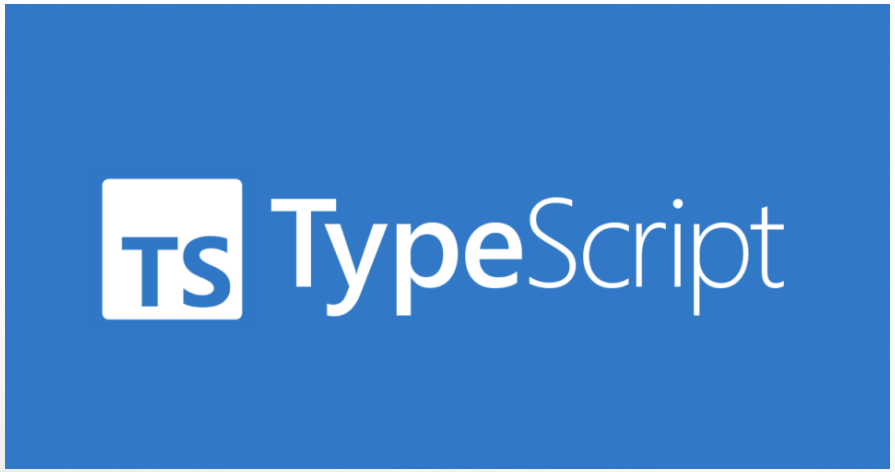
Humanbooster

ANGULAR TYPESCRIPT

Humanbooster

QU'EST CE QUE LE TYPESCRIPT ?



C'est un langage de programmation open source, il a pour but de sécuriser le code Javascript.
Il permet de une approche orienté objet meilleure que Javascript avec un typage fort.

Il respecte les standards EcmaScript6 (ES6).

Un code Javascript est un code Typescript, cependant ce n'est pas le cas pour l'inverse, il faut un compiler Typescript > Javascript

Humanbooster

SYNTAXE TYPESCRIPT

```
let result: string
let index: number
let isEmpty: boolean
const array: Array<number> = [5, 6, 8]
```

Voici un exemple de déclaration de variables en Typescript.
« let » et « const » sont les mots clés pour déclarer une variable en typescript.

- « **let** » est une variable modifiable
- « **const** » est une variable non modifiable, mais vous devrez l'affecter à sa déclaration

« result » est le nom de la variable, suivi de « : » et de son type, ici result est un « string ».

Une variable est utilisable uniquement dans la fonction où elle a été déclarée

Humanbooster

SYNTAXE TYPESCRIPT : LES TYPES

Il existe aussi les type :

- « number » : représente un chiffre (entier ou décimal)
- « Date » : représente une Date
- « boolean » : représente un booléen, true ou false (0 ou 1)
- « array » : représente un tableau, il est souvent accompagné du type contenu dans ce tableau (par exemple : « Array<number> »
- « undefined » : représente une variable non définie, souvent assimilé à « null »
- **« any » : oubliez le, on type nos variables en Typescript ;)**

Il est possible de dire qu'une variable peut-être d'un type ou d'un autre via le « | »

```
let result: string
let index: number|undefined
let isEmpty: boolean
const array: Array<number> = [5, 6, 8]
```

Humanbooster

SYNTAXE TYPESCRIPT : LES FONCTIONS

Une fonction est représentée par :

- « private » : sa visibilité
- « setSelectedCode » : son nom
- « code: string » : son ou ses paramètres, **ils doivent eux aussi être tous typés.**
- « : void » : la valeur de retour de la fonction, void représente le fait que la fonction ne renvoie juste rien.

```
private setSelectedCode(code: string): void {
  const selectedCode: string = 'Oui';
}
```

Humanbooster

SYNTAXE TYPESCRIPT : LES BOUCLES

« for..i » => boucle définie, on sait jusqu'où on doit aller.

« for..in » => boucle non définie, on ne sait pas exactement la taille du tableau, et on laisse Typescript gérer le parcours du tableau. L'itération aura l'index.

« for..of » => boucle non définie, on ne connait pas exactement la taille du tableau, et on laisse Typescript gérer le parcours du tableau. L'itération aura directement la valeur.

Humanbooster

SYNTAXE TYPESCRIPT : INSTALLATION 1/3

Nous allons créer un projet « basique » auquel nous ajouterons Typescript.

Pour cela nous avons besoin de « Node.js » : https://nodejs.org/en/download/

Afin de valider l'installation de « Node.js », ouvrez un terminal et tapez la commande : « **npm –v** »
Vous devriez «**14.18.0** »

Créez un projet vide avec votre IDE (Visual Studio : Code ou PhpStorm ou WebStorm).

Ouvrez un terminal : et tapez la commande : « **npm init –y** », vous devriez voir un fichier « **package.json** » à la racine de celui-ci ».

