exemple, `...args: string[]`

signifie que `args` sera un tableau de chaînes de caractères.

```typescript function logMessages(...messages: string[]) { messages.forEach(msg => console.log(msg)); }

logMessages("Hello", "World", "TypeScript"); // logMessages("Error", 404); // Erreur: L'argument de type 'number' n'est pas assignable au paramètre de type 'string'. ```

- **Utilisation avec des arguments optionnels ou par défaut** : Les paramètres rest peuvent être utilisés avec des paramètres optionnels ou des paramètres avec des valeurs par défaut, tant que le paramètre rest reste le dernier.

  ``typescript function greet(greeting: string = "Hello", ...names: string[]) { if (names.length === 0) { console.log(${greeting}, stranger!); } else { console.log(${greeting}, ${names.join(", ")}!`); } }

  greet(); // Hello, stranger! greet("Hi", "Alice", "Bob"); // Hi, Alice, Bob! ```

## 7.3 Différence avec l'opérateur Spread

Il est important de ne pas confondre les **paramètres rest** avec l'**opérateur spread** ( `...` ). Bien qu'ils utilisent la même syntaxe ( `...` ), leur fonction est différente :

- **Paramètres rest** : Utilisés dans la *définition* d'une fonction pour collecter un nombre indéfini d'arguments dans un tableau.
- **Opérateur spread** : Utilisé dans l'*appel* d'une fonction (ou lors de la création de tableaux/objets) pour étendre un itérable (comme un tableau) en éléments individuels.

  ```typescript // Paramètres rest (collecte) function myFunc(...args: number[]) { console.log(args); // [1, 2, 3] } myFunc(1, 2, 3);

  // Opérateur spread (extension) const numbers = [1, 2, 3]; myFunc(...numbers); // Équivalent à myFunc(1, 2, 3) ```

Les paramètres rest sont un outil essentiel pour créer des fonctions flexibles et polyvalentes en TypeScript, permettant de gérer élégamment les scénarios où le nombre d'arguments n'est pas fixe.

# 8. Nullish Coalescing

L'opérateur de coalescence nulle ( `??` ) est une fonctionnalité introduite en ECMAScript 2020 (ES11) et prise en charge par TypeScript. Il fournit un moyen concis de définir une valeur par défaut pour une variable qui pourrait être `null` ou `undefined`. Cet opérateur est particulièrement utile pour gérer les valeurs par défaut de manière plus précise que l'opérateur logique OR ( `||` ), qui peut avoir des comportements inattendus avec certaines valeurs "falsy" (fausses).

## 8.1 Fonctionnement de l'opérateur `??`

L'opérateur de coalescence nulle renvoie son opérande de droite si son opérande de gauche est `null` ou `undefined`. Dans tous les autres cas, il renvoie son opérande de gauche. Cela inclut les valeurs "falsy" comme `0`, `''` (chaîne vide), `false`, qui sont considérées comme valides par l'opérateur `??`.

```
 const value1 = null;
const value2 = undefined;
const value3 = 0;
const value4 = '';
const value5 = false;
const value6 = 'Hello';

const defaultValue = 'Default Value';

console.log(value1 ?? defaultValue); // 'Default Value'
console.log(value2 ?? defaultValue); // 'Default Value'
console.log(value3 ?? defaultValue); // 0 (0 n'est ni null ni undefined)
console.log(value4 ?? defaultValue); // '' (chaîne vide n'est ni null ni
undefined)
console.log(value5 ?? defaultValue); // false (false n'est ni null ni
undefined)
console.log(value6 ?? defaultValue); // 'Hello'
```

8.2 Différence avec l'opérateur logique OR (`||`)

Historiquement, l'opérateur logique OR (`||`) était souvent utilisé pour fournir des valeurs par défaut. Cependant, `||` renvoie son opérande de droite si son opérande de gauche est une valeur "falsy" (fausse), ce qui inclut `null`, `undefined`, `0`, `''`, et `false`.

```
const value1 = null;
const value2 = undefined;
const value3 = 0;
const value4 = '';
const value5 = false;
const value6 = 'Hello';

const defaultValue = 'Default Value';

console.log(value1 || defaultValue); // 'Default Value'
console.log(value2 || defaultValue); // 'Default Value'
console.log(value3 || defaultValue); // 'Default Value' (0 est falsy)
console.log(value4 || defaultValue); // 'Default Value' (chaîne vide est falsy)
console.log(value5 || defaultValue); // 'Default Value' (false est falsy)
console.log(value6 || defaultValue); // 'Hello'
```

Comme on peut le voir dans l'exemple ci-dessus, si vous souhaitez que `0`, `''`, ou `false` soient des valeurs valides et non remplacées par une valeur par défaut, l'opérateur `??` est le choix approprié. L'opérateur `||` est plus adapté lorsque vous voulez que toute valeur "falsy" soit traitée comme une absence de valeur.

