Module 1 : Introduction à TypeScript

- **Objectif :** Comprendre les bases de TypeScript et son utilité par rapport à JavaScript.

1.1. Introduction à JavaScript

- Comprendre les limitations de JavaScript dans les projets à grande échelle.
- Différences entre TypeScript et JavaScript.
- Installation de Node.js et de TypeScript.
- Configuration d'un projet TypeScript avec `tsconfig.json`.
- Conversion de fichiers `.ts` en `.js`.

1.2. Types de base

- Types primitifs : `number`, `string`, `boolean`, `null`, `undefined`.
- Le mot-clé `any` et pourquoi l'éviter.
- Inférence de type.
- Les tableaux et les tuples.

1.3. **Les structure de controls**

- `if` / `else if` / `else`
- switch
- for
- for...of
- for...in
- while
- do...while

1.4. **Les fonctions utilisateurs**

- Déclaration de fonctions avec typage
- Paramètres optionnels et valeurs par défaut
- Valeurs par défaut
- Fonctions anonymes et fléchées
- Type de retour
- Fonction avec des paramètres rest (`rest parameters`)
- Fonction avec des types de retour complexes

Module 2 : Types avancés et gestion des objets

- **Objectif :** Apprendre à utiliser les types complexes et gérer des objets en TypeScript.

2.1. Types complexes

- Les types personnalisés : `type`, `interface`.

- Différences entre `interface` et `type`.
- Types littéraux et énumérations (`enum`).
- Type d'union et d'intersection.
- Les types génériques.

2.2. Interfaces et objets

- Définir des interfaces pour structurer les objets.
- Implémentation de plusieurs interfaces.
- Étendre une interface.
- Les classes et héritage en TypeScript.
- Modificateurs d'accès : `public`, `private`, `protected`, `readonly`.

Module 3 : Programmation Orientée Objet (POO) avec TypeScript

- **Objectif :** Appliquer les concepts POO à TypeScript.

3.1. Classes et objets

- Définition de classes en TypeScript.
- Constructeurs, propriétés et méthodes.
- Initialisation des membres de classe.
- Classes abstraites et méthodes abstraites.

3.2. Héritage et polymorphisme

- Étendre une classe.
- Surcharge et redéfinition des méthodes.
- Utilisation de polymorphisme dans TypeScript.

3.3. Classes génériques

- Classes et méthodes génériques.
- Contraintes des types génériques.
- Avantages du typage générique pour les bibliothèques de fonctions.

Module 4 : Programmation asynchrone en TypeScript

- **Objectif :** Apprendre à gérer les promesses et l'asynchronisme avec TypeScript.

4.1. Introduction aux promesses

- Syntaxe des promesses : `Promise`, `then`, `catch`, `finally`.
- Conversion d'une fonction JavaScript en TypeScript avec les promesses.
- Le typage des promesses.

4.2. Utilisation de `async` / `await`

- Gérer les fonctions asynchrones avec `async` et `await`.
- Gestion des erreurs dans les fonctions asynchrones (`try`/`catch`).

4.3. Typage des fonctions asynchrones

- Type des fonctions retournant des promesses.
- Typage avancé pour les fonctions asynchrones.

Module 5 : Modules et gestion de code

- **Objectif :** Apprendre à structurer un projet TypeScript en modules.

5.1. Modules en TypeScript

- Importation et exportation de modules.
- Importation dynamique.
- Organisation du projet en modules.

5.2. Gestion des dépendances

- Gestion des dépendances avec `npm` et `yarn`.
- Intégration avec des librairies JavaScript.

Module 6 : Tests et vérification de type

- **Objectif :** Comprendre comment tester et garantir la sécurité du code TypeScript.

6.1. Introduction à la vérification de type

- Utilisation de `tsc` pour vérifier la conformité des types.
- Débogage des erreurs de type les plus fréquentes.

6.2. Tests unitaires

- Introduction aux tests unitaires avec `Jest` et TypeScript.
- Utilisation de `ts-jest` pour écrire des tests TypeScript.
- Écrire des tests pour vérifier le comportement du code.

Module 7 : Utilisation avancée de TypeScript

- **Objectif :** Approfondir les fonctionnalités avancées de TypeScript pour des applications robustes.

