Introduction à la programmation Orientée Objet

La programmation orientée objet ou **POO** est un paradigme de programmation qui vise à diviser le code en plusieurs partie appelée "objet". Un objet est composée de variables nommée attributs et de fonctions nommée méthodes. La structure d'un objet est basé sur une structure de donnée appelée classe.

Exemple de POO en TypeScript

```
// Une classe, un patron de conception de personne
class Person{
    // Atributs
    name : string;
    birthDate : Date;
    gender : string;

    //Methodes
    constructor(name : string, birthDate : Date, gender : string)
    {
        this.name = name;
        this.birthDate = birthDate;
        this.gender = gender;
    }

    hello(){
        console.log("Bonjour je m'appelle",name,", je suis un(e)",gender,"né(e) en",birt
    }
}
// Instanciation d'une classe via l'operateur new
const prof = new Person("Massinissa",new Date("12-31-1999"), "Homme");
prof.hello();   // Appel d'une méthode de la classe Person
```

La POO respecte 3 principes fondamentaux :

- **L'encapsulation.** Un objet doit être utilisable sans néccessitée la comprehension de son fonctionnnement interne. Un objet doit être utilisable sans néccessitée la comprehension de son fonctionnnement interne. L'encaplusation de l'objet est défini grâce à un classe, c'est à dire un

patron de conception réutilisable qui défini les méthodes et attributs comme entant privés ou accessibles publiquement. Au même titre qu'il n'est pas obligatoire de connaitre le fonctionnement interne d'une voiture pour utiliser le volant et les pédales, il n'est pas neccessaire de connaitre le fonctionnement interne d'un objet pour utiliser ses méthodes et attributs publiques.

- **L'héritage.** Une classe peut, lors de sa création, hériter d'une autre classe et ainsi posséder tout ses attributs et méthodes. La classe qui hérite est alors appelée classe enfant. Cette mécanique très puissante permet d'éviter la duplication de code et donc accélérer la vitesse de développement d'une application. Un voiture et une moto sont tout les deux des véhicules mais ne sont pas parfaitement identiques. Il peuvent tout les deux accelerer, tourner, démarrer, faire le plein d'essence mais une moto n'à pas la clim et une voiture possède des fenetres et un système de fermeture centralisé.
- **Le polymorphisme**, du grec polúmorphos (« multiforme »),la capacité à se présenter sous différentes formes. Une classe enfant peut modifier le contenu des méthodes de sa classe parent. La classe Moto et la classe Voiture héritent tout les deux de la classe Véhicule et ces classes vont effectuer une polymorphie de la méthode accélerer car une moto accélere grace à une chaine et la voiture grâce à un axe. Le polymorphisme prolonge la puissance de l'encapsulation, vu de l'exterieur deux véhicules qu'il soit de la classe Moto ou de la classe Voiture accelère via la même méthode `accélerer` , alors qu'en réalité le contenu de leurs méthodes diffère grandement.

Utilité de la POO et la programmation procédural

Jusque ici vous avez probalement uniquement fait de la programmation procéduale où l'encapsulation ne se fait que dans une fonction. C'est une manière de coder tout à fait viable et elle est d'ailleurs à la base de tout les systèmes d'exploitation moderne car le langage C est un langage procedurale.

Cepandant la complexité des projets de développement web ne cessent d'augementer depuis les années 2010 et l'époque où du PHP procédurale et quelques script JavaScript composait le web est bien lointaine. La complexité des projets d'aujourd'hui demande une code-base modulaire, évolutive et un rythme de développement rapide.

La POO est une solution à ces problèmes.

> Les exemples suivant utilise la syntaxe de TypeScript qui est proche de la syntaxe de Java.

L'encapsulation

L'encapsulation est défini dans une classe, un objet est l'instance d'une classe, une classe peut concevoir plusieurs instances et donc plusieurs objets.

