Rappels

Encapsulation

L'encapsulation consiste à protéger une partie des membres d'un objet, de façon à contrôler son utilisation et à rendre interne une partie de son comportement.

L'encapsulation est généralement mise en oeuvre à partir des types d'accès définis dans la classe sur ses membres.

Types d'accès

Accès	**Rôle**
Privé (private)	Accessibilité dans l'objet lui-même (this) uniquement.
Protégé (protected)	id Privé + Accessibilité dans les classes dérivées
Public (public)	id Protégé + Accessibilité de l'extérieur, sur une instance de la classe

Exemples java

Private

```
public class Base {
    private int member;
    public int getMember(){
        //Accès possible à this.member ou member sur l'instance this de la classe
        return this.member;
    }
}
```

Protected

```
public class Base {
    protected int pMember;

    public int getpMember() {
        return pMember;
    }
}

public class Derivee extends Base {
    public void maminpulePMember(){
        //Accès possible à this.member ou member sur l'instance this de la classe
dans la classe dérivée
        System.out.println(this.pMember);
    }
}
```

Public

```
public class Base {
    protected int pMember;
    public static int count;

    public int getpMember() {
        return pMember;
    }
}


public class Programme {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Base b=new Base();
        //Accès au membre public de classe count
        Base.count++;
        //Accès au membre public getpMember sur une instance de Base
        System.out.println(b.getpMember());
    }

}
```

Règles de mise en oeuvre

Les membres de données ne doivent jamais être à portée publique, excepté les membres de données statiques (de classe). Il est ensuite possible de contrôler l'accès à ces données protégées en ajoutant des accesseurs, en lecture et/ou en écriture.

Méthodes d'accès

Accesseur en lecture

Le rôle d'un accesseur en lecture est de retourner la valeur d'un membre privé.

```
public int getpMember() {
        return pMember;
}
```

Accesseur en écriture (modificateur)

Le rôle d'un accesseur en écriture est de permettre la modification d'un membre privé, par affectation de la valeur passée en paramètre.

```
public void setpMember(int pMember) {
        this.pMember=pMember;
}
```

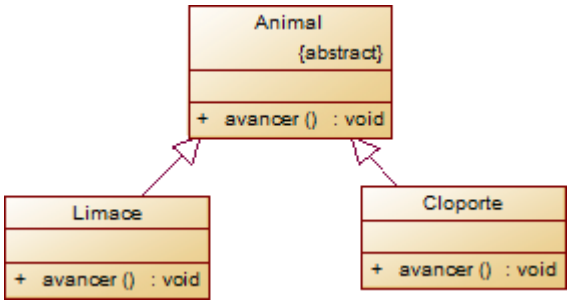
Construction d'objets

Un constructeur est une méthode d'une classe permettant l'instanciation d'objets. Plusieurs constructeurs peuvent être implémentés dans une classe.
On parle dans ce cas de surcharge (overloading).
Le rôle d'un constructeur est de permettre de créer des instances d'objets cohérentes, dont les membres sont correctement initialisés.

Polymorphisme d'héritage

Le polymorphisme est un concept permettant de simuler le changement de comportement ou de forme d'un objet, au cours de l'exécution d'un programme.

Exemple de mise en oeuvre :
Soit une classe **Animal** abstraite, ayant pour Classes dérivées **Cloporte** et **Limace**. Le polymorphisme se met en place grâce à la spécialisation (héritage + surdéfinition des méthodes).
Il permettra d'appeler la méthode **avancer** sur une instance d'**Animal** sans se préoccuper de savoir s'il s'agit d'une **Limace** ou d'un **Cloporte**.



La classe Animal est abstraite (il n'est pas possible d'instancier directement un Animal). Elle comporte une méthode abstraite avancer, que les classes dérivées de Animal devront implémenter (à moins qu'elles ne soient également abstraites).

```
public abstract class Animal{
    int position;
    public abstract void avancer();
}
```

Les classes **Limace** et **Cloporte** héritent de **Animal** et surdéfinissent la méthode **avancer** :

```
public class Limace{
    @Override
    public void avancer(){
        this.position++;
        this.laisserTraceDe(bave);
    }
}
```

```
public class Cloporte{
    @Override
    public void avancer(){
        this.position=this.position+2;
    }
}
```

Dans un programme, l'appel à la méthode avancer sur un animal pourra invoquer le avancer de la limace ou du cloporte, en fonction de ce qu'est l'animal à l'exécution.

```
Animal unAnimal;
...
unAnimal.avancer();
```

Interface

Une **Interface** permet en POO de définir le comportement que devront implémenter une ou plusieurs classes.