



Harouna KANE
CCI Laho

SOMMAIRE

Introduction à TypeScript

Installation et configuration

Les types de base

Les variables typées

Fonctions en TypeScript

Objets et interfaces

Les classes

Types avancés (intro légère)

Génériques (introductif)

Manipulation de modules

QU'EST-CE QUE TYPESCRIPT ?

TypeScript est un langage de programmation open-source développé par Microsoft. Il s'agit d'un sur-ensemble de JavaScript, ce qui signifie que tout code JavaScript valide est aussi du code TypeScript valide.

Mais TypeScript ajoute un système de types statiques, ce que JavaScript n'a pas. Cela permet de :

- détecter les erreurs avant l'exécution,
- mieux documenter son code,
- faciliter la maintenance des projets.

En résumé :

TypeScript = JavaScript + Typage + Outils de développement puissants

POURQUOI UTILISER TYPESCRIPT ?

Avantages :

- Moins d'erreurs : le typage permet de détecter des problèmes au moment de la compilation.
- Meilleure lisibilité : un code bien typé est plus facile à comprendre, même longtemps après l'avoir écrit.
- Autocomplétion : les éditeurs comme VS Code donnent des suggestions intelligentes.
- Travail en équipe : les types servent de contrat entre les développeurs.

Inconvénients (ou plutôt points d'attention) :

- Nécessite une compilation (pas directement exécutable comme du JS).
- Il faut apprendre des concepts supplémentaires si on vient seulement du JavaScript.

Mais ces efforts sont largement récompensés dès que le projet devient un peu plus complexe.

TYPESCRIPT VS JAVASCRIPT – DIFFÉRENCES CLÉS

Aspect	JavaScript	TypeScript
Typage	Dynamique et implicite	Statique et explicite
Compilation	Interprété directement	Doit être compilé en JavaScript
Détection d'erreurs	À l'exécution	À la compilation
Support éditeur	Basique	Avancé (autocomplétion, navigation)
Adoption	Natif du navigateur	Doit être transpilé

UN PETIT EXEMPLE POUR MIEUX COMPRENDRE

En **JavaScript** :

```
function addition(a, b) {  
    return a + b;  
}
```

addition("5", 3); // Résultat : "53" (erreur logique mais pas bloquante)

En **TypeScript** :

```
function addition(a: number, b: number): number {  
    return a + b;  
}
```

// addition("5", 3); // Erreur de compilation : "5" n'est pas un number

Résultat : TypeScript empêche des bugs courants avant même que le code ne s'exécute.

QU'EST-CE QUE LA **COMPILATION** EN TYPESCRIPT ?

En JavaScript, tu peux écrire ton code dans un fichier **.js** et l'exécuter directement dans un navigateur ou avec Node.js.

Mais TypeScript n'est pas compris nativement par les navigateurs ou Node. Il faut d'abord le convertir en JavaScript, car seul JavaScript est compris par les moteurs d'exécution.

C'est ce qu'on appelle la **compilation** (ou plus précisément, la transpilation dans ce cas).

DU *.TS* AU *.JS*

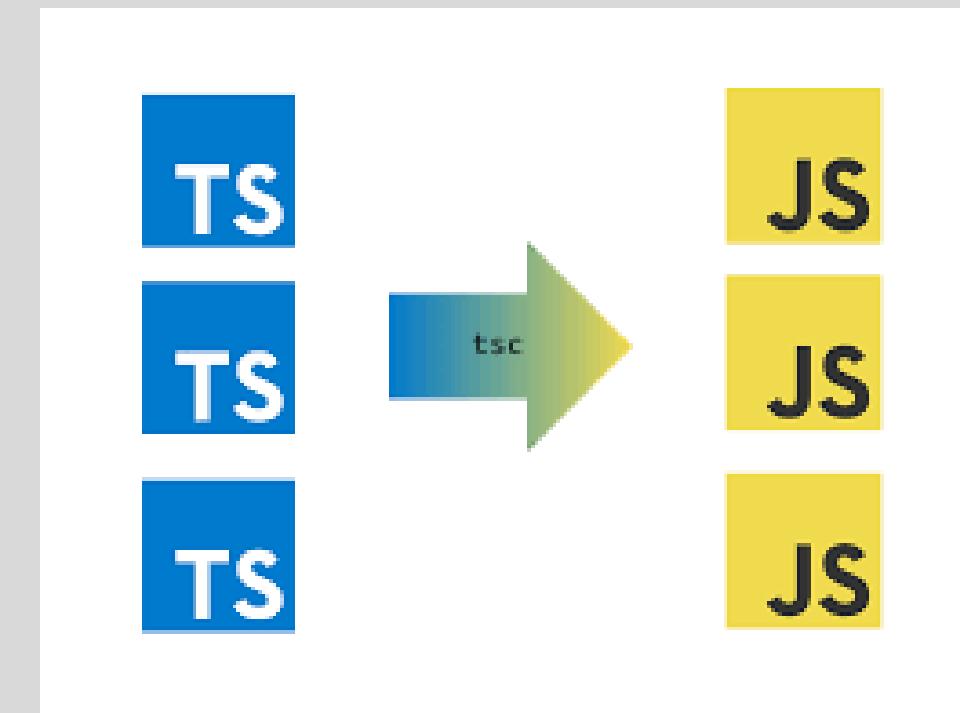
Fichier TypeScript – [hello.ts](#)