8.3 Cas d'utilisation

L'opérateur de coalescence nulle est particulièrement utile dans les scénarios suivants :

- **Configuration d'options** : Lorsque vous définissez des options pour une fonction ou un objet, et que vous voulez permettre à `0`, `''`, ou `false` d'être des valeurs valides.

  ```typescript interface UserSettings { theme?: string; fontSize?: number; notificationsEnabled?: boolean; }

  function applySettings(settings: UserSettings) { const theme = settings.theme ?? 'dark'; const fontSize = settings.fontSize ?? 16; // Si fontSize est 0, il sera conservé const notificationsEnabled = settings.notificationsEnabled ?? true; // Si notificationsEnabled est false, il sera conservé

  console.log(`Theme: ${theme}, Font Size: ${fontSize}, Notifications: ${notificationsEnabled}`); }

  applySettings({}); // Theme: dark, Font Size: 16, Notifications: true applySettings({ fontSize: 0 }); // Theme: dark, Font Size: 0, Notifications: true applySettings({
  ```

notificationsEnabled: false }); // Theme: dark, Font Size: 16, Notifications: false ```

- **Accès sécurisé aux propriétés imbriquées** : Bien que l'opérateur de chaînage optionnel (`?.`) soit plus adapté pour cela, `??` peut être utilisé en combinaison pour fournir une valeur par défaut si la propriété finale est `null` ou `undefined`.

  ```typescript interface UserProfile { name: string; address?: { street?: string; city?: string; }; }

  const user: UserProfile = { name: 'Alice' };

  const city = user.address?.city ?? 'Unknown City'; console.log(city); // 'Unknown City'

  const user2: UserProfile = { name: 'Bob', address: { city: 'Paris' } }; const city2 = user2.address?.city ?? 'Unknown City'; console.log(city2); // 'Paris' ```

En résumé, l'opérateur de coalescence nulle ( `??` ) est un ajout précieux à TypeScript et JavaScript, offrant un contrôle plus fin sur la gestion des valeurs par défaut en distinguant `null` et `undefined` des autres valeurs "falsy".

# 9. Déstructuration

La déstructuration est une fonctionnalité puissante introduite en ECMAScript 2015 (ES6) et pleinement prise en charge par TypeScript. Elle permet d'extraire des valeurs de tableaux ou des propriétés d'objets dans des variables distinctes de manière concise et lisible. Cette syntaxe simplifie grandement le code, en particulier lors de la manipulation de structures de données complexes ou lors de l'extraction de données à partir de réponses d'API.

## 9.1 Avec les tableaux

La déstructuration de tableaux permet d'extraire des éléments d'un tableau et de les assigner à des variables individuelles en fonction de leur position. La syntaxe utilise des crochets `[]` sur le côté gauche de l'affectation.

- **Extraction basique** : Vous pouvez extraire n'importe quel élément du tableau en spécifiant des variables pour les positions correspondantes.
  ```

```typescript const numbers = [10, 20, 30, 40, 50];

const [first, second] = numbers; console.log(first); // 10 console.log(second); // 20

const [, , third, fourth] = numbers; // Ignorer les deux premiers éléments console.log(third); // 30 console.log(fourth); // 40 ```

- **Valeurs par défaut** : Vous pouvez assigner des valeurs par défaut aux variables si l'élément correspondant n'existe pas dans le tableau ou est `undefined`.

  ```typescript const colors = ["red", "green"]; const [c1, c2, c3 = "blue"] = colors; console.log(c1); // "red" console.log(c2); // "green" console.log(c3); // "blue"

  const [c4, c5, c6 = "yellow"] = ["orange"]; console.log(c4); // "orange" console.log(c5); // undefined console.log(c6); // "yellow" ```

- **Paramètres rest dans la déstructuration** : Similaire aux paramètres rest dans les fonctions, vous pouvez utiliser l'opérateur spread (`...`) pour collecter les éléments restants d'un tableau dans un nouveau tableau.

  ```typescript const [head, ...tail] = [1, 2, 3, 4, 5]; console.log(head); // 1 console.log(tail); // [2, 3, 4, 5]

  const [a, b, ...rest] = ["apple", "banana", "cherry", "date"]; console.log(a); // "apple" console.log(b); // "banana" console.log(rest); // ["cherry", "date"] ```

- **Échange de variables** : La déstructuration de tableaux est une manière élégante d'échanger les valeurs de deux variables sans avoir besoin d'une variable temporaire.

  `typescript let x = 1; let y = 2; [x, y] = [y, x]; console.log(x); // 2 console.log(y); // 1`

## 9.2 Avec les objets

La déstructuration d'objets permet d'extraire des propriétés d'un objet et de les assigner à des variables individuelles en fonction de leurs noms de propriétés. La syntaxe utilise des accolades `{}` sur le côté gauche de l'affectation.

- **Extraction basique** : Vous spécifiez les noms des propriétés que vous souhaitez extraire. L'ordre n'a pas d'importance.
  ```

```typescript const person = { name: "Alice", age: 30, city: "New York" };

const { name, age } = person; console.log(name); // "Alice" console.log(age); // 30
```

- **Renommage de variables** : Si vous souhaitez que la variable extraite ait un nom différent de la propriété de l'objet, vous pouvez utiliser la syntaxe `propertyName: newVariableName`.