En TypeScript et dans la plupart des langages l'instanciation d'un objet se fait via l'operateur `new` .

```
// Déclaration de la classe Voiture
class Voiture{

}

// Instanciation de 3 instances(objets) de la classe Voiture.
let peugeot = new Voiture();
let renault = new Voiture();
let bmw = new Voiture();
```

Une classe est composée de deux élements : des attributs(variables) et des méthodes(fonctions).
Les attributs ne sont rien de plus que des variables innerente à l'objet, les méthodes quand à elle sont des fonctions.
L'encapsulation permet de définir les droits d'accès des élements d'un objet, les élements peuvent être privée ou publiques. Les élements publiques sont utilisables à l'exterieur de la l'objet alors que les élement privées ne sont connu que de l'objet.

Attributs publiques

Les attributs sont défini dans la classe, ici je défini les trois attributs de la classe Voiture comme publique, je vais donc pouvoir les modifier depuis l'exterieur de l'objet.

```
class Voiture{
    public marque : string;
    public vitesse : number;
    public nbPortes : number;
}
let voiture = new Voiture();
voiture.marque = "Renault";
voiture.vitesse = 20;
voiture.nbPortes = 5;
```

Constructeur

La plupart du temps il est neccéssaire d'initaliser un objet avant de l'instancier, l'initialisation d'un objet se fait dans une méthode publique appellée `construteur` .

```
class Voiture{
    public marque : string;
    public vitesse : number;
    public nbPortes : number;

    constructor(marque : string, nbPortes : number = 5){
        this.marque = marque;
        this.nbPortes = nbPortes;   // Par défaut une voiture à 5 portes
        this.vitesse = 0;           // On initalise la vitesse à 0kmh
    }
}
let clio = new Voiture("Renault");
let a1 = new Voiture("Audio",3);
let twingo = new Voiture("Renault",3);
```

> **Constructeur publique**
> **Par defaut le constructeur est publique** et il plutot rare de le rendre privé car cela empecherait d'instancier librement l'objet. Cependant certain cas l'exige comme le design pattern Singleton qui permet de s'assurer qu'un objet n'existe quand un et UN SEUL exemplaire.
> Plus d'info sur les singletons ici : https://refactoring.guru/fr/design-patterns/singleton

Attributs privé

La POO, dans sont application la plus stricte, exiges que tout les attributs soit privés celà permet de rendre l'objet plus robuste et d'eviter les non-senses.
Une voiture par exemple ne peut pas avoir un nombre de portes négatif ou encore un marque vide.

Pourtant la règlee `public` défini précedement sur nos attributs le permet.

```
class Voiture{
    public marque : string;
    public vitesse : number;
    public nbPortes : number;

    constructor(marque : string, nbPortes : number = 5){
        this.marque = marque;
        this.nbPortes = nbPortes;   // Par défaut une voiture à 5 portes
        this.vitesse = 0;           // On initalise la vitesse à 0kmh
    }
}
let clio = new Voiture("Renault");
/* La voiture n'à plus de marque alors que le constructeur
etait sensé rendre sa présence obligatoire */
clio.marque = "";
clio.nbPortes = -12;       // -12 portes ?!
clio.vitesse = 100000;  // 100 000 kmh c'est un peu beaucoup non ?
```

Pour palier à ça on va passer les atributs en privées et créer des méthodes publiques pour y accéder.

Je rajoute la regle privée sur mes attributs.

```
class Voiture{
    private marque : string;
    private vitesse : number;
    private nbPortes : number;

    constructor(marque : string, nbPortes : number = 5){
        this.marque = marque;
        this.nbPortes = nbPortes;   // Par défaut une voiture à 5 portes
        this.vitesse = 0;           // On initalise la vitesse à 0kmh
    }
}
let clio = new Voiture("Renault");
/* La voiture n'à plus de marque alors que le constructeur
etait sensé rendre sa présence obligatoire */
clio.marque = "";   // ERROR
```

Méthodes publiques

Une méthode est une fonction interne de l'objet. Au même titre que les attributs ils peut y avoir des méthodes privées ou publiques.

```
class Player{
    private pv : number = 10;

    public loosePv() : void
    {
        if(this.pv > 0) this.pv--;
    }
}
const mario = new Player();
mario.loosePv();
```