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

Compilation avec la commande : [tsc hello.ts](#)

Résultat – Fichier généré : [hello.js](#)

```
var message = "Bonjour TypeScript";
console.log(message);
```



Le compilateur supprime le typage et génère du JavaScript standard, exécutable partout.

AVANTAGES DE CETTE COMPIRATION

- Elle vérifie les erreurs de type et de logique avant même que le code ne soit lancé.
- Elle génère un code propre et souvent plus compatible avec d'anciens environnements.
- Elle permet aussi d'optimiser ou configurer le JS final via le fichier tsconfig.json.

INSTALLATION ET CONFIGURATION DE TYPESCRIPT

INSTALLER TYPESCRIPT

TypeScript s'installe via npm (le gestionnaire de paquets de Node.js).

Assure-toi d'abord que Node.js est installé sur ta machine <https://nodejs.org/fr/download>.

Tu peux vérifier avec : ***node -v***

Ensuite, installe TypeScript globalement avec cette commande :

npm install -g typescript

Tu peux vérifier que TypeScript est bien installé avec :

tsc -v

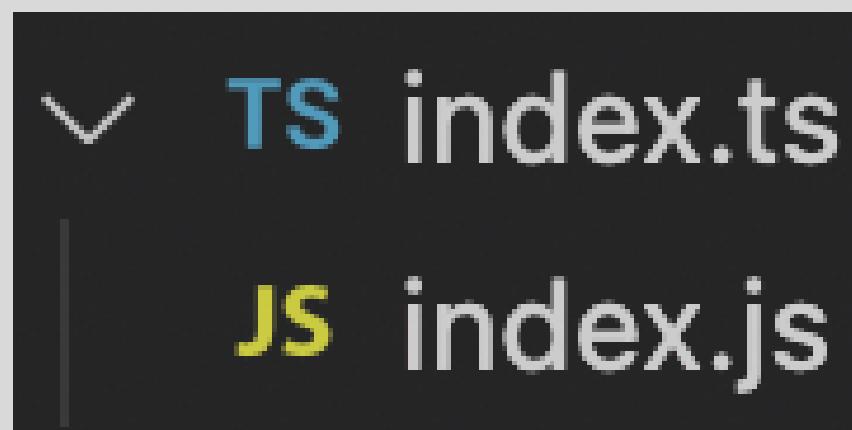
INITIALISER UN PROJET TYPESCRIPT AVEC TSCONFIG.JSON

Au lieu de compiler manuellement chaque fichier .ts, on peut configurer un projet complet grâce à un fichier tsconfig.json.

1. Place-toi dans un dossier de projet vide :
2. Initialise TypeScript dans ce projet : **tsc --init**

Cela crée un fichier tsconfig.json avec toutes les options de configuration (beaucoup sont commentées).

3. Compile tout le projet : **tsc**



RÉSUMÉ DES COMMANDES CLÉS

Action	Commande
Installer TypeScript	npm install -g typescript
Vérifier la version	tsc -v
Compiler un fichier	tsc fichier.ts
Initialiser un projet TS	tsc --init
Compiler tout un projet	tsc
Exécuter du JavaScript	node fichier.js

Les types de base en TypeScript

Comprendre et utiliser les types fondamentaux pour éviter les erreurs et rendre son code plus clair et plus sûr.

