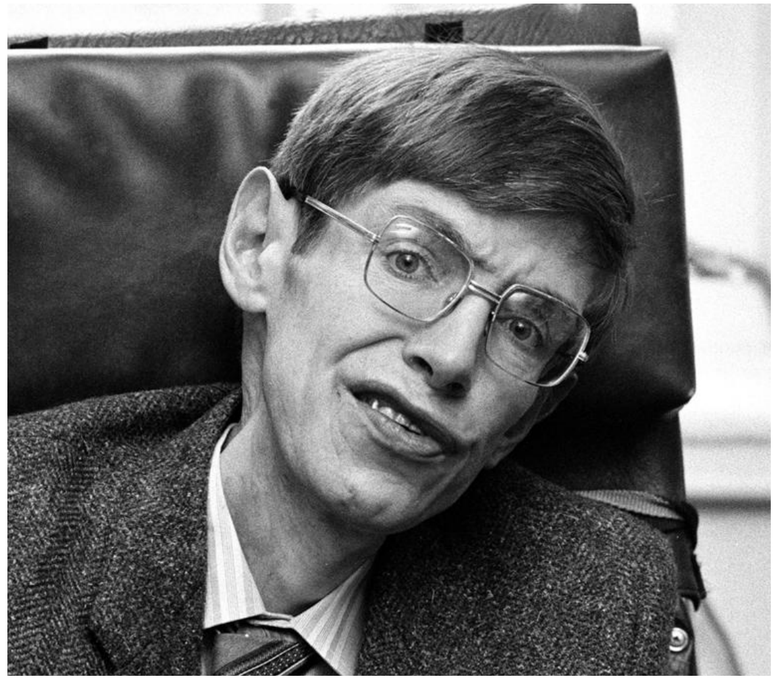
You are my type of value...

TypeScript et le TDD

COMBATTRE LE BUG PAR LE TEST

Tous les développeur qui ont déjà travaillé sur un projet de grands taille savent qu'ils doivent réserver entre 70 et 80 pourcent de son temps pour la pratique la plus frustrante de leur métier : les tests unitaires. Le pilotage de projet par le test, ou Test driven Development (TDD), répond en partie à la problèmatique en placant les tests dès le début du développement, ce qui permet de gagner beaucoup de temps, mais induit néanmoins de coder des fonctions de test avant de créer les fonctions du programme, ce qui peut allonger considérablement le temps de production. Le principe de test n'est pas nouveau, dès que les prémières lignes de codes ont été tapées il à fallu les tester, mais à l'origine il n'était pas à la charge du développeur.



> "THE GREATEST ENEMY OF KNOWLEDGE IS NOT IGNORANCE, IT IS THE ILLUSION OF KNOWLEDGE"
>
> **Stephen Hawking**

LE FEU ÇA BRÛLE ET L'EAU ÇA MOUILLE

La plupart des langages de programmation intègrent la notion de typage qui permet d'une certaine manière de contraindre des variables ou des fonctions. Cette notion manque au JavaScript, ce qui induit un temps d'écriture de test plus long lorque l'on utilise ce langage. Pour palier à ce manque, Microsoft a développé un "Super Set" de JavaScript qui, associé aux nouveautés de la version ES6, ajoute un ensemble de techniques pour typer un programme et ainsi réduire le temps passer à tester du code. L'utilisation de TypeScript ne supprime pas la nécessité de tester un programme mais apporte des solutions pour définir précisément le comportement des éléments qui composent un programme.

≠ 1 ≠ un ≠ `1`

TRAVAILLER AVEC TYPESCRIPT

Le TypeScript n'est pas un nouveau langage mais une surcouche de JavaScript, le principe et d'écrire un fichier au format .ts qui sera complier en JavaScript pour pouvoir être utilisé dans une page Web ou une application. Pour commencer à coder en TypeScript il faut donc configurer un environnement de développement spécifique pour intégrer TypeScript dans un environemment de travail spécifique.

CONFIGURATION DU DOSSIER DE TRAVAIL

Dans un premier temps, vous devez créer un dossier contenant deux sous-dossiers, **"typings"** et un **"js"**, ainsi qu'un fichier **"package.json"**.

```
MyTSproj
  - js
  - package.json
  - typings
```

INSTALLER LE TRANSPILLER

Pour commencer, il faut installer en globel le transpiller qui permet de compiller le code TypeScript.

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

`--g` ..... Installation en global
`--save-dev` ... Ajouté TypScript en dépendance de développement.

Toutes les option de TypeScript sont disponibles sur typescriptlang.org.

