TypeScript : une surcouche à JavaScript

Introduction

JavaScript est un langage :

- Interprété : il n'y a aucune phase de compilation, le code n'est pas vérifié avant son exécution.
- Faiblement typé : une variable n'est pas conditionnée à un seul type de données : il est possible de changer son type lors de l'exécution du programme.

Ces caractéristiques ont des avantages mais aussi des inconvénients :

- Nécessité de tester très rigoureusement son code
- Pas d'aides lors du développement
- Apparition de potentiels bugs au runtime

Pour pallier à ces inconvénients, il est possible :

- d'utiliser des librairies type flow (un vérificateur de types statique)
- d'utiliser un ***linter*** sur son projet (le plus connu : ESLint -voir bibliographie-) afin d'avoir une remontée d'alertes sous forme de warning sur son projet.

Une solution plus complète consiste à développer en TypeScript.

Introduction

TypeScript est une surcouche du langage JavaScript développée par Microsoft.

Le principe global du TypeScript est de typer toutes les variables.

Néanmoins, utiliser du TypeScript ne se résume pas simplement à cela.

Le langage vise à apporter un certain nombre d'améliorations vis à vis du JavaScript, de la programmation orientée objet à la programmation générique par exemple.

Lorsque l'on développe en TypeScript (extension de fichier conventionnelle : .ts) on doit utiliser un compilateur qui traduira finalement notre code en JavaScript.

On parle d'ailleurs plutôt de transcompilation, et non de compilation, puisque l'on traduit un code écrit dans un langage en un code d'un langage du même niveau (à l'inverse de la compilation où l'on passe d'un langage lisible humainement à un langage bas niveau).

TP JavaScript

Créez une page web permettant de calculer le résultat d'une addition entre deux champs de type nombre.

Ecrivez le code JavaScript permettant de calculer l'addition (créez une fonction dont l'objet est d'additionner deux nombres) et d'afficher le résultat.

Installation TypeScript

Installez typeScript via la ligne de commande :
npm install -g typescript

TP TypeScript

Créez un nouveau projet en reprenant le TP précédent.

Cette fois-ci, écrivez votre code dans un fichier .ts et ajoutez à vos nombres le type number :

```
function calculerSomme(operande1: number, operande2: number)
```

Utilisez la commande tsc <mon_fichier.ts> pour générer votre fichier JS.

TP TypeScript - Observations

TypeScript nous alerte sur les deux points suivants :

- Lorsque l'on récupère les éléments du DOM, il peut s'agir de n'importe quel type de noeud.
  La propriété value n'est donc pas forcément accessible. TypeScript nous invite explicitement à vérifier que l'on travaille bien avec une balise de type input.

- Notre fonction d'addition prend en paramètre 2 nombres, et la valeur récupérée de notre input est un string. Il nous force donc à faire la conversion, ce qui nous prémunit d'un bug à l'exécution si l'on n'avait pas fait attention.

→ C'est un exemple simple, mais dans une application plus complexe, le fait de typer explicitement nos variables nous aidera grandement à ne pas écrire d'erreurs (par exemple, accéder à une propriété d'un objet en l'écrivant mal).

Npm

Npm - Généralités

Npm est un **gestionnaire de dépendances.**

L'acronyme signifie "Node Package Manager".

Historiquement utilisé sur des projets Node (ie. JS hors Web), il l'est désormais également sur presque tous les projets de développement Web, y compris Front.

Tous les grands frameworks Front utilisent Node & Npm, car leur outillage est écrit en JS et que l'on utilise Npm pour gérer les dépendances du projet.

Lorsque l'on parle de NPM, on peut évoquer :

- Son site web
- Son registry
- Sa CLI (command line interface)

Npm - Généralités

Le principe général de NPM est de mettre à disposition une gigantesque base de données (le **registry** npm) contenant des librairies JS publiques ou privées.

Chaque librairie doit suivre le versionning **semver**, dont le principe général est le suivant :

- Un numéro de version est sous la forme MAJOR.MINOR.PATCH
- On incrémente le numéro :
  - Majeur lorsque l'on a un changement d'API qui introduit des incompatibilités avec le code utilisant la librairie dans sa version précédente ;
  - Mineur lorsque l'on ajoute de nouvelles fonctionnalités rétro-compatibles avec la version précédente ;
  - Patch lorsque l'on corrige des bugs qui n'introduisent pas d'incompatibilités.