Mot clé	Nom de la variable	Type de la variable	Valeur de la variable
let	maVariable	number	= 10

LES TYPES DE BASE EN TYPESCRIPT

1. **string** – Chaîne de caractères

```
let nom: string = "TypeScript";
```

Tu peux aussi utiliser des guillemets simples, doubles ou des backticks :

```
let message: string = `Bienvenue, ${nom}!`;
```

2. **number** – Nombres

```
let age: number = 30;
```

```
let prix: number = 19.99;
```

3. **boolean** – Booléens

```
let actif: boolean = true;
```

```
let connecté: boolean = false;
```

LES TYPES DE BASE EN TYPESCRIPT

4. **any** – Tout type (à éviter)

Le type any désactive la vérification. À utiliser avec prudence, car il annule l'intérêt de TypeScript.

```
let valeur: any = "bonjour";
valeur = 42;
valeur = true;
```

5. **unknown** – Type sûr pour données incertaines

Contrairement à any, TypeScript nous oblige à vérifier le type avant utilisation.

```
let valeur: unknown = "salut";

if (typeof valeur === "string") {
    console.log(valeur.toUpperCase());
}
```

LES TYPES DE BASE EN TYPESCRIPT

6. **void** – Aucun retour

Souvent utilisé dans les fonctions qui ne retournent rien :

```
function saluer(): void {  
    console.log("Salut !");  
}
```

7. **null** et **undefined**

```
let vide: null = null;  
let nonDefini: undefined = undefined;
```

En mode strict, tu dois les spécifier comme union si tu veux les autoriser :

```
let nom: string | null = null;
```

LES TYPES DE BASE EN TYPESCRIPT

8. Array – Tableaux typés

```
let nombres: number[] = [1, 2, 3];
let noms: string[] = ["Alice", "Bob"];
```

Autre syntaxe équivalente :

```
let noms2: Array<string> = ["Charlie", "David"];
```

9. tuple – Tableau à taille et types fixes

```
let personne: [string, number] = ["Alice", 30];
// personne[0] est un string, personne[1] un number
```

LES TYPES DE BASE EN TYPESCRIPT

10. **enum** – Énumérations : Permet de définir un ensemble de valeurs nommées :

```
enum Role {  
    Utilisateur,  
    Admin,  
    SuperAdmin  
}  
  
let monRole: Role = Role.Admin;  
console.log(monRole); // Affiche 1
```

Tu peux aussi donner des valeurs personnalisées :

```
enum Couleur {  
    Rouge = "rouge",  
    Vert = "vert",  
    Bleu = "bleu"  
}  
  
let c: Couleur = Couleur.Rouge;
```