7.1. Types conditionnels et utilitaires

- Les types conditionnels (`T extends U ? X : Y`).
- Types utilitaires fournis par TypeScript (`Partial`, `Readonly`, `Pick`, `Omit`).

7.2. Décorateurs

- Introduction aux décorateurs.
- Utilisation des décorateurs pour modifier le comportement des classes et des méthodes.
- Décorateurs dans les frameworks comme Angular.

Module 8 : Intégration avec les frameworks

- **Objectif :** Utiliser TypeScript avec des frameworks populaires comme Angular et React.

8.1. TypeScript avec Angular

- Configuration d'un projet Angular avec TypeScript.
- Types et interfaces dans les services et composants Angular.
- Communication entre les composants avec des types sûrs.

8.2. TypeScript avec React

- Configuration de TypeScript dans un projet React.
- Utilisation des interfaces pour typer les props et l'état (`state`) des composants React.
- Utilisation des types génériques dans les composants fonctionnels.

Module 9 : TypeScript et Node.js

- **Objectif :** Utiliser TypeScript avec des applications backend basées sur Node.js.

9.1. Introduction à Node.js avec TypeScript

- Initialisation d'un projet Node.js avec TypeScript.
- Utilisation de types pour `Express` ou d'autres frameworks back-end.
- Création d'une API REST typée avec TypeScript.

9.2. Typage des bibliothèques

- Gérer les types dans les bibliothèques JavaScript existantes.
- Utilisation des définitions de types avec `@types`.

Module 10 : Outils avancés et optimisation

- **Objectif :** Maîtriser les outils avancés et les bonnes pratiques pour un code TypeScript de production.

10.1. Linting et formatage du code

- Utilisation de `ESLint` pour vérifier les erreurs de code.
- Utilisation de `Prettier` pour formater le code TypeScript.

10.2. Configuration avancée de TypeScript

- Optimisation du fichier `tsconfig.json` pour les environnements de production.
- Utilisation de Webpack ou d'autres bundlers avec TypeScript.

Conclusion : Pratiques de développement avec TypeScript

- Bonnes pratiques pour écrire du code TypeScript maintenable et évolutif.
- Stratégies pour adopter TypeScript dans des projets existants.
- Mise à jour continue des compétences avec les nouvelles versions de TypeScript.

Objectifs généraux :

- Maitriser les bases de TypeScript et ses avantages par rapport à JavaScript.
- Appliquer les concepts de la programmation orientée objet avec TypeScript.
- Écrire du code TypeScript sûr, robuste et maintenable.
- Utiliser TypeScript dans des projets front-end et back-end.

Module 1

1.1 Introduction à JavaScript

Limitations de JavaScript dans les projets à grande échelle

JavaScript est dynamique, ce qui signifie qu'il n'y a pas de vérification de type lors de l'écriture du code. Cela peut rendre difficile la gestion des gros projets, car les erreurs de type ne sont détectées qu'à l'exécution. Exemple :

```
let user = "John";
user = 123;
// JavaScript permet cela, mais cela peut entraîner des bugs plus tard
```

Avec TypeScript, les types sont vérifiés à la compilation, ce qui aide à éviter des erreurs avant même que le code ne s'exécute.

Différences entre TypeScript et JavaScript

JavaScript est un langage dynamique, sans types, tandis que **TypeScript** ajoute des types statiques. En TypeScript, vous pouvez définir explicitement le type d'une variable, et cela améliore la sécurité du code.

Exemple en JavaScript :

Exemple en TypeScript :

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

print(greet(10)); // Cela fonctionne, mais n'a pas de sens (devrait être une
chaîne de caractères)
```

Installation de Node.js et TypeScript

1. **Installer Node.js** : Vous devez d'abord installer Node.js (il inclut `npm`, le gestionnaire de paquets). Vous pouvez le télécharger ici : https://nodejs.org.
2. **Installer TypeScript globalement** : Une fois Node.js installé, vous pouvez installer TypeScript globalement avec `npm` : npm install -g typescript

   Installer TypeScript localement :

   Créer un projet avec `package.json` :

- Dans le répertoire de votre projet, exécutez la commande suivante pour initialiser un projet Node.js : npm init -y

- Installez TypeScript en tant que dépendance de développement dans votre projet :

  npm install typescript --save-dev

  Cela installe TypeScript localement dans le dossier `node_modules`, et vous pourrez l'utiliser via les scripts définis dans votre `package.json` ou via `npx`.

3. **Vérifier l'installation locale** :

- Pour vérifier que TypeScript est bien installé localement et compiler des fichiers `.ts` :

  npx tsc -init

  Cela créera le fichier `tsconfig.json` dans votre projet.

Exécuter TypeScript dans l'environnement virtuel du projet

Une fois TypeScript installé localement dans le projet, vous pouvez exécuter le compilateur TypeScript via `npx`, qui permet de l'utiliser sans installation globale : npx tsc

Cela va compiler vos fichiers `.ts` en `.js` tout en utilisant la version locale de TypeScript dans votre projet.

Étapes pour créer et compiler un fichier `.ts` :

1. **Créer un fichier TypeScript :**
   - Créez un fichier nommé `app.ts` dans votre projet avec du code TypeScript, par exemple :