  `typescript const user = { id: 1, username: "john_doe" }; const { id: userId, username: userName } = user; console.log(userId); // 1 console.log(userName); // "john_doe"`

- **Valeurs par défaut** : Vous pouvez assigner des valeurs par défaut aux variables si la propriété correspondante n'existe pas dans l'objet ou est `undefined`.

  ```typescript const product = { productName: "Laptop", price: 1200 }; const { productName, price, quantity = 1 } = product; console.log(productName); // "Laptop" console.log(price); // 1200 console.log(quantity); // 1

  const { description = "No description provided" } = product; console.log(description); // "No description provided" ```

- **Paramètres rest dans la déstructuration d'objets** : Vous pouvez collecter les propriétés restantes d'un objet dans un nouvel objet en utilisant l'opérateur spread (`...`) avec un nom de variable.

  `typescript const settings = { theme: "dark", notifications: true, language: "en", version: "1.0" }; const { theme, notifications, ...otherSettings } = settings; console.log(theme); // "dark" console.log(notifications); // true console.log(otherSettings); // { language: "en", version: "1.0" }`

- **Déstructuration imbriquée** : Vous pouvez déstructurer des objets imbriqués pour extraire des propriétés de niveaux plus profonds.

  ```typescript const company = { name: "TechCorp", address: { street: "123 Main St", city: "San Francisco" }, employees: [ { id: 1, name: "Alice" }, { id: 2, name: "Bob" } ] };

const { name: companyName, address: { city: companyCity }, employees: [emp1] } = company; console.log(companyName); // "TechCorp" console.log(companyCity); // "San Francisco" console.log(emp1); // { id: 1, name: "Alice" } ```

La déstructuration est une fonctionnalité très utilisée en TypeScript et JavaScript moderne, en particulier dans les frameworks comme React, pour rendre le code plus propre, plus lisible et plus facile à maintenir.

## 10. Les décorateurs

Les décorateurs sont une fonctionnalité expérimentale de TypeScript qui permet d'ajouter des annotations et une méta-programmation à des classes, des méthodes, des accesseurs, des propriétés ou des paramètres. Ils sont une forme de métaprogrammation qui permet de modifier le comportement ou la structure du code au moment de la conception ou de l'exécution. Les décorateurs sont préfixés par le symbole `@` et sont placés juste avant la déclaration à laquelle ils s'appliquent.

Pour utiliser les décorateurs, vous devez activer l'option `experimentalDecorators` dans votre fichier `tsconfig.json` :

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true // Souvent utilisé avec
experimentalDecorators
  }
}
```

10.1 Ajout de décorateurs de classe

Un décorateur de classe est une fonction qui est appliquée à la déclaration d'une classe. Le décorateur de classe reçoit le constructeur de la classe comme seul argument. Il peut être utilisé pour observer, modifier ou remplacer une définition de classe.

Voici un exemple simple de décorateur de classe qui ajoute une propriété à la classe :

```
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}

@sealed
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}

// Tenter d'ajouter une nouvelle propriété à la classe Greeter après l'avoir scellée
// (new Greeter("world")).newProperty = "test"; // Erreur en mode strict: Cannot add property newProperty, object is not extensible
```

Dans cet exemple, le décorateur `@sealed` rend la classe `Greeter` et son prototype non extensibles, empêchant l'ajout de nouvelles propriétés. Le décorateur `sealed` est une fonction qui prend le constructeur de la classe comme argument. `Object.seal()` est une méthode JavaScript qui empêche l'ajout ou la suppression de propriétés d'un objet, et empêche la modification des attributs de propriété existants.

Un autre exemple courant est l'ajout de métadonnées ou de fonctionnalités à une classe. Par exemple, un décorateur de journalisation :

```
function logClass(constructor: Function) {
  console.log(`Class ${constructor.name} was created.`);
}

@logClass
class MyService {
  constructor() {
    console.log("MyService instance created.");
  }

  doSomething() {
    console.log("Doing something...");
  }
}

const service = new MyService();
service.doSomething();
// Output:
// Class MyService was created.
// MyService instance created.
// Doing something...
```

Les décorateurs de classe sont évalués au moment de la déclaration de la classe, avant même que les instances de la classe ne soient créées. Ils sont très utiles pour des tâches telles que l'enregistrement de classes, l'ajout de fonctionnalités transversales (comme l'injection de dépendances ou l'authentification) ou la modification du comportement des constructeurs.

10.2 Décorateurs de méthode

Un décorateur de méthode est appliqué à la déclaration d'une méthode. Il peut être utilisé pour observer, modifier ou remplacer une définition de méthode. Le décorateur de méthode reçoit trois arguments :

1. `target` : Soit la fonction constructeur de la classe pour un membre statique, soit le prototype de la classe pour un membre d'instance.
2. `propertyKey` : Le nom de la méthode.
3. `descriptor` : Le descripteur de propriété de la méthode.

Voici un exemple de décorateur de méthode qui logue l'exécution d'une méthode :

