# TypeScript

## Travaux Pratiques


2024-07-17#bfa6647

# Pré-requis

### Installation

- NODE

# TP 1 : Outillage

Dans ce TP, nous allons apprendre à manipuler les différents outils pour développer une application en TypeScript.

Dans un premier temps, nous nous familiariserons avec *Node.js* et *npm*.

Ensuite, nous allons pouvoir commencer à compiler notre code en JavaScript, valider la qualité de notre code et enfin automatiser avec des taches NPM.

### Initialiser le répertoire tp1

Créez un nouveau répertoire `tp1` dans le dossier `/home/ubuntu/workspaces`.

Ouvrez VS Code dans ce dossier puis ouvrez un nouveau terminal depuis VS Code.

Tapez les commandes suivantes:

```
node --version
npm --version
```

### Installation de node et npm (si nécessaire)

Si Node et npm ne sont pas installés, vous pouvez les installer au moyen de nvm.

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh | bash
```

Ouvrez ensuite un autre terminal et exécutez les commandes suivantes:

```
nvm install --lts
npm --version
node --version
```

### Initier un fichier package.json

Exécutez la commande pour initier un nouveau projet.

```
npm init -y
```

Un fichier `package.json` est créé dans votre répertoire.

### Compiler en Javascript

Pour compiler du TypeScript nous avons besoin d'avoir la CLI TypeScript.

- Installez-le avec npm :

```
npm install --save-dev typescript
```

- La configuration Typescript est définie dans un fichier `tsconfig.json`. Créer ce fichier comme suit:

```
{
  "compilerOptions": {
    "outDir": "./dist",
    "module": "commonjs",
    "target": "ES2019",
    "strict": true,
    "lib": ["dom", "ESNext"]
  }
}
```

- Copiez ce code dans un fichier nommé `src/app.ts` :

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

const user = "Zenika";

console.log(sayHello(user));
```

- Vous serez peut-être obligé d'installer le fichier de définitions pour NodeJS. Ne vous inquiétez pas, nous allons aborder ce sujet dans la suite de la formation.

```
npm install --save-dev @types/node
```

- Compilez le code en JavaScript avec `npx tsc` et exécutez le script via `node dist/app.js`.

Pour ne pas répéter systématiquement les commandes précédentes pour exécuter notre code TypeScript nous allons installer *ts-node*. Ce module permet d'exécuter le code TypeScript à la volée.

```
npm install --save-dev ts-node
```

Ensuite nous allons rajouter 2 tâches npm pour lancer notre application:

```
{
  "scripts": {
    "start": "ts-node src/app.ts",
```

```
    "build": "tsc"
  }
}
```

Par la suite vous avez juste à exécuter la commande `npm run start`.

### Watch mode (optionnel)

En complément, vous pouvez installer le module npm *nodemon* qui permet de relancer une tache npm dès que votre code est modifié. Pratique !

```
npm install --save-dev nodemon
```

Ensuite nous allons rajouter une nouvelle tâche npm :

```
{
  "scripts": {
    "start:dev": "nodemon -e ts -w ./src -x npm start"
  }
}
```

et la lancer avec `npm run start:dev`

### Linter

Maintenant, nous allons configurer l'analyse statique de code au moyen de typescript-eslint

```
npm -g i eslint-cli
npm install --save-dev @typescript-eslint/parser @typescript-eslint/eslint-
plugin eslint
```

Créez le fichier de configuration `.eslintrc.json`:

```
{
  "extends": ["eslint:recommended", "plugin:@typescript-eslint/recommended"],
  "parser": "@typescript-eslint/parser",
  "plugins": ["@typescript-eslint"],
}
```

Modifiez votre `package.json` afin d'ajouter un script de lint:

```
{
  "scripts": {
    "lint": "eslint ./src",
  }
}
```

Enfin exécutez `npm run lint` pour vérifier que la commande fonctionne.

## Les Tests

Nous allons écrire nos premiers tests unitaires. Cette partie vient au tout début de la formation, afin de vous laisser la possibilité d'écrire de nouveaux tests pour les fonctionnalités que nous allons implémenter dans les TPs suivants.

Le test que nous allons écrire permettra de s'assurer que les fonctionnalités définies dans une classe `HelloWorld` sont bien celles attendues.

Dans cette classe, nous avons une méthode `sayHello`, qui retourne :

- lorsqu'elle est appelée sans paramètre, **Hello World!**.
- lorsqu'elle est appelée avec un paramètre, *par exemple Zenika*, **Hello Zenika!**.

