

TYPESCRIPT

by :

mohamed@goumih.om

Histoire de TypeScript

En 2010, Anders Hejlsberg (le créateur de TypeScript) a commencé à travailler sur TypeScript chez Microsoft et en 2012, la première version de TypeScript a été rendue publique (TypeScript 0.8). Bien que la sortie de typescript ait été saluée par de nombreuses personnes à travers le monde, en raison du manque de soutien des principaux ides, elle n'a pas été largement adoptée par la communauté JavaScript.

La première version de typescript (typescript 0.8) a été rendue publique en octobre 2012.

La dernière version de typescript (typescript 5.3 beta) a été rendue publique en octobre 2023.
(version stable 5.2.2 septembre 2023).

Pourquoi typescript ?

3

Il y a deux objectifs principaux pour TypeScript :

- * Fournir un **système de type facultatif** pour JavaScript.
- * Fournir les fonctionnalités planifiées des futures éditions de JavaScript aux moteurs JavaScript actuels

TypeScript est open source.

TypeScript simplifie le code JavaScript, ce qui le rend plus facile à lire et à déboguer.

TypeScript est un sur-ensemble de ES3, ES5 et ES6 ..

TypeScript fera gagner du temps aux développeurs.

Le code TypeScript peut être compilé selon les normes ES5 et ES6 pour prendre en charge le navigateur le plus récent.

TypeScript peut nous aider à éviter les bogues pénibles que les développeurs rencontrent couramment lors de l'écriture de JavaScript en vérifiant le code par type.

TypeScript n'est rien d'autre que du JavaScript avec quelques fonctionnalités supplémentaires.

Pourquoi typescript ?

4

Les types ont prouvé leur capacité à améliorer la qualité et la compréhensibilité du code. De grandes équipes (Google, Microsoft, Facebook) sont continuellement parvenues à cette conclusion. Plus précisément :

- * Les types augmentent votre agilité lors de la refactorisation.
**Il est préférable pour le compilateur de détecter les erreurs que de faire échouer les choses au moment de l'exécution*.*
- * Les types sont l'une des meilleures formes de documentation que vous puissiez avoir. **La signature de la fonction est un théorème et le corps de la fonction en est la preuve**.

TypeScript fournit une sécurité de type pendant la compilation pour votre code JavaScript. Ce n'est pas une surprise étant donné son nom. Le truc sympa est que les types sont complètement facultatifs. Votre fichier de code JavaScript `*.js` peut être renommé en fichier `*.ts` et TypeScript vous rendra toujours l'équivalent `*.js` valide du fichier JavaScript d'origine. TypeScript est **intentionnellement** et strictement un sur-ensemble de JavaScript avec vérification de type facultative.

Fonctionnalités Typescript

5

Multiplateforme : typescript s'exécute sur n'importe quelle plate-forme sur laquelle JavaScript s'exécute. Le compilateur typescript peut être installé sur n'importe quel système d'exploitation tel que Windows, Mac OS et Linux.

Langage orienté objet : typescript fournit des fonctionnalités puissantes telles que des classes, des interfaces et des modules. Vous pouvez écrire du code orienté objet pur pour le développement côté client ainsi que côté serveur.

Vérification de type statique : typescript utilise le typage statique. Cela se fait à l'aide d'annotations de type. Il aide à la vérification de type au moment de la compilation. Ainsi, vous pouvez trouver des erreurs lors de la saisie du code sans exécuter votre script à chaque fois. De plus, à l'aide du mécanisme d'inférence de type, si une variable est déclarée sans type, elle sera déduite en fonction de sa valeur.

Typage statique facultatif : typescript permet également le typage statique facultatif si vous préférez utiliser le typage dynamique de JavaScript.

Manipulation du DOM : tout comme JavaScript, le typescript peut être utilisé pour manipuler le DOM afin d'ajouter ou de supprimer des éléments.

Caractéristiques d'ES 6 : typescript inclut la plupart des fonctionnalités d'ECMAScript 2015 prévu (ES 6, 7) telles que les fonctions de classe, d'interface, de flèche, etc.

Frameworks qui utilisent TypeScript

TypeScript est devenu très populaire dans le développement web moderne, principalement en raison de sa capacité à fournir un typage statique et une analyse de code à un écosystème JavaScript dynamiquement typé.

Voici quelques-uns des frameworks qui sont utilisés avec TypeScript ou qui ont un support officiel pour TypeScript :

1. Angular:

2. React:

3. Vue.js:

4. Next.js:

5. NestJS:

6. Express JS

7. Ionic:

10. Svelte (avec Svelte for TypeScript):

Ces frameworks et bibliothèques tirent parti des avantages de TypeScript, tels que le typage fort, les interfaces, les génériques et les décorateurs, pour améliorer la qualité du code, faciliter la maintenance et améliorer l'expérience de développement grâce à une meilleure autocomplétion et à la détection des erreurs en temps de compilation.

JavaScript vs TypeScript

mohamed@goumih.com

7

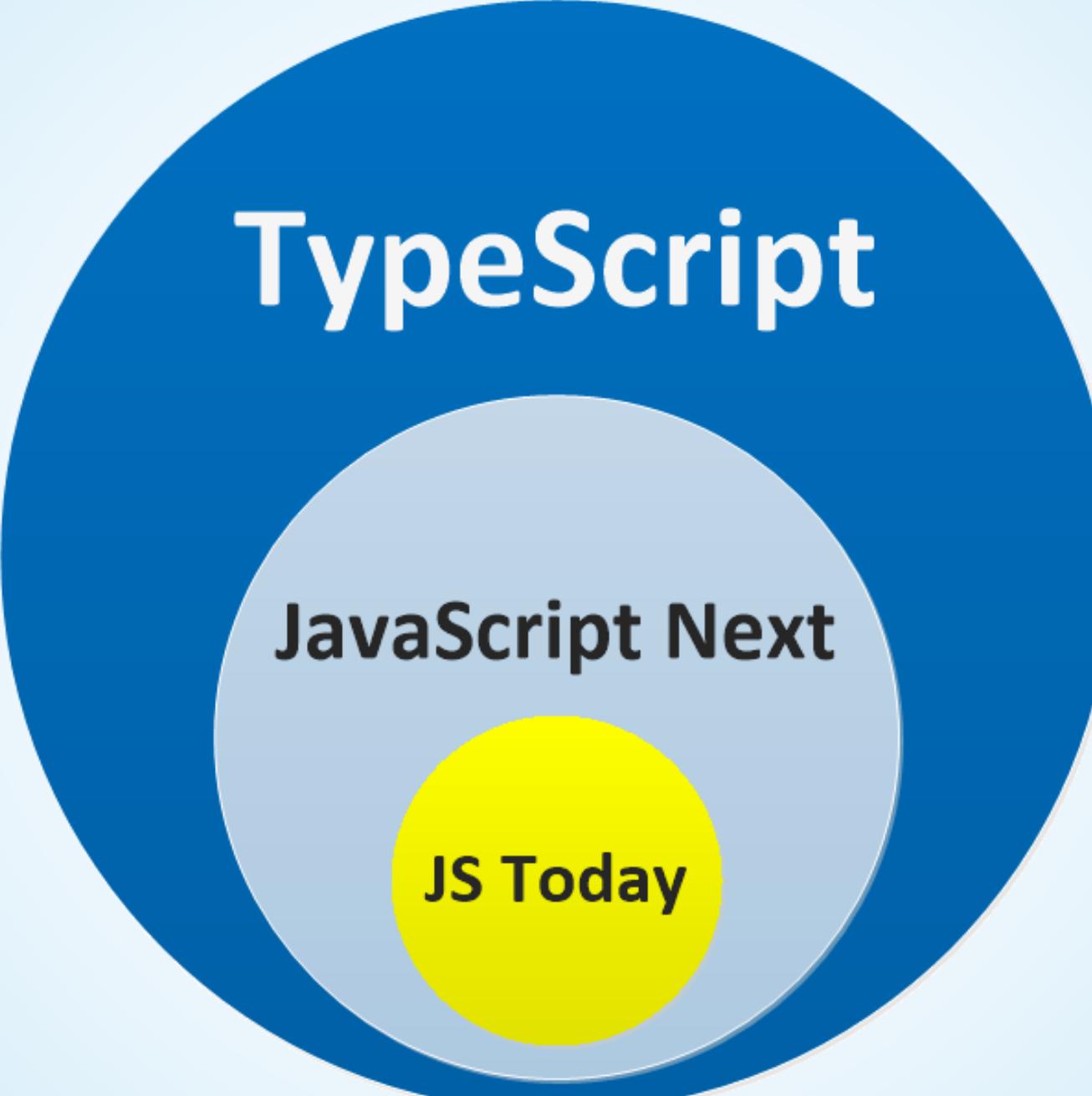
TypeScript standardise simplement toutes les façons dont vous fournissez *une bonne documentation* sur JavaScript.

TypeScript est juste du JavaScript avec de la documentation.

TypeScript essaiera de vous protéger contre des parties de JavaScript qui n'ont jamais fonctionné (vous n'avez donc pas besoin de vous souvenir de ce genre de choses) :

```
```ts
[] + []; // JavaScript vous donnera "" (ce qui n'a pas de sens), TypeScript affichera une
erreur
//
{} + []; // JS : 0, TS Error
[] + {}; // JS : "[object Object]", TS Error
{} + {}; // JS : NaN or [object Object][object Object] depending upon browser, TS Error
"hello" - 1; // JS : NaN, TS Error

function add(a,b) {
 return
 a + b; // JS : undefined, TS Error 'unreachable code detected'
}
```
```



TypeScript

JavaScript Next

JS Today

Premiers pas avec TypeScript

9

TypeScript se compile en JavaScript. JavaScript est ce que vous allez réellement exécuter (dans le navigateur ou sur le serveur). Vous allez donc avoir besoin des éléments suivants :

- * Compilateur TypeScript (OSS disponible [[dans les sources](#)](<https://github.com/Microsoft/TypeScript/>)) et sur [[NPM](#)](https://www.npmjs.com/package/typescript_))
- Un éditeur TypeScript :
• (vous pouvez utiliser :
[[vscode](#) 🐾](<https://code.visualstudio.com/>) avec une [[extension](#)](https://marketplace.visualstudio.com/items?itemName=ms-vscode.vscode-typescript-next_)).
- Aussi [[beaucoup d'autres IDEs le supportent également](#)](https://github.com/Microsoft/TypeScript/wiki>TypeScript-Editor-Support_))

Installer Typescript

10

Installez TypeScript à l'aide du nœud. Gestionnaire de paquets Js (npm).

Installez le plug-in typescript dans votre IDE (environnement de développement intégré).

➤ `npm install -g typescript`

➤ `tsc -v`

Version 5.3.0

TypeScript fournit un terrain de jeu en ligne <https://www.TypeScriptLang.Org/play> d'écrire et de tester votre code à la volée sans avoir besoin de télécharger ou d'installer quoi que ce soit.

Installer TypeScript

11

On recommande généralement aux gens d'utiliser la version nightly car
****la suite de tests du compilateur détecte plus de bugs au fil du temps**.**

Vous pouvez l'installer en ligne de commande en tapant :

```

```
npm install -g typescript@next
```

```

Et maintenant, la ligne de commande `tsc` sera la plus récente et la meilleure.
Divers IDE la prennent également en charge, par exemple :

* Vous pouvez demander à vscode d'utiliser cette version en créant un
``.vscode/settings.json`` avec le contenu suivant :

```json

{

  "`typescript.tsdk`": "./node\_modules/typescript/lib"

}

```

mohamed@goumih.com

Lancer un Projet TS

12

- Pour initialiser un projet TS .
➤ `tsc --init`

Il crée un fichier de configuration `tsconfig.json`.

- Pour créer un fichier Typescript ,utiliser l'extension `.ts`
- Exemple :`index.ts` .
- Pour compiler un fichier TS :
`tsc index.ts`
- Pour lancer le fichier js généré :
`node index.js`

- Pour activer le mode de compilation live sans besoin de recompiler à chaque fois lorsque vous faites la modification.
- `tsc -watch index.ts`

Configuration tsconfig.json

Pour exclure et inclure des fichiers de la compilation

```
"exclude": ["node_modules"],  
"include": ["main.ts"]
```

- ▶ Pour définir le dossier des sources : "rootDir": "./src",
- ▶ Pour définir le dossier de production "outDir": "./dist" /*
- ▶ Pour modifier le target du projet : "target": "es2016"

Type de données en TypeScript

- Number
- String
- Boolean
- Array
- Tuple
- Enum
- Union
- Any
- Void
- Never
- Unknown
- Type

Declaration des variables :

Les variables peuvent être déclarées à l'aide de :

var, let, const

Strings

Le type **string** en TypeScript représente une séquence de caractères.

```
let simpleString: string = "Hello, TypeScript!";
let doubleQuotes: string = "This is a string.";
let singleQuotes: string = 'This is also a string.'
```

```
let multiLineString :string = `Hello,
This is a multi-line string.
`;
```

```
let nom: string = "Mohamed";
let greeting: string = `Hello, ${nom}!`; // "Hello, mohamed!"
```

Toutes les méthodes et propriétés standard de JavaScript pour les chaînes sont également disponibles en TypeScript.
Par exemple :

```
let text: string = "TypeScript is great!";
let len: number = text.length;           // 19
let replaced: string = text.replace("great", "awesome"); // "TypeScript is awesome!"
```

Numbers

16

TypeScript possède un type de base appelé **number** qui représente tous les nombres entiers et flottants.

```
let integer: number = 42;  
let floating: number = 42.42;
```

Vous pouvez également définir des nombres en utilisant des notations hexadécimale, binaire ou octale :

```
let hexadecimal: number = 0xf00d; // 61453  
let binary: number = 0b1010; // 10  
let octal: number = 0o744; // 484
```

À partir de ECMAScript 2020 (ES11), JavaScript (et par extension TypeScript) supporte un nouveau type de nombre appelé **BigInt**. Il est conçu pour représenter des nombres entiers plus grands que **2^e53-1** qui est la plus grande valeur que JavaScript peut représenter en toute sécurité avec le type **number**.
n indique qu'il s'agit d'un nombre **bigint**

```
let bigIntValue: BigInt = 1234567890123456789012345678901234567890n;  
// il doit donner erreur si ES est inférieur à ES2020
```

Numbers

17

Comme en JavaScript, TypeScript supporte les valeurs **NaN** (Not a Number) et **Infinity**.

```
let notANumber: number = NaN;  
let positiveInfinity: number = Infinity;  
let negativeInfinity: number = -Infinity;
```

Toutes les fonctions mathématiques standard de JavaScript sont également disponibles en TypeScript, par exemple :

```
let square16: number = Math.sqrt(16); // 4
```

Vous pouvez convertir une chaîne en un nombre à l'aide de fonctions comme **parseInt** et **parseFloat**:

```
let strNumber: string = "42.42";  
let numFromStr: number = parseFloat(strNumber); // 42.42
```

Vous pouvez vérifier si une valeur est un nombre à l'aide de la fonction **isNaN**:

```
let isNumber = !isNaN(numFromStr); // true
```

Boolean ,Assertion,Union

```
let isPublished: boolean = true
```

Notez que le booléen avec un B majuscule est différent du booléen avec un b minuscule. Le booléen majuscule est un type d'objet tandis que le booléen minuscule est un type primitif. Il est toujours recommandé d'utiliser le booléen, le type primitif dans vos programmes. En effet, alors que JavaScript contraint un objet à son type primitif, le système de type typescript ne le fait pas. Typescript le traite comme un type d'objet.

