DÉVELOPPEMENT DES APPLICATIONS MOBILES HYBRIDES MULTIPLATEFORMES

CH 5 - Introduction au langage TypeScript, le futur de JavaScript

FORMATEUR : MAHAMANE SALISSOU YAHAYA : ysalissou@gmail.com

(TechnoLAB -ISTA)

4ième année

OCTOBRE 2020

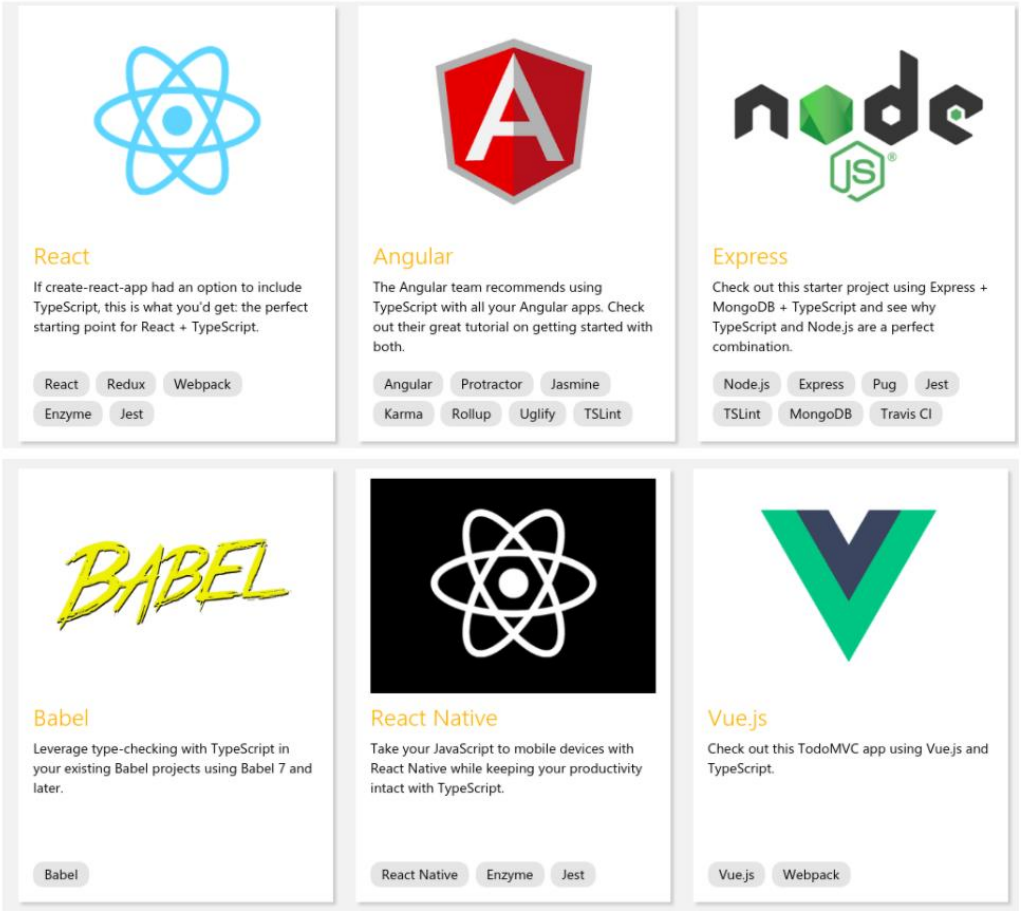
- TypeScript est un langage de programmation libre et open source développé par Microsoft qui a pour but d'améliorer et de sécuriser la production de code JavaScript.

- Sortie en 2012, Il est vu par beaucoup comme le futur du Javascript, car se basant sur la norme ECMAScript 6, celle déjà intégré au moteur JavaScript de la plupart des navigateurs et qui fera foi dans les prochaines années.

- TypeScript c'est donc du JavaScript, avec de supers pouvoirs.

- Il est utilisé dans la plupart des Frameworks JavaScript du moment : React, Angular, Express, VueJS,...