```
const greeting: string = "Hello, TypeScript!";
console.log(greeting);
```

2. **Compiler le fichier TypeScript en JavaScript :**

- Utilisez la commande suivante pour compiler votre fichier `app.ts` en JavaScript : npx tsc app.ts

Cela génère un fichier `app.js` dans le même répertoire avec le code JavaScript compilé :

```js
var greeting = "Hello, TypeScript!";
console.log(greeting);
```

3. **Exécuter le fichier JavaScript avec Node.js :**

- Exécutez le fichier JavaScript généré avec Node.js : node app.js

```
Hello, TypeScript!
```

1.2. Types de base

Dans cette section, nous allons explorer les types primitifs de TypeScript, le mot-clé `any`, l'inférence de type, et la gestion des tableaux et tuples.

Types primitifs

1. **`number`** : représente tous les types de nombres, qu'ils soient entiers ou à virgule flottante.

   Exemple :

```
let age: number = 25;
let price: number = 19.99;
```

`string` : représente les chaînes de caractères.

Exemple :

```
let name: string = "Dara Sow";
```

`boolean` : représente les valeurs booléennes, soit `true` ou `false`.

Exemple :

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

`null` et `undefined` : sont des types distincts. `undefined` signifie qu'une variable n'a pas été assignée, et `null` signifie qu'une variable a été explicitement vidée de sa valeur.

Exemple :

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

Le mot-clé `any` et pourquoi l'éviter

Le type **`any`** permet à une variable de prendre n'importe quelle valeur, sans vérification de type. Cela désactive les avantages de TypeScript en matière de vérification des types.

Exemple d'utilisation de `any` :

```
let randomValue: any = "hello";
randomValue = 42;
// Cela fonctionne, mais vous perdez la sécurité des types
// on l'utilise lors que c'est necessaire
```

Pourquoi éviter `any` : Utiliser `any` réduit la fiabilité et la sécurité du code, car TypeScript ne pourra plus vérifier les erreurs de type. Il est préférable de toujours utiliser des types explicites ou de s'appuyer sur l'inférence de type sauf si c'est nécessaire.

Inférence de type

TypeScript peut **déduire automatiquement** le type d'une variable si une valeur est assignée lors de sa déclaration. Cela vous permet d'éviter de spécifier le type explicitement.

Exemple d'inférence de type :

```
let message = "Hello"; // TypeScript déduit que c'est une string
let count = 10; // TypeScript déduit que c'est un number
```

TypeScript assigne automatiquement le type correspondant à la valeur initiale, donc vous n'avez pas besoin d'indiquer explicitement `string` ou `number`.

Les tableaux et les tuples

1. **Tableaux (`Array`)** : Les tableaux en TypeScript peuvent être de types homogènes (un seul type) ou hétérogènes (plusieurs types avec `any`).

   Exemple de tableau de nombres :

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

Exemple de tableau de chaînes de caractères :

```
let names: string[] = ["Alice", "Bob", "Charlie"];
```

Tuples : Les tuples sont similaires aux tableaux, mais vous pouvez spécifier différents types pour chaque élément. Ils sont utiles lorsque vous savez à l'avance combien d'éléments le tableau contiendra et quels seront leurs types.

Exemple de tuple :

```
let person: [string, number] = ["Alice", 25];
```

1.3. Les structures de contrôles

1. **`if` / `else if` / `else`**

L'instruction `if` permet d'exécuter un bloc de code en fonction d'une condition. Vous pouvez ajouter des conditions supplémentaires avec `else if` et un bloc par défaut avec `else`.

Exemple :

