Developpement Web

Chapitre 4: Introduction à TypeScript

M. Mohammed BELATAR

m.belatar@emsi.ma

avec Mme. FZ MOUTAI et M. YF EBOBISSE

3ème Année Ingénierie en Informatique et Réseaux

28 décembre 2024

Sommaire

Origine de TypeScript

- Développé par Microsoft et lancé en 2012.
- Contexte : Besoin d'un langage pour le développement à grande échelle avec JavaScript.
- **Inspiration** :
  - Influence de langages comme Java et C# pour la gestion des types.
  - Objectif : Améliorer la robustesse et la maintenabilité du code.

Caractéristiques de TypeScript

- **Typage Statique** : Détection des erreurs avant l'exécution.
  - Exemples de types de base : `number`, `string`, `boolean`.
- **Interfaces et Classes** : Support de la programmation orientée objet.
  - Exemples : Création d'une interface pour un utilisateur.
- **Compatibilité** : Tout JavaScript valide est également TypeScript.
- **Compilation** : Conversion en JavaScript standard.
  - Exemples : Configuration pour cibler différentes versions de JavaScript.

Adoption de TypeScript

- Utilisé par des entreprises comme Google (Angular), Slack, et Airbnb.
- Intégration dans les principaux frameworks front-end et back-end.
- **Écosystème** :
  - Outils : Visual Studio Code, Prettier...
  - Bibliothèques typées (ex. : DefinitelyTyped).

Comparaison TypeScript vs JavaScript

Caractéristique	**JavaScript**	**TypeScript**
Typage	Dynamique	Statique (peut être dynamique)
Détection des erreurs	À l'exécution	À la compilation
Syntaxe	Flexible, permissive	Plus stricte, inclut des types
Interfaces	Non supporté	Supporté, facilite la structuration
Génériques	Non supporté	Supporté, permet des fonctions et classes flexibles
Compilabilité	Pas de compilation	Doit être compilé en JavaScript
Mise à jour du code	Risque d'erreurs non détectées	Révisions plus sûres avec typage

Différences de syntaxe entre TypeScript et JavaScript

Fonctions

JavaScript :

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

TypeScript :

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

Interfaces

JavaScript :

```
// Pas d'interface native.
```

TypeScript :

```
interface Person { name: string; age: number; }
```

Avantages de TypeScript

- **Sécurité des Types** : Réduit les erreurs grâce à la vérification statique.
- **Outils de Développement** : Intellisense, suggestions de code, refactoring.
- **Facilité de Maintenance** : Code mieux structuré et plus lisible.
- **Interopérabilité** : TypeScript fonctionne avec les bibliothèques JavaScript existantes.

Cas d'utilisation

- **Projets à Grande Échelle** : Exemples d'applications complexes (ex. : applications d'entreprise).
- **Travail en Équipe** : Comment TypeScript facilite la collaboration.
- **Migration de Projets** : Avantages d'introduire TypeScript dans un projet JavaScript existant.

Résumé

- TypeScript améliore la qualité du code JavaScript.
- Favorise une meilleure organisation et maintenabilité.
- Encourage les bonnes pratiques de développement.
- **Discussion** : Quel aspect de TypeScript vous intéresse le plus jusqu'à présent ?

Frameworks Front-End populaires basés sur TypeScript

- **Angular** :
  - Développé par Google, Angular est entièrement écrit en TypeScript.
  - Utilisé pour créer des applications web complexes à grande échelle.
  - Supporte la programmation orientée objet et le typage statique.
- **React** :
  - Bien que React soit écrit en JavaScript, TypeScript est largement utilisé avec React pour des applications robustes.
  - Facilite le typage des props, des états, et des composants fonctionnels.
  - Améliore la sécurité du code dans les grandes applications.
- **Vue.js** :
  - Vue 3 a introduit un support natif pour TypeScript.
  - Améliore la maintenabilité et la lisibilité des projets Vue à grande échelle.
  - Utilisé pour des interfaces utilisateur interactives et performantes.

