Travaux Pratiques – TypeScript

Exercice 1 : Introduction

Installe TypeScript et configure-le. Crée un fichier hello.ts qui affiche "Hello TypeScript!" dans la console. Compile-le en JavaScript puis exécute-le avec Node.

Exercice 2 : Types de base

1. Déclare une variable name de type string, age de type number, et isAdmin de type boolean.
2. Crée un tableau scores contenant uniquement des nombres.
3. Crée un tuple [string, number] pour représenter un étudiant.
4. Crée un enum Role { User, Admin, SuperAdmin } et affecte-lui une valeur.

Exercice 3 : Types avancés

1. Crée une variable id qui peut être soit un number, soit un string.
2. Crée deux types A et B, puis fais-en une intersection.
3. Crée un alias Status qui peut valoir "pending", "done" ou "canceled".
4. Déclare une variable unknown et utilise une assertion de type pour accéder à sa longueur si c'est une chaîne.

Exercice 4 : Objets & Interfaces

1. Crée une interface User avec les propriétés id: number, name: string, email?: string (optionnelle), et isAdmin: boolean en lecture seule.
2. Crée un objet user1 conforme à cette interface.
3. Crée une interface Admin qui hérite de User et ajoute une propriété permissions: string[].

Exercice 5 : Fonctions

1. Crée une fonction add(a: number, b: number): number.
2. Crée une fonction greet(name: string, age?: number) qui affiche un message différent selon si l'âge est fourni ou non.
3. Crée une fonction power(base: number, exp: number = 2) qui calcule une puissance avec un exposant par défaut.
4. Crée une fonction combine qui soit peut additionner deux nombres, soit concaténer deux chaînes (surcharge).

Exercice 6 : Programmation Orientée Objet

1. Crée une classe Person avec les propriétés name et age, et une méthode greet().
2. Crée une classe Student qui hérite de Person et ajoute une propriété school.
3. Crée une classe abstraite Shape avec une méthode abstraite area(), puis implémente Circle et Rectangle.
4. Crée une interface Drivable avec une méthode drive(), puis une classe Car qui l'implémente.

Exercice 7 : Génériques

1. Crée une fonction générique identity<T>(value: T): T.
2. Crée une fonction générique getFirst<T>(arr: T[]): T qui retourne le premier élément d'un tableau.
3. Crée une classe générique Repository<T> avec les méthodes add, remove et getAll.
4. Crée une interface générique ApiResponse<T> avec les propriétés data: T et error?: string.

Exercice 8 : Modules & Organisation du code

1. Crée un fichier math.ts avec une fonction add.
2. Importe cette fonction dans main.ts et teste-la.
3. Crée un fichier index.ts qui ré-exporte plusieurs fonctions (add, subtract).
4. Utilise import type pour importer uniquement des types depuis un fichier types.ts.

Exercice 9 : Outils & Bonnes pratiques

1. Initialise un projet TypeScript avec tsc --init.
2. Active "strict": true dans tsconfig.json.
3. Installe ESLint et Prettier (npm install eslint prettier @typescript-eslint/parser @typescript-eslint/eslint-plugin -D).
4. Configure .eslintrc.json pour activer typescript-eslint.

Exercice 10 : TP final – Cas réel

Développe une application de gestion de bibliothèque en TypeScript.

Les livres ont : id, title, author, year, available.
Les utilisateurs ont : id, name, role (User ou Admin).
Une classe Library permet d'ajouter, retirer, rechercher et emprunter/rendre un livre.
Une classe abstraite Person factorise User et Admin.
Un Repository<T> générique gère livres et utilisateurs.

Un service API fictif renvoie une liste de livres.
Le projet doit être organisé en modules (models/, services/, utils/).