Nous allons nous assurer que la méthode `sayHello` retourne bien la chaîne de caractères désirée.

- l'implémentation de la classe que nous allons tester est la suivante. Recopiez cette classe dans un fichier `src/tp1.ts`

```
export class HelloWorld {
  public sayHello(name = "World"): string {
    return `Hello ${name}!`;
  }
}
```

Dans le code que vous venez de reprendre, plusieurs fonctionnalités du langage TypeScript sont utilisées (classe, paramètre par défaut, String Interpolation, fichiers de définition...), mais seront seulement expliquées dans les prochaines parties de cette formation.

- Installez le module Jest avec son support pour **typescript**

```
npm install -g jest
npm install --save-dev jest @types/jest ts-jest
```

Créez un fichier `jest.config.ts` de cette manière:

```
export default {
    rootDir: "./src",
    testEnvironment: "node",
    transform: {
        "^.+\\.tsx?$": "ts-jest",
    },
};
```

- Dans le répertoire `src/`, créez un fichier `tp1.spec.ts` (ou aller le chercher dans les **ressources**), en respectant la structure suivante :

```
import { HelloWorld } from "./tp1";
```

```
describe("TP1", () => {
  let helloWorld: HelloWorld;

  beforeEach(() => {
    helloWorld = new HelloWorld();
  });

  it("should say hello to Zenika", () => {
    expect(helloWorld.sayHello("Zenika")).toBe("Hello Zenika!");
  });

  it("should say hello to World", () => {
    expect(helloWorld.sayHello()).toBe("Hello World!");
  });
});
```

### Exercices

- Dans le fichier `package.json`, ajoutez le script `test` :

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Exécutez la commande `npm run test` afin de vérifier le bon fonctionnement de Jest sur votre projet.

# TP2 : Types et Inférence de types

Nous allons à présent commencer à développer en *TypeScript*. Nous allons tout d'abord nous habituer aux types primitifs proposés par le langage : `string`, `number`, `function`, `array` et `tuple`.

Une suite de tests unitaires est déjà disponible, permettant de tester le résultat de ce TP : `src/tp2.spec.ts`, le fichier est dans les **ressources**.

Copiez ce fichier dans votre dossier du TP précédent afin de pouvoir executer les tests.

Vous pouvez les lancer à tout moment en utilisant la commande (définie dans le TP précédent) :

```
npm test
```

**Astuce :** vous pouvez lancer la commande `npm run test -- --watchAll` pour laisser tourner les tests pendant que vous modifiez votre code.

## La méthode returnPeopleAndLength

- Dans un fichier `src/tp2.ts`, créez et exportez une méthode `returnPeopleAndLength`
  - qui prend en paramètre un tableau de `string`
  - qui retourne un tableau de tuples `[string, number]`
  - un tuple contenant un élément du tableau passé en paramètre et la taille (en nombre de caractères) de cet élément
- Le paramètre de la méthode `returnPeopleAndLength` possède une valeur par défaut égale à : `['Miles', 'Mick']`
- Exécutez les tests unitaires associés, à présent seulement 4 tests doivent échouer.

## La méthode displayPeopleAndLength

- Dans le fichier `src/tp2.ts`, créez et exportez une méthode `displayPeopleAndLength`
  - qui prend en paramètre un tableau de `string` optionnel
  - ne retourne aucune donnée
  - exécute la méthode `console.log` pour chaque élément retourné par la méthode `returnPeopleAndLength`
  - la chaîne de caractères affichée doit avoir cette structure : 'Miles contient 5 caractères'
  - utilisez le système d'*interpolation*
- Exécutez de nouveau les tests unitaires pour vérifier qu'aucune régression n'a été ajoutée.

- Nous allons à présent utiliser les types `enum` :
    - Définissez une enum `NumberToString` avec les valeurs suivantes : `zero`, `un`, `deux`, `trois`, `quatre`, `cinq`, `six`, `sept`, `huit`, `neuf`
    - Dans la méthode `displayPeopleAndLength`, ajoutez un paramètre optionnel de type `boolean` appelé `literal`
    - Si ce paramètre est égal à `false` ou `undefined`, le code implémenté précédemment sera exécuté
    - Si ce paramètre est égal à `true`, nous allons
        - filtrer le tableau passé en paramètre pour n'afficher que les chaînes de caractères dont la taille est inférieure ou égale à 9
        - la chaîne de caractères affichée aura la structure suivante : `'Miles contient cinq caractères'`
- Exécutez de nouveau les tests unitaires. Ils doivent tous s'exécuter avec succès.