Frameworks back-end populaires basés sur TypeScript

- **Node.js avec TypeScript** :
  - TypeScript est largement utilisé pour développer des applications Node.js, apportant le typage statique côté serveur.
  - Améliore la lisibilité du code et facilite le débogage des applications back-end.
- **NestJS** :
  - Inspiré par Angular, NestJS est un framework back-end basé sur TypeScript.
  - Conçu pour créer des applications server-side évolutives et modulaires.
  - Supporte les paradigmes de programmation orientée objet, fonctionnelle, et réactive.
- **Deno** :
  - Nouveau runtime pour JavaScript et TypeScript, créé par l'auteur de Node.js.
  - Intègre le support TypeScript de manière native, sans besoin de configuration supplémentaire.
  - Orienté vers la sécurité et la simplicité par rapport à Node.js.

Sommaire

Introduction : Utiliser TypeScript pour le Front-End

- TypeScript est utilisé pour écrire du code plus robuste en front-end.
- Pas besoin de Node.js pour exécuter du TypeScript dans le navigateur.
- Recommandé : **Visual Studio Code** comme éditeur.

Pourquoi utiliser VS Code ?

- Autocomplétion intelligente avec Intellisense.
- Terminal intégré pour compiler du TypeScript.

Installation de TypeScript

- Installer TypeScript via `npm` (nécessite d'abord d'installer Node.js).
- Télécharger Node.js ici : nodejs.org
- Commande pour installer TypeScript globalement :

Commande

```
npm install -g typescript
```

- Vérifiez l'installation avec :

Commande

```
tsc --version
```

Écrire du code TypeScript

- Les fichiers TypeScript ont l'extension `.ts`.
- Exemple de fichier TypeScript (`script.ts`) :

Exemple de Code

```
let message: string = "Bonjour TypeScript!";
console.log(message);
```

Ce fichier sera compilé en JavaScript pour être utilisé dans le navigateur.

Compiler du TypeScript en JavaScript

- Utiliser la commande suivante pour compiler le fichier TypeScript :

Commande

```
tsc script.ts
```

- Résultat : Un fichier `script.js` est généré.
- Inclure le fichier généré dans votre page HTML pour l'exécuter dans le navigateur.

Intégrer le fichier JavaScript dans une Page HTML

- Exemple d'intégration :

Exemple HTML

```
 <!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>TypeScript Front-End</title>
  <script src="script.js" defer></script>
</head>
<body>
  <h1>Bienvenue dans TypeScript</h1>
</body>
</html>
```

Résultat : Affiche "Bonjour TypeScript !" dans la console du navigateur.

Compilation automatique avec `tsc --watch`

- Utilisez le mode `tsc --watch` pour compiler automatiquement lors des modifications.

Commande

```
tsc script.ts --watch
```

- Utile pour un flux de développement fluide.

Utilisation de `tsconfig.json`

- Utilisez un fichier `tsconfig.json` pour configurer votre projet TypeScript.
- Créez-le avec la commande :

Commande

```
tsc --init
```

- Exemple de configuration :

Exemple de `tsconfig.json`

```
{
 "compilerOptions": {
   "target": "es6",
   "outDir": "./dist",
   "strict": true
 }
}
```

Déclaration de variables

- Utilisation des mots-clés : `let`, `const`, `var`.
- `let` et `const` ont une portée limitée au bloc.
- `var` a une portée globale ou fonctionnelle (à éviter).
- Préférez `let` pour les variables modifiables et `const` pour les constantes.

Exemple

```
let age: number = 30;
const PI: number = 3.14159;
var user: string = "Aïcha";
```

Note : Utilisez `const` pour les valeurs qui ne changent pas.

Types primitifs en TypeScript

- **Types supportés** : `number`, `string`, `boolean`, `any`, `void`, `null`, `undefined`.
- Type `number` : supporte les entiers, les flottants, les hexadécimaux, les binaires, et les octaux.
- Type `string` : supporte les chaînes de caractères.
- Type `boolean` : valeurs `true` ou `false`.

Type number

- TypeScript utilise `number` pour représenter à la fois les entiers et les nombres à virgule flottante.
- Supporte différentes bases : décimale, hexadécimale, binaire, et octale.