- Il est utilisé dans la plupart des Frameworks JavaScript du moment : React, Angular, Express, VueJS,...



- Les fichiers définis dans ce langage ont pour extension **.ts**. Les navigateurs web ne sachant pas encore interprété du code en TypeScript pur, il est nécessaire de le compiler en JavaScript : on parle alors de transtypage.

Comment TypeScript s'intègre à Ionic

```
export class HomePage {
  selected : any = '';
  notes : any = [];
  constructor(public navCtrl: NavController) {
    this.notes = [
      {'id':'1', 'title':'Une Note 1', 'content':'Le contenu de la note 1'},
      {'id':'2', 'title':'Une Note 2', 'content':'Le contenu de la note 2'},
      {'id':'3', 'title':'Une Note 3', 'content':'Le contenu de la note 3'}
    ];
  }

  itemSelected(item) {
    this.selected =item;
  }

}
```

Voyons à présent plus en détails comment fonctionne ce langage de programmation.

Installation et premier script

OCTOBRE 2020

Installation et premier script

Ionic utilise une version interne du compilateur TypeScript et pour nos tests nous allons devoir installer le package de manière globale.

```
> npm install -g typescript
```

Créons nouveau dossier qui nous permettra de faire nos tests avec TypeScript.

Ajoutez le fichier **demo_typescript.ts** et tant qu'à faire ajoutons-y du contenu :

demo_typescript.ts

```
function ditBonjour(person) {
    return "Bonjour, " + person;
}

let user = "Charles";

document.body.innerHTML = ditBonjour(user);
```

Installation et premier script

Il faut ensuite compiler ce code pour générer un fichier .js interprétable par le navigateur.

```
$ tsc demo_typescript.ts #compilation
```

npm install @types/node --save-dev

Si vous rencontrez des exceptions de ce type :

- **Cannot find name 'console'. Do you need to change your target library? Try changing the lib compiler option to include 'dom', ou encore**
- **error TS2339: Property 'find' does not exist on type 'any[]'.**

Créez simplement un fichier tsconfig.json dans le répertoire de votre projet, avec la configuration suivante :

tsconfig.json

```
{
  "compilerOptions": {
    "lib": [
      /* Permet d'utiliser le DOM (document) et
        la librairie standard inclus dans ES6 */
      "dom",
      "es6"
    ]
  },
  /* On ne compile pas (exclusion) les fichiers du dossiers npm
    et  ceux des tests unitaires (spec.ts)*/
  "exclude": [
    "node_modules",
    "**/*.spec.ts"
  ]
}
```

Installation et premier script

On obtient après compilation le fichier JavaScript suivant :

demo_typescript.js

```
function ditBonjour(person) {
    return "Bonjour, " + person;
}
var user = "Charles";
document.body.innerHTML = ditBonjour(user);
```

Bien évidemment le contenu du fichier demo_typescript.js est strictement le même que celui du fichier .ts, car ce code est relativement simple.

Installation et premier script

Ajoutons à présent un peu plus de choses, comme une Classe, ainsi que des méthodes.

demo_etudiants.ts

```
class Etudiant {
  private etudiants: string[] = [];
  constructor() {

  }

  addEtudiant(etudiant) {
    this.etudiants.push(etudiant);
  }

  getEtudiantByName(name) {
    return this.etudiants.find((etu) => etu.name == name);
  }

  getEtudiantsListe() {
    return this.etudiants;
  }
}
```

```
// On teste notre nouvelle classe ici
let objEtudiant = new Etudiant();

// On ajoute un l'étudiant Jean DUPONT
objEtudiant.addEtudiant({nom:'DUPONT', prenom:'Jean'});

// On ajoute un l'étudiante Jeanne DUPONT
objEtudiant.addEtudiant({nom:'DUPONT', prenom:'Jeanne'});

// On affiche l'ensemble des étudiants
let etudiants = this.getEtudiantsListe();
console.table('Liste de tous les étudiants ',etudiants);
```

