Héritage et Polymorphisme

Héritage

Principe de base

Dire qu'une classe hérite d'une autre, c'est établir une relation **EST UN** entre deux classes.

Soit une classe `Mamal` telle que :

```
public class Mamal {
    ...
}
```

La classe `Human` peut hériter de la classe `Mamal` avec la syntaxe suivante :

```
public class Human extends Mamal {
    ...
}
```

On dit que la classe `Mamal` est la **super classe** de `Human` et que `Human` est une **sous classe** de `Mamal`. Cela implique la logique suivante, un humain **est un** mamifère.

Quand une classe hérite d'une autre, cela implique que la classe fille récupère tous les membres (champs et méthodes) de la classe mère, quelle que soit leur visibilité. En Java, une classe peut hériter que d'une seule super classe.

> Attention, une sous classe possède tous les champs privés de sa super classe mais n'y a pas accès.

Note sur l'encapsulation

Le modificateur de visibilité `protected` existe et veut dire "privé sauf pour les sous classes". Cependant, en pratique il est peu utilisé, on préfère implémenter des getters et setters sur la super classe, afin d'avoir une maîtrise fine de l'encapsulation.

Constructeur

Si une classe n'a qu'un constructeur pas défaut, les constructeurs des classes filles appellent implicitement ce constructeur.

Si une classe a un constructeur définit autre qu'un constructeur par défaut, ses sous classes doivent impérativement appeler ce constructeur via `super()`, à la première ligne du constructeur.

Par exemple si on a une classe telle que :

```
public class Mamal {
    private String name;

    public Mamal(String name){
        this.name = name;
    }
}
```

Ses sous classes doivent implémenter ce constructeur, et faire appel à la logique ainsi :

```
public class Human extends Mamal {
    public Human(String name){
        super(id)name;
    }
}
```

Redéfinition de méthodes

Une sous classe peut redéfinir les méthodes de la super classe, afin de l'adapter à ce quelle est. Pour redéfinir une méthode, on la réécrit et on l'annote avec `@Override`. Par exemple une classe telle que :

```
public class Mamal {
    public void eat() {
        System.out.println("I eat");
    }
}
```

Ses sous classes peuvent redéfinir la méthode `eat()` pour qu'elle corresponde à ce qu'elles sont.

```
public class Human extends Mamal {
    public void eat() {
        System.out.println("I eat with a fork and a knife");
    }
}
```

Une méthode redéfinie peut faire appelle à la méthode originale de la super classe grâce à la référence `super` :

```
public class Human extends Mamal {
    public void eat() {
        System.out.println("I take a fork and a knife");
        super.eat();
    }
}
```

La trace de ce code sera :

```
I take a fork and a knife
I eat
```

Abstraction

La notion d'abstraction permet de définir de classes dites abstraites, qui ne peuvent être instanciées, mais établissent des **contrats de service** avec leur sous classes, c'est à dire qu'elles définissent le prototype de méthodes que les sous classes seront obligées de définir.

Pour définir une classe abstraite :

```
public abstract class Mamal {
    ...
}
```

Une classe abstraite peut définir tous les membres qu'une classe concrète peut définir, mais elle peut définir des méthodes abstraites. Ces méthodes sont les méthodes qui doivent être définies par les sous classes.

```
public abstract class Mamal {
    public abstract void eat();
}
```

La classe abstraite ne fournit pas d'implémentation de la méthode, mais oblige par **contrat de service** ses sous classe à la définir. Une classe qui étend une classe abstraite doit définir ses méthodes abstraites ou alors être abstraite également, sans quoi elle ne compilera pas.