Npm - Généralités

Les intérêts de NPM sont :

- De ne plus avoir besoin d'aller télécharger soi même ses dépendances
- De bien isoler ses sources de ses dépendances (on ne versionne **jamais** une librairie)
- De gérer les dépendances et les incohérences de versions
- D'être certain de récupérer les librairies dans les bonnes versions
- D'être certain d'avoir téléchargé correctement nos librairies (pas de fichiers corrompus)

En bref… l'époque du téléchargement manuel de nos librairies est révolu :)

Npm - En pratique

Dans un projet utilisant NPM, on retrouve deux fichiers principaux :

- package.json
- package-lock.json

Le fichier package.json contient :

- Des informations générales sur notre projet
- Des scripts (alias de commandes)
- La liste des dépendances de notre projet (on peut indiquer des dépendances de production, de développement, ou des dépendances transitives)

Le fichier package-lock.json contient des métadonnées issues de l'installation des packages.

Il existe également un répertoire **node_modules** qui contiendra l'ensemble des librairies téléchargées via npm sur un projet. **Ce dossier doit être ignoré par votre gestionnaire de sources.**

Npm - En pratique

Les commandes principales de Npm sont :

- npm init
- npm install
- npm remove
- npm pack / npm publish
- npx <nom d'un package npm>

Exemples d'installation de dépendances :

- npm install [--production] => installe toutes les dépendances d'un projet
- npm install @angular/core
- npm install @angular/core@latest => équivalent à la commande précédente
- npm install @angular/core@9.1.0 => permet d'obtenir une version spécifique
- npm install -g @angular/cli => Installation d'un package au niveau OS
- npx create-react-app helloworld => Exécution d'un package (ici init. d'une app react) sans l'installer
- npm install typescript --save-dev => Installation d'un package qui servira uniquement côté développement (ne sera pas inclus lors d'un packaging de production)

Npm - En pratique

Lorsque l'on ajoute une dépendance à notre projet, on peut contrôler la plage de versions autorisées lors de l'installation des dépendances sur un autre poste / répertoire (indispensable lorsque l'on travaille en équipe).

Voici les différentes options :

- Fixer une version :

```
"@angular.core": "9.1.0"
```

- Fixer le numéro de version majeur :

```
"@angular.core": "^9.1.0"
```

Autorise les versions 9.1.0 à 9.X.X

- Fixer les numéros de version majeur & mineur :

```
"@angular.core": "~9.1.0"
```

Autorise les versions 9.1.0 à 9.1.X

Il existe des variantes ("<x.x.x", "x.x.x - x.x.x").

Npm - En pratique

Lorsque l'on installe un package, npm récupère la dernière version autorisée de celui-ci en fonction des règles établies dans le fichier package.json.

Problème : deux développeurs peuvent se retrouver avec des versions différentes des librairies sur un même projet.

Cas simple : une dépendance dont vous avez spécifié une latitude sur le numéro de patch ou mineur.

Plus vicieux : vous avez spécifié des numéros de version précis, mais une dépendance de vos dépendances a été mise à jour sans que vous ne le sachiez.

Npm - En pratique

As an example, consider package A:

```
{
  "name": "A",
  "version": "0.1.0",
  "dependencies": {
    "B": "<0.1.0"
  }
}
```

package B:

```
{
  "name": "B",
  "version": "0.0.1",
  "dependencies": {
    "C": "<0.1.0"
  }
}
```

and package C:

```
{
  "name": "C",
  "version": "0.0.1"
}
```

Exemple extrait de la documentation NPM

Après un npm i :

```
A@0.1.0
`-- B@0.0.1
    `-- C@0.0.1
```

Si B est mis à jour en 0.0.2, une nouvelle installation donnera l'arbre de dépendances suivant :

```
A@0.1.0
`-- B@0.0.2
    `-- C@0.0.1
```

Npm - En pratique

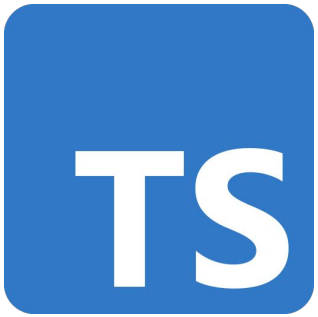
Pour répondre à ces problématiques, npm génère un fichier package-lock.json

Ce fichier a pour but de figer (d'où "lock") l'arbre de dépendances téléchargées lors d'un npm install.

De cette manière lorsqu'un développeur clone un repository contenant ce fichier package-lock on est certains de télécharger exactement le même arbre de dépendances.

Ce fichier doit être versionné et mis à jour régulièrement (à voir en fonction de votre politique projet).