```
let age: number = 18;
if (age < 18) {
  console.log("Mineur");
} else if (age === 18) {
  console.log("Juste majeur");
} else {
  console.log("Majeur");
}
```

2. **`switch`**

L'instruction `switch` est utilisée pour exécuter différentes branches de code en fonction d'une valeur spécifique.

Exemple :

```
let color: string = "red";
switch (color) {
  case "red":
    console.log("Couleur rouge");
    break;
  case "blue":
    console.log("Couleur bleue");
    break;
  default:
    console.log("Couleur inconnue");
    break;
}
```

Les boucles

1. `for`

La boucle `for` permet d'itérer sur une séquence de valeurs de manière contrôlée.

Exemple :

```
for (let i: number = 0; i < 5; i++) {
    console.log(i);
  }
```

2. `for...of`

La boucle `for...of` permet de parcourir les éléments d'un tableau ou d'une autre structure itérable.

Exemple :

```
let numbers: number[] = [10, 20, 30];
for (let num of numbers) {
  console.log(num);
}
```

3. for...in

La boucle `for...in` permet de parcourir les propriétés énumérables d'un objet.
Exemple :

```
let person = { name: "Dara", age: 25 };
for (let key in person) {
  // Utilisation de 'keyof typeof' pour indiquer que
  //'key' est une clé valide de 'person'
  console.log(`${key}: ${person[key as keyof typeof person]}`);
}
```

4. `while`

La boucle `while` exécute un bloc de code tant qu'une condition donnée est vraie. Exemple :

```
let count: number = 0;
while (count < 3) {
  console.log(count);
  count++;
}
```

5. **`do...while`**

Cette boucle fonctionne de manière similaire à `while`, sauf qu'elle exécute le bloc de code **au moins une fois**, puis vérifie la condition.

Exemple :

```
let index: number = 0;
do {
  console.log(index);
  index++;
} while (index < 3);
```

1.4. **Les fonctions utilisateurs**

Les fonctions sont des blocs de code réutilisables qui permettent d'exécuter une tâche spécifique. TypeScript offre une meilleure gestion des fonctions par rapport à JavaScript en ajoutant un typage strict aux paramètres et au retour de fonction, ce qui améliore la lisibilité et la sécurité du code.

1. Déclaration de fonctions avec typage

En TypeScript, on peut définir les types des paramètres et du type de retour d'une fonction.

Exemple simple :

```
function greet(name: string): string {
    return `Bonjour, ${name}!`;
  }
console.log(greet("Dara"));
```

- Ici, la fonction `greet` accepte un paramètre `name` de type `string` et retourne une chaîne de caractères (`string`).
- Si vous essayez de passer un type non `string`, TypeScript affichera une erreur.

2. Paramètres optionnels et valeurs par défaut

1. **Paramètres optionnels :** Un paramètre peut être rendu optionnel en utilisant `?` après son nom. Si ce paramètre n'est pas fourni, il sera `undefined`.

Exemple :

```
function greet(name: string, age?: number): string {
    if (age) {
      return `Bonjour, ${name}! Tu as ${age} ans.`;
    } else {
      return `Bonjour, ${name}!`;
    }
  }
  console.log(greet("Dara"));
  console.log(greet("Dara", 30));
```

2. **Valeurs par défaut :** Vous pouvez définir une valeur par défaut pour un paramètre si celui-ci n'est pas fourni.

Exemple :

```
function greet(name: string = "Invité"): string {
    return `Bonjour, ${name}!`;
  }
console.log(greet());
console.log(greet("Dara"));
```

3. Fonctions anonymes et fléchées

1. **Fonctions anonymes :** Ce sont des fonctions sans nom qui peuvent être assignées à des variables.

Exemple :

```
let add = function(a: number, b: number): number {
    return a + b;
  };
console.log(add(2, 3));
```

2. **Fonctions fléchées (`arrow functions`) :** Elles sont plus concises que les fonctions traditionnelles et permettent une syntaxe simplifiée, surtout pour les fonctions courtes.

Exemple :

```
let multiply = (a: number, b: number): number => a * b;
console.log(multiply(2, 3));
```

Les fonctions fléchées sont souvent utilisées dans des contextes comme des callbacks ou des itérations, car elles sont plus compactes.

4. Type de retour `void`

Lorsque votre fonction ne retourne rien, vous devez utiliser le type `void`.

Exemple :