Méthodes privée

Parfois nous avons besoin de créer une fonction sans la rendre publique, dans ce cas là il faut définir une méthode privée.

```
class Voiture{
    private vitesse : number;
    private carburant : number;     // Litre d'essence

    constructor(carburant : number){
        this.vitesse = 0;
        this.carburant = carburant;
    }

    /**
     * La méthode accelerer doit être publique pour que la voiture se
     *  déplace lors de l'appuie d'une touche du clavier.
     * */
    public accelerer() : void{
        if(this.alimenterMoteur() > 0){
            this.vitesse++;
        }
    }
    /**
     * Par contre la consommation de l'essence est innérent
     * au fonctionnement interne de la voiture.
     * C'est donc une méthode privée, innaccessible du reste du programme.
     * */
    private alimenterMoteur() : number{
        this.carburant-=0.1;
        return this.carburant;
    }
}
// Instanciation
let clio = new Voiture(10);
document.onkeydown = function(event){
    // Si l'utilisateur appuie sur Z
    if(event.key == "Z"){
        // La voiture accelere.
        clio.accelerer();
    }
};
```

Getter et Setter

Les attributs se doivent d'être privés, mais il faut parfois tout de même toujours pouvoir y accéder. Pour ceci on créer des méthodes publiques appellée Getter et Setter.
Les Getter renvoi la valeur de l'attribut demander alors que le setter modifier la valeur de l'attribut en s'assurant que la valeur fournit est correct.

```
class Personnage{
    private pv : number;
    private endurance : number;
    constructor(pv : number)
    {
        this.pv = pv;
        this.endurance = 100;
    }

    public getPv() : number
    {
        return this.pv;
    }

    public setPv(pv : number) : void
    {
        if(pv >= 0){
            this.pv = pv;
        }
    }
    public getEndurance() : number
    {
        this.endurance;
    }
    public setEndurance(endurance : number) : number
    {
        if(endurance >= 0 && endurance <= 100){
            this.endurance = endurance ;
        }
    }
}
```

La présence de getter et setter ralonge le nombre de ligne de code mais permet une encapsulation correct et améliore la robustesse du code. La programmation orientée objet ne cherche pas à réduire

le nombre de ligne mais à diminuer le temps de développement via des classes réutilissable et robuste. En programmation plus de ligne ne peut va forcement dire plus long à développer.

L'héritage

L'héritage c'est le partage des attributs et méthodes d'une classe mère vers des classes enfants.

> Pour la suite je ne précise pas sitématiquement les getter et setter pour réduire la taille des extraits de code.

```
class Personnage{
    private pv : number;
    private endurance : number;
    constructor(pv : number){
        this.pv = pv;
        this.endurance = 100;
    }
}
class Player extends Personnage{
    private inventaire : array = [];
    private pseudo : string;
}
class Monster extends Personnage{
    private zoneAgro : number;
}
const link = new Player(3);     // Link à 3 coeurs
const goblin = new Monster(1);  // Le goblin n'à qu'un coeur
```

Rajoutons les getter et setter aux attributs privés des classes enfants.

```
class Personnage{
    private pv : number;
    private endurance : number;
    constructor(pv : number){
        this.pv = pv;
        this.endurance = 100;
    }
}
class Player extends Personnage{
    private inventaire : array = [];
    private pseudo : string;

    public setPseudo(pseudo : string) : void
    {
        if(pseudo.length > 3)
            this.pseudo = pseudo;
    }

    public getPseudo() : string
    {
        return rhis.pseudo;
    }
    public loot(item : string){
        this.inventaire.push(item);
    }
}
class Monster extends Personnage{
    private zoneAgro : number;
    public setZoneAgro(rayon : number)
    {
        if(rayon > 0){
            this.zoneAgro = rayon;
        }
    }
}
const joueur = new Player(3);     // Le joueur à 3 pv
joueur.setPseudo("Xx_bastien_83");

const goblin = new Monster(1);  // Le goblin n'à qu'un pv.
goblin.setZoneAgro(3);      // Zone d'agro de 3 metre.
```

Surcharge du constructeur

Il est possible de surcharger le constructeur des classes enfants, c'est à dire créer un constructeur spécifique à une classe fille différent de celui de son parent.