Lancez ensuite la commande : « **npm install typescript --save-dev** »

```
"devDependencies": {
  "typescript": "^4.4.3"
}
```

Humanbooster

SYNTAXE TYPESCRIPT : INSTALLATION 2/3

```
{
  "exclude": [
    "node_modules",
  ],
  "compilerOptions": {
    "sourceMap": true,
    "downlevelIteration": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "module": "esnext",
    "moduleResolution": "node",
    "target": "es2015",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2018",
      "es2019",
      "dom",
      "dom.iterable",
      "es2015.proxy"
    ]
  }
}
```

Il faut ensuite indiquer à Typescript quel compileur utiliser, pour cela il faut créer à la racine du projet un fichier « **tsconfig.json** »
A l'intérieur de celui-ci ajoutez le code ci-joint.

Créez un dossier « src » et à l'intérieur un fichier « index.ts », faites juste un « **console.log('Oui');** » à l'intérieur.

Afin de compiler votre code Typescript en Javascript, toujours dans un terminal, tapez la commande : « **npx tsc** », vous venez de compiler votre premier code Typescript !

Humanbooster

SYNTAXE TYPESCRIPT : INSTALLATION 3/3

Le problème c'est que nous devons retaper la commande sans arrêt… C'est ennuyeux et pas optimal !

Afin d'améliorer ça, toujours dans un terminal, lancez la commande « **npm install --save-dev ts-node nodemon** », nous allons installer un « watcher » de nos sources, qui les recompilera dès que nous effectuerons un changement dedans, et il exécutera le code ! Toujours à la racine de votre projet, créez un fichier « **nodemon.json** » et ajoutez le code suivant à l'intérieur :

```
{
   "watch": ["src"],
   "ext": ".ts,.js",
   "ignore": [],
   "exec": "ts-node ./src/index.ts"
}
```

Enfin, dans le « package.json », remplacer la ligne « stripts » par cette ligne :

```
"scripts": {
  "start": "nodemon"
},
```

Lancez la commande : «**npm run start**»

Humanbooster

SYNTAXE TYPESCRIPT : LES CLASSES 1/2

Qu'est-ce qu'une classe ?

Il s'agit d'une représentation informatique d'un objet de la vie courante, afin de pouvoir l'utiliser en code.
Par exemple, une « Voiture ».

Une classe est composé d'attributs, ils sont là pour décrire l'objet, par exemple une voiture peut avoir un attribut de son type de carburant, sa couleur, sa marque et son modèle.

Humanbooster

SYNTAXE TYPESCRIPT : LES CLASSES 2/2

On déclare une classe via le mot-clé « class », il faut toujours mettre « export » devant, afin de pouvoir l'utiliser dans d'autres fichiers.
Dans cet exemple, « **Couvert** » est le nom de la classe.

```
export class Couvert {

}
```

Une classe représente un objet, autrement dit un nouveau type utilisable pour votre application !

```
const fourchette: Couvert
```

Humanbooster

SYNTAXE TYPESCRIPT : LES ATTRIBUTS DE CLASSES

```
export class User {

  private _name: string;
  private _email: string;

}
```

Les attributs de classes possèdent un indice de visibilité, ici « private ». Il en existe 3 :

- « private » : indique que l'attribut est visible uniquement dans la classe en cours
- « public » : indique que l'attribut est visible partout
- « protected » : indique que l'attribut est visible uniquement dans la classe en cours et ses filles (héritage)

Un attribut se nomme et se type comme une variable de fonction.

Humanbooster

SYNTAXE TYPESCRIPT : LES GETTER ET SETTER

```
export class User {

  private _name: string;
  private _email: string;

  get name(): string {
    return this._name;
  }

  set name(value: string) {
    this._name = value;
  }

  get email(): string {
    return this._email;
  }

  set email(value: string) {
    this._email = value;
  }
}
```

Lorsqu'un attribut est private, on ne peut pas l'utiliser en dehors de sa classe, ce qui est ennuyeux. Afin de pallier à ce problème on utilise des getters et setters, qui permettent de modifier ou d'afficher l'attribut en dehors de sa classe.

Ils sont représentés par le mot clé « **get** » (récupère l'attribut) et « **set** » (modifie l'attribut)

Le mot clé « this » représente à chaque fois « cette classe », ou la classe en cours, il est nécessaire pour représenter le fait que l'on fait appel à un attribut de la classe.