Exemples

```
let entier: number = 42;
let flottant: number = 3.14;
let hex: number = 0xABCD;
let binaire: number = 0b1010;
let octal: number = 0o744;
```

Note : Utilisez des nombres flottants pour les calculs avec des décimales.
Quastion : Comment afficher hex, binaire et octal sur la console ?.

Type string

- Le type `string` est utilisé pour représenter des chaînes de caractères.
- Supporte les guillemets simples, doubles, et les templates strings.
- Les templates strings utilisent des backticks (" `") pour inclure des expressions.

Exemples

```
let nom: string = "Aïcha";
let salutation: string = 'Bonjour';
let message: string = `Hello, ${nom}`; // Interpolation
```

Utilisation : Les templates strings sont pratiques pour construire des chaînes complexes.

Type boolean

- Utilisé pour représenter deux valeurs : `true` ou `false`.
- Pratique pour les conditions et les contrôles de flux.

Exemple

```
let isActive: boolean = true;
let isLoggedIn: boolean = false;
```

Utilisation : Les booléens sont souvent utilisés dans les instructions conditionnelles.

Les tableaux en TypeScript

- Un tableau est une collection ordonnée d'éléments du même type.
- Utilisé pour stocker plusieurs valeurs dans une seule variable.
- TypeScript garantit que tous les éléments du tableau respectent le type spécifié.

Syntaxe : `let nom: type[];`

Déclaration et initialisation d'un tableau

- Deux façons de déclarer un tableau :
  1. Utilisation des crochets : `let nombres: number[];`
  2. Utilisation de la syntaxe générique : `let nombres: Array<number>;`

Exemple

```
let fruits: string[] = ["Pomme", "Banane", "Orange"];
let nombres: Array<number> = [1, 2, 3, 4, 5];
console.log(fruits); // Affiche ["Pomme", "Banane",
"Orange"]
console.log(nombres); // Affiche [1, 2, 3, 4, 5]
```

Accéder aux éléments d'un tableau

- Utilisez des indices pour accéder aux éléments d'un tableau.
- Les indices commencent à zéro (0).
- Exemple d'accès à un élément :

Exemple

```
let couleurs: string[] = ["Rouge", "Vert", "Bleu"];
console.log(couleurs[0]); // Affiche "Rouge"
console.log(couleurs[2]); // Affiche "Bleu"
```

Modifier et jouter des éléments dans un tableau

- Utilisez l'index pour modifier un élément existant.
- Utilisez `push()` pour ajouter un nouvel élément.

Exemple

```
let nombres: number[] = [10, 20, 30];
nombres[1] = 25; // Modifie l'élément à l'indice 1
nombres.push(40); // Ajoute un élément à la fin
console.log(nombres); // Affiche [10, 25, 30, 40]
```

Parcourir un tableau avec une boucle

- Utilisez une boucle `for` pour parcourir un tableau.
- Utilisation de `for...of` pour un accès simplifié.

Exemple

```
let fruits: string[] = ["Pomme", "Banane", "Orange"];
for (let fruit of fruits) {
 console.log(fruit);
}
// Affiche "Pomme", "Banane", "Orange"
```

Boucles avec forEach et map en TypeScript

- forEach : Exécute une fonction pour chaque élément du tableau.
- map : Crée un nouveau tableau en appliquant une fonction à chaque élément.

Exemple avec forEach

```
let nombres: number[] = [1, 2, 3, 4, 5];
nombres.forEach((num) => {
 console.log(num * 2);
});
// Affiche 2, 4, 6, 8, 10
```

Exemple avec map

```
let carres = nombres.map((num) => num * num);
console.log(carres);
// Affiche [1, 4, 9, 16, 25]
```

Note : forEach ne retourne rien, tandis que map retourne un nouveau tableau.

Tableaux multidimensionnels en TypeScript

- TypeScript supporte les tableaux à plusieurs dimensions.
- Utiles pour représenter des grilles, matrices, etc.