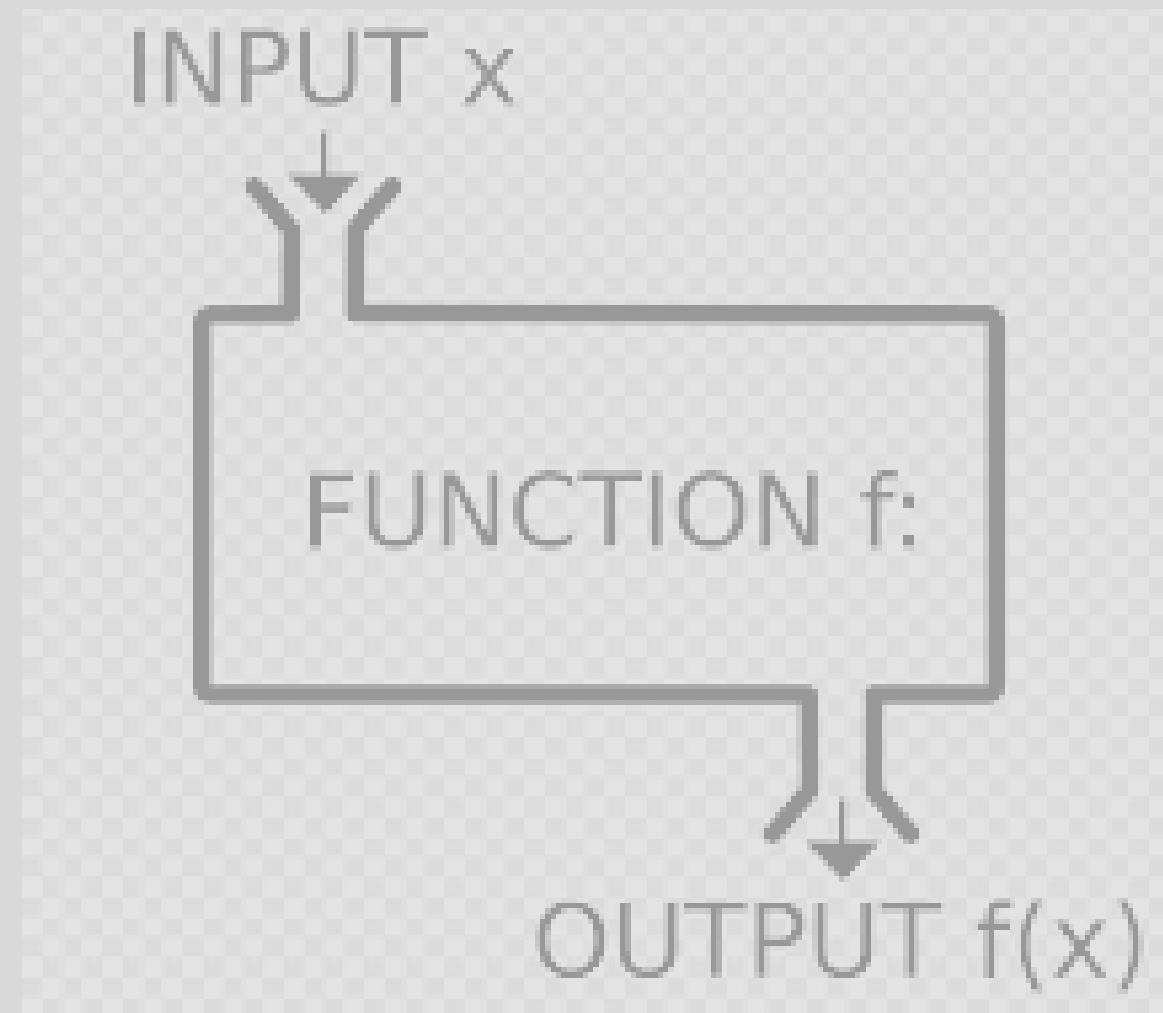
EXERCICE D'APPLICATION

1. Déclare une variable "*prenom*" de type string et donne-lui ton prénom
2. Déclare une variable "*age*" de type number
3. Crée une fonction "*retourneInfos*" qui retourne ces deux infos
4. Crée un *tableau* de nombres
5. Crée un *tuple* [string, number] pour une personne

FONCTIONS EN TYPESCRIPT

Apprendre à :

- déclarer des fonctions typées,
- utiliser des paramètres optionnels et des valeurs par défaut,
- comprendre l'importance du typage du retour.



DÉCLARATION TYPÉE SIMPLE

On précise le type des paramètres et le type de retour après `):`.

Exemple :

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

```
console.log(saluer("Alice")); // Bonjour Alice
```

- `nom: string` → le paramètre doit être une chaîne
- `: string` après la parenthèse → la fonction renvoie une chaîne

FONCTION SANS RETOUR (**VOID**)

Quand une fonction ne retourne **rien**, on utilise le type **void**.

Exemple :

```
function afficherMessage(message: string): void {  
    console.log("Message :", message);  
}
```

PARAMÈTRES OPTIONNELS (?)

On peut rendre un paramètre **facultatif** avec **?**.

Exemple :

```
function bonjour(prenom?: string): void {  
    if (prenom) {  
        console.log("Salut, " + prenom);  
    } else {  
        console.log("Salut !");  
    }  
}
```

```
bonjour();    // Salut !  
bonjour("Sam"); // Salut, Sam
```

VALEURS PAR DÉFAUT

On peut aussi définir une valeur par défaut pour un paramètre.

