Développement orienté objet

Encapsulation

Définition

L'encapsulation est un principe de programmation qui consiste à regrouper des données et les méthodes qui les manipulent dans une même entité. Cette entité est appelée **classe**.

Le but est de **restreindre** la visibilité des attributs (ainsi que de certaines méthodes). L'utilisateur d'une classe n'a pas besoin de connaître comment l'information est stockée, il doit juste savoir comment l'utiliser via les méthodes publiques.

Exemple : vous ne savez pas comment est stocké un tableau dans la classe `Array` mais vous savez comment l'utiliser via les méthodes `push`, `pop` ...

Visibilité des attributs et méthodes

Il existe différents niveau de visibilité des attributs et méthodes. Les deux les plus courantes sont :

- **privé (private) :** l'attribut ou la méthode n'est accessible que depuis les instances de la classe elle-même (c'est-à-dire depuis les méthodes de la classe).
- **public (public) :** l'attribut ou la méthode est accessible depuis n'importe où (c'était le cas jusque maintenant).

Remarque : dans le reste du cours, on se focalisera sur les attributs mais les méthodes sont concernées par les mêmes règles. On mettra une méthode privée si ce n'est pas une méthode destinée à être utilisée par l'utilisateur de la classe mais qu'elle sert plutôt à simplifier le code de la classe et à être appelée par d'autres méthodes de la classe.

Principe d'encapsulation

Dans le principe d'encapsulation, on rend **systématiquement** privé les attributs d'une classe. Le but est de garantir que la modification de ces attributs ne se fassent en respectant les règles de la classe.

Par exemple, si on a une classe `Personne` avec un attribut représentant son numéro d'INE. On veut donner ce numéro à la création de l'objet mais ensuite il ne doit plus être modifiable. Actuellement, rien n'empêche d'avoir le code suivant :

```
class Personne {
    ine: string;

    constructor(ine: string) {
        this.ine = ine;
    }
```

```
    // ...
}

let p = new Personne("123456789");
p.ine = "987654321"; // On peut modifier l'INE
```

On peut aussi vouloir imposer des règles lors de l'affectation d'une valeur à un attribut. Par exemple, on peut vouloir imposer qu'un attribut ne puisse pas être négatif.

```
class Personne {
    age: number;

    constructor(age: number) {
        if (age < 0) {
            throw new Error("L'âge ne peut pas être négatif");
        }
        this.age = age;
    }

    // ...
}

let p = new Personne(20);
p.age = -5; // On peut mettre un âge négatif
```

L'utilisateur n'ayant pas accès directement aux attributs (la donnée), il ne peut interagir avec la classe que via les méthodes publiques qui forme **l'interface publique** de la classe. On peut donc imposer des règles lors de l'appel de ces méthodes.

```
class Personne {
    private age: number; // private rend l'attribut privé

    constructor(age: number) {
        this.setAge(age);
    }

    setAge(age: number) {
        if (age < 0) {
            throw new Error("L'âge ne peut pas être négatif");
        }
        this.age = age;
    }

    // ...
}

let p = new Personne(20);
p.age = 12; // On ne peut pas accéder à l'attribut age -> erreur
p.setAge(-5); // On ne peut pas mettre un âge négatif -> levée d'une exception
```

Intérêt de l'encapsulation

- **masquer l'implémentation**
  - toute la décomposition du problème n'a pas besoin d'être connue par l'utilisateur
- **protéger**
  - l'objet a le contrôle de son état
  - préserver l'intégrité des objets
  - le "programmeur créateur" contrôle (et est responsable) de son interface par rapport au "programmeur utilisateur"
- **permettre l'évolutivité**
  - il est possible de modifier tout ce qui n'est pas public sans impacter les utilisateurs de la classe

En résumé, le principe d'encapsulation permet de **séparer** l'**implémentation** et l'**interface** d'une classe.

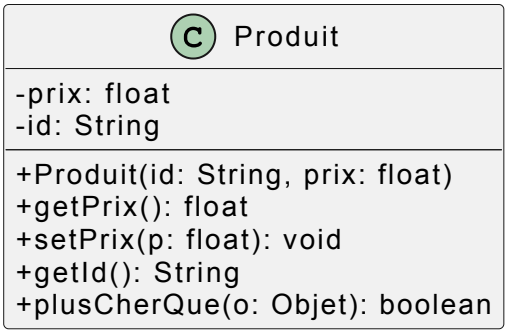
Règle : les attributs d'une classe doivent toujours être privés dans ce cours.

