Chapitre 4
TypeScript

1. JavaScript

Le langage JavaScript s'est imposé depuis quelques années comme le langage indispensable au développement d'applications web.

Initialement, JavaScript a été créé comme un langage de scripting utilisé pour apporter un peu de dynamisme aux sites web statiques. Mais au fil du temps, à l'aide de la popularisation de frameworks comme jQuery ou AngularJS, l'utilisation du langage JavaScript est devenue de plus en plus importante, jusqu'à permettre la création d'applications 100 % front, possédant des fonctionnalités identiques à ce que l'on peut faire en application client lourd (mode offline, notifications, etc.).

JavaScript est un langage assez différent de ce que l'on a l'habitude de rencontrer. Tout d'abord, JavaScript est un langage interprété, le code écrit étant directement exécuté sans phase de compilation préalable.

JavaScript est également un langage dynamiquement typé, c'est-à-dire qu'un élément peut changer de type en cours d'exécution. Une variable d'un certain type peut ainsi se voir affecter une valeur d'un autre type.

```
var maFonction = function() {
    console.log("I'm a function");
}
maFonction = 27;
```

Exemple du typage dynamique JavaScript. La variable est initialisée en tant que fonction puis prend une valeur entière.

Enfin, JavaScript est un langage par nature monothreadé, fortement asynchrone et basé sur un mécanisme d'événements non bloquants. Malgré sa nature monothreadée, JavaScript est donc par design un langage très performant, permettant de tirer parti de la totalité des performances des ressources lui étant allouées (au contraire du modèle de langage multithreadé permettant de paralléliser un ensemble de traitements).

Lorsque l'on développe une application en JavaScript, sa forte dynamicité devient rapidement un handicap. Comment faire pour s'assurer que le code que l'on écrit est syntaxiquement valide ? Comment effectuer des phases de refactoring sans apporter des régressions ?

```
var maFonction = function() {
    console.log("I'm a function");
}
maFunction();
```

Dans le code précédent, il y a une erreur d'orthographe à l'appel de la méthode `maFonction`. L'erreur ne sera visible qu'à l'exécution du code.

```
var maFonction = function(param) {
    console.log("I'm a function with a param " + param);
}
maFonction();
```

Dans le code précédent, la méthode `maFonction` est appelée sans paramètre. À l'exécution, la console affichera le message `"I'm a function with a param undefined"`.

```
var maFonction = function() {
    console.log("I'm a function");
}
maFonction = 4;
maFonction();
```

Dans le code précédent, on affecte un entier à la variable `maFonction`, qui était une fonction. L'exécution de ce code renverra l'erreur suivante : `"maFonction is not a function"`.

Il existe un ensemble d'outils, comme **jslint**, permettant de valider la qualité du code écrit mais ne permettant pas de se substituer à de vraies phases de compilation.

2. TypeScript

L'une des solutions à ces problématiques a été la création de langages qui se transcompilent en JavaScript, c'est-à-dire que la compilation génère des fichiers JavaScript. C'est notamment le cas de TypeScript, porté par Microsoft et utilisé par Angular.

D'autres langages similaires existent, comme **Dart** ou **CoffeeScript**, mais TypeScript est sans doute aujourd'hui l'un des plus complets.

TypeScript est donc un langage compilé et typé fortement qui génère du JavaScript compréhensible par tous les navigateurs.

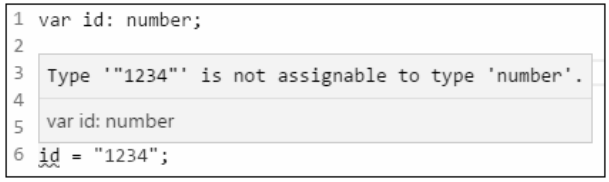
Remarque

D'autres solutions ont été trouvées, comme la création de nouveaux langages comme Flash ou Silverlight pouvant être interprétés par les navigateurs grâce à l'installation de plug-ins. Ces langages ont signé leur fin le jour où les navigateurs ont décidé de ne plus les supporter. L'avantage de TypeScript est qu'il génère du code JavaScript totalement compréhensible nativement, c'est-à-dire sans l'aide de plug-ins, par tous les navigateurs. Cette solution est donc pérenne dans le temps.

Le typage fort de ce langage va permettre d'associer un type à un élément et empêcher cet élément de changer de type. Ce typage fort permet donc une stabilité supérieure du code puisqu'il sera possible de prédire le type d'un élément, et en conséquence ses différentes valeurs possibles.

```
var id: number;
```

La phase de compilation de TypeScript permet de valider syntaxiquement le code. Si celui-ci n'est pas valide syntaxiquement, c'est-à-dire si le développeur a fait usage d'une propriété sur un objet qui n'existe pas ou qu'il utilise un type de manière non conforme, la compilation du code TypeScript indiquera cette erreur. Les phases de refactoring, qui s'avéraient extrêmement compliquées et dangereuses, sont beaucoup plus simples. De manière générale, le code sera beaucoup plus robuste et stable.