Exemple

```
let matrice: number[][] = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
console.log(matrice[1][2]); // Affiche 6
```

Types unions

- **Types Unions** : Permet à une variable d'accepter plusieurs types.
- Utilisé lorsque la valeur d'une variable peut être de différents types selon le contexte.
- Très utile pour les fonctions qui doivent accepter différents types d'arguments.

Exemples Simples

```
let id: number | string = 123;
id = "ABC";
console.log(typeof id);
id=33;
console.log(typeof id);
```

Types unions

Cas d'Utilisation : Fonctions avec Types Unions

```
function printId(id: number | string): void {
   if (typeof id === "string") {
     console.log("ID (string):", id.toUpperCase());
   } else {
     console.log("ID (number):", id.toFixed(2));
   }
}
```

- **Cas d'Utilisation** : Gérer des réponses d'API où un champ peut être soit une chaîne, soit un nombre.

Le Type "any"

- Le type 'any' permet à une variable de **contenir n'importe quel type**.
- Semblable à JavaScript : les variables ne sont pas typées de manière stricte.
- Utilisation du type 'any' **désactive le typage statique** de TypeScript.
- Utile pour le **débogage rapide** ou la **migration de projets JavaScript vers TypeScript**.
- **Usage déconseillé** dans l'absolu.

Exemples d'utilisation du type "any"

Exemple 1 : Variable avec Type 'any'

```
let data: any = 42;
data = "TypeScript"; data = true;
```

Exemple 2 : Fonction Acceptant 'any'

```
function processData(input: any): void {
   console.log("Input:", input);
}
processData(123); processData("Texte"); processData([1, 2,
3]);
```

Cas d'utilisation du type "any"

- Utilisé lors de la **migration d'un projet JavaScript** existant vers TypeScript.
- Utile pour des **données dynamiques** provenant de sources externes (API, fichiers JSON, etc.).
- Pratique pour le **débogage temporaire** avant de déterminer le type exact.

Exemple : Réponse d'API

```
let response: any = fetchDataFromAPI();
console.log(response.name);
```

Précautions avec le type "any"

- L'utilisation excessive de "any" **annule les avantages du typage** de TypeScript.
- Peut conduire à des **erreurs à l'exécution** difficiles à déboguer.
- Préférez utiliser des types plus spécifiques ('unknown', 'union types', etc.) si possible.

Alternative : Utiliser 'unknown' au lieu de 'any'

```
let input: unknown = getInput();
if (typeof input === "string") {
   console.log(input.toUpperCase());
}
```

Les Types 'any' et 'unknown' en TypeScript

- **'any'** : Permet à une variable d'accepter n'importe quel type de valeur.
- **'unknown'** : Peut contenir n'importe quel type de valeur, mais nécessite une vérification avant utilisation.
- **Différence clé** : 'any' désactive les vérifications de types, tandis que 'unknown' les impose.

Comparaison entre 'any' et 'unknown'

Caractéristique	**'any'**	**'unknown'**
Accepte tous les types	✓	✓
Vérification de type avant utilisation	✗	✓
Sécurité du typage	✗ (risque d'erreurs)	✓ (plus sécurisé)
Utilisation recommandée	Migration, prototypes	Entrées dynamiques

Exemples d'utilisation de 'any' et 'unknown'

Exemple avec 'any'

```
 let valeur: any = 10;
console.log(valeur.toUpperCase()); // Pas d'erreur à la
compilation
```

Problème : Pas de vérification, risque d'erreur à l'exécution.

Exemple avec "unknown"

```
 let valeur: unknown = 10;
valeur = "TypeScript";
if (typeof valeur === "string") {
   console.log(valeur.toUpperCase());
} else {
   console.log("La valeur n'est pas une chaîne");
}
```

Résultat : Sécurisé, nécessite une vérification de type.

Fonctions avec typage

- Spécifier les types pour les paramètres et le retour.
- Supporte les paramètres optionnels et les valeurs par défaut.

Exemple

```
function addition(a: number, b: number = 0): number{
 return a + b;
}
console.log(addition(5)); // Renvoie 5
```

Fonctions anonymes en TypeScript

- Une fonction anonyme est une fonction sans nom.
- Utilisée souvent comme argument à d'autres fonctions ou pour des callbacks.
- Pratique pour les opérations temporaires ou ponctuelles.