```
function logMethod(target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
  const originalMethod = descriptor.value;

  descriptor.value = function (...args: any[]) {
    console.log(`Calling method $`{propertyKey} with arguments:
`${JSON.stringify(args)}`);
    const result = originalMethod.apply(this, args);
    console.log(`Method $`{propertyKey} returned: `${JSON.stringify(result)}`);
    return result;
  };

  return descriptor;
}

class Calculator {
  @logMethod
  add(a: number, b: number): number {
    return a + b;
  }

  @logMethod
  subtract(a: number, b: number): number {
    return a - b;
  }
}

const calc = new Calculator();
calc.add(5, 3);    // Output: Calling method add with arguments: [5,3] ...
Method add returned: 8
calc.subtract(10, 4); // Output: Calling method subtract with arguments: [10,4]
... Method subtract returned: 6
```

Ce décorateur `logMethod` enveloppe la méthode originale et ajoute des logs avant et après son exécution. C'est un cas d'utilisation courant pour l'aspect-oriented programming (AOP), comme la journalisation, la gestion des erreurs, la mise en cache, etc.

10.3 Décorateurs d'accesseur (Getter/Setter)

Les décorateurs d'accesseur sont similaires aux décorateurs de méthode, mais ils sont appliqués aux accesseurs (getters et setters) d'une propriété. Ils reçoivent les mêmes trois arguments que les décorateurs de méthode.

```
function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
    descriptor.enumerable = value;
  };
}

class Person {
  constructor(public name: string) {}

  @enumerable(false)
  get greeting() {
    return `Hello, my name is ${this.name}`;
  }
}

const person = new Person("Alice");
for (let key in person) {
  console.log(key); // Affiche seulement 'name', car 'greeting' est non
énumérable
}
```

Le décorateur `@enumerable(false)` rend l'accesseur `greeting` non énumérable, ce qui signifie qu'il n'apparaîtra pas lors de l'itération des propriétés de l'objet avec `for...in`.

10.4 Décorateurs de propriété

Un décorateur de propriété est appliqué à la déclaration d'une propriété de classe. Il reçoit deux arguments :

1. `target` : Soit la fonction constructeur de la classe pour un membre statique, soit le prototype de la classe pour un membre d'instance.
2. `propertyKey` : Le nom de la propriété.

Les décorateurs de propriété ne reçoivent pas de descripteur de propriété, car une propriété n'est pas encore initialisée au moment où le décorateur est appliqué. Ils sont principalement utilisés pour enregistrer des métadonnées sur la propriété.

```
function format(formatString: string) {
  return function (target: any, propertyKey: string) {
    let value: string;

    const getter = function () {
      return value;
    };

    const setter = function (newVal: string) {
      value = formatString.replace("%s", newVal);
    };

    Object.defineProperty(target, propertyKey, {
      get: getter,
      set: setter,
      enumerable: true,
      configurable: true,
    });
  };
}

class User {
  @format("Mr./Ms. %s")
  name: string;

  constructor(name: string) {
    this.name = name;
  }
}

const user = new User("Alice");
console.log(user.name); // Mr./Ms. Alice

user.name = "Bob";
console.log(user.name); // Mr./Ms. Bob
```

Dans cet exemple, le décorateur `@format` modifie le comportement de la propriété `name` en lui appliquant un formatage spécifique lors de l'affectation. Il remplace le getter et le setter par défaut de la propriété pour intercepter les affectations et les lectures.

10.5 Décorateurs de paramètre

Un décorateur de paramètre est appliqué aux paramètres d'une méthode de classe ou d'un constructeur. Il reçoit trois arguments :

1. `target` : Soit la fonction constructeur de la classe pour un membre statique, soit le prototype de la classe pour un membre d'instance.
2. `propertyKey` : Le nom de la méthode.

3. `parameterIndex` : L'index ordinal du paramètre dans la liste des paramètres de la fonction.

Les décorateurs de paramètre sont principalement utilisés pour enregistrer des métadonnées sur un paramètre. Ils ne peuvent pas modifier le comportement du paramètre lui-même.

