Mémo: TypeScript

JS est à **typage dynamique** (le type est défini à l'exécution). TS est à **typage statique** (le type est défini à l'écriture).

- TS ne modifie pas le comportement de votre code au runtime (exécution).
- TS est un linter sous stéroïdes qui vous empêche de commettre des erreurs de type *avant* même de lancer le code.

La Syntaxe de Base (Cheat Sheet)

Voici comment on écrit les types. La règle est simple : **variable: type**.

Concept	Exemple de Code	Explication
Texte	let prenom: string = "Mario";	Uniquement des guillemets.
Nombre	let vie: number = 100;	Entiers ou décimaux, c'est pareil.
Vrai/Faux	let estGameOver: boolean = false;	true ou false uniquement.
Tableau	let scores: number[] = [10, 20];	Une liste qui ne contient QUE des nombres.
N'importe quoi	let inconnu: any = "Danger";	**INTERDIT !** (Sauf cas extrême). C'est revenir au JS classique.

Les Fonctions

En TS, une fonction ne laisse rien au hasard. Elle dit exactement ce qu'elle veut en entrée et ce qu'elle rend en sortie.

```
//     (Entrée 1)     (Entrée 2)    (Sortie)
function additionner(a: number, b: number): number {
   return a + b;
}
```

Si vous essayez d'appeler additionner("chat", 10), votre éditeur soulignera "chat" en rouge. Il

attend un nombre, point barre.

Fini les vérifications manuelles du style if (typeof a !== 'number') throw Error.... TS garantit que si la fonction est exécutée, a et b SONT des nombres. Vous pouvez supprimer tout votre code défensif inutile.

L'Inférence

Parfois, vous n'avez pas besoin d'écrire le type. TS est intelligent.

```
let score = 0;
// TS devine tout seul que 'score' est un number.

score = "Dix";
// ERREUR ! TS a deviné 'number', il refuse le string.
```

Conseil : Au début, forcez-vous à écrire les types explicitement (: string) pour apprendre. Plus tard, vous laisserez l'inférence travailler.

l'Exécution

Votre ordinateur ne sait pas lire le TypeScript. Il ne comprend que le JavaScript. C'est pourquoi on utilise un outil (ts-node ou tsc) qui traduit votre code TS en JS à la volée pour l'exécuter.

Le code TS n'existe pas en prod. Tout est transpilé.

- Si vous avez une erreur de type (rouge dans VS Code), la compilation échoue (selon la config).
- Le fichier tsconfig.json est votre nouveau meilleur ami : c'est lui qui décide si vous êtes en mode "cool" (strict: false) ou en mode "pro" (strict: true). **Pour ce cours, on est en mode PRO.**

Conseils

1. **"C'est tout rouge !"** ➔ Lisez le message au survol de la souris. Souvent, vous essayez de mettre un texte dans un nombre.
2. **"Commande introuvable"** ➔ Avez-vous bien fait npm install ? Êtes-vous dans le bon dossier ?
3. **"Je ne sais pas quel type mettre..."** ➔ Demandez-vous : "Qu'est-ce que cette variable est censée contenir dans la vraie vie ?". Si c'est un prix, c'est number. Si c'est un nom, c'est string.

Pour structurer une vraie application, on utilise des outils plus puissants.

1. Les Enums

Les Enums servent à limiter les choix possibles pour une variable. C'est comme un menu déroulant obligatoire.

```
enum Statut {
  EnCours = "EN_COURS",
  Termine = "TERMINE"
}

let etat: Statut = Statut.EnCours; //  Valide
// let etat: Statut = "En pause";   //  Interdit
```

Intérêt : Évite les fautes de frappe et rend le code plus lisible.

2. Les Classes & Encapsulation

Dans une Classe (un plan de construction d'objet), on peut protéger certaines données.

- **public** : La donnée est accessible et modifiable par tout le monde.
- **private** : La donnée est enfermée dans la classe. Seules les méthodes de cette classe peuvent la toucher.

```
class Compte {
   private solde: number = 0; //  Intouchable de l'extérieur

   public deposer(montant: number) {
      this.solde += montant; //  La classe a le droit de toucher à son 'private'
   }
}
const monCompte = new Compte();
// monCompte.solde = 1000000; //  ERREUR : Propriété privée !
```

3. Les Génériques <T>

Les Génériques permettent de créer des composants qui s'adaptent à différents types, sans perdre la sécurité. Imaginez une boîte sur laquelle on colle une étiquette au moment de l'achat.

- Le **<T>** est une variable qui remplace le Type.

```
class Boite<T> {
```

```
    contenu: T;
    constructor(valeur: T) { this.contenu = valeur; }
}

// Ici, T devient 'string'
const b1 = new Boite<string>("Hello");

// Ici, T devient 'number'
const b2 = new Boite<number>(123);
```

Intérêt : On écrit la logique de la Boite une seule fois, et elle marche pour tout le monde (Textes, Nombres, Utilisateurs...) tout en restant sécurisée.