Exemple :

```
function multiplier(a: number, b: number = 2): number {  
    return a * b;  
}
```

```
console.log(multiplier(5)); // 10  
console.log(multiplier(5, 3)); // 15
```

FONCTIONS FLÉCHÉES (*ARROW FUNCTIONS*)

TypeScript gère aussi les fonctions fléchées :

Exemple :

```
const addition = (x: number, y: number): number => {
    return x + y;
};
```

```
console.log(addition(3, 4)); // 7
```

FONCTIONS AVEC TABLEAU OU OBJET TYPÉ

Tu peux aussi typer des fonctions qui reçoivent un tableau ou un objet :

Exemple avec tableau :

```
function somme(liste: number[]): number {  
    return liste.reduce((acc, val) => acc + val, 0);  
}
```

```
console.log(somme([1, 2, 3])); // 6
```

Exemple avec objet :

```
function afficherUser(user: { nom: string; age: number }): void {  
    console.log(` ${user.nom} a ${user.age} ans`);
```

```
}
```

```
afficherUser({ nom: "Alice", age: 25 });
```

EXERCICE D'APPLICATION

- Crée une fonction bonjour qui prend un prénom (string) et retourne une phrase de salutation.
- Crée une fonction carre qui prend un number et retourne son carré.
- Crée une fonction afficherInfos avec deux paramètres : un prénom (string) et un âge (number), dont l'âge est facultatif.
- Crée une fonction fléchée soustraction qui prend deux number et retourne le résultat.

OBJETS ET INTERFACES

OBJETS ET INTERFACES

Objectif

- Créer des objets typés en TypeScript
- Comprendre et utiliser les interfaces pour structurer le code
- Savoir faire la différence entre type et interface

1. Déclaration d'un objet typé

On peut directement définir un objet avec un type explicite :

```
let utilisateur: { nom: string; age: number } = {  
  nom: "Alice",  
  age: 30  
};
```

```
console.log(utilisateur.nom); // Alice
```

OBJETS ET INTERFACES

2. Définir une interface

Une interface permet de définir une forme (ou contrat) pour un objet. Elle rend le code plus lisible et réutilisable.

```
interface Utilisateur {  
    nom: string;  
    age: number;  
}
```

```
let u1: Utilisateur = {  
    nom: "Bob",  
    age: 40  
};
```

OBJETS ET INTERFACES

3. Propriétés optionnelles (?)

Une propriété peut être facultative avec ?:

```
interface Produit {  
    nom: string;  
    prix: number;  
    description?: string;  
}
```

```
const p1: Produit = {  
    nom: "Stylo",  
    prix: 2.5  
};
```

OBJETS ET INTERFACES

4. Propriétés en lecture seule (*readonly*)

On peut empêcher la modification d'une propriété avec **readonly** :

```
interface Compte {  
    readonly id: number;  
    solde: number;  
}
```

```
const c1: Compte = { id: 101, solde: 500 };  
// c1.id = 102; // Erreur : id est en lecture seule
```

OBJETS ET INTERFACES

5. Fonctions qui prennent un objet typé

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

```
function saluer(p: Personne): void {  
    console.log(`Bonjour ${p.nom}, vous avez ${p.age} ans.`);  
}
```

```
saluer({ nom: "Claire", age: 28 });
```

OBJETS ET INTERFACES

6. Différence simple entre interface et type

interface	type
Pour décrire un objet	Pour objets, unions, fonctions...
Extensible (extends)	Plus flexible mais non extensible
Utilisé partout en React	Plus utilisé pour les types divers

```
interface Animal {  
  nom: string;  
}
```

```
type Vehicule = {  
  marque: string;  
};
```

EXERCICE D'APPLICATION

- Crée une interface Livre avec les propriétés :
- titre (string)
- auteur (string)
- disponible (boolean)
- Crée un objet monLivre de type Livre.
- Crée une fonction afficherLivre qui prend un Livre en paramètre et affiche ses infos.

LES CLASSES EN TYPESCRIPT

LES CLASSES EN TYPESCRIPT

Objectif

- Comprendre la structure d'une classe
- Savoir créer un objet à partir d'une classe
- Utiliser les modificateurs d'accès (public, private, etc.)
- Comprendre l'héritage simple

CRÉER UNE CLASSE SIMPLE

Une **classe** est un modèle d'objet. On y définit des propriétés et des méthodes.

```
class Personne {  
    nom: string;  
    age: number;  
  
    constructor(nom: string, age: number) {  
        this.nom = nom;  
        this.age = age;  
    }  
  
    saluer(): void {  
        console.log(`Bonjour, je m'appelle ${this.nom} et j'ai ${this.age} ans.`);  
    }  
}  
  
// Créer une instance (objet):  
const p1 = new Personne("Alice", 30);  
p1.saluer(); // Bonjour, je m'appelle Alice et j'ai 30 ans.
```

