TYPESCRIPT : L'HÉRITAGE

DÉFINITION

- Troisième pilier de la POO
- Permet de transmettre des attributs et méthodes d'une classe à lautre
- Les paramètres et méthodes seront transmises ou non en fonction de leur portée

SYNTAXE

```
class Class1 {
  // Définition
}

class Class1 extends Class2 {
    // Définiton
}
```

EXEMPLE

Reprennons maintenant la classe Formateur pour l'améliorer :

```
class Person {
  protected nom: string;
  protected prenom: string;
  protected dateNaissance: Date;

  constructor(nom: string, prenom: string, dateNaissance: Date) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNaissance = dateNaissance;
  }

  get identity(): string {
    return `${this.nom} ${this.prenom} a ${this.age}`;
  }

  get age(): number {
```

EXEMPLE

Reprennons maintenant la classe Formateur pour l'améliorer :

```
22      return age;
23    }
24  }
25
26  class Formateur extends Person {
27      private domaines!: string[];
28
29      constructor (nom: string, prenom: string, dateNaissance: Date, ...doma
30          super(nom, prenom, dateNaissance);
31          this.domaines = domaines
32      }
33
34      get allDomaines () :string {
35          const domainList: string = this.formatDomaines();
36          return `${this.prenom} ${this.nom} forme sur : ${domainList}`
```

EXEMPLE

Reprennons maintenant la classe Formateur pour l'améliorer :

```
23    }
24  }
25
26  class Formateur extends Person {
27      private domaines!: string[];
28
29      constructor (nom: string, prenom: string, dateNaissance: Date, ...dom
30          super(nom, prenom, dateNaissance);
31          this.domaines = domaines
32      }
33
34      get allDomaines () :string {
35          const domainList: string = this.formatDomaines();
36          return `${this.prenom} ${this.nom} forme sur : ${domainList}`
37      }
```

SUPER ?

- Le mot clef `super` permet d'invoquer (récupérer) les arguments de la super-classe
- Dans l'exemple précédent, on récupère les arguments du constructeur parent

SUPER !

- `super` peut être utilisé pour invoquer des méthodes statiques
- Il peut aussi être utilisé sur les objets littéraux

PARAMÈTRES ET MÉTHODES

DÉFINITIONS

- La classe parent est appelée **super classe**
- La classe enfant est appelée **sous classe**

PARAMÈTRES

- Les paramètres à portée **public** et **protected** sont hérités
- Les autres paramètres ne sont pas transmis

LES MÉTHODES

- Comme pour les paramètres, les méthodes sont transmises selon leur portée
- Une méthode héritée peut être redéfinie lors de la réception (override)

SURCHARGES ET REDÉFINITION

SURCHARGE (OVERLOAD)

- Une fonction est définie par son nom et le type de ses paramètres
- C'est ce qu'on appelle la signature d'une méthode
- La signature d'une méthode est unique au niveau de la classe
- Il n'est pas possible d'implémenter deux méthodes distinctes dans une même classe (contrairement à Java)

SURCHARGE (OVERLOAD)

- Finalement, l'action réalisée sera la même
- Le type de paramètres peut toutefois changer, ce qui impac le résultat

EXEMPLE SANS CLASSE :

Un exemple un peu bête et méchant :

```
1  // On déclare les paramètres de la fonction :
2  function mafonction(a:string, b:string):string;
3
4  // On déclare les paramètres alternatifs
5  function mafonction(a:number, b:number): number;
6
7  // L'opération à réaliser est la 'même' :
8  function mafonction(a: any, b:any): any {
9  return a + b;}
10
11 console.log(mafonction('Coucou ', 'toi !')) // Coucou toi !
12 console.log(mafonction(5, 10)) // 15
```

EXEMPLE SANS CLASSE :

Un exemple un peu bête et méchant :

```
1  // On déclare les paramètres de la fonction :
2  function mafonction(a:string, b:string):string;
3
4  // On déclare les paramètres alternatifs
5  function mafonction(a:number, b:number): number;
6
7  // L'opération à réaliser est la 'même' :
8  function mafonction(a: any, b:any): any {
9  return a + b;}
10
11 console.log(mafonction('Coucou ', 'toi !')) // Coucou toi !
12 console.log(mafonction(5, 10)) // 15
```

EXEMPLE SANS CLASSE :

Un exemple un peu bête et méchant :

```
1  // On déclare les paramètres de la fonction :
2  function mafonction(a:string, b:string):string;
3
4  // On déclare les paramètres alternatifs
5  function mafonction(a:number, b:number): number;
6
7  // L'opération à réaliser est la 'même' :
8  function mafonction(a: any, b:any): any {
9  return a + b;}
10
11 console.log(mafonction('Coucou ', 'toi !')) // Coucou toi !
12 console.log(mafonction(5, 10)) // 15
```