Une variable peut avoir plusieurs types optionnels en utilisant l'opérateur de ou |

```
// Union  
let pid: string | number  
pid = '22'
```

L'assertion de type vous permet de définir le type d'une valeur et d'indiquer au compilateur de ne pas l'inférer. C'est à ce moment-là que vous, en tant que programmeur, pourriez avoir une meilleure compréhension du type d'une variable que ce que Typescript peut déduire par lui-même

```
// Type Assertion  
let cid: any = true  
let customerId = <number>cid  
let customerId2 = cid as number
```

Egalité

19

```
//JS
console.log(5 == "5"); // true , TS Error
console.log(5 === "5"); // false , TS Error

//JS
console.log("") == "0"); // false ,TS Error
console.log(0 == ""); // true ,TS Error

console.log("") === "0"); // false ,TS Error
console.log(0 === ""); // false,TS Error
```

Utilisez toujours `==` et `!=` sauf pour les vérifications nulles, que nous couvrirons plus tard.

Si vous voulez comparer deux objets pour l'égalité structurelle, `==` / `!=` ne sont ***pas*** suffisants, par exemple :

```
//JS
console.log({a:123} == {a:123}); // False
console.log({a:123} === {a:123}); // False
```

npm install deep-equal

```
import * as deepEqual from "deep-equal";
console.log(deepEqual({a:123},{a:123})); // True
```

Array

20

1. Utilisation de crochets. Cette méthode est similaire à la façon dont vous déclareriez des tableaux en JavaScript

```
let numbers: number[] = [1, 2, 3, 4, 5];
```

2. Utilisation d'un type de tableau générique, type tableau<élément>.

```
let strings: Array<string> = ["a", "b", "c"];
```

Bien sûr, vous pouvez toujours initialiser un tableau comme indiqué ci-dessous, mais vous n'aurez pas l'avantage du système de types de typescript.

```
let tab=[1,3,"Apple",false]
```

Array

21

Si un tableau doit contenir des éléments de types différents, vous pouvez utiliser le type union :

```
//union
let mix: (number | string)[] = [1, "a", 2, "b"];
let mix2:Array<string | number> = [1, "a", 2, 2, "b"];
```

```
//tableau multidimensionnel
let matrix: number[][] = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
```

TypeScript introduit le concept de **tuple**, qui permet d'exprimer un tableau où le type des éléments est connu et ne change pas :

Les tuples sont utiles lorsque vous savez exactement combien d'éléments il y aura dans le tableau et quel type chaque élément aura.

Si vous ne savez pas à l'avance quel type d'éléments un tableau contiendra, vous pouvez utiliser le type **any[]** :

```
let randomItems: any[] = [1, "hello", true, {}, []];
```

```
// Tuple :
let person: [number, string, boolean] =
  [1, 'DEV', true]
// Tuple Array
let employees: [number, string][]

employees = [
  [1, 'Ali'],
  [2, 'Ahmed'],
  [3, 'Sara'],
]
```

Object

En TypeScript, tout comme en JavaScript, un objet est une collection de paires clé-valeur. TypeScript ajoute une couche de typage statique aux objets, ce qui vous permet de définir des structures d'objets de manière plus explicite et de bénéficier d'une vérification de type au moment de la compilation.

```
//Déclaration d'objets simples
let personne: { name: string; age: number; } = {
    name: "Alice",
    age: 30
};
//type
type Utilisateur = {
    id: number
    name: string
}
const usr1: Utilisateur = {
    id: 1,
    name: 'John',
}
```

Type littéral

mohamed@goumih.com

23

TypeScript vous permet de définir un **type littéral** de chaîne, ce qui signifie que vous pouvez restreindre une variable à avoir une valeur spécifique parmi un ensemble de chaînes :

```
//type litteral
type Color = "red" | "green" | "blue";
let myColor: Color = "red"; // Valid
//myColor = "yellow";
// Error: Type '"yellow"' is not assignable to type 'Color'.
```

```
//type similaire à une tableau
type tableau=[number,string]
let ar:tableau=[1,"aa"]
console.log(ar[0]) //1
```

```
//type object
type obj={
    name:string,
    age:number
}
const user:obj={name:"ahmed",age:35};
console.log(user.name) //ahmed
```

Type intersection

Les types d'intersection en TypeScript permettent de combiner plusieurs types en un seul. Cela signifie que vous pouvez prendre les propriétés de plusieurs types et les fusionner en un seul type. Les types d'intersection sont un outil puissant pour composer des types complexes et flexibles.

Pour créer un type d'intersection, utilisez le caractère & entre deux ou plusieurs types.
Par exemple :

```
type TypeA = {  
    a: string;  
    b: number;  
};  
  
type TypeB = {  
    b: number;  
    c: boolean;  
};  
  
type IntersectionType = TypeA & TypeB;
```

Dans cet exemple, IntersectionType sera un type qui possède toutes les propriétés de TypeA et TypeB.

```
type TypeC = {  
    a: string;  
    b: number;  
};  
  
type TypeD = {  
    a: number;  
    b: number;  
};  
  
type IntersectionTypeCD = TypeC & TypeD;
```

// IntersectionTypeCD a le type { a: never; b: number; } (car il n'y a aucun type qui soit à la fois string et number).

Type intersection

Utilisation avec des objets :

Les types d'intersection sont particulièrement utiles pour étendre des objets avec des propriétés supplémentaires.

```
type Person = {  
    name: string;  
    age: number;  
};  
  
type Employee = Person & {  
    employeeId: number;  
};  
  
let employee: Employee = {  
    name: "Ali",  
    age: 30,  
    employeeId: 12345  
};
```

Types des éléments DOM

26

En TypeScript, le DOM (Document Object Model) est représenté par une variété de types qui correspondent aux différents types d'éléments et de noeuds que vous pouvez trouver dans le DOM d'une page web. Ces types sont définis dans la librairie de types DOM fournie avec TypeScript, qui est basée sur les spécifications du DOM fournies par le WHATWG (Web Hypertext Application Technology Working Group) et le W3C (World Wide Web Consortium).

Voici quelques-uns des types d'éléments DOM les plus courants en TypeScript :

1. **HTMLElement**: Représente n'importe quel élément HTML. C'est une classe de base pour tous les autres types d'éléments HTML.
2. **HTMLInputElement**: Représente un élément HTML `<input>`. Il hérite de **HTMLElement** et ajoute des propriétés spécifiques aux éléments input, comme **value**, **checked**, etc.
3. **HTMLButtonElement**: Représente un élément HTML `<button>`. Il hérite de **HTMLElement** et est utilisé pour les interactions avec les boutons.
4. **HTMLDivElement**: Représente un élément HTML `<div>`. Il hérite de **HTMLElement** et est souvent utilisé pour la mise en page ou comme conteneur pour d'autres éléments.
5. **HTMLAnchorElement**: Représente un élément HTML `<a>`, c'est-à-dire un hyperlien. Il hérite de **HTMLElement** et contient des propriétés comme **href**, qui représente l'URL du lien.
6. **HTMLImageElement**: Représente un élément HTML ``. Il hérite de **HTMLElement** et contient des propriétés pour gérer les images, comme **src**, **alt**, etc.

Discriminated Unions

27

Un "Discriminated Union" (ou union discriminée) est un pattern qui combine des unions de types avec des champs littéraux pour faciliter l'affinement du type.

```
type Circle = {  
    kind: "circle";  
    radius: number;  
};  
type Square = {  
    kind: "square";  
    sideLength: number;  
};  
  
type Shape = Circle | Square;  
  
function getArea(shape: Shape) {  
    switch (shape.kind) {  
        case "circle":  
            return Math.PI * shape.radius ** 2;  
        case "square":  
            return shape.sideLength ** 2;  
    }  
}
```

Dans cet exemple, Shape est une union discriminée de Circle et Square, où le champ `kind` sert de discriminant. La fonction `getArea` utilise ce discriminant pour déterminer de quel `type de forme` il s'agit et comment calculer l'aire correspondante. TypeScript comprend que dans chaque branche du `switch`, `shape` doit être de type `correspondant` (`Circle` ou `Square`), permettant ainsi un accès sûr et direct aux propriétés pertinentes.

Types des éléments DOM

28

7.HTMLFormElement: Représente un élément HTML `<form>`. Il hérite de **HTMLElement** et est utilisé pour travailler avec des formulaires.

8.HTMLSelectElement: Représente un élément HTML `<select>`. Il hérite de **HTMLElement** et est utilisé pour créer une liste déroulante.

9.HTMLOptionElement: Représente un élément HTML `<option>`. Il est généralement utilisé en conjonction avec **HTMLSelectElement**.

10.HTMLCanvasElement: Représente un élément HTML `<canvas>`. Il hérite de **HTMLElement** et est utilisé pour dessiner des graphiques via JavaScript.

11.Document: Représente le document entier et peut être utilisé pour accéder à la racine du DOM.

12.Node: C'est un type plus générique qui représente n'importe quel nœud du DOM, pas seulement des éléments HTML. Il contient des propriétés et des méthodes communes à tous les types de nœuds.

13.EventTarget: C'est une interface que peuvent implémenter des objets qui peuvent recevoir des événements, comme **HTMLElement, Document et Window**.

14.Window: Représente la fenêtre contenant un document DOM; l'objet **window** en JavaScript est une instance de ce type. Pour utiliser ces types en TypeScript, vous pouvez les spécifier lors de la manipulation d'éléments DOM, pour bénéficier de la vérification de type statique et de l'autocomplétion de votre éditeur de code. Par exemple :

Types des éléments DOM

29

Pour utiliser ces types en TypeScript, vous pouvez les spécifier lors de la manipulation d'éléments DOM, pour bénéficier de la vérification de type statique et de l'autocomplétion de votre éditeur de code.

Par exemple :

```
const button1: HTMLButtonElement = document.querySelector('button');
button1.addEventListener('click', (event) => {
    // ...
});
```

Dans cet exemple, **button** est de type **HTMLButtonElement**, ce qui signifie que TypeScript connaît les propriétés et les méthodes spécifiques à cet élément, et vous avertira si vous essayez d'accéder à quelque chose qui n'existe pas sur un élément de bouton.

L'utilisation de **as** en TypeScript est un moyen d'effectuer une assertion de type. Cela indique à TypeScript que vous êtes sûr du type d'une certaine variable ou valeur, même si TypeScript ne peut pas le déduire automatiquement. Voici un exemple d'utilisation de **as** avec des éléments DOM en TypeScript :

```
// Supposons que nous avons un élément avec l'ID
// 'myInput' dans le DOM
const inputElement =
document.getElementById('myInput') as
HTMLInputElement;

// Maintenant, TypeScript sait que inputElement est
// un HTMLInputElement,
// et vous pouvez accéder en toute sécurité à ses
// propriétés spécifiques
inputElement.value = 'Hello, World!';
```

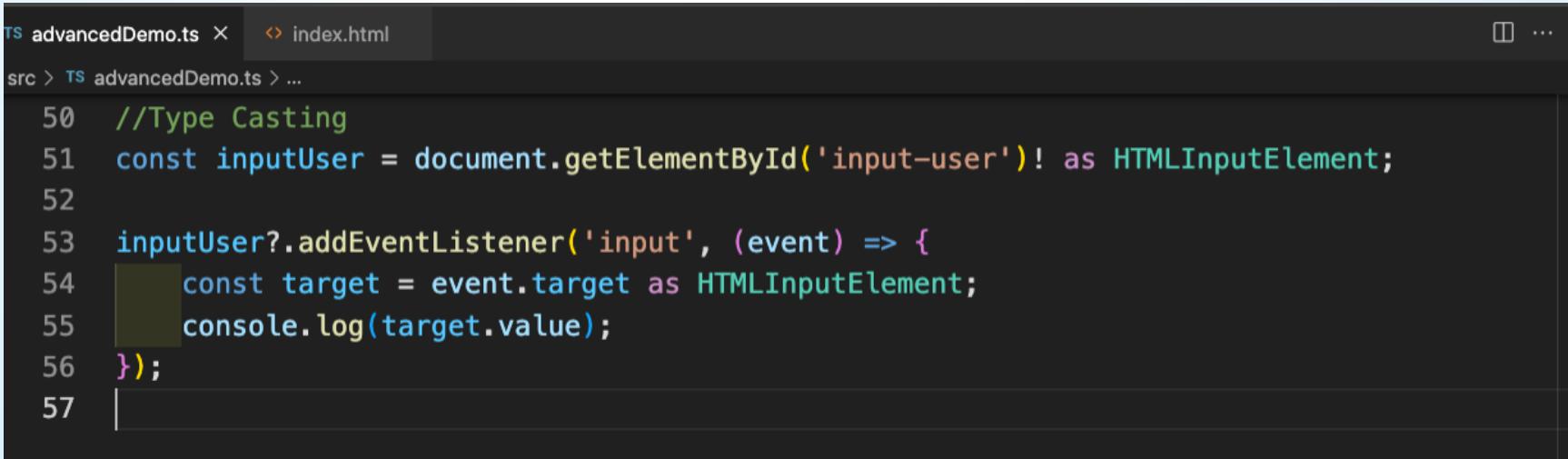
Types des éléments DOM

Type Casting

30

nous sélectionnons un élément avec l'habituel `getElementById`. Mais la différence c'est que nous avons un `!` après ça. Ça raconte l'expression qui la précède ne sera jamais nulle.