```
function required(target: Object, propertyKey: string | symbol,
parameterIndex: number) {
  console.log(`Parameter at index ${parameterIndex} of method
`${String(propertyKey)} is required.`);
}

class UserService {
  saveUser(@required name: string, age: number) {
    console.log(`Saving user: ${name}, ${age}`);
  }
}

const userService = new UserService();
userService.saveUser("Alice", 30);
// Output:
// Parameter at index 0 of method saveUser is required.
// Saving user: Alice, 30
```

Ce décorateur `@required` pourrait être utilisé dans un cadre de validation pour s'assurer qu'un paramètre n'est pas `null` ou `undefined` .

10.6 Ordre d'évaluation des décorateurs

L'ordre dans lequel les décorateurs sont évalués est important :

1. **Décorateurs de paramètre**, puis **décorateurs de méthode**, **accesseur** ou **propriété** sont appliqués pour chaque membre d'instance.
2. **Décorateurs de paramètre**, puis **décorateurs de méthode**, **accesseur** ou **propriété** sont appliqués pour chaque membre statique.
3. **Décorateurs de classe** sont appliqués.

Au sein de chaque groupe, les décorateurs sont évalués de haut en bas, et les fonctions renvoyées par les décorateurs sont appelées dans l'ordre inverse.

Les décorateurs sont un outil puissant pour étendre les fonctionnalités des classes et de leurs membres de manière déclarative, ce qui est particulièrement utile dans les frameworks comme Angular qui les utilisent intensivement pour la gestion des composants, des services et de l'injection de dépendances.

11. Map

L'objet `Map` est une collection de paires clé-valeur où les clés peuvent être de n'importe quel type (objets, fonctions, primitives, etc.), contrairement aux objets JavaScript traditionnels où les clés sont limitées aux chaînes de caractères ou aux symboles. `Map` maintient l'ordre d'insertion des éléments et offre des méthodes dédiées pour la manipulation des données, ce qui en fait un outil puissant pour la gestion de collections de données.

11.1 Création et manipulation d'une Map

Pour créer une nouvelle `Map`, vous utilisez le constructeur `new Map()`.

```
const myMap = new Map<string, number>(); // Map qui associe des chaînes à des nombres
```

Voici les méthodes principales pour interagir avec un objet `Map` :

- **`set(key, value)`** : Ajoute ou met à jour une paire clé-valeur dans la Map. Retourne la Map elle-même, ce qui permet le chaînage.

  `typescript myMap.set("apple", 1); myMap.set("banana", 2); myMap.set("orange", 3); console.log(myMap); // Map(3) { 'apple' => 1, 'banana' => 2, 'orange' => 3 }`

- **`get(key)`** : Retourne la valeur associée à la clé spécifiée, ou `undefined` si la clé n'existe pas dans la Map.

  `typescript console.log(myMap.get("apple")); // 1 console.log(myMap.get("grape")); // undefined`

- **`has(key)`** : Retourne un booléen indiquant si une clé existe dans la Map.

  `typescript console.log(myMap.has("banana")); // true console.log(myMap.has("grape")); // false`

- **`delete(key)`** : Supprime une paire clé-valeur de la Map par sa clé. Retourne `true` si l'élément a été supprimé avec succès, `false` sinon.

`typescript myMap.delete("banana"); console.log(myMap); // Map(2) { 'apple' => 1, 'orange' => 3 } console.log(myMap.has("banana")); // false`

- **`clear()`** : Supprime toutes les paires clé-valeur de la Map.

  `typescript myMap.clear(); console.log(myMap); // Map(0) {}`

- **`size`** : Une propriété qui retourne le nombre d'éléments (paires clé-valeur) dans la Map.

  `typescript myMap.set("apple", 1).set("banana", 2); console.log(myMap.size); // 2`

11.2 Itération sur une Map

L'objet `Map` est itérable, ce qui signifie que vous pouvez utiliser des boucles `for...of` pour parcourir ses éléments. Il fournit également des méthodes spécifiques pour itérer sur les clés, les valeurs ou les paires clé-valeur.

- **`for...of` (par défaut, itère sur les paires [clé, valeur])** :

  ```typescript const fruits = new Map([ ["apple", 10], ["banana", 20], ["orange", 30] ]);

  for (const [key, value] of fruits) { console.log( $ : ${value} ); } // Output: // apple: 10 // banana: 20 // orange: 30 ```

- **`keys()`** : Retourne un itérateur des clés de la Map.

  `typescript for (const key of fruits.keys()) { console.log(key); } // Output: // apple // banana // orange`

- **`values()`** : Retourne un itérateur des valeurs de la Map.

  `typescript for (const value of fruits.values()) { console.log(value); } // Output: // 10 // 20 // 30`

- **`entries()`** : Retourne un itérateur des paires `[clé, valeur]` de la Map (comportement par défaut de `for...of`).
  ```

`typescript for (const entry of fruits.entries()) { console.log(entry); // [ 'apple', 10 ], [ 'banana', 20 ], [ 'orange', 30 ] }`

- **`forEach()`** : Exécute une fonction de rappel pour chaque paire clé-valeur dans la Map.

  `typescript fruits.forEach((value, key) => { console.log(`$`{key} => `${value}`); }); // Output: // apple => 10 // banana => 20 // orange => 30`

11.3 Avantages de `Map` par rapport aux objets `Object`

`Map` offre plusieurs avantages par rapport à l'utilisation d'objets JavaScript (`{}`) comme collections de paires clé-valeur :

1. **Clés de n'importe quel type** : Les objets `Object` ne peuvent avoir que des chaînes de caractères ou des symboles comme clés. `Map` peut utiliser n'importe quel type de valeur (objets, fonctions, `null`, `NaN`, etc.) comme clé, ce qui est très puissant.

   ```typescript const objKey = { id: 1 }; const funcKey = () => {}; const specialMap = new Map(); specialMap.set(objKey, "Value for object"); specialMap.set(funcKey, "Value for function"); specialMap.set(NaN, "Value for NaN");

   console.log(specialMap.get(objKey)); // "Value for object" console.log(specialMap.get(funcKey)); // "Value for function" console.log(specialMap.get(NaN)); // "Value for NaN" (NaN est traité comme une clé unique dans Map) ```