# TP 3 : Classes, Interfaces, Héritage et Générique

Nous allons, à travers ce TP, manipuler tous les concepts de classes, interfaces, héritage et génériques proposés par TypeScript. Des tests unitaires sont déjà disponibles, permettant de tester les différentes étapes de ce TP : `src/tp3.spec.ts`.

Vous pouvez les lancer à tout moment en utilisant la commande définie dans les TPs précédents.

Toutes les entités seront à créer dans un nouveau fichier `src/tp3.ts`. Le découpage en module sera expliqué dans le prochain chapitre.

- Définissez une `enum Music` contenant les clés `JAZZ` et `ROCK`
- Définissez une classe `Musician` ayant les propriétés suivantes :
  - firstName (type string)
  - lastName (type string)
  - age (type number)
  - style (type `Music` ou `undefined`)
- Implémentez, dans la classe `Musician` une méthode `toString` qui doit, grâce au système *d'interpolation* retourner une chaîne de la forme `firstName lastName`.
- Définissez deux nouvelles classes `JazzMusician` et `RockStar` qui doivent hériter de la classe `Musician`.

Ces classes permettront de définir la propriété `style` dans leur constructeurs en utilisant les valeurs `Music.JAZZ` et `Music.ROCK`.

- Modifiez le scope de la propriété `style`. Elle doit être définie avec le scope `private`. Implémentez les accesseurs pour cette propriété (get/set).
- Modifiez la méthode `toString` pour qu'elle retourne une chaîne de caractère de la forme `firstName lastName plays style` lorsque `style` est défini.
- Créez une nouvelle classe `Album`. Cette classe aura une propriété `title` de type `string` et une méthode `toString` qui retournera title.
- Ajoutez à la classe `Musician` une propriété privée de type `Album[]` par défaut à `[]` et implémentez ses accesseurs (get/set).

Nous allons également définir une méthode `addAlbum(album: Album)`. Cette méthode sera définie dans une interface `IMusician` qui sera implémentée par la classe `Musician`.

- Pour terminer ce TP, nous allons implémenter une fonction générique `display` (à l'extérieur de la classe `Musician`) :

- Créez une interface `ElementToString` qui a une méthode `toString` retournant une string.
- La fonction `display` prendra en paramètre un tableau d'objets de type générique qui héritera de `ElementToString`.
- Pour chaque élément du tableau, nous afficherons sur la console le retour de la méthode `toString` pour l'objet courant.

🎉 Tous les tests unitaires doivent passer.

# TP 4 : Les Modules

Nous allons à présent découper la solution du TP précédent en différents modules, afin d'avoir du code simple et réutilisable.

- Copiez le contenu du fichier `src/tp3.ts` dans un nouveau fichier `src/tp4.ts`.
- Copiez le contenu du fichier `src/tp3.spec.ts` dans un nouveau fichier `src/tp4.spec.ts`.
- Créez un nouveau fichier `src/Log.ts`.
    - Ce fichier exporte une méthode `log` ayant la signature suivante : `<T>(value: T): void;`.
    - Dans cette méthode, vous devez appeler la méthode `log` de l'objet `console`.
- Créez un nouveau fichier `src/Display.ts` qui exposera une seule fonction (utilisation du mot clé `default`)
    - Cette méthode correspond à la méthode `display` implémentée dans le TP précédent.
    - Au lieu d'appeler directement la méthode `console.log`, vous devez faire appel à la méthode `log` du fichier `Log.ts`.
- Exportez tous les objets créés dans le TP précédent (`IMusician`, `Musician`, `Music`, `Album`, `RockStar` et `JazzMusician`), dans différents fichiers :
    - `Musician.ts`
    - `JazzMusician.ts`
    - `RockStar.ts`
    - `Album.ts`
    - `Utils.ts` pour y inclure les fonctions utilitaires
- Ajoutez une méthode `swing` à la classe `JazzMusician` et une méthode `shout` à la classe `RockStar`. Ces deux méthodes affichent respectivement `I'm swinging!` et `I'm shouting!` (utiliser la méthode `log` du fichier `Log.ts`)
- Dans le fichier `tp4.spec.ts`, les imports sont normalement cassés. Corriger les imports pour qu'ils utilisent l'ensemble des fichiers créés. Tous les tests doivent passer.
- Enfin, dans le fichier `src/tp4.ts` :
    - Utilisez la méthode `log` du fichier `Log.ts` pour afficher un message d'accueil : 'Bienvenue dans ma première application TypeScript.'
    - Créez une liste de 2 musiciens (un `JazzMusician` et un `RockStar`).