Après cela, nous disons à TypeScript avec le mot-clé `as` indiquant qu'il s'agit d'un `HTMLInputElement`. Après cela, nous avons le code habituel de l'écouteur d'événement. Ici, nous sommes aussi indiquant à TypeScript qu'il est facultatif par `?`. Ensuite, à l'intérieur de l'événement écouteur également, nous devons dire que la cible est `HTMLInputElement`.



```
ts advancedDemo.ts × index.html
src > ts advancedDemo.ts > ...
50 //Type Casting
51 const inputUser = document.getElementById('input-user')! as HTMLInputElement;
52
53 inputUser?.addEventListener('input', (event) => {
54     const target = event.target as HTMLInputElement;
55     console.log(target.value);
56 });
57
```

Fonctions

TypeScript ajoute des fonctionnalités de typage qui permettent de définir les types des paramètres et des valeurs de retour. Cela apporte plus de clarté et de sécurité à votre code.

Typage basique

Vous pouvez spécifier les types pour les paramètres et la valeur de retour d'une fonction :

```
function greet(name: string): string {  
    return "Hello, " + name;  
}
```

Paramètres optionnels

Les paramètres de fonction peuvent être rendus optionnels en TypeScript

en ajoutant un ? après le nom du paramètre :

```
function displayInfo(name: string, age?: number)  
: string {  
    if (age) {  
        return `${name} is ${age} years old.`;  
    }  
    return `Hello, ${name}!`;  
}
```

Paramètres par défaut

```
function createRectangle(width: number = 10,  
height: number = 5): { width: number; height:  
number; } {  
    return { width, height };  
}
```

Paramètres de type union

Un paramètre peut accepter des valeurs de plusieurs types à l'aide de types d'union :

```
function display(value: string | number) {  
    console.log(value);  
}
```

Fonctions

32

Fonctions comme types

En TypeScript, les fonctions peuvent elles-mêmes être typées :

```
type GreetingFunction = (name: string, age?: number) => string;
const sayHello: GreetingFunction = (name, age) => {
    return age ? `Hello, ${name}. You are ${age} years old.` : `Hello, ${name}!`;
```

Fonctions surchargées

TypeScript prend en charge la surcharge de fonctions, ce qui signifie que vous pouvez définir plusieurs signatures de fonction pour une seule fonction :

```
function add(a: string, b: string): string;
function add(a: number, b: number): number;
function add(a: any, b: any): any {
    if (typeof a === 'string' && typeof b === 'string') {
        return a + b;
    }
    if (typeof a === 'number' && typeof b === 'number') {
        return a + b;
    }
}
console.log(add(5, 3));      // 8
console.log(add('5', '3'));
```

Fonctions

33

Fonctions anonymes et fonctions fléchées

Tout comme en JavaScript ES6, TypeScript prend en charge les fonctions anonymes et les fonctions fléchées :

```
const square = function(x: number): number {  
    return x * x;  
};
```

```
const arrowSquare = (x: number): number => x * x;
```

Rest Parameters

Vous pouvez utiliser le paramètre "rest" pour travailler avec un nombre variable d'arguments :

```
function buildName(firstName: string, ...restOfName: string[]): string {  
    return firstName + " " + restOfName.join(" ");  
}
```

```
let fullName = buildName("John", "Doe", "Jr.");  
console.log(fullName); // John Doe Jr.
```

Fonctions :type void

34

En TypeScript, le type void est utilisé dans le contexte de type de retour de fonction pour signifier qu'une fonction ne retourne rien. En d'autres termes, une fonction qui est déclarée avec un type de retour void ne doit pas retourner de valeur.

Fonction sans valeur de retour

Si une fonction n'a pas d'instruction return ou si elle a une instruction return sans valeur, elle peut être typée comme void.

```
function logMessage(message: string): void {  
    console.log(message);  
}
```

```
logMessage("Hello, TypeScript!"); // Affiche "Hello, TypeScript!" dans la console
```

Utilisation du type void avec des variables

Il n'est généralement pas utile d'utiliser void comme type de variable, car une variable de type void ne peut accepter que undefined (et null si l'option strictNullChecks est désactivée).

```
let unusable: void = undefined;  
// unusable = "some string"; // Erreur: Type 'string' is not assignable to type 'void'
```

Fonctions :type void

35

Différence entre void et undefined

Bien que `void` et `undefined` puissent sembler similaires, il y a une différence subtile entre eux. `void` est utilisé comme `type de retour` pour signifier qu'une fonction ne retourne rien. D'autre part, `undefined` est un type `de données` qui représente une valeur non définie. Une fonction avec un type `de retour` `void` retourne effectivement `undefined`, mais le type `void` sert à indiquer l'intention de ne pas retourner de valeur.

Fonctions fléchées et void

Vous pouvez également utiliser `void` avec des fonctions fléchées.

```
const showAlert = (message: string): void => {
  alert(message);
};

showAlert("This is an alert!");
```

Utiliser le `type void` est une manière explicite de communiquer que la fonction est appelée pour ses effets secondaires (par exemple, logging, modification d'une variable externe, etc.) plutôt que pour sa valeur de retour.

Fonctions :type never

mohamed@goumih.com

36

Le type never en TypeScript représente une valeur qui ne doit jamais se produire. C'est un type qui est utilisé pour représenter des scénarios où une valeur ne sera jamais observée. Voici quelques utilisations courantes de never :

Différence entre void et never :
void est utilisé pour les fonctions qui ne retournent pas de valeur (ou qui retournent undefined). never est utilisé pour les fonctions qui ne retournent jamais ou qui lancent toujours une exception.

En résumé, le type never en TypeScript est un moyen de représenter des valeurs qui ne se produisent jamais. Il est utile pour les vérifications de type au moment de la compilation et pour s'assurer que certaines branches de code sont inaccessibles.

Fonction qui lance une exception:

Si une fonction lance toujours une exception et ne retourne jamais normalement, elle peut être typée avec never.

```
function throwError(message: string): never {  
    throw new Error(message);  
}
```

Fonction qui ne termine jamais:

Si une fonction entre dans une boucle infinie et ne termine jamais, elle peut également être typée avec never.

```
function infiniteLoop(): never {  
    while (true) {}  
}
```

Fonctions :type never

mohamed@goumih.com

37

Discrimination de types:

Le type `never` est souvent utilisé dans des discriminations de types pour s'assurer que tous les cas possibles d'un type d'union sont traités.

```
type Shape = { kind: 'circle', radius: number } | { kind: 'square', sideLength: number };
function getArea(shape: Shape): number {
    switch (shape.kind) {
        case 'circle':
            return Math.PI * shape.radius * shape.radius;
        case 'square':
            return shape.sideLength * shape.sideLength;
        default:
            const _exhaustiveCheck: never = shape;
            return _exhaustiveCheck;
    }
}
```

Dans l'exemple ci-dessus, le code garantit que tous les types possibles de `Shape` sont traités dans le `switch`. Si un nouveau type est ajouté à `Shape` et n'est pas traité, `TypeScript` générera une erreur.

EP01

- A. Concaténez deux chaînes de caractères, **firstName** et **lastName**, avec un espace entre elles, et stockez le résultat dans une variable appelée **fullName**.
- B. Trouvez la longueur de la chaîne de caractères: "TypeScript is awesome!".
- C. Remplacez le mot "TypeScript" par "JavaScript" dans la chaîne de caractères: "I love TypeScript!".
- D. Écrire un programme TypeScript qui compte le nombre de caractères dans une chaîne sans utiliser la propriété **length**.
- E. Écrire un programme TypeScript qui inverse une chaîne de caractères sans utiliser la fonction `reverse`.
- F. Écrire un programme TypeScript qui compte le nombre de voyelles dans une chaîne.
- G. Écrire un programme TypeScript qui trouve le caractère qui apparaît le plus souvent dans une chaîne de caractères.
- H. Écrire un programme TypeScript qui supprime tous les espaces d'une chaîne.
- I. Écrire un programme TypeScript qui compte combien de fois un sous-ensemble de caractères apparaît dans une chaîne.
- J. Écrire une fonction qui prend une chaîne de caractères représentant un titre de page et la formate pour l'affichage en capitalisant le premier caractère de chaque mot.
- K. Écrire une fonction qui tronque une chaîne de caractères à une longueur donnée et ajoute des points de suspension si elle est plus longue.
- L. Écrire une fonction qui convertit un titre en un slug .
- M. Écrire une fonction qui valide si une chaîne de caractères est une adresse email valide.(email valide doit contenir @ et .)

```
console.log(generateSlug("Full Stack 201"))
full-stack-201
```

EP02:

39

- A. Vérifier si un nombre est pair ou impair .
- B. Trouver le maximum de deux nombres.
- C. Écrire une fonction qui calcule la somme des chiffres d'un nombre entier.
- D. Écrire une fonction qui génère un nombre aléatoire entre deux bornes,max et min incluses.
- E. Ecrire une fonction qui Formate un nombre en devise : \$23.50 .
- F. Écrire une fonction qui trouve le nombre manquant dans une progression arithmétique donnée.
`console.log(findMissingInArithmeticSeq([2,4,8]))//6` .
- G. Écrire une fonction qui convertit un entier en sa représentation en chiffres romains
`:console.log(integerToRoman(55))` .

- H. Définissez une fonction **setupCounter** qui incrémente un compteur à chaque fois qu'un bouton est cliqué et affiche le compte mis à jour sur le bouton lui-même.
- I. Utilisez **localStorage** pour stocker et récupérer la valeur du compteur.
- J. Ajoutez une fonction pour réinitialiser le compteur et mettre à jour **localStorage**.

Le compteur est à 67

Réinitialiser

- L. Créer une fonction **convertSecondsToTimer** qui convertit un nombre en un **timer** de minutes et secondes :Exemple 3600 **60:00** .

60:00

- M. Implémentez un timer de 60 min en DOM avec cette fonction.

- N. On veut que l'utilisateur entre la valeur du timer dans un input :créer une fonction **startTimer** qui démarre le timer avec la valeur fournie par l'utilisateur.

- O. Implémentez le nouveau timer en DOM.

Entrez les secondes Démarrer le Timer

00:00

EP03:

40

- A. Créez une fonction pour Trouver les diviseurs d'un nombre.
- B. Écrivez une fonction qui prend un tableau de nombres et renvoie un nouveau tableau avec les éléments dans l'ordre inverse.
- C. Écrivez une fonction qui prend un tableau de nombres et renvoie le plus grand nombre du tableau.
- D. Écrivez une fonction qui prend un tableau de n'importe quel type et un type en chaîne de caractères (comme "string" ou "number"). La fonction doit renvoyer un nouveau tableau contenant uniquement les éléments du type spécifié.
- E. Écrivez une fonction qui prend un tableau de tableaux(matrices) de nombres et renvoie un tableau plat de tous les nombres.
- F. Écrivez une fonction qui prend un tuple [string, number] et renvoie un objet avec les propriétés name et age basées sur ce tuple.
- G. Écrivez une fonction qui prend plusieurs tableaux de nombres comme arguments et renvoie un nouveau tableau qui est la combinaison de tous les tableaux, sans doublons. **combineWithoutDuplicates(...arrays: number[][]): number[]** .
- H. Écrivez une fonction qui prend un tableau de chaînes et crée une liste non ordonnée () dans le DOM avec chaque chaîne comme élément de liste ().
- I. Supposons que vous avez déjà une liste non ordonnée dans le DOM avec des éléments de liste. Écrivez une fonction qui met à jour cette liste avec un nouveau tableau de chaînes, en ajoutant ou en supprimant des éléments au besoin.
- J. Écrivez une fonction qui trie les éléments d'une liste non ordonnée dans le DOM, soit en ordre alphabétique, soit en ordre inverse, en fonction d'un paramètre.
- K. Écrivez une fonction qui filtre les éléments d'une liste non ordonnée dans le DOM pour afficher uniquement ceux qui contiennent un texte donné.

EP04:

41

- A. Écrivez une fonction **combine** qui prend deux arguments et les retourne combinés.
- B. Si les deux arguments sont des nombres, ils doivent être ajoutés, si ce sont des chaînes, elles doivent être concaténées.
- C. Créez un alias de type appelé **StringOrNumber** qui est soit une chaîne, soit un nombre, ensuite, écrivez une fonction qui utilise cet alias de type pour déterminer si l'argument est une chaîne ou un nombre.
- D. Créez une fonction d'ordre supérieur qui accepte un nombre et renvoie une fonction closure qui ajoute ce nombre à un autre nombre passé en paramètre.
- E. Écrivez une fonction qui prend une fonction de rappel comme argument et l'exécute en passant un message.

- G. Définir un type **OperationCallback** de deux nombres.
- H. Définir une fonction callback **executeOperation** qui exécute une fonction des opérations arithmétiques .
- I. Tester la fonction précédente avec les différentes opérations de calcul : + , - , - * , / , % .

- J. Créez une fonction **calculateArea** qui peut être appelée avec des arguments différents pour calculer l'aire de différentes formes géométriques (cercle, carré, rectangle).
- K. Créez une fonction **Identite** qui accepte soit un prénom et un nom, soit un prénom seul, et renvoie le nom complet.
- L. (Surcharge) Écrivez une fonction **padLeft** qui ajoute des espaces à gauche d'une chaîne de caractères si un nombre est spécifié, ou une chaîne de caractères spécifiée si une chaîne est fournie comme premier argument.