2. **Ordre d'insertion** : `Map` maintient l'ordre dans lequel les éléments ont été insérés. Lors de l'itération, les éléments sont renvoyés dans cet ordre. Les objets `Object` ne garantissent pas l'ordre des propriétés (bien que les moteurs JavaScript modernes aient tendance à maintenir l'ordre pour les clés numériques et les clés de chaîne non numériques insérées).

3. **Performance pour les grandes collections** : Pour les scénarios impliquant de fréquentes additions et suppressions de paires clé-valeur, `Map` peut offrir de meilleures performances que les objets `Object`.
   ```

4. **Taille facile à obtenir** : La propriété `size` de `Map` fournit directement le nombre d'éléments. Pour les objets `Object`, vous devez itérer sur les propriétés ou utiliser `Object.keys().length`.

5. **Pas de conflits de clés par défaut** : Les objets `Object` ont des propriétés par défaut sur leur prototype (par exemple, `toString`, `hasOwnProperty`) qui peuvent potentiellement entrer en conflit avec vos propres clés si vous ne faites pas attention. `Map` n'a pas de telles propriétés par défaut.

En résumé, `Map` est une structure de données moderne et flexible en TypeScript/JavaScript, idéale pour les collections de paires clé-valeur où la flexibilité des types de clés, l'ordre d'insertion et la performance sont importants.

12. Ajout de décorateurs de propriété

Comme mentionné dans la section 10.4 sur les Décorateurs de propriété, les décorateurs de propriété sont une fonctionnalité de TypeScript qui permet d'attacher des métadonnées ou de modifier le comportement d'une propriété de classe. Ils sont appliqués directement au-dessus de la déclaration d'une propriété.

Pour rappel, un décorateur de propriété reçoit deux arguments :

1. `target` : Le prototype de la classe pour un membre d'instance, ou la fonction constructeur pour un membre statique.
2. `propertyKey` : Le nom de la propriété.

Ces décorateurs sont principalement utilisés pour enregistrer des métadonnées sur la propriété, qui peuvent ensuite être lues au moment de l'exécution pour implémenter des logiques spécifiques, comme la validation, la sérialisation/désérialisation, ou l'injection de dépendances. Ils ne peuvent pas modifier directement la valeur de la propriété au moment de la décoration, mais ils peuvent modifier le descripteur de propriété pour altérer son comportement (get/set).

Pour un exemple détaillé et une explication de l'implémentation d'un décorateur de propriété, veuillez vous référer à la section 10.4 de ce document.

13. Expliquer les clés du TSCONFIG

Le fichier `tsconfig.json` est le cœur de tout projet TypeScript. Il spécifie les options du compilateur et les fichiers racines qui composent le projet. Comprendre les différentes clés de configuration est essentiel pour configurer correctement votre environnement de développement et de compilation TypeScript.

13.1 Include

La propriété `include` dans `tsconfig.json` est utilisée pour spécifier un tableau de motifs de fichiers (glob patterns) qui indiquent au compilateur TypeScript quels fichiers doivent être inclus dans le projet. C'est une manière explicite de dire à TypeScript où trouver votre code source. Si cette propriété n'est pas spécifiée, le compilateur inclura par défaut tous les fichiers `.ts`, `.tsx`, et `.d.ts` dans le répertoire courant et ses sous-répertoires, à l'exception de ceux spécifiés dans `exclude` ou `node_modules`.

Syntaxe et utilisation :

La propriété `include` accepte un tableau de chaînes de caractères, où chaque chaîne est un motif de fichier. Les motifs peuvent inclure des caractères génériques (wildcards) :

- `*` : Correspond à zéro ou plusieurs caractères (sauf `/`).
- `**` : Correspond à zéro ou plusieurs répertoires.
- `?` : Correspond à un seul caractère.

Exemples :

1. **Inclure tous les fichiers TypeScript dans le répertoire `src` et ses sous-répertoires :**

   `json { "include": [ "src/**/*.ts" ] }`

2. **Inclure des fichiers TypeScript et des fichiers de définition (`.d.ts`) :**

   `json { "include": [ "src/**/*.ts", "typings/**/*.d.ts" ] }`

3. **Inclure des fichiers spécifiques :**

