formation.tech

TypeScript

TypeScript - Introduction

- TypeScript : JavaScript + Typage statique
  - TypeScript est un langage créé par Microsoft, construit comme un sur-ensemble d'ECMAScript
  - Pour pouvoir exécuter le code il faut le transformer en JavaScript avec un compilateur
  - A quelques exceptions près et selon la configuration, le JavaScript est valide en TypeScript
- Le principal intérêt de TypeScript est l'ajout d'un typage statique

TypeScript - Installation

- Installation
  - `npm install -g typescript`
- Création d'un fichier de configuration
  - `tsc --init`
- Compilation
  - `tsc`

TypeScript - Typage statique

- Le principal intérêt de TypeScript est l'introduction d'un typage statique

```ts
const lastName: string = 'Bohdanowicz';
const age: number = 32;
const isTrainer: boolean = true;
```

- Types basiques :
  - *boolean*
  - *number*
  - *string*

TypeScript - Typage statique

- Avantages
  - Complétion

```
const firstName: string = 'Romain';

function hello(firstName: string): string {
  return `Hello ${firstName.}`;
}
```

```
m charAt(pos: number)
m charCodeAt(index: number)
m concat(... strings: string)
m indexOf(searchString: string,
```

  - Détection des erreurs

```
const firstName: string = 'Romain';

function hello(firstName: string): string {
  return `Hello ${firstName}`;
}

hello({
  firstName: 'Romain',
});
```

TypeScript - Typage statique

- Tableaux

```ts
const firstNames: string[] = ['Romain', 'Edouard'];
const colors: Array<string> = ['blue', 'white', 'red'];
```

- Tuples

```ts
const email: [string, boolean] = ['romain.bohdanowicz@gmail.com', true];
```

- Enum

```ts
enum Choice {Yes, No, Maybe}

const c1: Choice = Choice.Yes;
const choiceName: string = Choice[1];
```

- Never

```ts
function error(message: string): never {
  throw new Error(message);
}
```

TypeScript - Typage statique

- Any

```ts
let anyType: any = 12;
anyType = "now a string string";
anyType = false;
anyType = {
  firstName: 'Romain'
};
```

- Void

```ts
function withoutReturn(): void {
  console.log('Do someting')
}
```

- Null et undefined

```ts
let u: undefined = undefined;
let n: null = null;
```

TypeScript - Assertion de type

- Le compilateur ne peut pas toujours déterminer le type adéquat :

```
const formElt = document.querySelector('#myForm');
const url = formElt.action; // error TS2339: Property 'action' does not exist on
type 'Element'.
```

- Il faut alors lui préciser, 3 syntaxes possibles

```
let formElt = <HTMLFormElement> document.querySelector('#myForm');
const url = formElt.action;
```

```
let formElt = document.querySelector<HTMLFormElement>('#myForm');
const url = formElt.action;
```

```
let formElt = document.querySelector('#myForm') as HTMLFormElement;
const url = formElt.action;
```

TypeScript - Inférence de type

- TypeScript peut parfois déterminer automatiquement le type :

```
const title = 'First Names';
console.log(title.toUpperCase());

const names = ['Romain', 'Edouard'];
for (let n of names) {
  console.log(n.toUpperCase());
}
```

TypeScript - Fonctions

- On peut typer les paramètres d'entrées et de retour d'une fonction

```ts
function hello(name: string): string {
  return `Hello ${name.toUpperCase()} !`;
}
```

- On peut également utiliser

```ts
function useCallback(cb: Function) {
  cb();
}

useCallback(() => {});
```

TypeScript - Interfaces

- Pour documenter un objet on utilise une interface
  - Anonyme

```
function helloInterface(contact: {firstName: string}) {
  console.log(`Hello ${contact.firstName.toUpperCase()}`);
}
```

  - Nommée

```
interface ContactInterface {
  firstName: string;
}

function helloNamedInterface(contact: ContactInterface) {
  console.log(`Hello ${contact.firstName.toUpperCase()}`);
}
```

TypeScript - Interfaces

- Les propriétés peuvent être :
  - optionnelles (ici *lastName*)
  - en lecture seule, après l'initialisation (ici *age*)
  - non déclarées (avec les crochets)

```ts
interface ContactInterface {
  firstName: string;
  lastName?: string;
  readonly age: number;
  [propName: string]: any;
}

function helloNamedInterface(contact: ContactInterface) {
  console.log(`Hello ${contact.firstName.toUpperCase()}`);
}
```

TypeScript - Classes

- Quelques différences avec JavaScript sur le mot clé class
  - On doit déclarer les propriétés
  - On peut définir une visibilité pour chaque membre : *public*, *private*, *protected*

```
class Contact {
  private firstName: string;

  constructor(firstName: string) {
    this.firstName = firstName;
  }

  hello() {
    return `Hello my name is ${this.firstName}`;
  }
}

const romain = new Contact('Romain');
console.log(romain.hello()); // Hello my name is Romain
```

TypeScript - Classes

- Une classe peut
  - Hériter d’une autre classe (comme en JS)
  - Implémenter une interface
  - Être utilisée comme type

```ts
interface Writable {
  write(data: string): void;
}

class FileLogger implements Writable {
  write(data: string): Writable {
    console.log(`Write ${data}`);
    return this;
  }
}
```

TypeScript - Génériques

- Permet de paramétrer le type de certaines méthodes

```
class Stack<T> {
  private data: Array<T> = [];
  push(val: T) {
    this.data.push(val);
  }
  pop(): T {
    return this.data.pop();
  }
  peek(): T {
    return this.data[this.data.length - 1];
  }
}

const strStack = new Stack<string>();
strStack.push('html');
strStack.push('body');
strStack.push('h1');
console.log(strStack.peek().toUpperCase()); // H1
console.log(strStack.pop().toUpperCase()); // H1
console.log(strStack.peek().toUpperCase()); // BODY
```

TypeScript - Décorateurs

- Permettent l'ajout de fonctionnalités aux classes ou membre d'une classe en annotant plutôt que via du code à l'utilisation
- Norme à l'étude en JavaScript par le TC39
  https://github.com/tc39/proposal-decorators
- Supporté de manière expérimentale en TypeScript
- Pour activer leur support il faut éditer le tsconfig.json ou passer une option au compilateur

```
{
  "compilerOptions": {
    "target": "es5",
    "experimentalDecorators": true
  }
}
```

TypeScript - Décorateurs

- Décorateur de classes

```
'use strict';

function Freeze(obj) {
  Object.freeze(obj);
}

@Freeze
class MyMaths {
  static sum(a, b) {
    return Number(a) + Number(b);
  }
}


try {
  MyMaths['substract'] = function(a, b) {
    return a - b;
  };
}
catch(err) {
  // Cannot add property substract, object is not extensible
  console.log(err.message);
}
```

TypeScript - Décorateurs

- Décorateur de propriétés

```
import 'reflect-metadata';

const minLengthMetadataKey = Symbol("minLength");

function MinLength(length: number) {
  return Reflect.metadata(minLengthMetadataKey, length);
}

function validateMinLength(target: any, propertyKey: string): boolean {
  const length = Reflect.getMetadata(minLengthMetadataKey, target, propertyKey);
  return target[propertyKey].length >= length;
}

class Contact {
  @MinLength(7)
  protected firstName;

  constructor(firstName: string) {
    this.firstName = firstName;
  }

  isValid(): boolean {
    return validateMinLength(this, 'firstName');
  }
}

const romain = new Contact('Romain');
console.log(romain.isValid()); // false
```