On compile le fichier :

```
$ tsc demo_etudiants.ts
```

Installation et premier script

Si tout s'est bien passé, vous devriez obtenir le contenu JavaScript suivant :

```
var Etudiant = /** @class */ (function () {
    function Etudiant() {
        // Tableau d'étudiants
        this.etudiants = [];
    }
    Etudiant.prototype.addEtudiant = function (etudiant) {
        this.etudiants.push(etudiant);
    };
    Etudiant.prototype.getEtudiantByName = function (name) {
        return this.etudiants.find(function (etu) { return etu.name === name; });
    };
    Etudiant.prototype.getEtudiantsListe = function () {
        return this.etudiants;
    };
    return Etudiant;
}());
```

```
// On teste notre nouvelle classe ici
var objEtudiant = new Etudiant();

// On ajoute un l'étudiant Jean DUPONT
objEtudiant.addEtudiant({ name: 'DUPONT', firstname: 'Jean' });

// On ajoute un l'étudiante Jeanne DUPONT
objEtudiant.addEtudiant({ name: 'DUPONT', firstname: 'Jeanne' });

// On affiche l'ensemble des étudiants
var etudiants = this.getEtudiantsListe();
console.table('Liste de tous les étudiants ', etudiants);
```

Voilà ! La différence entre les deux langages est un peu plus parlante cette fois-ci.

Types de base

OCTOBRE 2020

Types de base

Contrairement à JavaScript où les types sont définis au moment de l'initialisation d'une variable, TypeScript propose un typage de variable beaucoup plus fort.

La définition générale d'une variable se fait de la manière suivante :

```
let nomDeLaVariable: leTypeDeBase [= valeur par défaut - Optionnelle];
```

Booléens

```
let isConnected: boolean = false;
```

Nombres

```
let valeur_decimal: number = 6;
let valeur_binary: number = 0b1010;
let valeur_octal: number = 0o744;
let valeur_hex: number = 0xf00d;
```

Types de base

String

```
let color: string = "blue";
color = 'red'; # On a le choix entre les guillemets simples ou doubles
```

Chose intéressante pour les chaînes de caractères, il est possible de les utiliser sous forme de template, un truc que l'on rencontrait jusqu'à lors dans des langages de haut niveau comme Python ou PHP.

```
let fullName: string = `Charles EDOU NZE`;
let age: number = 32;
let year: number = 2019;
let sentence: string = `Salut, mon nom est ${ fullName }. Nous sommes en ${ year }.
                        J'aurai ${ age + 1 } ans l'année prochaine.`;
```

Résultat : Salut, mon nom est Charles EDOU NZE. Nous sommes en 2019. J'aurai 33 ans l'année prochaine.

On aurait pu obtenir le même rendu en concaténant des chaînes de caractères avec le signe "+". Mais avouons-le, c'est quand même moins pratique !

Types de base

Les tableaux

```
let list_nombres_premiers: number[] = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31];
```

Any

On l'utilise quand on ne sait pas trop qu'elle type de données on aura à traiter. C'est souvent le cas quand on utilise un API propriétaire.

```
let variableMystere: any = 4;
variableMystere = "c'est une chaine de caractère";
variableMystere = false; // ou un booléen finalement
```

Attention, si l'on définit une variable avec un type de base, autre que any, un changement de type à la volée renverrait une exception.
Par exemple, si vous avez définit la variable isConnected en tant que booléen.
Il ne sera pas possible d'attribuer une valeur de type chaîne de caractère (string) à cette variable

Lorsque aucun type n'est précisé lors de la déclaration d'une variable, celle-ci est considérée comme étant de type any.

Les Classes et les méthodes

OCTOBRE 2020

Les Classes et les méthodes

Un des gros avantages de TypeScript est la possibilité de créer des classes, ce qui n'est pas tout à fait le cas (au sens strict) en JavaScript, où l'on passe alors par l'attribut prototype.

