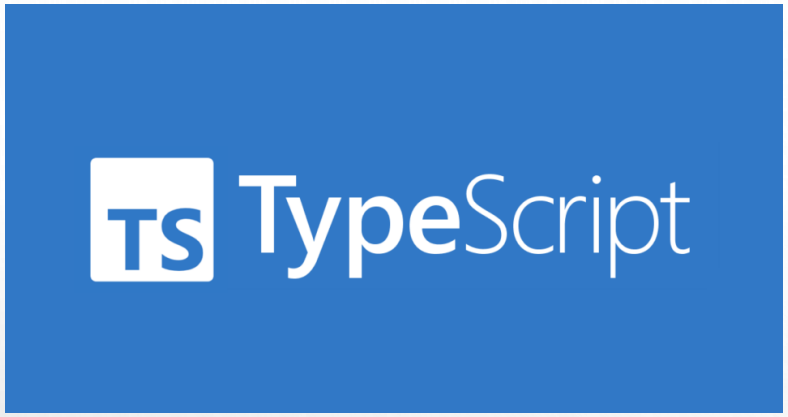
TYPESCRIPT

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 vers Javascript

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 de type « **string** ».

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

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>** ». Il est aussi possible de l'écrire de cette manière : « **number[]** », les deux syntaxes sont identiques.
- « **undefined** » : représente une variable non définie, souvent assimilé à « null » dans d'autres langages

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

Il est possible de dire qu'une variable peut-être d'un type ou d'un autre via le « | ». On s'en sert souvent pour indiquer qu'une variable peut-être undefined.

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.
- Par défaut, si on ne précise pas de visibilité, la fonction est « **publique** »

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

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.

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, est un **nouveau type** utilisable pour votre application !

```
const fourchette: Couvert
```

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.

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.

```
var myFoo = new foo();
if(myFoo.bar) {          // calls the getter
    myFoo.bar = false;   // calls the setter and passes false
}
```

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@drosalys.fr');
```

Un constructeur fonctionne comme une fonction pour ses paramètres.

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 comportement 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.

Afin de l'appeler il faut l'appeler avec un « . » sur un objet du type ayant la méthode, ici « Car » :

```
let car: Car = new Car();
car.marcheAvant();
```

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.

```
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.

Une interface peut aussi simplement contenir des méthodes, sans implémentations. Il faut voir une interface comme un « contrat » à respecter.

SYNTAXE TYPESCRIPT : L'HERITAGE 1/5

```
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 dites « 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** ».

SYNTAXE TYPESCRIPT : L'HERITAGE 2/5

```
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** ».

SYNTAXE TYPESCRIPT : L'HERITAGE 3/5

A l'intérieur d'une classe fille, on peut surcharger les méthodes de la classe mère.
Le mot clé « **override** » apparaît et on peut rappeler les comportements de la classe mère avec le mot-clé « **super** ».

```
override moveBackward() {
  super.moveBackward();
  console.log('Dans une voiture');
}
```

SYNTAXE TYPESCRIPT : L'HERITAGE 4/5

```
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.

SYNTAXE TYPESCRIPT : L'HERITAGE 5/5

```
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.