Le soucis dans l'exemple de code précedent c'est qu'il est obligatoire d'utiliser les méthodes setter des classes Monster et Player pour initialiser correctement les objets.

```
const joueur = new Player(3);     // Le joueur à 3 pv
joueur.setPseudo("Xx_bastien_83");

const goblin = new Monster(1);  // Le goblin n'à qu'un pv.
goblin.setZoneAgro(3);      // Zone d'agro de 3 metre.
```

Seulement, le pseudo du joueur et la zone d'agro du goblin sont des informations obligatoire au fonctionnement des l'objets,il faud donc les demander dans le constructeur au moment de l'initialisation de l'objet.

C'est possible grâce à la surcharge du constructeur.

```
class Personnage{
    private pv : number;
    private endurance : number;
    constructor(pv : number){
        this.pv = pv;
        this.endurance = 100;
    }
}
class Player extends Personnage{
    private inventaire : array = [];
    private pseudo : string;
    /**
     * Constructeur surchargé
     * */
    constructor(pv : number, pseudo : string){
        /*
        La fonction super en JavaScript permet d'appeler le constructeur parent.
        Ce qui est obligatoire lors de la surcharge d'un constructeur.
        */
        super(pv);
        this.setPseudo(pseudo);
    }

    public setPseudo(pseudo : string) : void
    {
        if(pseudo.length > 3)
            this.pseudo = pseudo;
    }

    public getPseudo() : string
    {
        return rhis.pseudo;
    }
    public loot(item : string){
        this.inventaire.push(item);
    }
}
```

En TypeScript et JavaScript pour surchargé le constructeur il faut utiliser la fonction `super()` dans le constructeur enfant pour appeler le constructeur parent.

```
...
super(pv);
...
```

Il sera également probalement néccessaire de demander dans le constructeur enfant les paramètres obligatoire du constructeur parent.

```
constructor(pv : number, pseudo : string) {
    super(pv);
}
```

Le code précedent,

```
const joueur = new Player(3);
joueur.setPseudo("Xx_bastien_83");
```

devient donc,

```
const joueur = new Player(3,"Xx_bastien_83");
```

Exercice - Constructeur surchargé.

Consigne : Surchargez le constructeur de la classe `Monster` pour rendre obligatoire la présence d'une zone d'agro.

```
class Personnage{
    private pv : number;
    private endurance : number;
    constructor(pv : number){
        this.pv = pv;
        this.endurance = 100;
    }
}
class Monster extends Personnage{
    private zoneAgro : number;
    public setZoneAgro(rayon : number)
    {
        if(rayon > 0){
            this.zoneAgro = rayon;
        }
    }
    public getZoneAgro() : number{
        return this.zoneAgro;
    }
}

/*
Ce code à un défaut, la zone d'agro devrait être obligatoire
mais l'utilisateur de la classe peut oublier d'utiliser la méthode setZoneAgro().
*/
const goblin = new Monster(1);  // Le goblin n'à qu'un pv.
goblin.getZoneAgro();      // Error : Undefined
```

Règle protected

La rêgle `protected` permet de définir des attributs ou méthodes privées, donc inaccessible de l'exterieur de l'objet, mais tout de même accessible au enfant.

En effet, vous ne vous en êtes peut être pas rendu compte mais la règle `private` bloque l'accès à tout objet exterieur à lui même y compris aux enfants de la classe.

Par exemple, ici les attributs `pv` et `endurance` sont inutilisables par la classe `Monster` .

```
class Personnage{
    private pv : number;
    private endurance : number;
    constructor(pv : number){
        this.pv = pv;
        this.endurance = 100;
    }
}
class Monster extends Personnage{
    private zoneAgro : number;
    constructor(zoneAgro : number){
        super(10);  // Les monstre on 1 pv par défaut
        this.zoneAgro = zoneAgro;

        this.endurance = 100;
        // ERROR : endurance est un attribut private et donc exclusivement
        // accessible à la classe Personnage.

    }
    public setZoneAgro(rayon : number)
    {
        if(rayon > 0){
            this.zoneAgro = rayon;
        }
    }
    public getZoneAgro() : number{
        return this.zoneAgro;
    }
}
```

Une solution serait d'utiliser des `setter` et des `getter` publique qui rendrait accessible à tous, y compris les classes filles, les attributs pv et endurance. Mais dans le cas où la regle `public` est trop permisive on utilise la rêgle `protected` .