MODIFICATEURS D'ACCÈS

- **public** : accessible partout (par défaut)
- **private** : accessible uniquement dans la classe
- **protected** : accessible dans la classe et ses enfants

```
class CompteBancaire {  
    private solde: number;  
  
    constructor(soldeInitial: number) {  
        this.solde = soldeInitial;  
    }  
  
    consulterSolde(): number {  
        return this.solde;  
    }  
}  
  
const compte = new CompteBancaire(1000);  
console.log(compte.consulterSolde()); // ✓ 1000  
// console.log(compte.solde); ✗ Erreur : solde est privé
```

HÉRITAGE AVEC EXTENDS

Une **classe** peut **hériter** d'une **autre** :

```
class Animal {  
  
    nom: string;  
  
    constructor(nom: string) {  
  
        this.nom = nom;  
  
    }  
  
    faireDuBruit(): void {  
  
        console.log(`${this.nom} fait un bruit`);  
  
    }  
  
}
```

```
class Chien extends Animal {  
  
    aboyer(): void {  
  
        console.log(` ${this.nom} aboie`);  
  
    }  
  
    }  
  
const rex = new Chien("Rex");  
  
rex.faireDuBruit(); // Rex fait un bruit  
  
rex.aboyer(); // Rex aboie
```

IMPLÉMENTER UNE INTERFACE

Une classe peut aussi implémenter une interface, ce qui garantit qu'elle respecte une structure :

```
interface Identifiable {  
    id: number;  
    afficherId(): void;  
}
```

```
class Utilisateur implements Identifiable {  
    id: number;  
  
    constructor(id: number) {  
        this.id = id;  
    }  
  
    afficherId(): void {  
        console.log("ID :", this.id);  
    }  
}
```

EXERCICE D'APPLICATION

- Crée une classe Voiture avec :
- une propriété marque (string)
- une propriété vitesse (number)
- une méthode accelerer() qui augmente la vitesse
- Crée une instance de Voiture et appelle accelerer() deux fois.
- Ajoute une classe VoitureSport qui hérite de Voiture et ajoute une méthode turbo() qui augmente fortement la vitesse.

LES TYPES AVANCÉS

TYPES AVANCÉS

Objectif

Découvrir les types plus flexibles que propose TypeScript pour :

- représenter plusieurs possibilités,
- combiner des types,
- restreindre les valeurs à une liste précise.

TYPE UNION (|)

Permet de dire qu'une variable peut avoir plusieurs types possibles.

```
let identifiant: string | number;
```

```
identifiant = "ABC123"; // OK
```

```
identifiant = 456; // OK
```

```
// identifiant = true; // ✗ Erreur
```

Exemple avec fonction :

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

TYPE INTERSECTION (&)

Combine plusieurs types pour exiger toutes leurs propriétés.

```
type Identifiable = { id: number };
```

```
type Nommé = { nom: string };
```

```
type Utilisateur = Identifiable & Nommé;
```

```
const user: Utilisateur = {  
  id: 1,  
  nom: "Alice"  
};
```

TYPES LITTÉRAUX ("...")

On peut limiter une variable à certaines valeurs précises :

```
let direction: "gauche" | "droite";
```

```
direction = "gauche"; // OK
direction = "droite"; // OK
// direction = "haut"; // ✗ Erreur
```

C'est souvent utilisé pour les états ou actions fixes (ex. : statut d'un utilisateur).

ALIAS DE TYPE ("TYPE")

Tu peux donner un nom à un type personnalisé avec **type** :

```
type Statut = "actif" | "inactif" | "suspendu";
```

```
let monStatut: Statut = "actif";
```

TYPEOF POUR RÉCUPÉRER UN TYPE

Si tu as une variable, tu peux en récupérer le type pour autre chose :

```
const nom = "Jean";
type NomType = typeof nom; // string
```

Utile dans les tests ou avec des objets JSON.