EP05:

mohamed@goumih.com

42

```
Product:product={{ name: 'Banana', category: 'Fruit', price: 1 }  
const products:product[] = [  
  { name: 'Banana', category: 'Fruit', price: 1 },  
  { name: 'Carrot', category: 'Vegetable', price: 2 },  
  { name: 'Apple', category: 'Fruit', price: 3 }  
];
```

- A. Déclarer le type **Product**
- B. Créez une fonction qui ajoute une nouvelle propriété à un objet existant. **addProperty(obj: any, key: string, value: any)**.
- C. Écrivez une fonction qui supprime une propriété d'un objet si elle existe.
- D. Écrivez une fonction qui copier tous les attributs d'un objet.
- E. Écrivez une **fonction findByKey(array: any[], key: string, value: any)** qui prend un tableau d'objets et une clé, et renvoie le premier objet qui a une propriété avec une valeur spécifique pour cette clé.
- F. Créer une fonction qui Filtre les propriétés d'un objet pour ne garder que celles dont la valeur est de type **string**.
- G. Écrivez une fonction qui prend un tableau d'objets et renvoie un nouveau tableau contenant uniquement les objets qui répondent à un critère de filtrage sur une propriété spécifiée. `const fruits = filterByProperty(products, 'category', (value) => value === 'Fruit');`
- H. Écrivez une fonction qui met à jour la valeur d'une propriété spécifique d'un objet dans un tableau, en fonction de la valeur d'une autre propriété.
- I. Écrivez une fonction qui calcule la somme des valeurs d'une propriété numérique pour tous les objets d'un tableau.
- J. Écrivez une fonction qui prend un tableau d'objets et extrait une liste des valeurs pour une propriété spécifiée de chaque objet.
- K. Créez un objet qui représente une **calculatrice** avec des fonctions pour ajouter, soustraire, multiplier et diviser. Ces fonctions doivent utiliser **this** pour accéder à deux propriétés de l'objet qui contiennent les nombres à opérer.

EP06:

supposons que nous avons une structure HTML de base **ep05.html** :
Nous voulons créer l'application suivante :

```
<body>
  <div id="app"></div>
  <script src="app.js"></script>
</body>
```

• MOHAMED GOUMIH	mohamed.goumih@ofppt.nr	Editer	Supprimer
• MOHAMED GOUMIH	mohamed@goumih.com	Editer	Supprimer

Name	Email	Ajouter
------	-------	-------------------------

Vous avez un tableau d'objets représentant des utilisateurs, chaque utilisateur ayant un **id**, un **nom**, et un **email**:

- A. Créer le type **User**.
- B. Créer un tableau **users** avec le type **User** avec quelques objets.
- C. Créez une fonction **displayUsers** qui prend ce tableau et l'affiche dans le DOM sous forme de liste.

On veut ajouter une fonctionnalité qui permet aux utilisateurs d'ajouter un nouvel utilisateur à la liste en utilisant un formulaire HTML simple. L'utilisateur doit être ajouté à la fois au tableau et à la liste affichée dans le DOM :

- D. Ajoutez un formulaire HTML à votre fichier index.html.
- E. Écrivez la fonction **addUser** pour gérer l'ajout d'un nouvel utilisateur.
- F. Créer une fonction **saveUsersToLocalStorage** qui permet de stocker un tableau dans le localStorage.
- G. Créer une fonction **loadUsersFromLocalStorage** qui permet de récupérer un tableau de localStorage.
- H. Créer une fonction **deleteUser** qui permet de supprimer un élément de la liste.
- I. Créer une fonction **createUserListItem** qui permet de crée et retourne un élément de liste (****) pour chaque utilisateur, avec des champs de saisie pour le nom et l'email, ainsi que des boutons pour éditer et supprimer.
- J. Créer une fonction **createUserListItem** qui permet de modifier un élément de la liste.

Enum

44

Les énumérations sont un nouveau type de données pris en charge dans typescript.

Les enum en TypeScript sont un moyen puissant de créer :des ensembles nommés de constantes.

Ils rendent le code plus lisible et garantissent la sécurité du typage lors de l'utilisation de ces constantes.

Il existe trois types d'énumérations :Énumération numérique ,Énumération de chaînes, Enumération hétérogène

```
enum Days {
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}
```

```
let today: Days = Days.Saturday;
console.log(today); // 7
/*(parce que Saturday est le 7eme élément,
et l'indexation commence à 0)
*/
//Vous pouvez également obtenir le nom d'un
élément enum
//à partir de sa valeur :

console.log(Days[3]); // "Saturday"
```

Les éléments d'un **enum** sont numérotés par défaut en commençant par 0, mais vous pouvez leur attribuer des valeurs spécifiques :

```
enum StatusCode {
    OK = 200,
    NotFound = 404,
    InternalServerError = 500
}
```

```
console.log(StatusCode.OK); // 200
console.log(StatusCode[404]); // "NotFound"
```

Enum

45

```
//!Vous pouvez également utiliser des chaînes  
//!comme valeurs pour un enum :  
//enum string  
enum Colors {  
    Red = "RED",  
    Green = "GREEN",  
    Blue = "BLUE"  
}  
let favoriteColor: Colors = Colors.Red;  
console.log(favoriteColor); // "RED"
```

//!Même s'il est généralement recommandé de ne pas le faire, TypeScript permet d'avoir des enum avec des valeurs mixtes :

```
enum Mixed {  
    Yes = "YES",  
    No = 0,  
    Maybe = 1  
}
```

//!Un enum peut contenir des membres constants et des membres calculés

```
enum EnumWithCalc {  
    A,  
    B = A * 2,  
    C,  
    D = B + 3  
}
```

Interfaces

Au lieu de définir le type directement lors de la déclaration de la variable, vous pouvez utiliser une interface pour définir la structure de l'objet. C'est particulièrement utile lorsque vous avez plusieurs objets partageant la même structure.

Les interfaces ne peuvent être utilisées que dans les objets.

```
interface Person {  
    name: string;  
    age: number;  
}  
  
let ali: Person = {  
    name: "Alice",  
    age: 30  
};  
  
let ahmed: Person = {  
    name: "Bob",  
    age: 25  
};
```

Propriétés optionnelles: Dans une interface, vous pouvez marquer certaines propriétés comme optionnelles en utilisant le symbole ?.

```
interface Product {  
    id: number;  
    name: string;  
    description?: string;  
}  
  
let book: Product = {  
    id: 1,  
    name: "TypeScript Guide"  
    // Pas de description ici, et c'est OK car  
    c'est une propriété optionnelle  
};
```

Interfaces

Types indexés

Si vous ne savez pas à l'avance quelles seront les clés de votre objet,

mais que vous connaissez le type des valeurs, vous pouvez utiliser un type indexé.

```
interface StringDictionary {  
  [key: string]: string;  
}
```

```
let colors: StringDictionary = {  
  red: "#FF0000",  
  green: "#00FF00",  
  blue: "#0000FF"  
};
```

Type d'union dans les objets

Vous pouvez utiliser des types d'union pour permettre à une propriété d'avoir plusieurs types possibles.

```
interface Artwork {  
  name: string;  
  year: number | string;  
  // Peut être un nombre ou une chaîne de caractères  
}  
  
let painting: Artwork = {  
  name: "The Starry Night",  
  year: "1889" // C'est une chaîne,  
  //mais cela pourrait aussi être un nombre  
};
```

interfaces

48

Propriétés en lecture seule

Avec **TypeScript**, vous pouvez marquer des propriétés comme étant en lecture seule, ce qui signifie qu'**une fois qu'elles ont été initialisées**, elles ne peuvent plus être modifiées.

```
interface Config {  
    readonly apiUrl: string;  
    readonly port: number;  
}  
  
let myConfig: Config = {  
    apiUrl: "https://api.example.com",  
    port: 443  
};  
  
// myConfig.apiUrl = "https://other-url.com";  
// Erreur ! apiUrl est en lecture seule
```

Extension d'interfaces

Les interfaces peuvent être étendues, ce qui permet de créer de nouvelles interfaces basées sur des interfaces existantes.

```
interface Animal {  
    name: string;  
}  
  
interface Bird extends Animal {  
    wingspan: number;  
}  
  
let eagle: Bird = {  
    name: "Eagle",  
    wingspan: 200  
};
```

Interfaces

49

Une interface peut également définir le type d'un tableau où vous pouvez définir le type d'index ainsi que des valeurs.

```
interface numList{
  [index:number]:number
}
let numArr:numList=[1,2,3];
numArr[0]=3;
numArr[2]=34;
numArr[3]=34;
console.log(numArr)
```

```
interface stringList{
  [index:string]:string
}
let strAr:stringList={};
strAr["TS"]="TypeScript";
strAr["JS"]="JavaScript";
console.log(strAr)
```

interface numlist définit un type de tableau avec index comme nombre et valeur comme type nombre. De la même manière, stringlist définit un tableau de chaînes avec index comme chaîne et value comme chaîne.

Définir une interface de typefunction :

```
interface MathFunc {
  (x: number, y: number): number
}
const add: MathFunc = (x: number, y: number): number => x + y
const sub: MathFunc = (x: number, y: number): number => x - y
```

Types optionnels

Les types optionnels en TypeScript sont une caractéristique qui vous permet de déclarer qu'une propriété ou un paramètre peut être soit d'un certain type, soit absent (c'est-à-dire **undefined**). Cette fonctionnalité est très utile pour décrire des structures de données ou des signatures de fonctions où certains éléments ne sont pas toujours requis.

Dans les interfaces et les types:

Vous pouvez marquer une propriété comme optionnelle en utilisant un point d'interrogation (?) après le nom de la propriété.

```
interface Person {  
    name: string;  
    age?: number; // Age est optionnel  
}  
let person: Person = { name: "Ali" };  
// Valide, car age est optionnel
```

Types optionnels

51

Utilisation avec Prudence:
Bien que les types optionnels soient pratiques, ils peuvent également introduire de l'incertitude dans votre code. Si vous accédez à une propriété optionnelle sans vérifier d'abord si elle existe, vous risquez de rencontrer des erreurs à l'exécution. TypeScript offre des mécanismes tels que l'opérateur de coalescence nulle (??) et l'opérateur de chaînage optionnel (?.) pour gérer ces situations de manière sûre.

```
interface Person {  
    name: string;  
    age?: number;  
}  
  
function getAge(person: Person): string {  
    return person.age?.toString() ?? "Unknown";  
}  
  
let person: Person = { name: "Ali" };  
  
console.log(getAge(person)); // Affiche "Unknown"  
  
Dans cet exemple, person.age?.toString() renvoie undefined si age n'est pas présent, et l'opérateur de coalescence nulle (??) est utilisé pour fournir une valeur par défaut ("Unknown").
```

Classes

52

Les classes sont un élément fondamental de la programmation orientée objet, et TypeScript offre une syntaxe robuste pour définir et travailler avec des classes, enrichissant la syntaxe de classe introduite par ECMAScript 2015 (ES6) avec un typage statique et d'autres fonctionnalités.

Définition de base d'une classe :

```
class Animal {  
    name: string;  
  
    constructor(name: string) {  
        this.name = name;  
    }  
  
    move(distance: number) {  
        console.log(` ${this.name} moved  
${distance} meters. `);  
    }  
}
```

Héritage:

Comme dans la plupart des systèmes de classes basés sur des prototypes, TypeScript prend en charge l'héritage basé sur des classes :

```
class Dog extends Animal {  
    bark() {  
        console.log('Woof! Woof!');  
    }  
  
    const dog = new Dog("Rex");  
    dog.bark(); // Woof! Woof!  
    dog.move(10); // Rex moved 10 meters.
```

Classes

53

Modificateurs d'accès:

TypeScript supporte les modificateurs d'accès **public**, **private** et **protected** :

public (par défaut) : La propriété/méthode peut être accédée de n'importe où.

private : La propriété/méthode peut être accédée uniquement à l'intérieur de la **classe qui la définit**.

protected : La propriété/méthode peut être accédée dans la classe qui la définit et dans ses sous-classes.

```
class Person {  
    private socialSecurityNumber: string;  
    constructor(ssn: string) {  
        this.socialSecurityNumber = ssn;  
    }  
    protected getSSN(): string {  
        return this.socialSecurityNumber;  
    }  
}
```

Vous pouvez marquer des membres de classe comme étant en lecture seule avec le modificateur **readonly** :

```
class Circle {  
    readonly PI = 3.141592653589793;  
  
    constructor(public radius: number) {}  
  
    area() {  
        return this.PI * this.radius *  
            this.radius;  
    }  
}
```

Classes:

Accesseurs (getters et setters)

TypeScript prend en charge les accesseurs pour obtenir ou définir les valeurs des propriétés :

```
class Rectangle {  
    constructor(private _width: number, private _height: number)  
{}  
  
    get area() {  
        return this._width * this._height;  
    }  
  
    set width(value: number) {  
        this._width = value;  
    }  
  
    get width() {  
        return this._width;  
    }  
}
```

```
let rect = new Rectangle(10, 5);  
console.log(rect.area); // 50  
rect.width = 20;  
console.log(rect.area); // 100
```

Classes:

Paramètres de propriété dans le constructeur Au lieu de définir explicitement les membres de classe, puis de les initialiser dans le constructeur, **TypeScript** vous permet de les faire simultanément :

```
class Cat {  
    constructor(public name: string,  
private age: number) {}  
  
    describe() {  
        console.log(` ${this.name} is  
${this.age} years old.`);  
    }  
}
```

Classes abstraites:

Les classes abstraites servent de base pour d'autres classes. Elles ne peuvent pas être instanciées directement :

```
abstract class Shape {  
    abstract area(): number;  
}  
class Square extends Shape {  
    constructor(public sideLength: number) {  
        super();  
    }  
    area() {  
        return this.sideLength *  
this.sideLength;  
    }  
}
```

les interfaces en typescript peuvent être implémentées avec une classe. La classe implementant l'interface doit être strictement conforme à la structure de l'interface.

Une classe peut implémenter une ou plusieurs interfaces.

```
interface PersonInterface {  
    id: number  
    name: string  
    register(): string  
}
```

```
class Person implements PersonInterface {  
    id: number  
    name: string  
  
    constructor(id: number, name: string) {  
        this.id = id  
        this.name = name  
    }  
  
    register() {  
        return `${this.name} is now registered`  
    }  
}  
  
const ahmed = new Person(1, 'Ahmed')  
const sara= new Person(2, 'Sara')
```

Interface étend classe

En TypeScript, une interface peut étendre une classe, héritant ainsi des membres de cette classe. Cela est souvent utilisé pour exprimer que l'interface adoptera la forme de la classe mais sans sa mise en œuvre. C'est un moyen d'adopter la "forme" d'une classe tout en permettant à d'autres classes d'implémenter cette interface.