La rêgle `protected` fonctionne de la même façon que `private` MAIS sans bloquer l'accès au enfant.

```
class Personnage{
    protected pv : number;
    protected endurance : number;
    constructor(pv : number){
        this.pv = pv;
        this.endurance = 100;
    }
}
class Monster extends Personnage{
    private zoneAgro : number;
    constructor(zoneAgro : number){
        super(10);  // Les monstre on 1 pv par défaut
        this.zoneAgro = zoneAgro;

        this.endurance = 100; // OK

    }
    public setZoneAgro(rayon : number)
    {
        if(rayon > 0){
            this.zoneAgro = rayon;
        }
    }
    public getZoneAgro() : number{
        return this.zoneAgro;
    }
}
```

Les attibuts `pv` et `endurance` sont maintenant accessible à la classe `Monster` sans pour autant être rendu vulnérable par une rêgle d'accès `public` .

Le polymorphisme

Le polymorphisme permet à une classe enfant de modifier les méthodes `protected` et `public` de sa classe parent.

```
class Animal{
    private name : string;
    private espece : string;

    constructor(name : string, espece : string){
        this.name = name;
        this.espece = espece;
    }

    public info(){
        console.log(this.name,"est un",this.espece);
    }

    public crier(){
        console.log("L'animal crie !");
    }
}
class Chat extends Animal{
    // La classe Chat redifini la méthode crie
    crier(){
        console.log("Miaou !");
    }
}
class Chien extends Animal{
    // La classe Chien redifini la méthode crie
    crier(){
        console.log("Ouaf !");
    }
}
const chat = new Chat("Caramel","Felis silvestris");
const chien = new Chien("Médor","Canis lupus");
const dinosaure = new Animal("Trex","Tyrannosaurus");

chat.crier();        // => Miaou !
chien.crier();       // => Ouaf !
dinosaure.crier();   // => L'animal crie !
```

> La polymorphie est possible pour les méthodes public ou protected de la classe mère uniquement. Les méthodes private sont par définition innaccessible il nous est donc impossible de les modifier.

La polymorphie permet de définir un comportement par défini et de s'assurer une constence dans le nommage des méthodes de classes filles. Peut importe la classe tant qu'elle hérite de la classe Animal le crie se produit avec la méthode `crier()` .

C'est d'autant plus pratique quand une fonction neccéssite un Animal en paramètre mais na pas besoin de savoir si on parle d'un chat ou d'un chien.

```
class Zoo{
    private animals : Animal[] = [];
    addAnimal(newAnimal : Animal){
        this.animals.push(newAnimal);
    }
    showAnimals(){
        console.log("Bonjour mes petits :)");
        this.animals.forEach(animal=>{
            animal.crier();
            animal.info();
        });
    }
}
const zoo = new Zoo();
zoo.addAnimal(new Chat("Minette","Felis silvestris"));
zoo.addAnimal(new Chien("Patapouf","Canis lupus"));
zoo.addAnimal(new Animal("Raptor","Velociraptor"));
zoo.showAnimals();
```

Résultat dans la console

```
Bonjour mes petits :)
Miaou !
Minette est un Felis silvestris
Ouaf !
Patapouf est un Canis lupus
L'animal crie !
Raptor est un Velociraptor
```

La généricité

La généricité c'est la capacité d'un code à être modulaire, plusieurs classes peuvent être utiliser de façon interchangable grace à l'héritage et le polymorphisme permet de garder une constance dans le nommage des méthodes à travers toutes ces classes.

```
// addAnimal(newAnimal : Animal){...}
zoo.addAnimal(new Chat("Minette","Felis silvestris"));
zoo.addAnimal(new Chien("Patapouf","Canis lupus"));
zoo.addAnimal(new Animal("Raptor","Velociraptor"));
```

Dans le cas de la POO une classe comme Animal peut être utiliser comme type attendu des paramètres d'une méthode même si La classe effective est un Chat ou Chien.

L'héritage et le polymorphisme permettent donc une généricité de notre code en complement l'encapsulation permet rendre les objets facile d'utilisation via des méthodes publiques.
Le tout forme une code base évolutive et plus rapide à developper qu'une code base procédurale classique.