Introduction à TypeScript

Thèmes	JavaScript TypeScript
Description	Introduction au typage avec TypeScript.
Date de création	@23 juin 2025 14:52
Créée par	Kévin Wolff

Chapitres

Sur le même sujet

Installer Node.js et NVM

Démarrer un projet TypeScript

Introduction

TypeScript est une surcouche de JavaScript développé par Microsoft. Cela signifie que tout code JavaScript est également valide en TypeScript, mais que TypeScript ajoute des fonctionnalités supplémentaires, principalement autour du typage statique.

👉 en résumé : TypeScript = JavaScript + types

- Typage statique : détecte les erreurs à l'écriture avant l'exécution
- Auto-complétion améliorée : grâce à l'inférence de types
- Documentation implicite : les types rendent le code plus lisible
- Interopérable avec JavaScript : migration progressive possible (on peut mixer les deux)

Les différents types

... À écrire

Les types primitifs

boolean

```
let isDone: boolean = true;
```

number

```
let count: number = 42;
```

string

```
let username: string = "Alice";
let greeting: string = Hello, ${username};
```

`null` et `undefined`

```
let empty: null = null;
let notDefined: undefined = undefined;
```

`any` à éviter autant que possible ⚠️

```
let anything: any = "Hello";
anything = 42;
```

`array`

```
let scores: number[] = [1, 2, 3];
let tags: Array<string> = ["typescript", "javascript"];
let scores: Array<number> = [1, 2, 3];
```

Interfaces et types

En TypeScript, `interface` et `type` permettent tous les deux de décrire la forme d'un objet, mais ils ont des usages différents.

Une interface est idéale pour structurer des objets et peut être étendue ou fusionnée facilement, ce qui la rend pratique pour les projets à long terme.

Un type est plus flexible : il permet non seulement de décrire des objets, mais aussi de créer des unions (voir plus bas dans le document), des intersections, ou des alias plus complexes.

En résumé : utilisez `interface` pour les objets, et `type` pour tout le reste ou quand vous avez besoin de plus de souplesse.

Exemple interface :

```
// Une interface décrit la forme d'un objet
interface User {
  id: number;
  name: string;
  email: string;
  isAdmin: boolean;
}

// On peut utiliser l'interface pour typer des variables
const user1: User = {
  id: 1,
  name: "Alan",
  email: "alan@simplon.com"
```

```
  isAdmin: false,
};
```

Exemple type :

```
type Status = "success" | "error" | "loading";

// On peut utiliser le type pour typer des variables
let currentStatus: Status = "success";
```

Tableau des différences

Critère	`interface`	`type`
Définition	Structure d'un objet	Alias pour n'importe quel type
Extension (héritage)	✅ `extends`	✅ avec `&` (intersection)
Fusion déclarative	✅ oui (interfaces se combinent)	❌ non (une seule déclaration possible)
Utilisation avec objets	✅ recommandé	✅ possible
Utilisation avec union/literal	❌ non (limité à des objets)	✅ oui
Génériques	✅ oui	✅ oui

Les paramètres optionnels et valeurs par défaut

Les paramètres optionnels

En TypeScript, on peut rendre un paramètre de fonction optionnel en ajoutant un `?` après son nom. Cela signifie que l'appelant n'est pas obligé de le fournir, et qu'il vaudra `undefined` s'il est omis.

Dans cet exemple, le second paramètre `title` est optionnel :

```
function greet(name: string, title?: string): void {
   if (title) {
      console.log(`${title} ${name}`);
   } else {
      console.log(`Bonjour ${name}`);
   }
}
```

Lorsque j'appel cette fonction vous n'êtes donc pas obligé de satisfaire ce paramètre :

```
greet("Alan");          // "Bonjour Alan"
greet("Alan", "Mr.");   // "Mr. Alan"
```

Les valeurs par défaut

En TypeScript, on peut attribuer une valeur par défaut à un paramètre. Cela le rend automatiquement optionnel, puisqu'il sera utilisé même si aucun argument n'est fourni lors de l'appel de la fonction.

Dans cet exemple, le second paramètre `author` est optionnel est recevra une valeur par défaut :

```
function showMessage(message: string, author: string = "anonyme"): void {
    console.log(`${message} envoyé par : ${author}`);
}
```

Lorsque j'appel cette fonction vous n'êtes donc pas obligé de satisfaire ce paramètre :

```
greet("Un super msg");              // "Un super msg envoyé par : anonyme"
greet("Un super msg", "Un random");   // "Un super msg envoyé par : Un random"
```

Fonctions typées en TypeScript

Typage des paramètres et du retour :

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

Valeurs par défaut avec typage :

```
function add(a: number, b: number = 10): number {
    return a + b;
}
```

Fonction fléchée typée :

```
const multiply = (x: number, y: number): number => x * y;
```

Fonction avec plusieurs types (union) :

```
function format(input: string | number): string {
    return input.toString();
}
```

Fonction sans retour (void) :

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

Union de type

Pour créer une union de type on utilise `|` pour dire : "ce type peut être ceci ou cela".

Tableau contenant soit des strings soit des nombres :

```
let mixedValues: Array<string | number> = ["hello", 42, "world", 7];
```

Une fonction qui accepte un identifiant sous forme de string ou de number :

```
function printId(id: string | number): void {
console.log("ID:", id);
}
```

Intersection de type

Pour créer une intersection de type on utilise `&` (pas `&&`) pour dire : "ce type doit respecter les deux formes en même temps".

```
interface WithId {
  id: number;
}

interface WithName {
  name: string;
}

type IdentifiedPerson = WithId & WithName;

const user: IdentifiedPerson = {
  id: 1,
  name: "Alice",
};
```