Visibilité des attributs et méthodes en UML

En UML, on représente la visibilité des attributs et méthodes par des symboles devant le nom de l'attribut ou de la méthode :

- `+` : public
- `−` : privé

Exemple :



Accesseur / modificateur

On peut vouloir accéder à la valeur d'un attribut privé ainsi que de vouloir la modifier. Pour cela, on utilise des méthodes spéciales appelées **accesseurs** et **modificateurs** (**getter** et **setter** en bon franglais).

La convention est de nommer les accesseurs et modificateurs de la manière suivante :

- `get` + nom de l'attribut pour l'accesseur (en respectant le camelCase)
- `set` + nom de l'attribut pour le modificateur (en respectant le camelCase)

Exemple : dans la classe `Objet` ci-dessus, `getPrix` (respectivement, `getId`) est un accesseur à l'attribut `prix` (respectivement, `id`) et `setPrix` est un modificateur de l'attribut `prix`.

Remarque : il n'y a pas nécessairement besoin d'avoir un accesseur et un modificateur pour chaque attribut. On peut très bien avoir un attribut privé sans accesseur ni modificateur si son accès / modification n'est pas souhaitable.

Visibilité des attributs et méthodes en TypeScript

En TypeScript, on utilise le mot-clé `private` pour rendre un attribut ou une méthode privée. On utilise `public` pour rendre un attribut ou une méthode publique. Si on ne met rien, l'attribut ou la méthode est publique par défaut.

Exemple :

```
class Produit {
    private prix: number;
    private id: string;

    constructor(id: string, prix: number) {
        this.id = id;
        this.prix = prix;
    }

    getPrix(): number {
        return this.prix;
    }

    setPrix(p: number): void {
        this.prix = p;
    }

    getId(): string {
        return this.id;
    }

    plusCherQue(p: Produit): boolean {
        return this.prix > p.prix;
    }
}
```

Unité d'encapsulation

Si on considère la méthode `plusCherQue` de la classe `Objet` ci-dessus, on voit qu'elle utilise l'attribut `prix` de l'objet courant et l'attribut `prix` de l'objet passé en paramètre. Or, cet attribut est privé. Cela ne pose pas de problème. Cela vient du fait que l'unité d'encapsulation en TypeScript (et dans de nombreux langages orientés objet) est la classe. Une méthode d'une classe est capable d'accéder directement à tous les attributs privés d'un objet de la classe.

Méthodes d'accès et de modification en TypeScript

En TypeScript, on peut définir des méthodes d'accès et de modification pour un attribut via une syntaxe particulière. On utilise pour cela les mots-clés `get` et `set`. Ainsi, la classe `Objet` ci-dessus peut être réécrite de la manière suivante :

```
class Produit {
    private _prix: number;
    private _id: string;

    constructor(id: string, prix: number) {
        this._id = id;
        this._prix = prix;
    }

    get prix(): number {
        return this._prix;
    }

    set prix(p: number): void {
        this._prix = p;
    }

    get id(): string {
        return this._id;
    }

    plusCherQue(p: Produit): boolean {
        return this._prix > p._prix;
    }
}
```

Par convention, on met un `_` devant le nom de l'attribut privé et on utilise le même nom sans ce tiret pour la méthode.

Cela permet d'utiliser la syntaxe suivante lorsque l'on manipule un objet de la classe `Objet` :

```
let p = new Produit("123456789", 20);

p.prix = 12; // Appel de la méthode set prix
console.log(p.prix); // Appel de la méthode get prix
console.log("Le produit " + p.id + " coûte " + p.prix + "€");
```

Remarque : c'est aussi le cas par exemple pour `a.length` quand `a` est un tableau. `length` est un attribut privé de la classe `Array` mais on peut y accéder directement via la syntaxe `a.length` car un accesseur a été défini pour cet attribut selon la syntaxe TypeScript.