Lorsqu'une interface étend une classe, elle hérite des membres publics et protégés de cette classe, mais pas de ses implémentations.

```
class Point {  
    x: number;  
    y: number;  
}  
  
interface ThreeDPoint extends Point  
{  
    z: number;  
}  
  
class DetailedPoint implements ThreeDPoint {  
    x: number;  
    y: number;  
    z: number;  
    metadata: string;  
}
```

```
let dp: DetailedPoint = {  
    x: 1,  
    y: 2,  
    z: 3,  
    metadata: "Some data"  
};
```

Il est important de noter que lorsqu'une interface étend une classe avec des membres privés ou protégés, cette interface ne peut être implémentée que par cette classe ou ses sous-classes.

static

58

Les membres statiques d'une classe sont accessibles à l'aide du nom de la classe et de la notation par points, sans créer d'objet, par exemple

Les membres statiques peuvent être définis à l'aide du mot-clé **static**

```
class circle{
    static PI:number=3.14;

    static calculSurface(r:number):number{
        return PI*Math.pow(r,2)
    }
}
console.log(circle.PI)//3.14
console.log(circle.calculSurface(5)) //78.5
```

generics

59

Les génériques (ou "generics" en anglais) sont un outil puissant en TypeScript qui permet aux développeurs de créer des composants réutilisables tout en conservant la sécurité du typage. Les génériques sont largement utilisés dans les langages de programmation typés statiquement tels que C# et Java, et TypeScript les introduit dans le monde de JavaScript.

Supposons que vous vouliez créer une fonction qui renvoie le même type qu'elle reçoit. Sans génériques, vous pourriez être tenté de faire quelque chose comme ceci :

```
function identity(arg: any): any {  
    return arg;  
}
```

Cette fonction utilise le type `any`, ce qui signifie qu'elle perd la sécurité du typage spécifique. Avec les génériques, vous pouvez capturer le type que l'utilisateur fournit :

```
function identity<T>(arg: T): T {  
    return arg;  
}
```

Ici, `T` est un type de paramètre. Vous pouvez l'utiliser comme une variable pour le type. Ainsi, la fonction `identity` peut maintenant être appelée de l'une des manières suivantes :

```
let output1 = identity<string>("myString");  
let output2 = identity("myString"); //
```

generics

60

Utilisation avec des tableaux:

Les génériques peuvent également travailler avec des tableaux :

```
function loggingIdentity<T>(arg: T[]): T[] {  
    console.log(arg.length);  
    return arg;  
}
```

Ou en utilisant la notation de type générique d'array :

```
function loggingIdentity<T>(arg: Array<T>):  
Array<T> {  
    console.log(arg.length);  
    return arg;  
}
```

```
function mergeObjects<T,U>(obj1:T,obj2:U):T & U{  
return {...obj1,...obj2};  
}
```

Classes génériques:

Vous pouvez également avoir des classes génériques :

```
class GenericNumber<T> {  
    zeroValue: T;  
    add: (x: T, y: T) => T;  
}  
  
let myGenericNumber = new  
GenericNumber<number>();  
myGenericNumber.zeroValue = 0;  
myGenericNumber.add = function(x, y) { return x  
+ y; };
```

generics

61

Contraintes génériques:

Parfois, vous voudrez travailler avec une partie des capacités du type, mais vous ne saurez pas nécessairement à l'avance ce que sera ce type. Pour cela, vous pouvez définir une interface comme contrainte pour le type générique :

```
interface Lengthwise {  
    length: number;  
}  
  
function loggingIdentity<T extends Lengthwise>(arg: T): T {  
    console.log(arg.length);  
    return arg;  
}
```

Avec cette fonction, arg doit avoir une propriété length.

Génériques avec des interfaces et des types:

Les génériques peuvent également être utilisés avec des interfaces et des alias de type pour définir des types de manière flexible :

```
interface GenericContainer<T> {  
    value: T;  
    retrieve: () => T;  
}  
  
type AnotherContainer<U> = {  
    data: U;  
    extract: () => U;  
};
```

Type Guards

62

Les "Type Guards" en TypeScript sont des expressions qui permettent de vérifier le type d'une variable au moment de l'exécution. Ces vérifications de type influencent la façon dont le compilateur TypeScript traite ces variables dans le bloc de code où la vérification est effectuée. Cela rend le code plus sûr, en s'assurant que les opérations effectuées sur les variables sont appropriées pour leur type réel.

Il existe plusieurs façons de créer et d'utiliser des Type Guards en TypeScript :

Utilisation de `typeof`

Le Type Guard le plus simple est `typeof`, que vous utilisez probablement déjà. TypeScript comprend les vérifications `typeof` et ajuste le type dans ces blocs de code.

```
function padLeft(padding: number | string, input: string): string {
  if (typeof padding === "number") {
    return new Array(padding + 1).join(" ") + input;
  }
  return padding + input;
}
```

Dans cet exemple, TypeScript sait que `padding` est un `number` dans le premier bloc `if`.

Type Guards

63

Utilisation de instanceof

Un autre Type Guard courant est `instanceof`, qui est utilisé pour vérifier si un objet est une instance d'une classe particulière.

```
class Bird {  
    fly() {  
        console.log("Flying");  
    }  
}  
class Fish {  
    swim() {  
        console.log("Swimming");  
    }  
}  
function move(animal: Bird | Fish) {  
    if (animal instanceof Bird) {  
        animal.fly();  
    } else {  
        animal.swim();  
    }  
}
```

```
let myBird = new Bird();  
move(myBird); // Flying
```

Dans cet exemple, TypeScript comprend qu'à l'intérieur du bloc `if`, `animal` doit être un `Bird`.

Type Guards

Création de User-Defined Type Guards

Vous pouvez également définir vos propres Type Guards.

Ce sont des fonctions qui renvoient un type de prédicat, comme `arg is T`.

```
interface Cat {  
    meow: number;  
}  
  
interface Dog {  
    bark: number;  
}  
  
function isCat(pet: Cat | Dog): pet is Cat {  
    return (pet as Cat).meow !== undefined;  
}  
  
function makeNoise(pet: Cat | Dog): string {  
    if (isCat(pet)) {  
        return 'Meow! '.repeat(pet.meow);  
    } else {  
        return 'Bark! '.repeat(pet.bark);  
    }  
}
```

Dans cet exemple, `isCat` est un Type Guard qui vérifie si `pet` est un `Cat`.

À l'intérieur du bloc `if` où `isCat(pet)` renvoie `true`, TypeScript sait que `pet` doit être un `Cat`.

Les Type Guards sont un outil puissant pour gérer les types d'union et les scénarios où le type d'une variable peut être incertain. Ils permettent de s'assurer que le code est sûr et fonctionne

EP07:

65

- A. Définissez une interface **Voiture** avec des propriétés pour **marque**, **modèle**, et **année**.
- B. créez une classe **Garage** qui contient une liste de voitures .
- C. implémentez une méthode **addCar** pour ajouter une voiture au garage,
- D. Implémentez une méthode **removeCar** pour retirer une voiture du garage.
- E. Implémentez une méthode **getDataCar** pour récupérer une voiture par marque et modèle ,année.

- F. Créez une classe Personne avec les attributs id,name,age.
- G. Créer le constructeur
- H. Créer une classe Personnes qui contient une liste des personnes.
- I. Créer une méthode **addPersonne** qui ajoute une personne à la liste personnes
- J. Créer une méthode **getPersonne** qui trouve une personne par son id
- K. Créer une méthode **updatePersonne** qui mis à jour les données d'une Personne .
- L. Créer une méthode **deletePersonne** qui supprime une personne de la liste personnes
- M. Créer une méthode **listePersonnes** qui retourne la liste des personnes.
- N. Créer 5 Personnes
- O. Afficher la liste des Personnes
- P. Trouver la Personne avec id=2
- Q. Supprimer La Personne avec id=4
- R. Modifier la Personne avec l'id=5.

EP08:

66

- A. Créez une fonction générique **getArrayLength** qui prend un tableau de n'importe quel type et retourne sa longueur, assurez-vous que la fonction soit utilisable avec des tableaux de n'importe quel type de données.
- B. Écrivez une fonction générique qui prend un tableau de n'importe quel type et un type en chaîne de caractères (comme "string" ou "number"), la fonction doit renvoyer un nouveau tableau contenant uniquement les éléments du type spécifié.
- C. Écrivez une fonction générique qui prend deux tableaux et renvoie un nouveau tableau qui est l'union des deux tableaux sans doublons.
- D. Écrivez une fonction générique qui prend deux tableaux et renvoie un tableau contenant les éléments qui sont à la fois dans le premier et dans le second tableau.
- E. Écrivez une fonction G qui prend deux tableaux et renvoie la différence symétrique des deux tableaux (éléments qui sont dans un des deux tableaux, mais pas dans les deux).

Promesse/async/await

Une **promesse** est un objet qui représente l'achèvement ou l'échec éventuel d'une opération asynchrone et sa valeur résultante.

async et **await** sont des mots-clés introduits pour simplifier le travail avec les promesses.

- Fonctions Async** : Une fonction déclarée avec le mot-clé **async** renvoie toujours une promesse.

- Await** : Le mot-clé **await** est utilisé pour attendre la résolution d'une promesse à l'intérieur d'une fonction **async**. Il suspend l'exécution de la fonction jusqu'à ce que la promesse soit résolue ou rejetée.

```
const myPromise = new
Promise<string>((resolve, reject) => {
    // Opération asynchrone
    setTimeout(() => {
        resolve("Data received");
    }, 2000);
});
```

myPromise

```
.then(data => {
    console.log(data);
})
.catch(error => {
    console.error(error);
});
```

```
async function myAsyncFunction(): Promise<string> {
    return "Hello, World!";
}

async function fetchData(): Promise<void> {
    try {
        const data = await myPromise; // Attendre la résolution de la promesse
        console.log(data);
    } catch (error) {
        console.error(error); // Gérer l'erreur si la promesse est rejetée
    }
}
```

fetchData();

Promesse/async/await

68

Voici un exemple qui combine l'utilisation de **promesses** et **async/await** pour réaliser une opération asynchrone :

```
function getData(): Promise<string> {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Données récupérées avec succès");
            // reject("Failed to fetch data");
        }, 1000);
    });
}

async function process1() {
    try {
        const data = await getData(); // Attendre la promesse
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}

process1()
```

Type unknown

Le type `unknown` en TypeScript est un type qui représente n'importe quelle valeur, mais, contrairement à `any`, il est beaucoup plus sûr car il ne permet pas de faire des opérations arbitraires sur ses valeurs sans d'abord s'assurer ou affiner leur type.

Vous ne pouvez pas effectuer d'opérations arbitraires sur une valeur de type `unknown`. Vous devez d'abord vérifier le type de la valeur avant de procéder à des opérations spécifiques.

Tout type en TypeScript peut être assigné à `unknown`.

Mais `unknown` ne peut être assigné directement à un autre type (sauf `any` et `unknown`).

Idéal pour représenter des données provenant de sources externes où le type exact n'est pas connu à l'avance.

```
let value: unknown;  
  
value = "Hello"; // Valide  
value = 42; // Valide  
value = true; // Valide
```

Vérification du type avant l'utilisation :

```
let value: unknown = "Hello World";  
  
// Vous ne pouvez pas directement utiliser une méthode spécifique à un type sur une variable de type `unknown` sans vérifier son type.  
if (typeof value === "string") {  
    console.log(value.toUpperCase()); // Valide après vérification  
}
```

Type unknown

70

Si une fonction accepte un argument de type `unknown`, vous devez vérifier le type de cet argument avant de le manipuler.

```
function process(value: unknown) {  
    if (typeof value === "number") {  
        return value * 2;  
    }  
    return value;  
}
```

Le `type unknown` est particulièrement utile dans les scénarios où vous voulez maintenir un haut niveau de typage sans sacrifier la sécurité. Il vous oblige à effectuer des vérifications explicites, ce qui peut aider à prévenir les erreurs à l'exécution.

L'utilisation du type `unknown` avec `fetch` en TypeScript est un excellent moyen d'aborder la gestion de réponses HTTP dont le type peut ne pas être connu à l'avance. Cela vous oblige à valider et à affiner le type de la réponse avant de l'utiliser, ce qui est une pratique courante dans la programmation défensive.

Voici un exemple montrant comment vous pourriez utiliser `unknown` avec `fetch` pour récupérer des données depuis une API :

```
async function fetchData(url: string): Promise<unknown> {
  const response = await fetch(url);
  if (!response.ok) {
    throw new Error('Network response was not ok');
  }
  return response.json();
}
```

Type unknown

```
async function handleData(url: string) {
  try {
    const data: unknown = await fetchData(url);

    if (typeof data === 'object' && data !== null &&
      'id' in data && 'name' in data) {
      console.log(`User ID: ${data as { id: number }}.id`);
      console.log(`User Name: ${data as { name: string }}.name`);
    } else {
      console.log("Invalid data format");
    }
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}
```

```
const url = "https://jsonplaceholder.typicode.com/users/1";
handleData(url);
```

EP09:

72

- A. Écrivez une fonction asynchrone **fetchData(url)** en TypeScript qui utilise **fetch** pour récupérer des données depuis une API et retourne ces données sous forme d'un objet JSON.
- B. Écrivez une fonction asynchrone **getData(url)** qui utilise **axios** pour faire une requête GET à une API. Gérez les erreurs potentielles et retournez une structure de données spécifique en cas d'erreur.
- C. Créez une fonction asynchrone **fetchAndProcessData(url): qui retourne un résultat de type interface pour une personne(id,name..)** il utilise **fetch** pour récupérer des données.utilisez le type **unknown** pour les données initialement récupérées et effectuez une assertion de type pour les convertir en un type spécifique.

Décorateurs

73

Un décorateur est une sorte de déclaration qui peut être attachée à une classe, une méthode, un accesseur, une propriété, ou un paramètre. Les décorateurs utilisent la forme `@expression`, où expression doit évaluer une fonction qui sera appelée au moment de l'exécution avec des informations sur la déclaration décorée.

Les décorateurs sont un concept avancé et sont largement utilisés dans des frameworks comme [Angular](#) pour la définition de composants, services, etc. Ils offrent un moyen puissant de gérer la métaprogrammation et de personnaliser le comportement des classes et de leurs membres.