```
json { "include": [ "src/main.ts", "src/utils.ts" ] }
```

Interaction avec `exclude` et `files` :

- **`include` vs `exclude`** : Les fichiers qui correspondent aux motifs dans `exclude` sont toujours ignorés, même s'ils correspondent à un motif dans `include`. `exclude` a une priorité plus élevée.
- **`include` vs `files`** : La propriété `files` est utilisée pour spécifier une liste explicite de fichiers individuels à inclure. Si `files` est spécifié, `include` et `exclude` sont ignorés. Il est généralement recommandé d'utiliser `include` pour la plupart des projets, car il est plus flexible et facile à maintenir pour les grandes bases de code.

L'utilisation judicieuse de `include` permet de s'assurer que seuls les fichiers pertinents sont compilés, ce qui peut améliorer les performances de compilation et éviter les erreurs dues à l'inclusion de fichiers non désirés.

13.2 File

La propriété `files` dans `tsconfig.json` est utilisée pour spécifier une liste explicite de chemins de fichiers individuels qui doivent être inclus dans le projet TypeScript. Contrairement à `include` qui utilise des motifs génériques pour inclure des groupes de fichiers, `files` exige que chaque fichier soit listé explicitement par son chemin relatif ou absolu.

Syntaxe et utilisation :

La propriété `files` accepte un tableau de chaînes de caractères, où chaque chaîne est le chemin d'accès à un fichier spécifique.

Exemples :

1. **Inclure des fichiers spécifiques :**

```
json { "files": [ "src/main.ts", "src/utils/helpers.ts", "typings/global.d.ts" ] }
```

Interaction avec `include` et `exclude` :

- **Priorité** : Si la propriété `files` est présente dans `tsconfig.json`, les propriétés `include` et `exclude` sont complètement ignorées par le compilateur TypeScript. Cela signifie que seuls les fichiers listés dans `files` seront compilés, et aucun autre fichier ne sera inclus ou exclu via les motifs.
- **Utilisation recommandée** : La propriété `files` est généralement utilisée pour les projets plus petits ou pour les bibliothèques qui ont un nombre fixe et bien défini de fichiers d'entrée. Pour les projets de plus grande envergure ou ceux dont la structure de fichiers est susceptible de changer fréquemment, l'utilisation de `include` est préférée car elle est plus flexible et moins sujette aux erreurs (pas besoin de mettre à jour manuellement la liste des fichiers à chaque ajout ou suppression).

En résumé, `files` offre un contrôle très précis sur les fichiers à compiler, mais au prix d'une gestion manuelle plus importante. Pour la plupart des projets modernes, `include` est l'option privilégiée.

13.3 RootDir

La propriété `rootDir` dans la section `compilerOptions` de `tsconfig.json` est utilisée pour spécifier le répertoire racine de tous les fichiers d'entrée TypeScript. Lorsque `outDir` est spécifié, `rootDir` est utilisé pour déterminer la structure de répertoire de la sortie. Le compilateur TypeScript utilise `rootDir` pour s'assurer que la structure de répertoire des fichiers de sortie dans `outDir` reflète la structure de répertoire des fichiers d'entrée par rapport à `rootDir`.

Fonctionnement :

Si `rootDir` n'est pas spécifié, le compilateur détermine le `rootDir` commun de tous les fichiers d'entrée. Cela peut parfois entraîner des structures de répertoire de sortie inattendues si vos fichiers sources sont dispersés dans plusieurs répertoires sans une racine commune claire.

Exemples :

Supposons la structure de projet suivante :

```
project/
├── src/
│   ├── components/
│   │   └── Button.ts
│   └── pages/
│       └── Home.ts
├── tests/
│   └── Button.test.ts
└── tsconfig.json
```

Si votre `tsconfig.json` est configuré comme suit :

```
{
  "compilerOptions": {
    "outDir": "./dist",
    "rootDir": "./src"
  },
  "include": [
    "src/**/*.ts"
  ]
}
```

Après la compilation, la structure de sortie dans `dist` reflétera la structure de `src` :

```
dist/
├── components/
│   └── Button.js
└── pages/
    └── Home.js
```

Si `rootDir` n'avait pas été spécifié, et que le compilateur avait déterminé `project/` comme `rootDir` commun (parce que `tests/` contient aussi des fichiers `.ts`), alors la structure de `dist` pourrait inclure un répertoire `src/` inutile.

Avantages de l'utilisation de `rootDir` :

- **Contrôle de la structure de sortie** : Permet de définir explicitement la base pour la structure de répertoire des fichiers transpilés, garantissant une sortie prévisible.
- **Prévention des répertoires inutiles** : Aide à éviter la création de répertoires vides ou superflus dans le dossier de sortie lorsque vos fichiers sources ne sont pas tous sous une seule racine logique.
- **Clarté du projet** : Rend l'intention de la structure du projet plus claire pour les autres développeurs.

Il est recommandé de définir `rootDir` explicitement, surtout dans les projets de taille moyenne à grande, pour maintenir une structure de sortie propre et organisée.

13.4 RemoveComments

La propriété `removeComments` dans la section `compilerOptions` de `tsconfig.json` est une option booléenne qui, lorsqu'elle est définie sur `true`, indique au compilateur TypeScript de supprimer tous les commentaires des fichiers JavaScript transpilés. Par défaut, cette option est `false`, ce qui signifie que les commentaires sont conservés dans la sortie JavaScript.

Utilisation :