Exemple de Fonction Anonyme

```
const saluer = function(nom: string): string {
 return "Bonjour, " + nom;
};
console.log(saluer("Aïcha")); // Bonjour, Aïcha
```

Utilisation : Souvent utilisée avec des méthodes comme `array.forEach` ou avec des évènements.

Fonctions fléchées : écriture minimale

- Les fonctions fléchées peuvent être écrites de manière très concise.
- Idéales pour les fonctions simples sans paramètres.

Exemples

Fonction avec un paramètre :

```
const saluer = (nom: string) => `Bonjour, ${nom}`;
console.log(saluer("Aïcha")); // Bonjour, Aïcha
```

Fonction sans paramètre :

```
const direBonjour = () => "Bonjour tout le monde";
console.log(direBonjour()); // Bonjour tout le monde
```

Note : Pas besoin d'accolades ou de mot-clé `return` pour une seule expression.

Fonction fléchée vs fonction anonyme

Comportement du mot-clé this.

```
 class Utilisateur {
   nom: string;
   constructor(nom: string) { this.nom = nom; }
   saluerAnonyme() {
     setTimeout(function() { console.log("Bonjour " + this.nom);
}, 1000);
   }
   saluerFleche() {
     setTimeout(()=>{console.log("Bonjour " + this.nom);}, 1000);
   }
}
const user = new Utilisateur("Aïcha");
user.saluerAnonyme(); // Affiche "Bonjour undefined" après 1
seconde
user.saluerFleche(); // Affiche "Bonjour Aïcha" après 1 seconde
```

Explications des résultats : fonction anonyme vs fléchée

- **Fonction Anonyme** :
  - La fonction anonyme définit son propre contexte `this`.
  - Ici, `this` fait référence à l'objet `setTimeout`, non à l'instance `Utilisateur`.
  - Résultat : `"Bonjour undefined"` car `this.nom` n'existe pas dans ce contexte.
- **Fonction Fléchée** :
  - La fonction fléchée hérite du contexte `this` de la classe `Utilisateur`.
  - Ici, `this` fait référence à l'instance `Utilisateur`.
  - Résultat : `"Bonjour Aïcha"` car `this.nom` est bien défini.

Conclusion : Utilisez des fonctions fléchées pour les callbacks afin d'éviter les problèmes avec `this`.

Fonctions avec paramètres optionnels

- Les paramètres optionnels sont marqués avec point d'interrogation (?)
- Utiles lorsque certains paramètres ne sont pas toujours nécessaires.
- Si le paramètre n'est pas fourni, il sera défini à `undefined`.

Exemple Détaillé

```
 function afficherInfo(nom: string, age?: number, ville?:
string): string {
   let info = `Nom : ${nom}`;
   if (age !== undefined) { info += `, Age : ${age}`; }
   if (ville) { info += `, Ville : ${ville}`; }
   return info;
}
console.log(afficherInfo("Aïcha")); // Nom : Aïcha
console.log(afficherInfo("Hamid", 25)); // Nom : Hamid, Age : 25
console.log(afficherInfo("Karima", 30, "Paris")); // Nom :
Karima, Age : 30, Ville : Paris
```

Note : Utilisez des conditions pour vérifier si un paramètre optionnel est défini.

Fonctions avec paramètres par défaut en TypeScript

- TypeScript permet de définir des paramètres avec des valeurs par défaut.
- Si aucun argument n'est fourni, la valeur par défaut sera utilisée.

Syntaxe

```
function nom(param: type = valeurParDéfaut): typeRetour {}
```

Exemple

```
 function saluer(nom: string = "inconnu"): void {
   console.log(`Bonjour, ${nom}! `);
}
saluer(); // Affiche : Bonjour, inconnu!
saluer("Mohammed"); // Affiche : Bonjour, Mohammed!
```

Paramètres par défaut et optionnels

- Les paramètres par défaut peuvent être combinés avec des paramètres optionnels.
- Un paramètre avec valeur par défaut est toujours évalué lors de l'appel.
- **Attention** : Les paramètres avec valeur par défaut doivent être placés après les paramètres obligatoires.