TypeScript - Bases du langage



Nous allons désormais aborder quelques bases du TypeScript :

- Types primitifs
- Tuples
- Énumérés
- Any
- Union
- Type littéral
- Alias
- Retours de fonctions
- Function

Types primitifs et généralités

En TypeScript, le type d'une variable est déclaré ou inféré :

```
let unNombre: number = 10; //Type déclaré
let unAutreNombre = 10; //Type inféré
unAutreNombre = "test"; //Le type de la variable est bien number
```

Généralement on utilise l'inférence de types lorsque l'initialisation est immédiatement consécutive à la déclaration de la variable. On prendra pour bonne habitude de typer explicitement toutes les variables non initialisées immédiatement (exemple : paramètres d'une fonction).

TypeScript et JavaScript fonctionnent de la même manière pour les types :

- number
- string
- boolean

Types primitifs et généralités

Lorsque l'on manipulera des objets, TypeScript va inférer ses propriétés :

```
let etudiant = {
    prenom : "Pierre",
    nom : "Dupont"
};

etudiant.nomComplet = "Pierre Dupont";
```

```
any
Property 'nomComplet' does not exist on type '{ prenom: string; nom: string; }'. ts(2339)
Peek Problem (Alt+F8)    No quick fixes available
```

L'objectif est de s'assurer de l'existence des propriétés / méthodes auxquelles on accède.

Types primitifs et généralités

Il est possible de “surcharger” l’inférence de types pour un objet :

```
let etudiant: {prenom: string, nom: string, nomComplet: string};
```

```
let etudiant: {
    prenom: string;
    nom: string;
    nomComplet: string;
}
```

```
Property 'nomComplet' is missing in type '{ prenom: string; nom: string; }' but required in
type '{ prenom: string; nom: string; nomComplet: string; }'. ts(2741)

index.ts(17, 45): 'nomComplet' is declared here.

Peek Problem (Alt+F8)    No quick fixes available
```

```
etudiant = {
    prenom : "Pierre",
    nom : "Dupont"
};
```

Cela présente néanmoins assez peu d’intérêt, souvent on passera plutôt par un type / une classe personnalisé(e).

Types primitifs et généralités

Lorsque l'on manipule des tableaux, on déclare un type de variable associé :

```
let tabNotes: number[];
let tabNotesInfere = [10, 12, 14];
tabNotesInfere.push("test");
```

Dans une boucle sur un tableau, le type de la variable de boucle est automatiquement déduit :

```
let tabNotes: number[];
let tabNotesInfere = [10, 12, 14];

        let note: number
for (let note of tabNotesInfere) {
    console.log(note);
}
```

Tuples

En TypeScript il est possible de créer des **tuples**.

Un tuple est un ensemble de 2 valeurs. En quelque sorte, c'est un tableau qui ne contiendra que 2 éléments.

Exemples :

```
let role: [number, string];

role = [0, "User"];

role = ["User", 0]; //incohérence de types

role[0].toFixed(2); //le type est connu
role[0].substring(1); //substring est applicable sur un string, pas un number

role[2] = "test"; //impossible

role.push("test"); //Par contre, on reste sur le prototype array !
```

Énumérés

Un énuméré TypeScript est en quelque sorte une liste d'alias.

```
enum Color {
    RED,
    GREEN,
    BLUE
}

let rouge = Color.RED;

console.log(rouge); //0
console.log(Color.GREEN); //1
console.log(Color[2]); // "BLUE"
```

Par défaut, chaque valeur de l'enum est traduite en un nombre (sa position dans l'enum), en partant de 0.

Il est possible de démarrer à 1 :

```
enum Color {
    RED = 1,
    GREEN,
    BLUE
}
```

Énumérés

Il est possible d'utiliser un enum avec pour valeur des strings et non des numbers :

Attention par contre dans ce cas, ce n'est pas un dictionnaire où l'on peut passer facilement de la clef à la valeur et vice versa…

```
enum Roles {
    ADMIN = "Administrateur",
    USER = "Utilisateur"
}

console.log(Roles.ADMIN); //Administrateur
console.log(Roles["Administrateur"]); //undefined
```