Activation des Décorateurs

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

```
{  
  "compilerOptions": {  
    "target": "ES5",  
    "experimentalDecorators": true  
  }  
}
```

Décorateurs

74

Décorateurs de Classe:

Ils sont appliqués à la déclaration d'une classe. Le décorateur de classe reçoit le constructeur de la classe comme argument.

```
function MyDecorator(constructor: Function)
{
    console.log("Decorator called");
}
@MyDecorator
class MyClass { }
```

Décorateurs de Méthode:

Ils sont appliqués à des méthodes d'une classe. Le décorateur de méthode reçoit trois arguments : le prototype de la classe, le nom de la méthode, et la description de la propriété de la méthode.

```
function MethodDecorator(target: Object,
propertyKey: string, descriptor: PropertyDescriptor) {
    console.log("MethodDecorator called");
}
class MyClass {
    @MethodDecorator
    myMethod() {}
```

Décorateurs d'Accesseur: Ils sont similaires aux décorateurs de méthode, mais appliqués aux accessoires get et set.

Décorateurs de Propriété: Ils peuvent être utilisés pour surveiller, modifier ou remplacer les définitions de propriétés.

Décorateurs de Paramètre: Ils sont appliqués à un paramètre d'une méthode et reçoivent le prototype de la classe, le nom de la méthode, et l'indice du paramètre dans la liste des arguments.

Décorateurs(Exemple)

75

```
function Log(target: any, propertyName: string) {  
    console.log("Property decorator!");  
    console.log(target, propertyName);  
}  
  
class Product {  
    @Log  
    title: string;  
  
    constructor(t: string) {  
        this.title = t;  
    }  
}
```

Dans cet exemple, `@Log` est un décorateur de propriété appliqué à la propriété `title` de la classe `Product`. Chaque fois que la classe `Product` est utilisée, le décorateur sera appelé.

```
const product:Product = new Product("Book");  
console.log(product.title);
```

Espace de noms

En TypeScript, les espaces de noms (anciennement appelés "modules internes") sont un moyen d'organiser votre code en regroupant logiquement les fonctions, classes, interfaces et types associés. Ils permettent de structurer le code de manière à éviter les conflits de noms dans les grands projets et d'offrir une meilleure lisibilité et une organisation claire.

Pour définir un espace de noms, utilisez le mot-clé `namespace` suivi du nom que vous souhaitez donner à l'espace de noms :

```
namespace MyNamespace {  
    export class MyClass {  
        constructor(public message: string) {}  
    }  
    export function myFunction() {  
        console.log("Function inside namespace");  
    }  
}
```

Notez l'utilisation du mot-clé `export` devant la classe `MyClass` et la fonction `myFunction`. Cela les rend accessibles en dehors de l'espace de noms.

Pour utiliser les éléments d'un espace de noms, vous devez les préfixer par le nom de l'espace de noms :

```
let instance = new MyNamespace.MyClass("Hello  
from namespace!");  
MyNamespace.myFunction();
```

Espace de noms

Référencer des fichiers externes:

Si votre espace de noms est réparti sur plusieurs fichiers, vous pouvez utiliser la directive

`/// <reference ... />` pour indiquer les dépendances :

```
// Dans le fichier `MyNamespacePart1.ts`  
namespace MyNamespace {  
    export function functionPart1() {  
        ...  
    }  
  
// Dans le fichier `MyNamespacePart2.ts`  
/// <reference path="MyNamespacePart1.ts" />  
namespace MyNamespace {  
    export function functionPart2() {  
        functionPart1();  
        ...  
    }  
}
```

Espace de noms

Espaces de noms imbriqués:

Vous pouvez également avoir des espaces de noms imbriqués pour une organisation plus détaillée :

```
namespace ParentNamespace {  
    export namespace ChildNamespace {  
        export function myFunction() {  
            console.log("Function in child  
namespace");  
        }  
    }  
}  
ParentNamespace.ChildNamespace.myFunction()
```

Aliases pour les espaces de noms

Si vous trouvez que vous devez écrire de longs noms d'espaces de noms, vous pouvez créer un alias pour simplifier leur utilisation :

```
namespace VeryLongNamespaceName {  
    export function logMessage() {  
        console.log("Inside a very long  
namespace name");  
    }  
}  
  
import shortName = VeryLongNamespaceName;  
shortName.logMessage();
```

Bien que les espaces de noms soient toujours une caractéristique valide de TypeScript, avec l'avènement des modules ES6 et la capacité de TypeScript à travailler avec eux, l'utilisation de modules est souvent recommandée pour organiser et scoper le code dans les applications modernes. Ils offrent une encapsulation et une organisation plus naturelle et sont plus adaptés à l'architecture et aux outils de construction modernes.

Modules

79

Les modules en TypeScript sont étroitement alignés sur les modules ECMAScript 2015 (ou ES6) et sont une manière d'organiser et d'encapsuler du code.

Contrairement aux espaces de noms, qui sont spécifiques à TypeScript, les modules sont une fonctionnalité standard de JavaScript moderne.

Avec les modules, vous **export** des éléments que vous souhaitez rendre disponibles à l'extérieur du module, et vous **import** des éléments provenant d'autres modules.

math.ts

```
export function add(x: number, y: number):  
number {  
    return x + y;  
}  
  
export function subtract(x: number, y: number):  
number {  
    return x - y;  
}
```

app.ts

```
import { add, subtract } from './math';  
console.log(add(5, 3));          // 8  
console.log(subtract(5, 3));     // 2
```

Modules

80

Exportation par défaut:

Chaque module peut avoir un export par défaut. C'est utile pour, par exemple, exporter une seule classe ou fonction.

Renommage lors de l'importation

Si vous voulez renommer une fonction, une classe ou une variable lors de l'importation, vous pouvez le faire avec `as`

`greeting.ts`

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

`app.ts`

```
import greet from './greeting';
console.log(greet("TypeScript"));
// "Hello, TypeScript!"
```

```
import { add as addition } from './math';
```

```
console.log(addition(5, 3)); // 8
```

Importer tout

Si vous souhaitez importer tous les exports d'un module, vous pouvez le faire avec `*`.

```
import * as MathFunctions from './math';
```

```
console.log(MathFunctions.add(5, 3)); // 8
```

```
console.log(MathFunctions.subtract(5, 3)); // 2
```

Vite

mohamed@goumih.com

81

Vite est un outil de **build moderne** pour les projets web qui offre une expérience de développement plus rapide ,plus performant et plus efficace. Voici quelques caractéristiques clés et avantages de Vite :

- 1.Démarrage Rapide du Serveur de Développement :**
- 2.Rechargement à Chaud (Hot Module Replacement, HMR):**
- 3.Construction Optimisée : Pour la production**
- 4.Prise en Charge des Frameworks Modernes :**
- 5.Support du TypeScript :**
- 6.Plugin Écosystème :**
- 7.Optimisation des Dépendances :**
- 8.Facile à Configurer :**



Vite + TypeScript

Vite :installation

Installer Vite : Ouvrez votre terminal ou invite de commande et exécutez la commande suivante pour installer Vite globalement sur votre système

```
npm install -g create-vite
```

Créer un Nouveau Projet avec Vite :

Pour créer un nouveau projet TypeScript avec Vite, utilisez la commande suivante dans votre terminal, en remplaçant **mon-projet** par le nom que vous souhaitez donner à votre projet :

```
create-vite mon-projet --template  
typescript
```

Navigation vers le Répertoire du Projet :

Changez de répertoire pour aller dans le dossier de votre nouveau projet :

```
cd mon-projet
```

Installation des Dépendances :

Une fois dans le dossier de votre projet, installez les dépendances nécessaires en exécutant :

```
npm install
```

Lancement du Serveur de Développement :

Pour démarrer le serveur de développement et voir votre application, exécutez :

```
npm run dev
```

Vite: démarrage

83

le serveur de développement local s'ouvre ouvrira généralement par défaut à l'adresse `http://localhost:5173/`.

Commencer à Développer :

Maintenant, vous pouvez commencer à développer votre application en TypeScript avec Vite.

Le serveur de développement prend en charge le rechargement à chaud, donc toute modification que vous apportez au code sera immédiatement reflétée dans le navigateur.

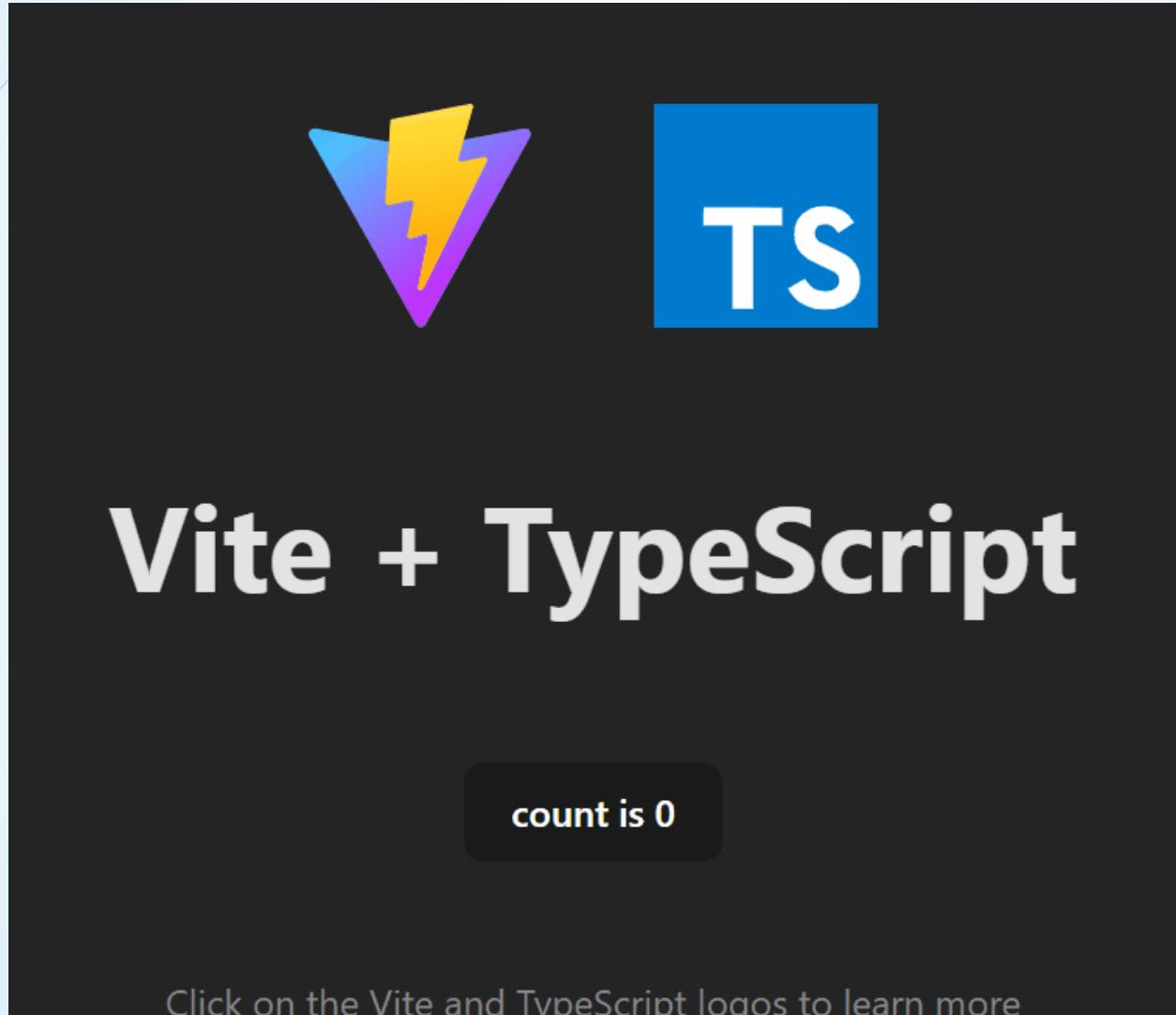
Construire et Déployer :

Une fois que vous êtes prêt à déployer votre application, vous pouvez construire une version de production en utilisant :

```
npm run build
```

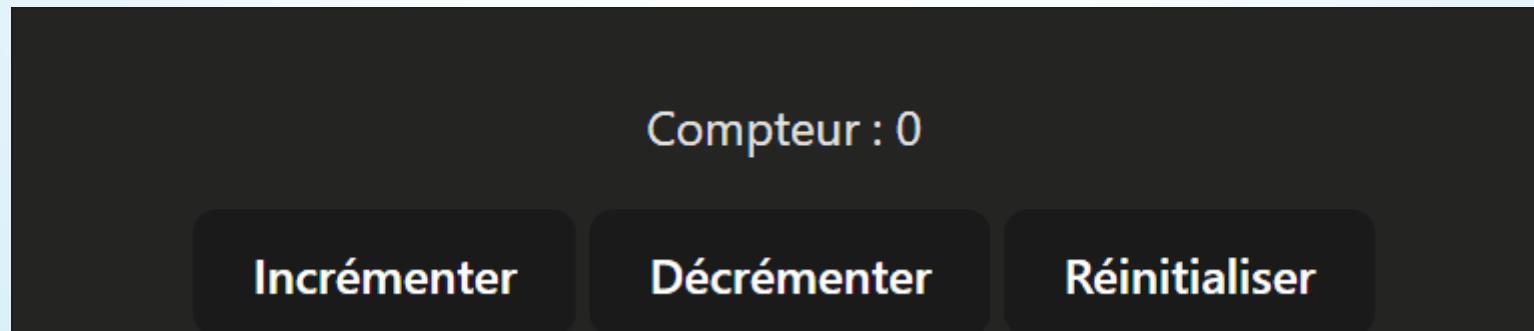
Cela générera des fichiers statiques dans le dossier `dist` que vous pourrez déployer sur n'importe quel serveur web.

Vite+TypeScript



EP10:

1. Modifier le project de départ vite+TypeScript pour créer une autre fonction sur un autre button pour décrémenter le compteur .
2. Modifier les codes pour créer 3 buttons pour appliquer sur un seul div incrémentation ,décrémentation et réinstallation à 0 :
 - Créer une fonction **updateDisplay** qui modifier la valeur du compteur .
 - Créer une fonction **increment** qui incremente le compteur .
 - Créer une fonction **decrement** qui décremente le compteur .
 - Créer une fonction **reset** qui initialise le compteur à 0 .