Polymorphisme

Le polymorphisme est l'un des principes fondamentaux de la programmation orientée objet, qui consiste à utiliser les contrats de service pour manipuler des classes qui partagent une super classe dont ils redéfinissent certains comportements. En effet, une référence du type d'une certaine classe, peut recevoir une référence de n'importe quelle sous classe de la classe en question. Avec les classes suivantes :

```
public abstract class Animal {
    public abstract void shout(){
    }
}
```

```
public class Dog extends Animal {
    public void shout(){
        System.out.println("wof wof !")
    }
}
```

```
public class Cat extends Animal {
    public void shout(){
        System.out.println("mew mew !")
    }
}
```

Le code suivante est possible :

```
Animal animal1 = new Dog();
Animal animal2 = new Cat();
```

Si on appelle `eat()`, la définition de la méthode correspondant à chaque sous classe sera appelée :

```
animal1.shout();
animal2.shout();
```

La trace de ce code sera :

```
wof wof !
mew mew !
```

Ce principe est très puissant, car il permet au code appelant d'ignorer les sous classes et leur implémentation, et d'utiliser juste ce dont il a besoin pour effectuer son travail, permettant de **séparer les responsabilités**. Par exemple, étant donné la classe `Human` suivante :

```
public class Human {
    private List<Animal> pets = new ArrayList<Animal>();

    public void adopt(Animal animal){
        animal.shout();
        this.pets.add(animal);
    }
}
```

La classe `Human` ainsi que le méthode `adopt()` n'ont à connaître des animaux que le fait qu'ils peuvent crier, car savoir de quel type sont les animaux, ou de savoir comment ils crient ne sont sont pas de sa responsabilité. Ainsi, si on change la façon de crier des animaux, si on rajoute des nouvelles classes d'animaux, pas besoin de modifier la classe `Human` ni la méthode `adopt()`.

Interfaces

Les interfaces sont des classes purement abstraites. Elles ne contiennent pas d'attributs et seulement des méthodes `public` et `abstract` (si bien qu'il n'y a pas besoin de le préciser). Les interfaces permettent de définir des contrats de service en s'affranchissant des contraintes de l'héritage. En effet, une classe peut implémenter plusieurs interfaces. Ainsi, on va privilégier cette stratégie, et utiliser l'héritage uniquement lorsque l'on a besoin de factoriser des membres communs à plusieurs sous classes.

Définir une interface :

```
public interface Engine {
    void start();
    void accelerate();
    void stop();
}
```

Implémenter une interface :

```
public class ElectricEngine implements Engine {
    public void start() {

    }

    public void accelerate(){

    }

    public void stop(){

    }
}
```

```
public class CombustionEngine implements Engine {

    public void start(){

    }

    public void accelerate(){

    }

    public void stop(){

    }
}
```

L'utilisation des interfaces permet, de séparer au maximum les contrats de service de l'implémentation, afin de séparer les responsabilité.

> Attention à ne pas centraliser trop de comportements dans une seule interface. Une interface doit disposer uniquement des comportements qui doivent être exposés au code appelant, tout en ayant une valeur sémantique.

Inversion de dépendance

Il est très important de réduire les dépendances au maximum en utilisant les interfaces. Il vaut toujours mieux dépendre d'une interface que d'une implémentation. Cela permet de rendre le code plus facile à changer mais aussi plus facile à tester.

Par exemple :

```
public class Car {
    private CombustionEngine engine;

    public Car(){
        engine = new CombustionEngine();
    }
}
```

Ici, la classe `Car` dépend de la classe `CombustionEngine`. Il vaudrait utiliser notre interface `Engine` pour cacher son implémentation à la classe `Car`. Cela permet de changer d'`Engine` sans modifier `Car` ainsi que de tester la classe `Car` indépendement de son `Engine`.

```
public class Car {
    private Engine engine;

    public Car(Engine engine){
        this.engine = engine;
    }
}
```

Et dans le code utilisant `Car` :

```
Car car = new Car(new CombustionEngine());
```

Revision #1
Created 2024-11-21 15:27:13 UTC by Nicolas
Updated 2025-02-10 12:13:14 UTC by Nicolas