```
{
  "compilerOptions": {
    "removeComments": true
  }
}
```

Exemple :

Si vous avez un fichier TypeScript `example.ts` :

```
// Ceci est un commentaire en TypeScript
function greet(name: string) {
  /*
   * Ceci est un commentaire multi-lignes
   * qui sera supprimé.
   */
  return `Hello, ${name}!`;
}

const message = greet("World");
console.log(message);
```

Avec `"removeComments": true`, le fichier JavaScript transpilé `example.js` ressemblera à ceci :

```
function greet(name) {
  return `Hello, ${name}!`;
}
const message = greet("World");
console.log(message);
```

Avantages et inconvénients :

- **Avantages :**
    - **Taille de fichier réduite** : La suppression des commentaires peut légèrement réduire la taille des fichiers JavaScript générés, ce qui peut être bénéfique pour les applications web où chaque octet compte pour le temps de chargement.
    - **Obfuscation légère** : Bien que ce ne soit pas une mesure de sécurité robuste, la suppression des commentaires peut rendre le code un peu moins lisible pour ceux qui tentent de le rétro-ingénierie.
- **Inconvénients :**
    - **Débogage plus difficile** : Les commentaires sont souvent utiles pour comprendre le code, surtout lors du débogage en production ou de l'analyse de traces d'erreurs. Sans commentaires, le code généré peut être plus difficile à interpréter.
    - **Perte d'informations** : Si vous utilisez des outils qui s'appuient sur des commentaires spécifiques (par exemple, des annotations JSDoc pour la génération de documentation), la suppression des commentaires peut entraîner une perte de fonctionnalité.

En général, pour les environnements de production où la taille des fichiers est critique, `removeComments: true` peut être envisagé. Cependant, pour le développement et le débogage, il est souvent préférable de laisser cette option à `false` ou de s'appuyer sur des outils de minification qui gèrent la suppression des commentaires de manière plus sophistiquée (par exemple, en ne supprimant que les commentaires non essentiels).

13.5 NoEmitOnError

La propriété `noEmitOnError` dans la section `compilerOptions` de `tsconfig.json` est une option booléenne qui, lorsqu'elle est définie sur `true`, empêche le compilateur TypeScript de générer des fichiers JavaScript de sortie si des erreurs de compilation sont détectées. Par défaut, cette option est `false`, ce qui signifie que le compilateur générera des fichiers JavaScript même s'il y a des erreurs, bien que ces fichiers puissent être incomplets ou incorrects.

Utilisation :

```
{
  "compilerOptions": {
    "noEmitOnError": true
  }
}
```

Exemple :

Considérons un fichier TypeScript `error.ts` avec une erreur de type :

```
function greet(name: string) {
  return `Hello, ${name}!`;
}

// Erreur: L'argument de type 'number' n'est pas assignable au paramètre de type 'string'.
const message = greet(123);
console.log(message);
```

- **Avec `"noEmitOnError": false` (par défaut) :** Le compilateur affichera l'erreur de type, mais générera quand même un fichier `error.js` (qui pourrait contenir `greet(123)` et potentiellement échouer à l'exécution).
- **Avec `"noEmitOnError": true` :** Le compilateur affichera l'erreur de type et *ne générera pas* de fichier `error.js`. Le processus de compilation s'arrêtera, indiquant qu'il y a des problèmes qui doivent être résolus avant que le code JavaScript ne puisse être produit.

Avantages et inconvénients :

- **Avantages :**
  - **Prévention des builds cassés** : C'est une mesure de sécurité importante pour s'assurer que vous ne déployez pas de code JavaScript qui contient des erreurs de type TypeScript. Cela garantit que votre build de production est toujours exempt d'erreurs de compilation.
  - **Intégration continue (CI)** : Très utile dans les pipelines CI/CD, où vous voulez que le build échoue immédiatement si des erreurs TypeScript sont présentes, plutôt que de générer un code potentiellement défectueux.
  - **Développement plus sûr** : Encourage les développeurs à corriger les erreurs de type dès qu'elles apparaissent, car le code ne sera pas transpilé tant que toutes les erreurs ne sont pas résolues.

- **Inconvénients :**
    - **Ralentissement du cycle de développement (parfois)** : Dans certains scénarios de développement rapide, les développeurs peuvent vouloir voir le résultat de leur code même avec des erreurs mineures, et `noEmitOnError: true` les en empêcherait.

Recommandation :

Il est fortement recommandé de définir `"noEmitOnError": true` dans la plupart des projets TypeScript, en particulier pour les builds de production et les environnements d'intégration continue. Cela garantit la qualité et la fiabilité du code JavaScript généré. Pour le développement local, vous pouvez temporairement le désactiver si vous le souhaitez, mais il est préférable de s'habituer à corriger les erreurs de type dès qu'elles surviennent.