TP1:liste des taches

- On veut créer un projet avec **typescript** pour créer des listes des **Taches** en utilisant VITE:

Index.html

```
<H1>Liste des taches:</H1>
<ul id="list">
  <form action="" id="task-form">
    <input type="text" placeholder="ajouter une tache" id="task-input">
    <button type="submit">Ajouter</button>
  </form>
</ul>
<script type="module" src="/src/main.ts"></script>
</body>
```

Style.css

```
body{
  display: grid;
  place-items: center;
}
#list{
  list-style-type: none;
  padding: 0;
}
```

Liste des taches:

tache3	Ajouter
<input checked="" type="checkbox"/> tache1	
<input checked="" type="checkbox"/> tache2	
<input type="checkbox"/> tache3	

- Dans **main.ts** récupérer les éléments du form et liste .
- Créer une interface pour définir un type **Tache(id,titre,etat,dateCreation)**
 - Créer une fonction **ajouterTache(tache)** qui permet d'ajouter une tache à un div avec **checkbox** pour définir l'état du tache.
 - Installer et importer le module v4 de **uuid** qui permet de générer un id aléatoire.
 - Ajouter une ligne à l'intérieur lorsque le **checkbox** est coché pour indiquer que la tache est finie et vider l lorsque une tache est ajoutée.

TP1:liste des taches

main.ts

1,2,3

```
import './style.css'
import {v4} from "uuid";

//Récuperation des éléments:
const list=document.querySelector<HTMLULListElement>("#list");
const form=document.querySelector<HTMLFormElement>("#task-form");
const input=document.querySelector<HTMLInputElement>("#task-input");

//interface type tache
interface Tache{
    id:String
    titre:String,
    etat:boolean,
    dateCreation:Date,
}

//Ajouts des evenements
form?.addEventListener("submit",e=>{
    e.preventDefault();
    if (input?.value=="" || input?.value==null)
        return ;
    // @ts-ignore
    const newTache:Tache={
        id:v4(),
        titre:input.value,
        etat:false,
        dateCreation:new Date()
    }
    ajouterTache(newTache);
})
```

```
//fonction pour ajouter une tache
const ajouterTache=(tache:Tache):void=>{
    const
    item=document.createElement("li");
    const
    label=document.createElement("label");
    const
    checkbox=document.createElement("input")
    ;
    checkbox.type="checkbox";
    // @ts-ignore
    label.append(checkbox,tache.titre);
    item.append(label);
    list?.append(item);
}
```

TP1:liste des taches

main.ts

4

```
ajouterTache(newTache);
//4
input.value=""
```

Liste des taches:

ajouter une tache

Ajouter

```
checkbox.type="checkbox";
//4
checkbox.addEventListener("change", ()=>{
    if (checkbox.checked)
    {
        Label.style.textDecoration="line-through";
        Label.style.color="green";
    }
    else{
        Label.style.textDecoration="none";
        Label.style.color="red";
    }
checkbox.checked=tache.etat;
```

- TAF :

5.Ajouter des méthodes pour supprimer et modifier une tache ,et aussi l'implementation pour sauvegarder les taches dans localStorage.

TP2:Drag-Drop project

- Réaliser l'application suivante :

Nom du module	<input type="text"/>
Description	<input type="text"/>
Nombre de Stagiaire	<input type="text"/> 
Ajouter	<input type="button" value="Ajouter"/>

ACTIVE PROJECTS	
<input type="text"/>	<input type="text"/>

FINISHED PROJECTS	
<input type="text"/>	<input type="text"/>

TP3:React avec TypeScript

Nous allons créer une simple application de **liste de evenements** avec React et TypeScript. Alors, ouvrez votre terminal et exécutez la commande ci-dessous. Remarquez que nous devons spécifier **--template typescript** pour créer un projet ReactJS avec TypeScript."

Création d'un Nouveau Projet avec Vite :

Exécutez la commande suivante pour créer un nouveau projet **React avec TypeScript** :

```
npm create vite@latest mon-projet-react -- --template react-ts
```

Remplacez **mon-projet-react** par le nom de votre choix pour le dossier du projet.

Pour notre projet :

```
npm create vite@latest evenement-app -- --template react-ts
```

```
cd evenement-app
npm install
npm run dev
# pour déployer votre application
npm run build
```

TP3:React avec TypeScript

91

La seule différence avec un composant ReactJS normal en JavaScript, c'est que nous spécifions le type. Ici, nous définissons que les personnes sont un tableau d'objets, contenant le nom, l'âge, l'image et la fonction.

L'objectif est de créer la vue suivante

Invités d'un evenement		
 Ali	30 ans	Medecin
 Sara	32 ans	Professeur
 Ahmed	27 ans	Ingenieur

TP3:React avec TypeScript

92

1.Dans **src** ,créer un fichier **types.tsx** pour définir le type personne.

```
ts types.ts x
1 export type personne = {
2   name: string;
3   age: number;
4   img: string;
5   fonction: string;
6 };
```

2.Dans **src** ,créer un fichier **ListesPersonnes.tsx** qui définit une interface qui contient un tableau des personnes puis en créer le composant **Invités** pour boucler sur les éléments du tableau et l'afficher dans une liste.

```
import React from 'react';
import { personne } from './types';

interface IProps {
  people: personne[];
}

const Invites: React.FC<IProps> = ({ people }) => {
  return (
    <ul>
      {people.map(p => (
        <li className="list" key={p.name}>
          <div className="list-header">
            <img className="list-img" src={p.img} alt={p.name}/>
            <h2>{p.name}</h2>
          </div>
          <p>{p.age} ans</p>
          <p className="list-fonction">{p.fonction}</p>
        </li>
      ))}
    </ul>
  );
}

export default Invites;
```

TP3:React avec TypeScript

93

3.Dans **App.tsx** on fait appel au composant **Invites** et on définir une liste des personnes et on la passe à la props **people**

```
import './App.css';
import Invites from './ListePersonnes';
import { personne } from './types';
const listePersonnes: personne[] = [
  {
    name:"Ali",
    age:30,
    img:"https://randomuser.me/api/portraits/men/75.jpg",
    fonction:"Medecin"
  },
  {
    name:"Sara",
    age:32,
    img:"https://randomuser.me/api/portraits/women/10.jpg",
    fonction:"Professeur"
  },
  {
    name:"Ahmed",
```

```
    age:27,
    img:"https://randomuser.me/api/portraits/men/20.jpg",
    fonction:"Ingenieur"
  }
];
```

3 usages

```
function App(): JSX.Element {
  return (
    <>
      <h1>Invités d'un evenement</h1>
      <div className="App">
        <Invites people={listePersonnes}/>
      </div>
    </>
  );
}

export default App;
```

TP3:React avec TypeScript

94

4.Pour rendre l'affichage plus bonne en utilise un fichier css :App.css

```
App{  
  text-align: center;  
}  
.list{  
  list-style: none;  
  display: flex;  
  align-items: center;  
  width: 50rem;  
  margin: 0px auto;  
  border: 1px solid rgba(0,0,0,0.233);  
  padding: 1rem;  
  justify-content: space-between;  
}
```

```
.list-header{  
  display: flex;  
  align-items: center;  
}  
.list-header h2{color: white;}  
.list-img{  
  width: 4rem;  
  height: 4rem;  
  border-radius: 100%;  
  margin-right: 0.5rem;  
}  
.liste-fonction{  
  width: 30%;  
  text-align: left;  
}  
.list:hover{  
  background: black;  
}
```

TP3:React avec TypeScript

95

5. Nous allons ensuite créer un composant **AddPersonne.tsx**. Ici, nous avons 4 champs de saisie pour recueillir les informations sur le nom, l'âge ,l'image et fonction et nous avons un bouton pour soumettre ces éléments. Dans la fonction **handleChange**, nous utilisons l'astuce commune de React consistant à utiliser **e.target.name** pour avoir une fonction commune.

La chose principale à remarquer ici est le type de **e**, qui est communément connu comme événement. Ici, nous devons le considérer comme un **React.ChangeEvent**, mais il peut avoir deux valeurs. Ce sont **HTMLInputElement** ."

```
import React, { useState } from "react";
import { personne } from "./types";
interface IProps {
  people: personne[],
  setPeople: React.Dispatch<React.SetStateAction<personne[]>>
}

export const AddPersonne: React.FC<IProps> = ({ setPeople, people }) => {
  const [input, setInput] = useState<personne>({ name: "", age: 0, img: "", fonction: "" });

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setInput({
      ...input,
      [e.target.name]: e.target.name === "age" ? Number(e.target.value) : e.target.value
    });
  };
}
```

AddPersonne.tsx

TP3:React avec TypeScript

96

AddPersonne.tsx

```
return (
  <div className="add-people">
    <input type="text" onChange={handleChange} className="add-input" name="name"
      value={input.name} placeholder="nom"/>
    <input type="number" onchange={handleChange} className="add-input" name="age"
      value={input.age} placeholder="age"/>
    <input type="text" onChange={handleChange} className="add-input" name="img"
      value={input.img} placeholder="url"/>
    <input type="text" onChange={handleChange} className="add-input" name="fonction"
      value={input.fonction} placeholder="fonction"/>
    <button onClick={handleClick} className="add-button">Ajouter</button>
  </div>
);
```

App.tsx

On ajoute un useState pour modifier la liste des personnes lorsque on ajoute une nouvelle personne

```
const [people, setPeople] = useState<personne[]>(listePersonnes);
```

On importe le composant AddPersonne

```
import { AddPersonne } from "./AddPersonne";
```

On l'insere dans div App;

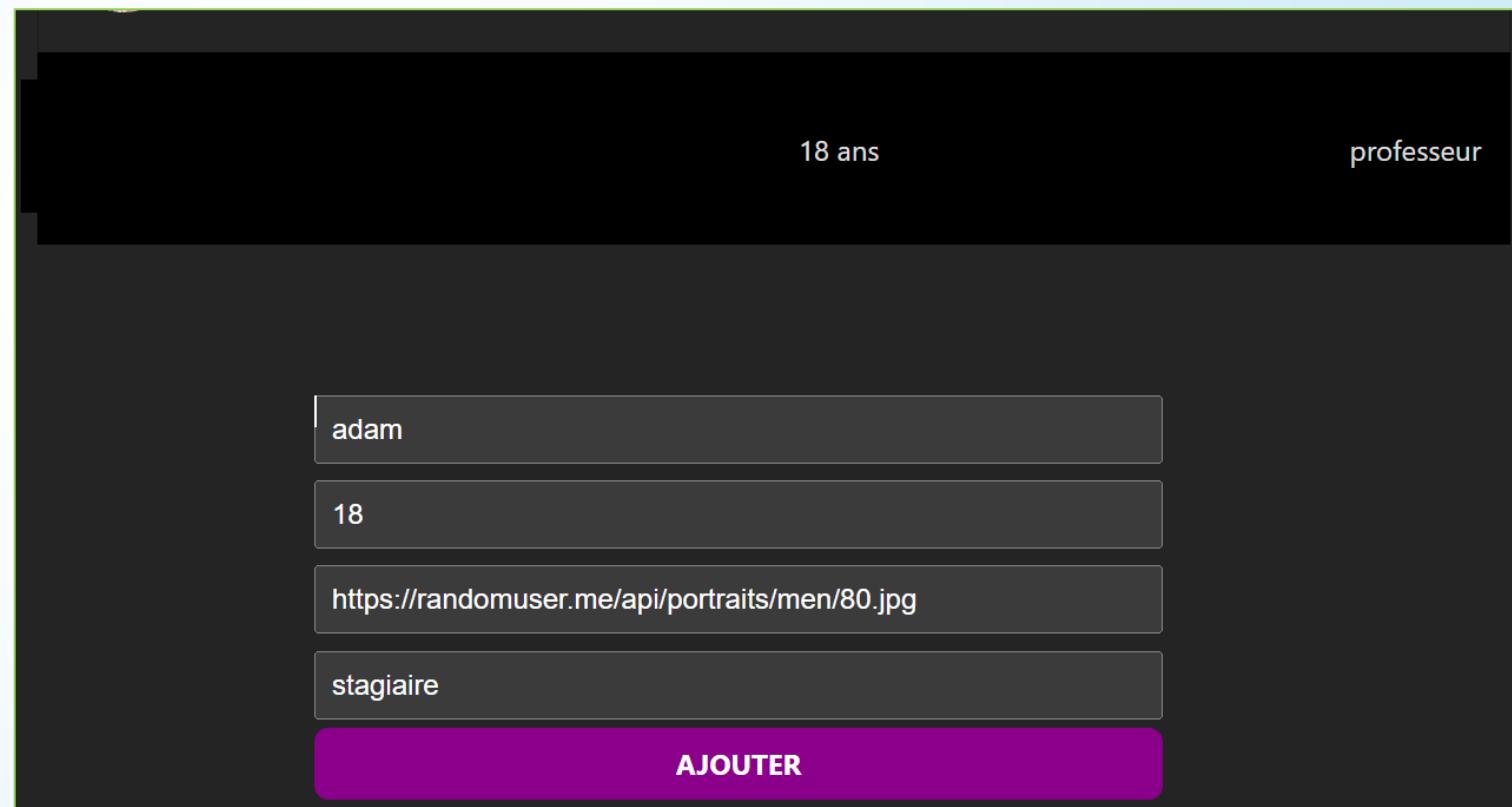
```
<AddPersonne people={people} setPeople={setPeople}/>
```

TP3:React avec TypeScript

97

6. Ajouter la modification dans **App.css** pour rendre le formulaire à une bonne vue :

```
.add-people{  
    display: flex;  
    flex-direction: column;  
    width: 30rem;  
    margin: 5rem auto;  
}  
.add-input{  
    padding: 0.5rem;  
    font-size: 1rem;  
    margin: 0.3rem 0rem;  
}  
.add-button{  
    cursor: pointer;  
    background: darkmagenta;  
    font-weight: 700;  
    color: white;  
    border: none;  
    border-radius: 0.5rem;  
    text-transform: uppercase;  
}
```



TP3:React avec TypeScript

98

Rôle de React.FC

1. Typage du Composant : React.FC définit le type d'un composant fonctionnel en React. Il indique que le composant est une fonction qui peut recevoir des props et retourne un élément React (JSX).

2. Props et Enfants : En utilisant **React.FC**, les props passées au composant sont automatiquement typées avec l'interface ou le type spécifié.

3. De plus, **React.FC** inclut la propriété **children** par défaut, ce qui signifie que vous n'avez pas besoin de définir explicitement **children** dans les types de props si votre composant peut les recevoir.