Humanbooster

SYNTAXE TYPESCRIPT : LE CONSTRUCTEUR

```
export class User {

  private _name: string;
  private _email: string;

  get name(): string {
    return this._name;
  }

  set name(value: string) {
    this._name = value;
  }

  get email(): string {
    return this._email;
  }

  set email(value: string) {
    this._email = value;
  }

  constructor(name: string, email: string) {
    this._name = name;
    this._email = email;
  }
}
```

Une classe peut, ou ne peut pas avoir de constructeur, si elle n'en a pas, par défaut il s'agira d'un constructeur vide, c'est-à-dire qui ne modifie pas notre objet à sa création.

Ici lorsque l'on créera un « User », on lui indiquera son « name » et son « email ».

Afin de créer un objet de type « User », on doit utiliser le mot-clé « **new** » :

```
let user: User = new User('Kevin', 'kevin.hb@hb.fr');
```

Un constructeur fonctionne comme une fonction pour ses paramètres.

Humanbooster

SYNTAXE TYPESCRIPT : LES METHODES

```
export class Car {

  private brand: string;
  private model: string;

  marcheAvant(): void {

  }

  marcheArriere(): void {

  }

}
```

Au sein d'une classe on peut déclarer des méthodes, équivalent des fonctions. Ici, elles représentent un compotement de notre objet.

Par exemple pour une voiture on pourrai avoir une méthode « marcheAvant », « marcheArriere ».

Dans le cadre d'une classe, vous n'avez pas à préciser le mot-clé « function » avant le nom de celle-ci.

Humanbooster

SYNTAXE TYPESCRIPT : LES INTERFACES

```
export interface IUser {
   'name': string;
   'age': number;
}
```

Une interface peut s'assimiler à une classe, **à la différence que celle-ci ne s'instancie pas** et ses attributs n'ont pas de visibilité.
Ses attributs doivent être déclarées entre simple quote et toujours typés.

```
const user: IUser = {
   'name': "Jean-Michel",
   'age': 42
}
```

Afin de réutiliser notre interface, on affecte ses valeurs en reprenant les noms de ses attributs entre simple quote suivi de « : » et de la valeur de l'attribut.

Humanbooster

SYNTAXE TYPESCRIPT : L'HERITAGE 1/4

```
export class Admin extends User {

}
```

Le but de l'héritage est de déclarer du contenu (attributs ou méthodes) dans une classe, et via l'héritage de pouvoir en faire bénéficier ses classes « filles ». Ainsi, cela évite de répéter du code lorsque plusieurs objets ont un comportement en commun.

Afin d'écrire qu'une classe hérite d'une autre, on utilise le mot-clé « **extends** » puis le nom de la classe que l'on souhaite hériter.

Ainsi ma classe « **Admin** » aura les mêmes attributs que la classe « **User** », on peut dire qu'un « **Admin** » est un « **User** », mais un « **User** » n'est pas un « **Admin** ».

Humanbooster

SYNTAXE TYPESCRIPT : L'HERITAGE 2/4

```
export class Admin extends User {

}
```

Le but de l'héritage est de déclarer du contenu (attributs ou méthodes) dans une classe, et via l'héritage de pouvoir en faire bénéficier ses classes « filles ».
Ainsi, cela évite de répéter du code lorsque plusieurs objets ont un comportement en commun.

Afin d'écrire qu'une classe hérite d'une autre, on utilise le mot-clé « **extends** » puis le nom de la classe que l'on souhaite hériter.

Ainsi ma classe « **Admin** » aura les mêmes attributs que la classe « **User** », les mêmes méthodes **et il sera possible de redéfinir des méthodes propres à un « Admin », que le « User », n'aura pas d'accès.**

On dit qu'un « **Admin** » est un « **User** », mais un « **User** » n'est pas un « **Admin** ».

Humanbooster

SYNTAXE TYPESCRIPT : L'HERITAGE 3/4

```
export abstract class User {

}
```

En héritage, il est possible d'avoir des classes dites « **abstraites** », elles sont représentées par le mot clé « **abstract** ».

A quoi servent-elles ?

Il s'agit d'une classe qui ne s'instancie pas, selon l'exemple on peut pas directement instancier de classe « User ».
On peut instancier ses classes filles, une classe abstraite est utilisée lorsque ses filles ont des méthodes très spécifique à redéfinir, et que leurs attributs sont communs.

Humanbooster

SYNTAXE TYPESCRIPT : L'HERITAGE 4/4

```
export abstract class User {

  abstract getRight(): string;

}

export class Admin extends User {

  getRight(): string {
    return "J'ai les droits Admin";
  }

}
```

Il est aussi possible de déclarer une méthode « **abstraite** », seulement dans une classe abstraite.
Dans ce cas, on ne déclare pas de code dans celle-ci, juste ses éventuels paramètres et son type de retour.
Par contre, **ses classes filles ont obligation de la redéfinir.**

L'intérêt est d'imposer un comportement (méthode) à toutes les classes filles, et chacune pourra définir son propre comportement.