Les méthodes de classe elles sont équivalentes aux fonctions JavaScript, mais sans le mot clé "function".

```
class classeMmi {
    intervenant: string;
    etudiants: string[];
    constructor(intervenant: string, etudiants:string[]) {
        this.intervenant = intervenant;
        this.etudiants = etudiants;
    }
    getIntervenant() {
        return "L'intervenant actuel s'appelle " + this.intervenant;
    }
    getEtudiants() {
         return this.etudiants;
    }
}

let classeLPMim = new classeMmi("charles", ['etudiant1','etudiant_n']);
classeLPMim.getIntervenant() // Retourne L'intervenant actuel s'appelle charles
```

Les itérateurs

OCTOBRE 2020

Les itérateurs

Boucle for..of

Cette boucle permet d'afficher les éléments d'une liste pure.

```
let unArray = [1, "deux", 'trois'];

for (let valeur of unArray) {
    console.log(valeur); // 1, "deux", "trois"
}
```

Une fois votre code transpilé en JavaScript (commande tsc), il ressemblera plus à ceci :

```
var unArray = [1, "deux", 'trois'];
for (var idx = 0; idx < unArray.length; idx++) {
    var valeur = unArray[idx];
    console.log(valeur);
}
```

Boucle for..of

La boucle for..of est aussi celle qu'il faudra utiliser quand on voudra boucler sur des listes d'objets :

```
let unArrayObjets = [{'nom':'etudiant 1', 'num':1}, {'nom':'etudiant 2', 'num':2}];

for (let item of unArrayObjets) {
    console.log(item.nom); // etudiant 1 etudiant 2
}
```

Les itérateurs

Boucle for..in

Reprenons le code précédent, mais en utilisant la boucle for..in

```
let unArray = [1, "deux", 'trois'];

for (let index in unArray) {
    console.log(index); // 0, 1, 2
}
```

La boucle renvoie en fait les indexes de chaque valeur et ne doit être utilisé dans ce cas là que si l'on souhaite vraiment récupérer les indexes d'un tableau.
Mais le véritable intérêt pour moi de la boucle **for..in** est de pouvoir boucler sur les propriétés d'un objet.

```
let etudiant = {'nom':'Etudiant 001', 'ine':1};
for (let idx in etudiant) {
    console.log(`${idx} : ${etudiant[idx]}`);
    // nom : Etudiant 001
    // ine : 1
}
```

Les itérateurs

Et un petit résumé de ce que renvoie l'une ou l'autre des deux boucles :

```
let list = [3, 4, 5];
let etudiant = {'nom':'Etudiant', 'ine':1};

for (let val of list) {
   console.log(val); // "3", "4", "5"
}

for (let idx in list) { // moyen mnémotechnique : "in" comme "in-dex"
   console.log(idx); // "0", "1", "2",
}

for (let idx in etudiant) {
    console.log(`${idx} : ${etudiant[idx]}`);
    // nom : Etudiant 001
    // ine : 1
}
```

L'objet Set

L'objet Set (Ensemble en français) permet de stocker des valeurs uniques, de n'importe quel type, que ce soit des valeurs d'un type primitif (string, booléen,...) ou des objets.

npm install @types/node --save-dev // tsc nom_du_fichier.ts --downlevelIteration

```
const monObjSet = new Set([1, 2, 3, 4, 5, 5, 5, 6]); // Liste d'éléments
```

L'objet set supprimera les doublons, ainsi en bouclant sur la liste, on aura qu'une seule valeur pour le chiffre 5

```
for (let chiffre of monObjSet) {
  console.log(chiffre) // 1, 2, 3, 4, 5, 6
}
```

On peut également vérifier la présence d'une valeur dans notre liste avec la méthode **has** :

```
console.log(monObjSet.has(1)); // Renvoie "true", car la liste contient bien le chiffre 1
```