Exemple

```
function creerUtilisateur(nom: string, role: string =
"utilisateur"): void {
  console.log(`Nom: ${nom}, Rôle: ${role}`);
}
creerUtilisateur("Aïcha"); // Affiche : Nom: Aïcha, Rôle:
utilisateur
creerUtilisateur("Hamid", "admin"); // Affiche : Nom: Hamid,
Rôle: admin
```

Note : Si un paramètre est optionnel, utilisez le symbole ?.

Résumé

- TypeScript apporte de la sécurité au développement JavaScript.
- Le typage statique aide à éviter les erreurs courantes.
- Recommandé pour les projets complexes ou collaboratifs.

Sommaire

Sécurité de type en TypeScript

- TypeScript effectue une vérification des types à la compilation.
- Permet de détecter les erreurs avant même d'exécuter le code.
- Réduit le risque de bugs en production.

Exemple : Erreur de Typage

```
 let age: number = 25;
age = "trente"; // Erreur : Type 'string' n'est pas
assignable au type 'number'
```

Résultat : L'erreur est détectée à la compilation.

Pourquoi la sécurité de type est-elle importante ?

- Prévention des erreurs courantes liées aux types.
- Améliore la robustesse du code.
- Facilite le refactoring du code sans introduire de nouvelles erreurs.

Exemple : Sécurité des fonctions

```
 function addition(a: number, b: number): number {
   return a + b;
}
console.log(addition(5, "3")); // Erreur à la compilation
```

Résultat : L'erreur est détectée avant l'exécution.

Amélioration de la lisibilité avec TypeScript

- Les types explicites rendent le code plus compréhensible.
- La documentation est facilitée grâce aux annotations de type.
- Les IDE comme VS Code offrent une autocomplétion basée sur les types.

Exemple : Code Lisible avec TypeScript

```
 interface Utilisateur {
   nom: string;
   age: number;
   email: string;
}
const user: Utilisateur = { nom: "Aïcha", age: 30, email:
"aïcha@example.com" };
```

Avantage : Le code est auto-documenté et facile à comprendre.

Meilleure maintenabilité du code avec TypeScript

- TypeScript aide à maintenir de grands projets grâce à la vérification des types.
- Facilite la collaboration en équipe en réduisant les erreurs dues aux modifications.
- Favorise l'utilisation de pratiques comme le refactoring en toute sécurité.

Avant (JavaScript)

```
 let produit = { id: 1, nom: "PC", prix: 1000 };
produit.prix = "mille"; // Pas d'erreur en JavaScript
console.log("Le prix est :", produit.prix);
```

Problème : En JavaScript, il n'y a pas d'erreur, même si "prix" est supposé être un "number".

Meilleure maintenabilité du code (suite..)

Après (TypeScript)

```
 type Produit = { id: number; nom: string; prix: number };
let produit: Produit = { id: 1, nom: "PC", prix: 1000 };
produit.prix = "mille"; // Erreur : Type 'string' n'est pas
assignable au type 'number'
console.log("Le prix est :", produit.prix);
```

Résultat : Le système de types de TypeScript détecte automatiquement l'erreur avant l'exécution, ce qui rend le code plus sûr et plus facile à maintenir.

Sommaire

Types personnalisés en TypeScript

- Les types personnalisés simplifient et structurent le code.
- Utilisation du mot-clé `type` pour créer des types personnalisés.

Exemple de Type Personnalisé

```
 type ID = string | number;
let identifiant: ID = "123";
identifiant = 456;
console.log(identifiant);
```

Résultat : Affiche "456"

Enums en TypeScript

- Les enums définissent des ensembles de valeurs nommées pour améliorer la lisibilité.