KEYOF POUR RÉCUPÉRER LES CLÉS D'UN TYPE

```
type Personne = {  
    nom: string;  
    age: number;  
};
```

```
type Propriete = keyof Personne; // "nom" | "age"
```

```
let propriete: Propriete = "nom"; // OK
```

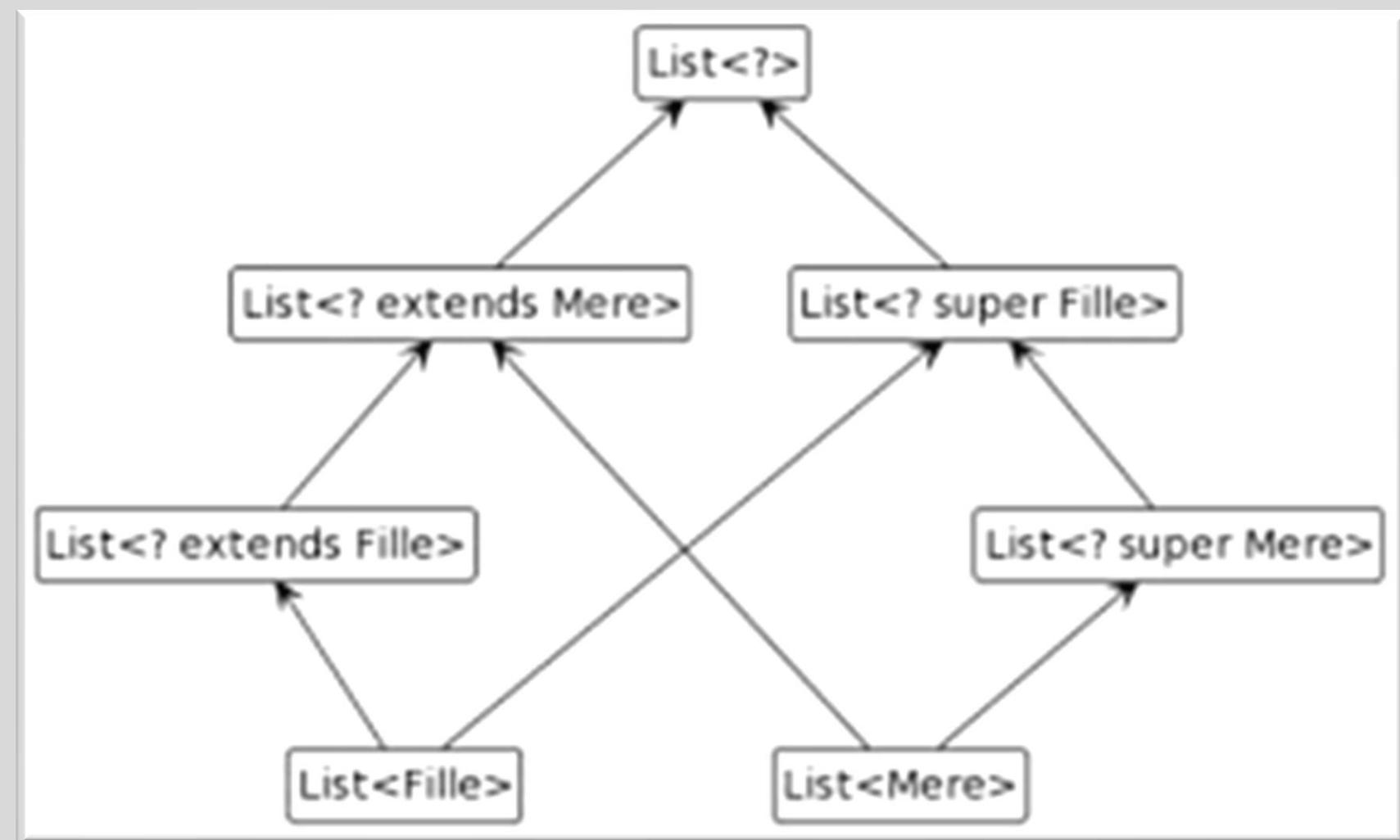
EXERCICE D'APPLICATION

- Crée un type Sexe qui accepte uniquement "homme" ou "femme"
- Crée une variable genre de type Sexe
- Crée deux types : Employé (avec id) et Profil (avec nom)
- Crée un type combiné EmployéComplet avec intersection (&)
- Déclare un objet e1 de type EmployéComplet

LA GÉNÉRICITÉ <*T*>

Objectif

Comprendre comment écrire des fonctions, types ou classes réutilisables, sans sacrifier la sécurité des types.



POURQUOI UTILISER DES GÉNÉRIQUES ?