```
function logMessage(message: string): void {
    console.log(message);
  }
logMessage("Ceci est un message");
```

Ici, la fonction `logMessage` ne retourne rien, donc on utilise le type `void`.

5. Fonction avec des paramètres rest (`rest parameters`)

Les paramètres `rest` permettent de passer un nombre variable d'arguments à une fonction. Ils sont représentés par `...` suivi du nom du paramètre.

Exemple :

```
function sum(...numbers: number[]): number {
    return numbers.reduce((acc, curr) => acc + curr, 0);
  }
console.log(sum(1, 2, 3, 4)); // Affiche 10
console.log(sum(5, 10)); // Affiche 15
```

6. Fonction avec des types de retour complexes

TypeScript permet d'indiquer des types de retour plus complexes, comme des objets ou des tableaux.

Exemple de fonction retournant un objet :

```
function createPerson(name: string, age: number): { name: string; age: number}
{
    return { name, age };
}
let person = createPerson("Dara", 30);
console.log(person);
```

Module 2 : Types avancés et gestion des objets

Objectif : Apprendre à utiliser les types complexes et gérer des objets en TypeScript.

2.1. Types complexes

Dans cette section, nous allons explorer les types personnalisés et comment ils facilitent la gestion de structures plus complexes en TypeScript. Nous aborderons aussi les différences entre les interfaces et les types, les énumérations, et des concepts avancés comme les types d'union, d'intersection et génériques.

---

1. Types personnalisés : `type` et `interface`

1. **Déclaration de types avec `type`**

Le mot-clé `type` permet de définir des alias pour des types complexes ou des combinaisons de types. Cela permet de donner un nom à une structure de données.

Exemple :

```
type Person = {
    name: string;
    age: number;
  };

  let person1: Person = {
    name: "Dara",
    age: 30
  };
console.log(person1);
```

Ici, le type `Person` décrit la structure d'un objet avec un nom et un âge.

2. **Déclaration d'interfaces avec `interface`**

Une interface définit un contrat pour les objets. Elle est souvent utilisée pour définir les propriétés qu'un objet doit respecter.

Exemple :

```
interface Car {
    brand: string; model: string; year: number;
  }
  let car1: Car = {
    brand: "Toyota",
    model: "Corolla",
    year: 2022
  };
```

```
console.log(car1);
```

Ici, l'interface `Car` décrit les propriétés qu'une voiture doit avoir.

2. Différences entre `interface` et `type`

- **Extensions multiples** : Les interfaces permettent de définir des contrats et de les étendre avec d'autres interfaces. Les types, quant à eux, ne supportent pas l'héritage multiple de la même manière, mais permettent la composition de types plus complexes via les types d'intersection.
- **Merging (Fusion)** : Les interfaces peuvent être fusionnées (merged). Si vous déclarez deux interfaces avec le même nom, TypeScript les combinera automatiquement. Les types ne se fusionnent pas.

Exemple de fusion d'interfaces :

```
interface User {
    name: string;
  }
interface User {
    age: number;
  }
let user: User = {
    name: "Dara",
    age: 30
  };
console.log(user);
```

En pratique, on préfère utiliser des **interfaces** pour les objets, et des **types** pour les alias de types primitifs ou complexes.

3. Types littéraux et énumérations

1. **Types littéraux**

Les types littéraux permettent de restreindre une variable à certaines valeurs spécifiques, souvent utilisées pour les valeurs constantes.

Exemple :

```
type Direction = "left" | "right" | "up" | "down";
let move: Direction = "left"; // Valide
//let move: Direction = "forward"; // Erreur, car non autorisé
console.log(move);
```

Ici, `move` ne peut prendre que les valeurs `"left"`, `"right"`, `"up"`, ou `"down"`.

2. **Énumérations (`enum`)**

Les énumérations définissent un ensemble de constantes nommées, associées à des valeurs numériques ou de chaînes de caractères.

Exemple :

```
enum Color {
    Red = "RED",
    Green = "GREEN",
    Blue = "BLUE"
  }
  let color: Color = Color.Red;
  console.log(color); // Affiche "RED"
```

Veillez vous même poursuivre le programme si necessaire