Énumérés

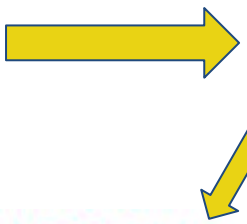
Regardons ce qui est généré en JavaScript :

```
enum Roles {
    ADMIN,
    USER
}
```

```
var Roles;
(function (Roles) {
    Roles[Roles["ADMIN"] = 0] = "ADMIN";
    Roles[Roles["USER"] = 1] = "USER";
})(Roles || (Roles = {}));
```

```
▼ {0: "ADMIN", 1: "USER", ADMIN: 0, USER: 1}
    0: "ADMIN"
    1: "USER"
    ADMIN: 0
    USER: 1
  ▶ __proto__: Object
```

```
enum Roles {
    ADMIN = "Administrateur",
    USER = "Utilisateur"
}
```



```
var Roles;
(function (Roles) {
    Roles["ADMIN"] = "Administrateur";
    Roles["USER"] = "Utilisateur";
})(Roles || (Roles = {}));
```

```
▼ {ADMIN: "Administrateur", USER: "Utilisateur"}
    ADMIN: "Administrateur"
    USER: "Utilisateur"
  ▶ __proto__: Object
```

Any

Il est possible d'utiliser le mot clef any si l'on ne souhaite pas préciser de type. Cela revient à développer en JS natif donc l'intérêt reste limité.

Any est surtout utilisé lorsque les déclarations de type ne sont pas connues (exemple une librairie qui ne les fournit pas).

```
let uneVariableSansType: any;

uneVariableSansType = "test";
uneVariableSansType = 34;
```

Union

Il est possible d'utiliser une union de type, c'est à dire de définir le fait qu'une variable puisse être d'un type **ou** d'un autre.

On peut changer de type comme on le souhaite.

```
let chaineOuNombre: string | number;

chaineOuNombre = 13;
console.log(typeof chaineOuNombre); //number
chaineOuNombre = "test";
console.log(typeof chaineOuNombre); //string

chaineOuNombre = false; //interdit !
```

On peut indiquer autant de types possibles qu'on le souhaite :

```
let chaineOuNombre: string | number | boolean ;
```

Type littéral

Le type littéral permet de définir des valeurs exactes.

Cela peut être pratique en combinaison avec l'union de types. Par exemple, si l'on souhaite que la valeur d'une variable soit comprise dans un ensemble de valeurs possibles :

```
let maVariable : "valeur_possible_1" | "valeur_possible_2" | 3;

maVariable = "autre valeur"; //erreur
maVariable = 3;
maVariable = "valeur_possible_1";
```

Alias de type

Afin d'éviter de répéter une définition de type, on peut créer un alias.

```
type MonAliasDeTypes = string | number;
type MonAutreAliasDeTypes = "valeur_1" | "valeur_2";
```

Fonctionne également pour des objets :

```
type Produit = { titre: string; prix: number };

const p1: Produit = { titre: "Chaussures", prix: 45};

const p2: Produit = { title: "test", price: 20, stock: 40 } //génère une erreur
```

Retours de fonctions

Le type de retour d'une fonction peut être inféré ou défini.

Préférence personnelle : définir pour chaque fonction son type de retour. Cela aide à la lisibilité.

```
function uneFonction() : string {
    return "Hello function";
}

function uneFonction(): string
uneFonction();
```

```
function uneFonction() {
    return "Hello function";
}

function uneFonction(): string
uneFonction();
```

Lorsqu'une fonction ne retourne rien, le type inféré par TypeScript est **void**. Void signifie "rien" et est différent de undefined.

```
function uneFonction() { }

function uneFonction(): void
uneFonction();
```

Function

Il est possible d'utiliser le type **Function**.

Exemple :

```
function uneFonction() { }

uneFonction();

let monPointeurDeFonction: Function;
monPointeurDeFonction = uneFonction;
monPointeurDeFonction = false; //erreur
```

On peut préciser la signature attendue de la fonction.

Exemple :

```
function uneFonction() { }
function uneAutreFonction(p1:number) { return p1 };

let monPointeurDeFonction: (p1: number) => number;

monPointeurDeFonction = uneFonction; //signature incorrecte
monPointeurDeFonction = uneAutreFonction;
```

Function

Dans le cadre de la programmation asynchrone, on utilise souvent cette notation.

Exemple :

```
function addAndHandle(n1: number, n2: number, callback: (num: number) =>  void) {
    callback(n1 + n2);
}
```

Ecrire void ici signifie que l'on ne s'intéresse pas à l'éventuel type de retour du callback s'il en existe un.

```
function addAndHandle(n1: number, n2: number, callback: (num: number) =>  number) {
    let resultatCallback = callback(n1 + n2);

                let resultatCallback: number
    console.log(resultatCallback);
}
```