Exemple d'Enum Numérique

```
 enum Direction { Haut, Bas, Gauche, Droite };
let dir: Direction = Direction.Gauche;
console.log(dir);
```

Résultat : Affiche "2"

Enums de chaînes

- On peut aussi personnaliser les valeurs d'une énumération de la manière suivante :

Exemple d'Enum de Chaînes

```
 enum Role { Admin = "ADMIN", Utilisateur = "USER", Invite =
"GUEST" };
let role: Role = Role.Admin;
console.log(role);
```

Résultat : Affiche "ADMIN"

Tuples en TypeScript

- Un tuple est un tableau avec un nombre fixe d'éléments de types prédéfinis.
- Contrairement aux tableaux classiques, les types des éléments d'un tuple sont définis à l'avance.
- Utile pour regrouper des valeurs hétérogènes dans un ordre précis.

Exemple de Tuple Simple

```
 let utilisateur: [string, number];
utilisateur = ["Aïcha", 30];
console.log(utilisateur);
```

Résultat : Affiche '["Aïcha", 30]'

Accéder aux éléments d'un Tuple

- Utilisez la notation d'index pour accéder aux éléments d'un tuple.
- Les tuples ont un nombre fixe d'éléments, ce qui garantit l'ordre des types.

Exemple

```
 let produit: [string, number, boolean] = ["Ordinateur",
1000, true];
console.log(produit[0]); // "Ordinateur"
console.log(produit[1]); // 1000
console.log(produit[2]); // true
```

Note : Un accès hors limites génère une erreur.

Destructuration des Tuples

- TypeScript supporte la destructuration des tuples pour un accès simplifié.

Exemple de Destructuration

```
 let utilisateur: [string, number] = ["Hamid", 45];
let [nom, age] = utilisateur;
console.log(nom); // "Hamid"
console.log(age); // 45
```

Avantage : Simplifie l'accès aux valeurs dans un tuple.

Tuples avec paramètres optionnels

- Les tuples peuvent contenir des paramètres optionnels.
- Pratique pour gérer des valeurs facultatives dans un ensemble ordonné.

Exemple

```
 let employe: [string, number, string?] = ["Samir", 35];
console.log(employe); // ["Samir", 35]
employe = ["Samir", 35, "Développeur"];
console.log(employe); // ["Samir", 35, "Développeur"]
```

Tuples avec les fonctions

- Les tuples peuvent être utilisés comme types de retour pour les fonctions.
- Utile pour retourner plusieurs valeurs de types différents.

Exemple de Fonction Retour Tuples

```
 function getUtilisateur(): [string, number] {
   return ["Aïcha", 30];
}
const [nom, age] = getUtilisateur();
console.log(nom); // "Aïcha"
console.log(age); // 30
```

Résultat : Retourne un tuple avec deux valeurs.

Tuples immutables avec `readonly`

- Utilisez le mot-clé `readonly` pour rendre un tuple immuable.
- Empêche la modification des valeurs après l'initialisation.

Exemple d'un Tuple Immuable

```
 const config: readonly [string, number] = ["Serveur",
8080];
console.log(config);
// config[1] = 3000; // Erreur : Impossible de modifier un
tuple en lecture seule
```

Avantage : Prévenir les modifications accidentelles.

Introduction aux Classes en TypeScript

- Une classe est un plan pour créer des objets avec des propriétés et des méthodes.
- TypeScript améliore les classes JavaScript en ajoutant le typage statique.
- Supporte les modificateurs d'accès comme `public`, `private` et `protected`.

Déclaration d'une Classe

Exemple de Classe

```
class Personne {
  public nom: string;
  private age: number;
  constructor(nom: string, age: number) {
    this.nom = nom;
    this.age = age;
  }
  public sePresenter(): void {
    console.log(`Bonjour, je m'appelle ${this.nom}`);
  }
}
const personne = new Personne("Aïcha", 30);
personne.sePresenter();
```

Résultat : Affiche "Bonjour, je m'appelle Aïcha"

Héritage avec les Classes

- TypeScript supporte l'héritage avec le mot-clé `extends`.

Exemple d'héritage

```
 class Employe extends Personne {
   private poste: string;
   constructor(nom: string, age: number, poste: string) {
     super(nom, age);
     this.poste = poste;
   }
   public afficherPoste(): void {
     console.log(`Poste : ${this.poste}`);
   }
}
const employe = new Employe("Hamid", 40, "Développeur");
employe.sePresenter();
employe.afficherPoste();
```

Résultat : Affiche "Bonjour, je m'appelle Hamid" et "Poste : Développeur"

Interfaces en TypeScript

- Une interface définit la structure d'un objet sans implémentation.
- Utilisée pour garantir la cohérence des types.