- Ajoutez 2 albums au `JazzMusician`.
- Affichez la liste des musiciens et la liste des albums ( `Display.ts` ).
- Bouclez sur la liste des musiciens et affichez `I'm swinging!` / `I'm shouting!` selon leur type.
- Compilez votre code et exécutez-le dans un environnement *node* à l'aide de *ts-node* : vous devez voir le message d'accueil, les deux listes et ce que les musiciens ont à dire.

# TP 5 : Fichiers de définitions

Nous allons dans ce TP, intégrer la librairie *lodash* afin de bénéficier de la méthode `each` pour itérer sur une collection.

Nous allons utiliser cette méthode à la place des boucles `for` / `forEach`

- Installez via *npm* la librairie *lodash*
- Importez le module téléchargé dans le fichier `src/tp5.ts` de votre application.

```
import * as _ from 'lodash';
```

- Le système d'autocomplétion de votre IDE ne doit pas fonctionner pour cette librairie. Ce problème est normal, car le compilateur ne connaît pas la structure de la librairie `lodash`.

```
npm install --save-dev @types/lodash
```

- Remplacez la boucle `for` / `forEach` utilisée dans le fichier `src/tp4.ts`, par cette toute nouvelle méthode. Observez l'autocomplétion qui permet d'accéder aux informations que nous venons de définir. Faites de même pour la boucle contenue dans le fichier `src/Display.ts`

# TP 6 : Les Décorateurs

Dans ce TP, nous allons créer plusieurs décorateurs permettant d'améliorer le code que nous venons d'implémenter dans les TPs précédents :

- Explication :
  - Un décorateur `@logged` que nous allons utiliser sur une méthode. Elle permettra d'utiliser la méthode `log` du fichier `Log.ts` avec en paramètre la valeur de retour.
  - `@StyleMusic`, un décorateur paramétrable qui, utilisé sur une classe, permettra de définir la propriété `style`
- Travail à réaliser :
  - Dans la configuration du compilateur :

```
{
  "experimentalDecorators": true,
  "emitDecoratorMetadata": true
}
```

- Dans le fichier `Log.ts`, créez un décorateur `@logged`, qui sera une annotation qui devra être utilisée sur une méthode
- Cette méthode permettra d'appeler la méthode `log` du module `utils` en utilisant **le résultat** de la méthode sur laquelle nous avons utilisé le décorateur.
- Modifiez les implémentations des méthodes `swing` et `shout` des classes `JazzMusician` et `RockStar`. Elles doivent à présent retourner la chaîne de caractères précédemment passée en paramètre de la méthode `log`.
- Ajoutez le décorateur `@logged` sur ces méthodes.
- Vérifiez que les tests unitaires s'exécutent toujours avec succès.

# TP 7 : Alias, Omit et Pick

Dans ce TP nous allons manipuler quelques fonctions TS utilitaires.

- Créez une liste de 2 musiciens (un `JazzMusician` et un `RockStar`).
- Créez un alias `MusicianData` qui représente les propriétés d'un musicien en utilisant l'utilitaire `Omit`.
- Créez un alias `OnlyFirstAndLastName` qui représente le prénom et le nom d'un musician (utiliser `Pick`).

# TP 8 : Unknown and user-defined type guards

Dans ce TP nous allons manipuler le type `unknown` et les guard "custom" afin de valider les données que nous avons.

- Créez un fichier `tp8.ts` et y mettre la variable suivante :

```
const dataInString = `[
    {
        "firstName": "Benoit",
        "lastName": "Vasseur",
        "age": 29,
        "style": "JAZZ"
    },
    {
        "firstName": "Johnny",
        "lastName": "Hallyday",
        "age": 74,
        "style": "ROCK"
    },
    {
        "firstName": "Rocky",
        "lastName": "Balboa",
        "age": 32,
        "style": "??"
    }
]`;
```

- Parsez la chaine de caratère et afficher les données brutes ( `JSON.parse()` ). Typez le résultat du parse à `unknown[]` .
  Le but ici est de valider les données afin de créer soit un JazzMusician soit une RockStar, puis d'invoquer la méthode `swing` ou `shout` .

