

Ne faites pas du trafic d'organes (encapsulation et bonnes pratiques)

Encapsuler ses attributs

```
public class Group {  
    // Ces attributs sont private donc ne peuvent pas être directement modifié par un tiers objet  
    private String name;  
    private String[] groupMembers;  
    private int score;  
  
    // Pour pouvoir quand même y accéder on va donc définir un accesseur (getter) qui commence  
    toujours par "get"  
    public int getScore() {  
        return this.score;  
    }  
  
    // Pour pouvoir quand même le modifier on va donc définir un modificateur (setter) qui  
    commence toujours par "set"  
    // Cela permet ainsi d'utiliser nos propre conditions et de garantir que l'objet est toujours  
    dans un état cohérent  
    public void setScore(int score) {  
        if (score >= 0) {  
            this.score = score;  
        }  
    }  
  
    // On fait de même pour "groupMembers"  
    // sauf que groupMembers est un tableau et comme les objets, cela signifie que c'est la  
    *référence* qui sera passée  
    // Par conséquent cela pourrait tout de même permettre à l'utilisateur de modifier le contenu  
    de l'objet
```

```
// Nous allons donc faire une "copie défensive"
public void getGroupMembers() {
    return Arrays.copyOf(this.groupMembers);
}

// Un String est aussi un objet sauf que c'est un objet immuable donc nous n'avons pas besoin
de faire de copie défensive
public String getName() {
    return this.name;
}

// Et nous n'allons pas définir de setter pour le nom et le groupe car nous souhaitons qu'il
ne puisse plus être changé après la construction de l'objet
// Une classe ou un attribut peut être rendu immuable avec l'utilisation du mot clé "final"
}
```

Ainsi `private` permet de limiter l'accès à quelque chose (méthode, attribut, etc) à seulement la classe courrante. Mais il y a d'autres niveaux également :

Nom	Effet
<code>private</code>	Seul la classe courrante peut y accéder
<code>protected</code>	Toutes les classes dans le même package et les classes qui héritent de la classe actuelle peuvent y accéder
<code>public</code>	Tout le monde peut y accéder
Ne rien mettre (par default)	Seul les classes qui sont dans le même package peut y accéder

Il est conseillé de surtout utiliser `public` et `private`.

N'exploitez pas vos amis

Un objet a ses responsabilités, elle ne doit pas simplemenet stoquer des données mais doit aussi avoir des fonctionnalités.

- ⚠ Ne pas faire ça

```
// Cette classe ne fait que stoquer des objets et ça ne devrait pas être le role des autres
classe d'implémenter ses fonctions
public class Apple {
    private int x;
```

```

private int y;

public int getX() {
    return x;
}

public void setX(int x) {
    this.x = x;
}

public int getY() {
    return y;
}

public void setY(int y) {
    this.y = y;
}

```

On peut donc ajouter une nouvelle méthode `locateApple` pour placer une pomme dans une certaine position par exemple:

```

public void locateApple(int dotSize, int randPos) {
    int r = (int) (Math.random() * randPos);
    this.x = ((r * dotSize));

    r = (int) (Math.random() * randPos);
    this.y = ((r * dotSize));
}

```

En résumé une classe doit implémenter des fonctionnalités et éviter de demander aux autres classes de faire son travail.

Ne kidnapez pas les objets

La "loi de déméter" sert à protéger les pauvres objets que vous maltraitez.

Elle défini que vous ne devez interagir directement qu'avec vos amis et ne pas parler aux inconnus. Et vos amis sont uniquement :

- Les objets en paramètres
- Les objets en attributs
- Les objets de la même classe que vous
- Les objets que vous créez

En revanche les objets qui sont retournés par des méthodes d'une autre classe ne peuvent pas être utilisés directement.

Donc ça c'est juste non...

```
int rank = game.getActivePlayer().getHand().getCardAt(i).getRank();  
//           ↓           ↓           ↓           ↓           ↓  
//           Game      Player      CardHand   Card       int
```

Dans cet exemple, nous avons un objet de classe `Game` mais on va récupérer et aussi dépendre aussi sur les classes `Player`, `CardHand` et `Card`. Ce qui n'est vraiment pas une bonne idée et rends l'infrastructure du code beaucoup plus complexe.

On pourrait par exemple créer une méthode `getActivePlayerCard(int i)` dans `Game` pour obtenir un `Card` et réduire le nombre de dépendences (notre classe est amie avec `Game` et `Game` (où notre nouvelle méthode est) est amie avec `CardHand`).

Revision #1

Created 27 April 2023 04:31:43 by SnowCode

Updated 27 April 2023 04:35:12 by SnowCode