Les itérateurs

Il existe d'autres méthodes permettant de manipuler un ensemble : add, clear, delete, entries , foreach, values.

```
console.log(monObjSet.add(7)); // Ajoute le chiffre 7


console.log(monObjSet.delete(6)); // Supprime le chiffre 7 de la l'ensemble


console.log(monObjSet.entries()); // renvoie un iterateur avec l'ensemble des entrées
let iterator1 = monObjSet.entries();
console.log(iterator1.next().value); // renvoie [1, 1]
console.log(iterator1.next().value); // renvoie [2, 2]
console.log(monObjSet.forEach(funct)); permet d'exécuter une fonction donnée pour chaque valeur de
monObjSet.forEach(function(value1, value1, set){
    console.log(value1, value2);
})


console.log(monObjSet.values()); // renvoie un iterateur avec les valeurs de chaque élément de l'e
let iterator2 = monObjSet.values();
console.log(iterator2.next().value); // renvoie 1
console.log(iterator2.next().value); // renvoie 2
console.log(monObjSet.clear()); // Permet de retirer tous les éléments d'un ensemble Set.
```

Les itérateurs

Pour connaitre la taille d'un ensemble, on utilise la propriété **size** :

```
const chiffrePairs = new Set();
chiffrePairs.add(2);
chiffrePairs.add(2);
chiffrePairs.add(6);

console.log(chiffrePairs.size); // Renvoie 2, car on supprime les doublons
```

L'objet Set a aussi la particularité de mixer objet et liste. On peut ainsi assigner une valeur de type objet à la variable monObjSet :

```
monObjSet['type'] = {category:'Nombres entiers', root:'Nombres réels'};
```

```
for (let type in monObjSet) {
  console.log(monObjSet[type].category) // Nombres entiers
}
```

Les itérateurs : Calculs et transformations sur les listes avec map, filter et reduce

Map

Considérons la liste de politiciens suivante :

```
let politiciens = [{
        'prenom': 'Emmanuel',
        'nom': 'Macron',
        'age': 40,
      },
      {
        'prenom': 'Edouard',
        'nom': 'Philippe',
        'age': 47,
      },
      {
        'prenom': 'Bruno',
        'nom': 'Le Maire',
        'age': 49,
      },
      {
        'prenom': 'Virginie',
        'nom': 'Calmels',
        'age': 47,
      },
      {
        'prenom': 'Alain',
        'nom': 'Juppé',
        'age': 72,
}];
```

Pour extraire uniquement les noms en majuscule de chaque politicien, il existe différentes solutions comme celle-ci :

```
let politiciens_noms = [];

for (let pol of politiciens) {
    politiciens_noms.push(pol.nom.toUpperCase());
}

// politiciens_noms
// (5) ["MACRON", "PHILIPPE", "LE MAIRE", "CALMELS", "JUPPÉ"]
```

Ou encore celle là :

```
let politiciens_noms = [];

politiciens.forEach(function (politicien) {

    politiciens_noms.push(politicien.nom.toUpperCase());

});
```

Map

Mais avec la méthode Map, la récupération se fait de manière beaucoup plus efficace :

```
let politiciens_noms = politiciens.map(function (politicien, index, array) {
    return politicien.nom.toUpperCase();
});
// politiciens_noms
// (5) ["MACRON", "PHILIPPE", "LE MAIRE", "CALMELS", "JUPPÉ"]
```

La méthode donne accès dans son callback à chaque item du tableau depuis la variable **politicien**, à sa position (**index**) à l'intérieur du tableau, et enfin du tableau lui-même (**array**).

Avec cette méthode, vous n'aurez pas à vous inquiéter de l'index de la boucle (en JavaScript classique) ou d'utiliser la méthode push pour stocker vos éléments.

De plus, la méthode renvoyant un Array, il est tout à fait possible d'appliquer une autre méthode sur celui-ci juste après la méthode map.

Filter

Cette méthode fait exactement ce que nom semble suggérer : à partir d'un tableau reçu en entrée, il le filtre en éliminant les éléments non désiré selon une condition déterminée.