Vous pouvez suivre ce plan route :

1. Créer un type ( `MusicianData` ) qui représente les données sérialisées d'un musicien valide.
2. Ecrire la guard `isValidMusicianData` qui respecte la signature suivante : `function isValidMusicianData(o: unknown): o is MusicianData` . Rappel : la fonction doit renvoyer true si l'argument est bien du bon type.
3. Parser les données => garder les données valides => instantier un JazzMusician ou une RockStar (Aide : Parse, filter puis map)
4. Parcourir le tableau de `JazzMusician | RockStar` et invoquer la méthode `swing` ou `shout` suivant le type de l'objet.

## logo: Logo Zenika Horizontal_White.png theme: Zenika Theme v1

# Validation des acquis - TYPESCRIPT

## Rappel des Objectifs :

- Discerner la différence en JavaScript et TypeScript
- Appréhender les bases du langage (types, classes, interfaces, décorateurs, transpilation)
- Écrire des applications en s'aidant de TypeScript
- Utiliser les différents outils de l'écosystème TypeScript

## Discerner la différence en JavaScript et TypeScript

### Laquelle de ces affirmations est vraie ?

1. ☐ Votre navigateur peut lire et interpreter du TypeScript.
2. ☐ TypeScript doit être transpilé en JavaScript avant d'être utilisé dans le Web ou sur Node.
3. ☐ Tout programme TypeScript est lisible par un moteur JavaScript nativement.
4. ☐ Seul le nom entre JavaScript et TypeScript change, sinon le code est le même.
5. ☐ TypeScript ne s'utilise qu'avec un framework.

### TypeScript est un superset de JavaScript, quelle affirmation est fausse ?

1. ☐ Tout programme JavaScript est compatible avec TypeScript (comme les librairies externes en JS).
2. ☐ TypeScript ne prend pas en charge la programmation orientée objet.
3. ☐ TypeScript ajoute des fonctionnalités supplémentaires à JavaScript.
4. ☐ TypeScript peut cibler différentes versions ECMAScript au moment de la transpilation.

## Appréhender les bases du langage (types, classes, interfaces, décorateurs, transpilation)

### Type : Quelle est la différence entre "String" et "string" ou entre "Number" et "number" ?

1. ☐ Aucune différence.
2. ☐ Les premiers sont des types de JavaScript et les deuxièmes sont les types primitifs de TypeScript.
3. ☐ Les premiers servent pour les fonctions et les deuxièmes pour les variables uniquement.
4. ☐ Les premiers sont utilisés uniquement dans les fichiers `.tsx` et les deuxièmes peuvent être utilisés partout.

**Type : Quels objets littéraux ci-dessous sont correctement typés avec `interface Person { name: string; age: number; genre: "male" | "female" }` ?**

- [ ]
```ts
const marc: Person = {
  name: "Assin",
  age: 42,
  genre: "male",
}
```
- [ ]
```ts
const daisy: Person = {
  name: "Draté",
  genre: "female",
}
```
- [ ]
```ts
const nobody: Person = {
  name: "",
  age: -100,
  genre: "male",
}
```
- [ ]
```ts
const al: Person = {
  name: "Batrosse",
  age: .5,
  genre: "oiseau" as unknown as "male",
}
```

**Type : Quel type est le plus sûr/robuste à utiliser quand on ne connait pas le type d'un objet ?**

1. [ ] `any`
2. [ ] `all`
3. [ ] `unknown`
4. [ ] `undefined | null`

**Classes : TypeScript dispose de notion de setters/getters**

1. [ ] Oui
2. [ ] Non

**Classes : Une classe peut hériter de plusieurs autres classes ?**

1. [ ] Oui
2. [ ] Non

### Classes : Comment accéder à `staticField`, un attribut `static` au sein de la classe `ClassName` ?

1. ☐ `this.staticField`
2. ☐ `ClassName.staticField`
3. ☐ `this.static.staticField`
4. ☐ `#staticField`

### Classes : Est-ce que cet objet littéral `{name: "Jean", age: 42}` peut-être typé avec cette classe `class Person {public name:string; public age: number}` ?

1. ☐ Oui
2. ☐ Non

### Interfaces : Parmi ces phrases, lesquelles sont vraies ?

1. ☐ Deux interfaces peuvent avoir le même nom.
2. ☐ Deux interfaces peuvent définir exactement la même structure.
3. ☐ Une interface peut utiliser une autre interface pour typer un attribut.
4. ☐ Une interface peut définir la signature d'une fonction.