Quand tu écris une fonction ou une structure qui doit fonctionner avec plusieurs types différents, sans perdre la vérification de types.

Sans générique (version rigide) :

```
function retournerNombre(val: number): number {  
    return val;  
}
```

Avec any (pas sécurisé) :

```
function retourner(val: any): any {  
    return val;  
}
```

let res = retourner(5); // res est de type any → pas de vérification ensuite

POURQUOI UTILISER DES GÉNÉRIQUES ?

Fonction générique simple :

```
function retourner<T>(val: T): T {  
    return val;  
}  
  
// Appels avec type explicite  
let res1 = retourner<string>("Hello");  
let res2 = retourner<number>(123);  
  
// Appels avec inférence automatique  
const texte = retourner("Bonjour"); // type string  
const nombre = retourner(42); // type number
```

Tableaux génériques:

```
function afficherTableau<T>(tab: T[]): void {  
    tab.forEach((item) => console.log(item));  
}  
  
afficherTableau<string>(["un", "deux"]);  
afficherTableau<number>([1, 2, 3]);
```

POURQUOI UTILISER DES GÉNÉRIQUES ?

Types génériques personnalisés:

Tu peux créer un type générique pour structurer des objets flexibles :

```
type Boite<T> = {  
    contenu: T;  
};
```

```
let boite1: Boite<string> = { contenu: "Texte" };  
let boite2: Boite<number> = { contenu: 42 };
```

Contraintes sur les génériques :

Tu peux limiter les types autorisés avec *extends* :

```
function longueur<T extends { length: number }>(x: T): number {  
    return x.length;  
}
```

```
longueur("texte");      // OK (string a une longueur)  
longueur([1, 2, 3]);    // OK (array a une longueur)  
// longueur(123); ✗ Erreur
```

POURQUOI UTILISER DES GÉNÉRIQUES ?

Classe générique :

```
class BoiteGenerique<T> {  
    contenu: T;  
  
    constructor(contenu: T) {  
        this.contenu = contenu;  
    }  
  
    afficher(): void {  
        console.log("Contenu :", this.contenu);  
    }  
}  
  
const b1 = new BoiteGenerique<string>("Livre");  
const b2 = new BoiteGenerique<number>(123);  
  
b1.afficher(); // Contenu : Livre  
b2.afficher(); // Contenu : 123
```

EXERCICE D'APPLICATION

- Crée une fonction inverser<T> qui prend un tableau de type T[] et retourne le même tableau à l'envers.
- Crée un type générique Paire<T, U> qui contient deux propriétés : premier et second.
- Crée une variable de type Paire<string, number>.

LES MODULES

MANIPULATION DES MODULES

Objectif

- Organiser le code TypeScript en fichiers/modules réutilisables
- Comprendre comment importer/exporter du code
- Découvrir les types externes (@types) pour les bibliothèques JavaScript

Pourquoi utiliser des modules ?

Quand un **projet grandit**, on veut :

- **séparer les fonctions**, classes ou types dans plusieurs fichiers,
- **réutiliser** des fonctions ailleurs,
- **limiter le mélange** de responsabilités dans le code.

TypeScript utilise le système de modules ES6, comme en JavaScript moderne.

EXPORTER/IMPORTER UNE FONCTION, UNE CONSTANTE, UNE CLASSE

1. Exporter

Fichier math.ts :

```
export function addition(a: number, b: number): number {  
    return a + b;  
}
```

```
export const PI = 3.14;
```

2. Importer dans un autre fichier

Fichier main.ts :

```
import { addition, PI } from "./math";  
  
console.log(addition(5, 2)); // 7  
console.log("PI =", PI); // PI = 3.14
```

Astuce : on utilise le chemin relatif (ex. ./) et l'extension .ts est omise.

EXPORT PAR DÉFAUT (DEFAULT)

Tu peux aussi exporter une valeur par défaut :

util.ts :

```
export default function direBonjour(nom: string): void {  
    console.log("Bonjour, " + nom);  
}
```

main.ts :

```
import direBonjour from "./util";  
  
direBonjour("Alice");
```

EXERCICE D'APPLICATION

- Crée un fichier calcul.ts avec une fonction multiplier(a: number, b: number): number
- Exporte cette fonction
- Dans un fichier index.ts, importe-la et appelle-la
- Compile et exécute avec Node.js

M E R C I