Reprenons notre liste de politiciens et ne retenons que ceux de moins de 50 ans. Grâce à la méthode filter, il suffit de faire :

```
let politiciens_U50 = politiciens.filter((politicien) => politicien.age <= 50 );


// (4) [{…}, {…}, {…}, {…}]
// 0:{prenom: "Emmanuel", nom: "Macron", age: 40}
// 1:{prenom: "Edouard", nom: "Philippe", age: 47}
// 2:{prenom: "Bruno", nom: "Le Maire", age: 49}
// 3:{prenom: "Virginie", nom: "Calmels", age: 47}
// length:4
// __proto__:Array(0)
```

Reduce

Si la fonction map permet de créer un nouveau tableau en transformant chaque élément d'un tableau, et si filter permet quant à lui de créer un nouveau tableau en supprimant des éléments selon une condition donnée, la méthode reduce permet de prendre tous les éléments du tableau pour les "réduire" à une unique valeur.

Considérons le tableau de politiciens de moins de 50 ans précécent. On aimerait cette fois calculer la moyenne d'âge de ces "jeunes" faiseurs de lois.

```
let moyenne_age = politiciens_U50.reduce(function (previous, current, index) {
    return (previous + current.age);
}, 0)/politiciens_U50.length;

// previous est la valeur de l'âge à un moment t dans la boucle. Cette valeur est initialié à 0.
// current est la valeur actuelle de l'élement de la boucle, ex: {prenom: "Emmanuel", nom: "Macron

// moyenne_age : 45.75
```

Filtre et reduce

Vous aurez souvent l'occasion d'utiliser ces méthodes qui sont extrêment puissante. On peut comme dit précédemment les concaténer les unes derrière les autres. On peut par exemple calculer la moyenne d'âge des politiciens de moins 50 ans précédents en une seule ligne de code. Oui je sais, je suis cruel de vous l'avoir caché jusque là :-).

```
politiciens.filter((politicien) => politicien.age <= 50 )
           .reduce(function (previous, current, index) {
                      return previous + current.age;
           }, 0)/politiciens.filter((politicien) => politicien.age <= 50 )
.length;


// Renvoie bien : 45.75
```

Find

Cette méthode permet de retrouver un élément d'une liste en fonction d'un critère donné.

On ne renvoie que le premier élément trouvé, même si la condition de filtrage correspond à plusieurs items de la liste.

Reprenons l'exemple de nos politiciens. Alors que la méthode filter renverrait plusieurs politiciens de moins de 50 ans, avec find on ne renvoi que le premier élément.

```
politiciens.find((politicien) => politicien.age <= 50 );

// Renvoi {prenom: "Emmanuel", nom: "Macron", age: 40}
```

Les conditions

OCTOBRE 2020

Les conditions

Comme en JavaScript. On retrouve les traditionnels "**if..else**" et "**switch...case**" :

```
if (une_conditon) {
   // La condition est vraie
} else {
   // Elle est fausse
}
```

```
switch(meteo) {
   case 'soleil': {
      //Il fait beau
      break;
   }
   case 'pluie': {
      // Il fait moins beau
      break;
   }
   default: {
      // devine
      break;
   }
}
```

Les constantes

OCTOBRE 2020

Les constantes

Une constante comme son nom l'indique est censée de ne pas être modifiée ou redéfinie par la suite. On pourra par exemple y stocker l'URL d'une API :

```
const apiUrl = 'https://api.charlesen.fr';
```

Ou encore se créer un fichier de configuration pour notre application :

src/config/config.ts

```
const is_localhost = window.location.hostname == 'localhost';
const is_production = false;

// Urls API
const url_local = '/api';
const url_test = 'https://api.charlesen.fr/api_v1';
const url_dev = is_localhost ? url_local : url_test;
const url_prod = is_localhost ? url_local : 'https://api.charlesen.fr/api_v2';

export const Config = {
  app_id: 'b457q7za', // générer au hasard les amis ;-)
  production: !is_localhost && is_production,
  api_url: is_production ? url_prod : url_dev
  db_name:'nom_de_ma_base_de_donnees'
};
```

Les constantes

Que l'on peut ensuite appeler n'importe où, depuis la page d'accueil par exemple :

src/pages/home/home.ts

```
// config
import { Config } from '../../config/config';
```