### Interfaces : Une interface peut étendre une autre interface ?

1. ☐ Oui
2. ☐ Non

### Interfaces : Une interface peut être générique ?

1. ☐ Oui
2. ☐ Non

### Décorateurs : Sur quel élément ne pouvons-nous pas placer un décorateur ?

1. ☐ Les classes
2. ☐ Les méthodes
3. ☐ Les paramètres d'une fonction
4. ☐ Les attributs d'une classe
5. ☐ Les boucles `for/while`

### Décorateurs : Quelle est la limite d'un décorateur ?

1. ☐ Les décorateurs ne peuvent pas prendre d'arguments.
2. ☐ Les décorateurs ne peuvent changer ni récupérer la valeur d'une propriété.
3. ☐ Plusieurs décorateurs ne peuvent pas être placés sur le même objet/fonction/paramètre.

4. ☐ Je ne sais pas

### Transpilation : Qu'est-ce que la transpilation en TypeScript ?

1. ☐ Le processus de conversion du code TypeScript en code JavaScript.
2. ☐ Le processus de vérification statique du code TypeScript.
3. ☐ Le processus de compression du code JavaScript généré par TypeScript.
4. ☐ Le processus d'exécution du code TypeScript directement dans un navigateur.

### Transpilation : Les interfaces sont enlevées complétements du code une fois la transpilation vers JS effectuée.

1. ☐ Oui
2. ☐ Non

### Transpilation : Quel(s) outil(s) peut(peuvent) être utilisé(s) pour effectuer la transpilation TypeScript ?

- ☐ TypeScript Compiler (tsc)
- ☐ Babel
- ☐ Webpack
- ☐ Node.js

## Écrire des applications en s'aidant de TypeScript

### Dépendances : Quelle(s) technique(s) permet(permettent) d'utiliser une librairie JS en TypeScript ?

1. ☐ Utiliser une librairie déjà écrite en TypeScript.
2. ☐ Utiliser `require` pour importer la librairie JS dans un fichier TypeScript.
3. ☐ Transpiler la librairie JS en TypeScript avant de l'utiliser.
4. ☐ Avoir un fichier de définition dans le code source, lu par la configuration via `include`.
5. ☐ Avoir une dépendance de type `@types/<librairie-js>` dans le package.json.
6. ☐ Il n'est pas possible de le faire.

### Général : Que permet TypeScript sur l'écriture d'une application ?

1. ☐ Garantir la compatibilité avec tous les navigateurs web.
2. ☐ Détecter les erreurs de code plus rapidement (vérification statique).
3. ☐ Optimiser les performances de l'application.
4. ☐ Réduire la taille des fichiers JavaScript générés.

### Général : Peut-on avoir des fichiers JS dans un projet en TypeScript ?

1. ☐ Oui
2. ☐ Non

### Général : Quand écrire des types dans une application ?

- ☐ Il faut en écrire le plus possible, dans chaque retour de fonctions, pour chaque variable.
- ☐ Lors de la définition des modèles de données.
- ☐ Pour servir de documentation, avec parcimonie.
- ☐ Je ne sais pas.

### Migration : Puis-je migrer progressivement une ancienne application JavaScript en TypeScript ?

1. ☐ Oui
2. ☐ Non

## Utiliser les différents outils de l'écosystème TypeScript

### Outils : Trouver l'intrus

1. ☐ Eslint
2. ☐ CoffeeScript
3. ☐ Jest / Vitest
4. ☐ TSC
5. ☐ Prettier

### Package Manager : Lequel n'est pas pour l'écosystème JS ?

1. ☐ NPM
2. ☐ YARN
3. ☐ Maven
4. ☐ PNPM

### Configuration : Quel est le nom du fichier de configuration de TypeScript ?

1. ☐ tsconfig.json
2. ☐ package.json
3. ☐ config.ts
4. ☐ typescript.config.js

### Configuration : Quel est le nom de l'option qui permet d'exclure des fichiers de la transpilation ?

1. ☐ ignore
2. ☐ exclude
3. ☐ no-transpilation
4. ☐ Je ne sais pas

### Configuration : À quoi servent les sources map ?

- ☐ Permettre de debugger plus facilement en associant le code source TypeScript au code JavaScript transpilé.
- ☐ Créer une carte de votre code pour mieux s'y retrouver.
- ☐ Compresser le code JavaScript généré pour réduire sa taille.
- ☐ Sécuriser le code JavaScript en empêchant les utilisateurs de voir le code original.