



DIGINAMIC

FORMATION

Introduction à Angular : Typescript



Introduction à TypeScript

- Superset de JavaScript créé par Microsoft
- Ajoute des types statiques au langage
- Se compile en JavaScript classique
- Utilisé dans des frameworks comme Angular, NestJS
- <https://www.typescriptlang.org/fr/>



Pourquoi TypeScript

- Meilleure détection d'erreurs
- Code plus lisible et maintenable
- Autocomplétion plus précise
- Facilite le travail en équipe
- Recommandé dans les projets professionnels
- Plus de **robustesse** et de **sécurité**

Projet TypeScript minimal

Projet TS minimal (Node Js déjà installé)

1. Initialiser le projet
2. Installer TypeScript pour le projet
3. Initialiser TypeScript
4. Créer le fichier .ts
5. Compiler en JavaScript
6. Exécuter le JS avec Node

1. Initialiser le projet

- `npm init -y` permet d'initialiser un projet node

2. Installer Typescript pour le projet

- Une fois le projet crée, on lance la commande `npm install --save-dev typescript` permet d'installer localement TypeScript pour le projet

3. Initialiser TypeScript

- Il est nécessaire de créer un fichier de configuration TypeScript, on peut le générer via la commande `npx tsc --init`
- Voici un exemple de fichier tsconfig.json

```
Uploaded using RayThis Extension

{
  "compilerOptions": {
    "target": "ES2020",
    "module": "CommonJS",
    "rootDir": "src",
    "outDir": "dist",
    "strict": true
  }
}
```

4. Créer le fichier .ts

- Créer un dossier src (`mkdir src` sur powershell)
- Dans le dossier src, créer un fichier `index.ts` (par exemple) qui servira de point d'entrée du projet node

5. Compiler en JavaScript

- Comme expliqué, TypeScript est un langage qui doit être compilé en javascript
- On lancera donc la commande `npx tsc` qui transformera les fichiers .ts en .js vers un nouveau dossier dist (voir tsconfig.json “outDir”)

6. Exécuter le JS avec Node

- On lance la commande `node dist/index.js` pour lancer le projet une fois compilé
- Option : ajouter les scripts dans package.json

```
Uploaded using RayThis Extension

"scripts": {
  "build": "tsc",
  "start": "node dist/index.js"
}
```

Les types en TypeScript

string	number	boolean	any	unknown	null	undefined
--------	--------	---------	-----	---------	------	-----------



Uploaded using RayThis Extension

```
let nom: string = "Alice";  
let age: number = 30;  
let actif: boolean = true;
```

Tableaux et tuples

- TS introduit les tuples
- Tableau : aucun contrôle sur l'ordre ou la longueur.
- Tuple : fixe le nombre d'éléments et leur type à chaque position



Uploaded using RayThis Extension

```
let fruits: string[] = ["pomme", "banane"];  
let utilisateur: [string, number] = ["Bob", 42]; // Tuple
```

Fonctions typées

- On définit le type des paramètres et du retour



Uploaded using RayThis Extension

```
function saluer(nom: string): string {  
    return "Bonjour " + nom;  
}
```

Exercices

Enums

- Fonctionnalité TS utile pour représenter un ensemble de valeurs nommées
- Rend le code plus lisible, permet d'éviter des erreurs de typage

```
enum Role {  
  ADMIN,  
  USER,  
  GUEST,  
}  
  
let monRole: Role = Role.ADMIN;
```

Interfaces

- Fonctionnalité TS pour définir une structure d'objet typée
- Dans cet exemple, le compilateur vérifie bien que utilisateur correspond bien **exactement** à l'interface Personne (nom et age)

```
Uploaded using RayThis Extension

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

let utilisateur: Personne = {
  nom: "Alice",
  age: 25
};
```

Classes

- Ajout de types
- Modificateurs d'accès (public, private, protected)

```
Uploaded using RayThis Extension

class Animal {
  constructor(public nom: string) {}

  crier(): void {
    console.log(`${this.nom} crie !`);
  }
}

let chat = new Animal("Mimi");
chat.crier();
```

Types personnalisés

- Permet par exemple l'union de plusieurs types
- Le **type** est préféré à l'interface si l'on a besoin de **plus de flexibilité**



Uploaded using RayThis Extension

```
type Identifiant = string | number;  
  
let id: Identifiant = 123; // ou "ABC123"
```

Exercices

Génériques

- Un générique est un type paramétré : prend un paramètre qui sera générique
- Permet d'écrire une fonction qui marche avec n'importe quel type

```
Uploaded using RayThis Extension

function identite<T>(val: T): T {
    return val;
}

let resultat = identite<string>("test");
```

Génériques : exemple

- Filtrer un tableau

```
Uploaded using RayThis Extension

function filterArray<T>(arr: T[], predicate: (item: T) => boolean): T[] {
    return arr.filter(predicate);
}

const numbers = [1, 2, 3, 4, 5];
const evens = filterArray(numbers, n => n % 2 === 0); // evens: number[]

const words = ["apple", "banana", "cherry"];
const filteredWords = filterArray(words, w => w.startsWith("b")); // filteredWords: string[]
```

Exercices

Bonnes pratiques

- Toujours typer arguments et retours
- Utiliser les interfaces pour vos objets
- Eviter le “passe-partout” any



TP : Validation des acquis