CONFIGURATION DU COMPILATEUR

Une fois TypeScrpt installé, ajouter un script dans votre fichier package.json pour configurer le compileur.

```
..
  "scripts": {
    "build": "tsc typings/* --outDir js -w --pretty"
  },
..
```

`-w` ..... Compliation du/des fichier/s à la vollée
`--pretty` ..... Affichage des erreurs simplifié

COMPILATION DES FICHIERS

Tous les fichier TypeScript du dossier **"typings"** seront compilés en JavaScript dans le dossier **"js"** en tapant la commande suivante dans votre terminal :

```
npm run build
```

LE PRINCIPE DE TYPAGE

Le principal avantage du TypeScript tient dans son nom : en plus d'accocier un nom et une valeur à une variable nous pouvons déffinir le type de la variable. Cette notion de typage est présente dans de nombreux langages de programmation, nottamment le Java, mais était très compliqué en ES5. Le TypeScript propose une solution plus rapide à mettre en place et qui répond à la problématique de Test Driven Developement. Le typage d'une variable en TypeScript ce fait à l'aide du signe ":" suivie du type de la variable entre le nom et la valeur de la variable.

TYPAGE IMPLICITE

Lorsqu'une valeur est assignée à une variable, la valeur défini le type de la variable

TYPAGE EXPLICITE

A la créaion d'une variable, il faut définir son type de la façon suivante : variable: type

```
let firstName = `Abdel`
firstName = 38

let age: string
age = `Carole`
age = 17
```

FIRSTNAME EST DE TYPE STRING

Le compileur indique que la valeur `38` n'est pas assignable à la variable `firstName`

AGE EST DE TYPE NUMBER

Le compileur indique que la valeur `Carole` n'est pas assignable à la variable `age`

```
firstName = 38
Type '38' is not assignable to type 'string'.

age = `Carole`
Type '"Carole"' is not assignable to type 'number'.
```

TYPER UN TABLEAU ET SES VALEURS

Contrairement à des variables classiques, un tableu recoit plusieurs variables différentes qui peuvent être également de types différents. La définition du type d'un tableau et de ces valeurs ce fait de la même manière que pour des variables classiques mais si les type de valeurs sont différents il faut tous les définir.

TABLEAU A TYPE DE VALEUR UNIQUE

Les valeurs du tableau doivent toutes être de type **`string`**

```
let firstArray: Array<string>
firstArray = [`Abdel`, `Carole`, `Louis`]
```

TABLEAU À TYPE DE VALEUR MULTIPLE

En séparant les valeur par **un pipe** il est possible de définir plusieur types de valeur possible.

```
let secondeArray: Array<string | boolean>
secondeArray = [true, `Carole`, `Louis`]
```

TABLEAU À TOUT TYPES DE VALEUR

Le mot clé **`any`** permet de ne pas définir de type

```
let thirdArray: Array<any>
thirdArray = [true, `Carole`, 38]
```

TABLEAU NON-MODIFIABLE

En utilisant le mot clé **ReadonlyArray** le tableau devient une constante

```
let fourthArray: ReadonlyArray<number> = [3, 10]
fourthArray.push = 16 // Erreur : fourthArray est une constante
```

Pour un tableau à type de valeur unique il est également possible de déclarer la variable de la façon suivante :
let firstArray: String[]

TYPER UNE FONCTION ET SES PARAMÉTRES

Les fonctions peuvent également être typées, que ce soit au niveau du retour de la fonction comme des paramétres qui la compose. Pour des fonctions contenant un grand nombre de paramètres, il est conseillé d'utiliser un objet associé à une interface que nous verrons dans les pages suivantes.

TYPER LA VALEUR DE RETOUR

Il est indiqué à la création de la fonction qu'elle doit renvoyer une valeur de type **string**

La valeur du retour de la fonction est bien une string

```
function sayHello(): string {
    return `Hello World`
}
sayHello() === `Hello World`` // true
```

TYPER LES PARAMÈTRE DE LA FONCTION

De la même manière que le typage de variable, il est possible de définir les paramètre de la fonction

Une valeur de type string n'est pas accèptée en paramètre

```
function isPair(x: number, y: number): string {
    if( typeof x === `number` && typeof y === `number` {
        return `La somme de x + y est ${x + y}`
    }
}
isPair(5, 2) ===  `La somme de x + y est 7` // true
isPair(5, `2`) // Argument of type '"2"' is not assignable
```

TYPER LE CALLBACK D'UNE FONCTION

Certaines fonctions utilise en paramètre une fonction de callback pour qu'elle s'exécute à la fin du traitement de la fonction principale. En TypeScript il faut définir la valeur du callback et celle du retour de la fonction de callback.

DÉFINITION DU CALLBACK

Le paramètre est typé de la même manière que le autre mais il faut également typer le retour de la fonction

Création d'un tableau typé

UTILISATION DU CALLBACK

Les paramètres de la fonction principale sont disponible dans la fonction de callback

```
function twice(names: Array<string>, callback: () => string): number{
    return names.length
}

let users: Array<string> = [`Carole`, `Abdel`, `Lisa`]

twice(users, () => {
    return `Liste des utilisateurs : ${users}`
})
```

Le principe des fonction de **callback en paramètre n'est pas spécifique à TypeScrypt**, il est question ici du typage de la fonction de callback en TypeScript.