Exemple d'Interface

```
interface Utilisateur {
  nom: string;
  age: number;
  email?: string;
}
const user: Utilisateur = { nom: "Aïcha", age: 30 };
console.log(user);
```

Résultat : Affiche { nom : "Aïcha", age : 30 }

Extension d'Interfaces

- Les interfaces peuvent être étendues pour créer des objets plus complexes.

Exemple d'Extension

```
 interface Employe extends Utilisateur {
   poste: string;
}
const employe: Employe = { nom: "Hamid", age: 25, poste:
"Développeur" };
console.log(employe);
```

Résultat : Affiche { nom : "Hamid", age : 25, poste : "Développeur" }

Interfaces vs. Classes en TypeScript

- **Interfaces** :
  - Utilisées pour définir des contrats (structure d'objets).
  - Ne contiennent pas d'implémentation.
  - Idéales pour le typage statique et la documentation du code.
- **Classes** :
  - Utilisées pour créer des objets avec des états et des comportements.
  - Peuvent contenir des méthodes, des constructeurs et des modificateurs d'accès.
  - Supportent l'héritage pour la réutilisation du code.
  - Peuvent implémenter les Interfaces pour réutiliser leurs structures.

Exemple Combiné : Interface et Classe

Interface + Classe

```
 interface Drivable {
   drive(): void;
}
class Car implements Drivable {
   drive(): void {
     console.log("The car is driving.");
   }
}
const myCar = new Car();
myCar.drive();
```

Résultat : Affiche "The car is driving"

Résumé : Quand utiliser Interfaces et Classes

- Utilisez les **interfaces** pour :
  - Définir des structures d'objets.
  - Assurer la cohérence du typage à grande échelle.
- Utilisez les **classes** pour :
  - Créer des objets avec des états et des méthodes.
  - Gérer la logique métier et l'héritage.

Introduction aux génériques

Pourquoi Utiliser des Génériques ?

- Les génériques permettent d'écrire du code réutilisable pour différents types.
- Utilisés avec des fonctions, des classes et des interfaces.

Syntaxe Générique

```
function nom<T>(param: T): T {...}
```

Fonctions Génériques

- Les génériques permettent d'écrire des fonctions flexibles.

Exemple de Fonction Générique

```
 function identite<T>(valeur: T): T {
   return valeur;
}
console.log(identite(42)); // 42
console.log(identite("TypeScript")); // TypeScript
```

Résultat : Affiche ‘42‘ et ‘"TypeScript"‘

Génériques avec des tableaux

Exemple avec des tableaux

```
 function filtrer<T>(elements: T[], filtre: (el: T) =>
boolean): T[] {
   return elements.filter(filtre);
}
const nombres = [1, 2, 3, 4, 5];
const pairs = filtrer(nombres, n => n % 2 === 0);
console.log(pairs);
```

Résultat : Affiche ‘[2, 4]‘

Classes génériques

Exemple de Classe Générique

```
 class Boite<T> {
   contenu: T;
   constructor(contenu: T) { this.contenu = contenu; }
   afficher() { console.log(this.contenu); }
}
const boite = new Boite<number>(123);
boite.afficher(); // 123
```

Résultat : Affiche ‘123‘

Sommaire

Conclusion générale

- Nous avons exploré les bases de TypeScript, ses avantages par rapport à JavaScript, ainsi que ses types de données.
- La syntaxe stricte de TypeScript permet de sécuriser le code et d'améliorer la maintenabilité des projets.
- TypeScript est devenu incontournable pour les grands projets, aussi bien en Front-End qu'en Back-End.

Prochaines étapes : Utiliser TypeScript pour des projets Back-End avec Node.js et Express, et pour des applications Front-End avec React.

Références

- Documentation officielle de TypeScript : typescriptlang.org
- Ouvrages recommandés : "TypeScript Quickly" de Yakov Fain, Anton Moiseev.