3. Fonctionnalités Explicites : L'utilisation de **React.FC** rend le code plus explicite et plus facile à comprendre, en particulier pour d'autres développeurs qui peuvent travailler sur le même code.

1. React.Dispatch : C'est un type générique fourni par React pour représenter une fonction de dispatch. En contexte React, "dispatch" fait généralement référence à la fonction fournie par le hook **useState** ou **useReducer** pour mettre à jour l'état.

2. React.SetStateAction : C'est un type générique qui représente les valeurs possibles que vous pouvez passer à la fonction de dispatch. Il peut prendre une valeur directe du nouvel état ou une fonction qui reçoit l'état précédent et renvoie le nouvel état.

3. personne[] : C'est le type de l'état que vous gérez. Dans votre cas, il s'agit d'un tableau d'objets de type **personne**.

Nécessité de la Déclaration en TypeScript

- **Précision du Typage** : En TypeScript, déclarer explicitement le type de la fonction **setPeople** comme

React.Dispatch<React.SetStateAction<personne[]>> garantit que la fonction est utilisée correctement dans le composant. Cela aide à prévenir les bugs en s'assurant que vous ne mettez à jour l'état qu'avec les bonnes valeurs.

- **Autocomplétion et Vérification d'Erreurs** : Avec TypeScript, vous bénéficiez de l'autocomplétion et de la vérification d'erreurs en temps réel. En définissant explicitement le type, l'éditeur de code peut vous avertir si vous essayez de mettre à jour l'état avec des valeurs inappropriées.

TP3:React avec TypeScript

99

7. on veut stocker les données dans **localStorage** pour les conserver

On peut utiliser l'effet de bord **useEffect** pour surveiller les changements l'état **people** et mettre à jour **localStorage** en conséquence.

Dans **App.tsx**, ajoutez ce qui suit :

```
//(listePersonnes);
(
  initialState: ()=>{
    // Récupérer les données de localStorage au démarrage de l'application
    const savedPeople :string | null = localStorage.getItem(key: 'people');
    return savedPeople ? JSON.parse(savedPeople) : listePersonnes;
  }
);
useEffect( effect: () :void => {
  // stocker les données dans localStorage chaque fois que `people` change
  localStorage.setItem('people', JSON.stringify(people));
}, [deps: [people]]);
```

TP3:React avec TypeScript

100

8. On veut ajouter les fonctionnalités de suppression et modification :

Pour intégrer les fonctionnalités de modification et de suppression dans le projet React ,on suit les modifications suivantes.

Modification du composant AddPersonne : Ajoutez une propriété **editingPerson** pour gérer la personne actuellement en cours de modification.

- **Modification de l'état input** : Lorsque l'utilisateur clique sur les informations d'une personne, définissez l'état **input** avec les données de cette personne.

- **Changement du libellé du bouton** : Changez le texte du bouton en fonction de l'état (ajouter ou modifier).

- **Ajout d'une icône de suppression** : Dans le composant qui affiche chaque personne, ajoutez une icône ou un bouton pour la suppression.

- **Gestion de l'événement de suppression** : Lorsque l'icône/bouton est cliqué, supprimez la personne de la liste **people**.

Dans ListePersonnes.tsx :Modifier le composant par **Invites.tsx**

- Transmettez une fonction **onDelete** et **onEdit** depuis **App.tsx** pour gérer les événements.

- **Dans AddPersonne.tsx** Gérez un nouvel état pour déterminer si l'utilisateur est en train d'ajouter ou de modifier une personne,modifiez le texte du bouton et la logique en conséquence.

TP3:React avec TypeScript

101

9.Le code final :addPersonne.tsx

```
import React, { useState, useEffect } from "react";
import { personne } from "./types";

interface IProps {
    people: personne[];
    setPeople: React.Dispatch<React.SetStateAction<personne[]>>;
    editingPerson?: personne;
}
export const AddPersonne: React.FC<IProps> = ({ setPeople, people, editingPerson }) => {
    const [input, setInput] = useState<personne>({ name: "", age: 18, img: "", fonction: "" });

    useEffect(() => {
        if (editingPerson) setInput(editingPerson);
    }, [editingPerson]);

    const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
        setInput({ ...input, [e.target.name]: e.target.name === "age" ? Number(e.target.value) : e.target.value });
    };

    const handleClick = () => {
        if (!input.name || input.age === 0 || !input.img || !input.fonction) return;
        if (editingPerson) {
            setPeople(people.map(p => (p.name === input.name ? input : p)));
        } else {
            setPeople([...people, input]);
        }
        setInput({ name: "", age: 18, img: "", fonction: "" });
    };
}
```

TP3:React avec TypeScript

102

9.Le code final :addPersonne.tsx

```
return (
    <div className="add-people">
        <input type="text" onChange={handleChange} className="add-input"
name="name" value={input.name} placeholder="nom"/>
        <input type="number" onChange={handleChange} className="add-input"
name="age" value={input.age} placeholder="age"/>
        <input type="text" onChange={handleChange} className="add-input"
name="img" value={input.img} placeholder="url"/>
        <input type="text" onChange={handleChange} className="add-input"
name="fonction" value={input.fonction} placeholder="fonction"/>
        <button onClick={handleClick}>{editingPerson ? "Modifier" :
"Ajouter"}</button>
    </div>
);
};
```

TP3:React avec TypeScript

9.Le code final :Invites.tsx

```
import React from 'react';
import { personne } from './types';

interface IProps {
  people: personne[];
  onDelete: (name: string) => void;
  onEdit: (person: personne) => void;
}
```

```
const Invites: React.FC<IProps> = ({ people, onDelete, onEdit }) => {
  return (
    <ul>
      {people.map(p => (
        <li className="list" key={p.name}>
          <div className="list-header" onClick={() => onEdit(p)}>
            <img className="list-img" src={p.img} alt={p.name}/>
            <h2>{p.name}</h2>
          </div>
          <p>{p.age} ans</p>
          <p className="list-fonction">{p.fonction}</p>
          <button onClick={() => onDelete(p.name)}>Supprimer</button>
        </li>
      ))}
    </ul>
  );
}

export default Invites;
```

TP3:React avec TypeScript

104

9.Le code final :App.tsx

```

import { useEffect, useState } from "react";
import './App.css';
import Invites from "./Invites";
import { AddPersonne } from "./AddPersonne";
import { personne } from './types';

const listePersonnes: personne[] = [
    //vous pouvez supprimer les données de listePersonnes
];
function App() {
    const [people, setPeople] = useState<personne[]>(() => {
        const savedPeople = localStorage.getItem('people');
        return savedPeople ? JSON.parse(savedPeople) : listePersonnes;
    });

    const [editingPerson, setEditingPerson] =
    useState<personne | undefined>(undefined);

    useEffect(() => {
        localStorage.setItem('people', JSON.stringify(people));
    }, [people]);

    const handleDelete = (name: string) => {
        const updatedPeople = people.filter(p => p.name !== name);
        setPeople(updatedPeople);
    };

    const handleEdit = (person: personne) => {
        setEditingPerson(person);
    };
}

return (
    <>
        <h1>Invités d'un événement</h1>
        <div className="App">
            <Invites
                people={people}
                onDelete={handleDelete}
                onEdit={handleEdit}>
            </Invites>
            <AddPersonne
                people={people}
                setPeople={setPeople}
                editingPerson={editingPerson}>
            </AddPersonne>
        </div>
    </>
);
}

export default App;

```

TP3:React avec TypeScript

105

9.Résultat final

The screenshot displays a user interface for managing profiles. At the top, there are two cards representing users:

- User 1:** Name: mohamedgoumih, Age: 18 ans, Description: ddsdsds. Includes a "Supprimer" button.
- User 2:** Name: MOHAMED GOUMIH, Age: 18 ans, Description: vcvc. Includes a "Supprimer" button.

Below these cards is a form with four input fields:

- nom: mohamedgoumih
- age: 18
- url: (empty)
- fondation: (empty)

At the bottom of the form is a large blue "Ajouter" button.

TP3:React avec TypeScript

106

10.On peut utiliser d'autres hooks React comme **useRef** et **useReducer** dans le projet, Voici quelques idées sur la manière dont vous pourriez les utiliser dans le contexte l'application de gestion des invités :

useRef est souvent utilisé pour référencer des éléments du DOM ou conserver une valeur mutable qui ne déclenche pas de re-render lorsqu'elle change. Dans votre application, vous pourriez l'utiliser pour :

- **Focaliser un champ de formulaire** : Après la modification d'une personne, vous pouvez utiliser **useRef** pour mettre automatiquement le focus sur le premier champ du formulaire, ce qui améliore l'expérience utilisateur.

- **Conserver une référence à une valeur non modifiée** : Si vous voulez comparer la version actuelle d'une personne avec sa version originale avant modification, vous pouvez utiliser **useRef** pour conserver une référence à la version originale.

useReducer est utile pour gérer des états complexes ou interdépendants. Dans votre application, il pourrait remplacer **useState** pour gérer l'état global des personnes (**people**). Cela est particulièrement utile si la logique de mise à jour de l'état devient plus complexe

```
export const AddPersonne: React.FC<IProps> = ({ setPeople, people, editingPerson }) => {
  const nameInputRef = useRef<HTMLInputElement>(null);

  useEffect(() => {
    if (editingPerson) {
      setInput(editingPerson);
      nameInputRef.current?.focus();
    }
  }, [editingPerson]);

  // ... Reste du composant ...

  return (
    <div className="add-people">
      <input
        ref={nameInputRef}
        type="text"
        onChange={handleChange}
        // ... autres props...
      />
      {/* ... autres champs de saisie... */}
    </div>
  );
};
```

Vous travaillez sur une application web de gestion de tâches en utilisant ReactJS.

L'application contient déjà un certain nombre de composants et utilise le système de routing de ReactJS pour afficher différents composants en fonction de l'URL.

L'application contient déjà un répertoire **Components** :

- **Contact** : contient un simple header (exemple : <h1>Contact</h1>)

- **Home** : contient un simple header (exemple : <h1>Home page</h1>)

- **NoPage** : contient le message : <h1>Page non trouvée</h1>

- **TaskList**: permet d'afficher la liste des tâches avec une case à cocher à côté de chaque tâche.

- **Form** : contient le formulaire d'ajout d'une nouvelle tâche. :

- **Layout** : contient les liens de route vers les autres composants.

1- Donner le code qui permet de déclarer le menu de l'application en utilisant BrowserRouter Spécifier pour chaque fichier le code que vous allez ajouter pour attendre le résultat ci-dessous :

2.Le composant ‘TaskList consomme l’API

<https://jsonplaceholder.typicode.com/todos> qui retourne le json des tâches sous la forme suivante :Donner le code du composant **TaskList**, qui permet d'afficher les tâches dans une simple liste (li) en affichant le titre de la tâche ainsi que sa priorité.

3.Le composant Form permet d'ajouter une nouvelle tâche en spécifiant son titre et sa priorité



TP4:correction

108

App.tsx

```

import { BrowserRouter, Routes, Route, Form } from 'react-router-dom'
import Layout from './Layout'
import Contact from './Contact';
import NoPage from './NoPage';
import TaskList from './TaskList';
import Home from './Home';
const App: React.FC = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />} />
        <Route index element={<Home />} />
        <Route path="/tasks" element={<TaskList />} />
        <Route path="/add-task" element={<Form />} />
        <Route path="/contact" element={<Contact />} />
        <Route path="*" element={<NoPage />} />
      </Routes>
    </BrowserRouter>
  );
}
export default App

```

Layout.tsx

```

import { Outlet, Link } from 'react-router-dom';
const Layout: React.FC = () => {
  return (
    <>
      <nav>
        <ul>
          <li><Link to="/">Home</Link></li>
          <li><Link to="tasks">Tâches</Link></li>
          <li><Link to="add-task">Ajouter une
            tâche</Link></li>
          <li><Link to="contact">Contact</Link></li>
        </ul>
      </nav>
      <Outlet/>
    </>
  )
}
export default Layout;

```

TP4:correction

109

TaskList.tsx

```

import { useState, useEffect } from 'react';
import Layout from './Layout';

// Définir le type pour une tâche individuelle
type Task = {
  id: number;
  title: string;
  cost?: number; // Le '?' indique que 'cost' est optionnel
};
const TaskList: React.FC = () => {
  const [tasks, setTasks] = useState<Task[]>([]); // useState utilise ici le type Task[]
  return (
    <div>
      <Layout/>
      <h1>Liste des tâches</h1>
      <ul>
        {tasks.map(task => (
          <li key={task.id}>
            {task.title} - Coût: {task.cost || 'Non spécifié'}
          </li>
        ))}
      </ul>
    </div>
  );
}
export default TaskList;

```

```

useEffect(() => {
  // La fonction pour charger les tâches depuis l'API
  const fetchTasks = async () => {
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/todos');
      if (!response.ok) {
        throw new Error('Network response was not ok');
      }
      const data = await response.json();
      setTasks(data); // Mettre à jour l'état avec les nouvelles tâches
    } catch (error) {
      console.error('There was a problem with the fetch operation:', error);
    }
  };
  fetchTasks();
}, []); // Le tableau vide signifie que cet effet ne s'exécute qu'une fois, similaire à componentDidMount

```

TP4:correction

110

Form.tsx

```
import { useState, FormEvent } from 'react';
export default function Form() {
  // Définition des états avec TypeScript
  const [titre, setTitre] = useState<string>('');
  const [priority, setPriority] = useState<string>('');

  // Gestionnaire de soumission de formulaire avec TypeScript
  const handleSubmit = (event: FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    // Logique de soumission de formulaire, comme l'envoi des
    données à une API
    console.log('Titre:', titre, 'Priorité:', priority);

    // Réinitialiser les champs du formulaire si nécessaire
    setTitre('');
    setPriority('');
  };
}
```

```
return (
  <div>
    <h1>Ajouter une nouvelle tâche</h1>
    <form onSubmit={handleSubmit}>
      <p>
        <label>Titre : </label>
        <input
          type="text"
          name="titre"
          value={titre}
          onChange={(e) => setTitre(e.target.value)}
          required
        />
      </p>
      <p>
        <label>Priorité : </label>
        <input
          type="number"
          name="priority"
          value={priority}
          onChange={(e) => setPriority(e.target.value)}
          required
        />
      </p>
      <p>
        <input type="submit" value="Ajouter" />
      </p>
    </form>
  </div>
);
```