TYPER UN OBJET AVEC UNE INTERFACE

Proche du modèle objet (constructeur), les interfaces sont principallement utilisées dans le cadre de fonctions ayant en paramètre un objet. Plutôt que de typer chacune des propriétés de l'objet dans le parenthèse de la fonction, nous créons une interface réutilisable qui s'en charge.

CRÉATION DE L'INTERFACE
Une interface se présente sous la forme d'un objet dont les paramètres sont typés

Il est possible de rendre un paramètre constant

```
interface User {
    fullName: string,
    age: number,
    readonly isMajor: boolean,
    skills: Array<string>
}
```

INTERFACÉ LE PARAMÈTRE
Une fois créée, l'interface est utilisée comme un type de variable ce qui allège l'écriture de la finction

```
function createUser(object: User): string {
    return `Création d'un nouvel utilisateur`
}
```

DÉFINITION D'UNE VARIABLE
Il est recommandé de créer une variable qui utilise l'interface plutôt que de directement renseigner l'objet dans la fonction.

```
let newUser: User = {
    fullName: `Sophia`,
    age: 27,
    isMajor: true,
    skills: [`ES6`, `TypeScript`]
}
```

Le paramètre isMajor n'est pas modifiable

```
newUser.isMajor = false // Erreur : isMajor est une constante
```

Lorsque tout est typé, la fonction ne peut que s'exécuter correstement

```
createUser(newUser)
```

ETENDRE UNE INTERFACE

Comme pour les classes que nous verrons dans les pages suivantes, il est possible de créer des interfaces qui intégrent d'autres interfaces ce qui est très utile pour séparer le code d'un programme. Cette technique permet une maintenance facilité et une réutilisation plus rapide du code lorsque les interfaces sont exportées depuis un fichier et importer dans ceux qui les utilisent.

CRÉATION DE DEUX INTERFACES
Les interface sont séparées par type de données

```
interface UserInfos {
    userName: string,
    age: number
}

interface SkillsLevel {
    es6: number,
    ts: number
}
```

ETENDRE LES INTERFACE
L'interface User intégre dans sa construction les paramètres des interfaces UserInfos et SkillsLevel

```
interface User extends UserInfos, SkillsLevel {
    isMajor: boolean
}
```

UTILISATION DE L'INTERFACE ÉTENDUE

```
let newUser = <User>{}
```

Définitions des valeurs des paramètres

```
newUser.userName = `Steve`
newUser.es6 = 6
newUser.isMajor = true
```

LES CLASSES EN TYPESCRIPT

Depuis la version ES6 les classes sont apparues dans la programmation JavaScript qui en manquait curellement. Il est donc à présent possible de développer des programme complèxes organisés par classes et TypeScript apporte des solutions pour les configurer correctement.

CRÉATION DE LA CLASSE

DÉFINITION DES PARAMÈTRES

Les paramètres peuvent être de plusieurs types (public, static, private, protected) selon leur niveau d'accessibilité.

LE CONSTRUCTEUR

Permet d'ajouter des valeurs dès l'instanciation d'un objet.

Un paramètre static est accessible via la classe

LES MÉTHODES

De plusieurs type comme les paramètres

GETTER ET SETTER

Pour définir la valeur d'un paramètre de type private il faut utiliser les méthodes get et set

INSTANCIATION DE L'OBJET

Le setter est utilisé comme un paramètre publique

Un paramètre static est accessible via la classe

```
class Users{
    public firstName: string
    static numOfUsers: number = 0
    private _nickname: string

    constructor(userName: string){
        this.firstName = userName
        Users.numOfUsers++
    }

    userInfo(): string { return `My name is ${this.firstName}` }
    static howManyUsers(): number { return Users.numOfUsers }

    get nickname(): string { return this._nickname }
    set nickname(pseudo: string){ this._nickname = pseudo }
}

let newUser = new Users(`John`)

newUser.nickname = `The Turtle`
Users.numOfUsers === 1
```

ETENDRE UNE CLASSE

Les classes offrent une multitude de possibilité en terme de programmation mais l'erreur la plus classique lorsqu'on les utilise est d'écrire des classes trop volumineuse. La logique de programmation fonctionn et de séparer un maximum le code pour permettre une mainteannce plus efficace. Partant de cette logique, il est recommandé d'écrire des classes et des sous-classes qui ont de comportements spécifiques.

CRÉATION D'UNE CLASS
Cette classes peut contenir autant de variables et de méthode que nécessaire

Création d'une méthode statique

ETENDRE UNE CLASSE

LA MÉTHODE SUPER
Cette méthode permet de faire référence au constructeur de la classe Vehicle

La fonction howManyVehicle() est directement accessible sur la classe Car

```
class Vehicle{
    static totalVehicle: number = 0

    constructor(private name: string){}

    static howManyVehicle(): number { return Vehicle.totalVehicle }
}

class Car extends Vehicle {
    public carName: string

    constructor(name: string){
        super(name)
        this.carName = name
    }
}

let newCar = new Car(`Fiat 500`)

Car.howManyVehicle() // 1
```