EXEMPLE DANS UNE CLASSE :

```
class Birthday {

    birthday!: Date

    setBirthday(birthday: string): void;
    setBirthday(birthday: Date): void;

    setBirthday(birthday: any): void {
        if (birthday instanceof Date) {
            this.birthday = birthday
        }
        else if (typeof birthday === "string") {
            this.birthday = new Date (birthday)
        }
    }
}
```

Ici, avec la même fonction : On peut traiter l'information de deux manières différentes

EXEMPLE DANS UNE CLASSE :

```
class Birthday {

    birthday!: Date

    setBirthday(birthday: string): void;
    setBirthday(birthday: Date): void;

    setBirthday(birthday: any): void {
        if (birthday instanceof Date) {
            this.birthday = birthday
        }
        else if (typeof birthday === "string") {
            this.birthday = new Date (birthday)
        }
    }
}
```

Ici, avec la même fonction : On peut traiter l'information de deux manières différentes

EXEMPLE DANS UNE CLASSE :

```
4
5       setBirthday(birthday: string): void;
6       setBirthday(birthday: Date): void;
7
8       setBirthday(birthday: any): void {
9           if (birthday instanceof Date) {
10              this.birthday = birthday
11          }
12          else if (typeof birthday === "string") {
13              this.birthday = new Date (birthday)
14          }
15      }
16  }
17
18  let myBirthday: Birthday = new Birthday
19
```

Ici, avec la même fonction : On peut traiter l'information de deux manières différentes

EXEMPLE DANS UNE CLASSE :

```
10              this.birthday = birthday
11          }
12          else if (typeof birthday === "string") {
13              this.birthday = new Date (birthday)
14          }
15      }
16  }
17
18  let myBirthday: Birthday = new Birthday
19
20  myBirthday.setBirthday('1992/07/20')
21  console.log(myBirthday.birthday)
22
23  myBirthday.setBirthday(new Date('1992/07/20'))
24  console.log(myBirthday.birthday)
```

Ici, avec la même fonction : On peut traiter l'information de deux manières différentes

REDÉFINITION (OVERRIDE)

- Une redéfinition peut avoir lieu lors d'un héritage
- La fonction de la super-classe peut ne pas être adaptée à sa sous-classe
- On conservera le même nom de fonction, en changeant le code qui la compose
- Cette fois les paramètres ne peuvent pas changer !

```
class Bonjour {
    protected nom: string
    constructor(nom: string){
        this.nom = nom
    }
    sayHello(): string {
        return `Bonjour ${this.nom} ! Je suis la super-classe`
    }
}

class OverrideBonjour extends Bonjour {
    sayHello(): string {
        return `Bonjour ${this.nom} ! J'ai été override !`
    }
}
```

```
class Bonjour {
    protected nom: string
    constructor(nom: string){
        this.nom = nom
    }
    sayHello(): string {
        return `Bonjour ${this.nom} ! Je suis la super-classe`
    }
}

class OverrideBonjour extends Bonjour {
    sayHello(): string {
        return `Bonjour ${this.nom} ! J'ai été override !`
    }
}
```

ABSTRACTION

DÉFINITION

- Quatrième pilier de la POO
- Le terme **classe abstraite** désigne une super classe qui n'a pas lieu d'être instanciée
- La classe sert 'uniquement' à transmettre attributs et méthodes à ses enfants

DÉFINITION

- Permet de créer un comportement générique, qui sera utilisé par des classes dérivées
- Les classes abstraites permettent de définir des méthodes sans implémentation

SYNTAXE

```
1 abstract class Nom {
2   // Paramètres
3   // Constructeur
4   // Méthodes
5 }
```

SYNTAXE

```
1 // Déclarer une classe abstraite
2 abstract class Nom {
3   // Avec les méthodes que ses sous classes doivent posséder :
4   abstract methode (param1: type, ...): type;
5 }
```

- On peut aussi passer des méthodes abstraites, non implémentées
- La sous classe devra implémenter elle même la logique de cette fonction
- Elle devra en revanche respecter les types et nombres de paramètres

SYNTAXE

```
// Déclarer une classe abstraite
abstract class Nom {
  // Avec les méthodes que ses sous classes doivent posséder :
  abstract methode (param1: type, ...): type;
}
```

- On peut aussi passer des méthodes abstraites, non implémentées
- La sous classe devra implémenter elle même la logique de cette fonction
- Elle devra en revanche respecter les types et nombres de paramètres

EXEMPLE

```
abstract class ClasseAbstraite {
    abstract doSomething(): void;
}

class ClasseConcrete extends ClasseAbstraite {
    doSomething(): void {
        console.log('something')
    }
}
```

DÉMONSTRATION !

Have fun with abstraction !

EXERCICE !

Réalisez l'exercice 4

LA SUITE

Par ici !