De nombreux éditeurs, notamment **Visual Studio**, **VSCode** ou **Sublime-Text** pour n'en citer que quelques-uns, comprennent le langage TypeScript et proposent de nombreux outils, comme l'autocomplétion, les erreurs de compilation directement dans l'éditeur, etc.

2.1 Syntaxe

La syntaxe de TypeScript est assez différente de ce qu'on a l'habitude de voir en JavaScript, voici un tour d'horizon des principaux éléments.

2.1.1 Variables

La déclaration d'une variable se fait de la façon suivante :

```
var maVariableBooleenne : boolean;
```

Le type est défini en suffixant la définition d'une variable par son type, séparé par ':'. Il existe un ensemble de types de base : `boolean`, `number`, `string`, `[]` pour un tableau, etc.

Dans l'optique de garder la possibilité de profiter du typage dynamique de JavaScript, TypeScript introduit le type `any`.

```
var maVariableDynamique : any;
```

Dans ce cas, TypeScript ne vérifiera pas le type de cette variable à la compilation.

Si le type d'une variable n'est pas précisé, TypeScript la considérera comme étant de type `any`.

2.1.2 Fonctions

La déclaration d'une fonction se fait de manière classique.

```
function maFonction() : void {
   ...
}
```

Le type de retour de la fonction se définit de manière identique au type d'une variable. Le type `void` indique qu'il n'y a aucun retour.

Si la fonction prend des paramètres en entrées, ces paramètres se typent de manière identique aux variables :

```
function maFonction(monParametre : string, monAutreParametre :
boolean) : void {
   ...
}
```

2.1.3 Classes

La notion de classe est apparue en JavaScript avec EcmaScript 6. Cette notion existe depuis plusieurs versions de TypeScript.

Remarque

À ce jour, la notion de classe introduite par EcmaScript 6 n'est pas encore supportée par tous les navigateurs récents, notamment par Internet Explorer 11. Il n'est donc pas encore possible de l'utiliser si l'on souhaite cibler tous les navigateurs récents.

```
class Person {
    name : string;
    age: number;

    constructor(name : string, age: number) {
        this.name = name;
        this.age = age;
```

```
    toString() {
        return `Hi I'm ${this.name} and I'm ${this.age} years old!`;
    }
}
```

Le mot-clé `class` permet d'indiquer que l'on est en train de créer une classe.

Il est également possible de créer de l'héritage entre classes :

```
class Developer extends Person {
    constructor(name, age, language) {
        super(name, age);
        this.language = language;
    }

    toString() {
        return super.toString() + ` :: I'm a Developer who likes
${this.language}`;
    }
}
```

La relation d'héritage se déclare via le mot-clé `extends`. Il est ensuite possible d'accéder aux éléments de la classe parente via la méthode `super`.

TypeScript introduit également la notion de visibilité sur les propriétés. Il est possible de déclarer une propriété `private`, qui ne sera accessible que depuis la classe courante, `protected`, qui ne sera accessible que depuis la classe courante ou les classes héritant de la classe courante, et enfin une propriété `public` qui sera accessible depuis l'extérieur de la classe. Par défaut, si aucune visibilité n'est spécifiée, une propriété sera `public`.

```
class Modifier {
    public myPublicProperty: string;
    protected myProtectedProperty: string;
    private myPrivateProperty: string;
}
```

2.1.4 Arrow Function

L'un des pièges que l'on rencontre souvent quand l'on débute avec le JavaScript est l'utilisation de `this`.

En JavaScript, le `this` se comporte légèrement différemment des autres langages de programmation. Sa valeur est déterminée à partir de la façon dont la fonction est appelée.

```
function Person(age) {
    this.age = age;

    this.growOld = function() {
        this.age++;
    }
}

var person = new Person(1);
setTimeout(person.growOld(), 0);
```

Dans l'exemple précédent, le `this` utilisé dans la fonction `growOld` n'est pas le `this` attendu. Le `this` correspondra à l'objet Windows et non à la propriété `person`.

Il est courant en JavaScript de devoir enregistrer la valeur du `this` avant d'appeler une fonction. Cela permet d'y avoir accès même à l'intérieur de la cette fonction.

```
function Person(age) {
    var that = this;
    this.age = age;

    this.growOld = function() {
        that.age++;
    }
}

var person = new Person(1);
setTimeout(person.growOld(), 0);
```

La variable `that` permet d'enregistrer la référence du `this`, et donc de pouvoir s'y référer plus tard.