

Programmation Objet - Cours 2

Méthodes génériques

Licence 3 Informatique

Résumé des cours précédents

- Les **classes** représentent le cœur de la programmation orientée objet. Elles définissent les caractéristiques qu'auront tous les objets du type de cette classe. On définit une classe avec le mot clé **class**
 - **Class** Chien {
 String nom;
 void aboie{...};
}
- On **instancie** (i.e. on crée) un **objet** (une variable ayant pour type une classe) avec le mot clé **new**:
 - Chien variableChien = **new** Chien();
- On définit les caractéristiques d'une classe sous forme d'**attributs** (variables internes) et de **méthodes** (fonctions), dont l'union est appelée **membres**. On manipule un objet au travers de ces membres par le « . »:
 - variableChien.nom = « Medore »; variableChien.aboie();

Résumé des cours précédents

- La programmation orientée objet s'articule autour de trois principes fondamentaux:

- L'encapsulation :

- désigne le principe de regrouper les données avec les traitements qui les concernent directement
- s'accompagne du masquage des données pour offrir une interface à l'utilisateur dont on contrôle les entrées et sorties.



- Pour masquer un membre, on place devant lui un mot clé :

Mot clef	Classe	Package	Classe(s) fille(s)	Partout
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
par défaut	✓	✓	✗	✗
private	✓	✗	✗	✗

Résumé des cours précédents

- La programmation orientée objet s'articule autour de trois principes fondamentaux :

- L'héritage :

- Permet à une classe fille d'hériter des membres d'une classe mère. Pour hériter, il suffit d'utiliser le mot clé **extends** :

```
Class Chien extends Animal{  
    ...  
}
```

- Le polymorphisme :

- désigne le fait de pouvoir manipuler la classe mère et ses classes filles sans modification de code :

```
public int nourrir(Animal a){  
    ...  
}  
public static int main(String[] args){  
    Berger fidjie = new Berger();  
    Gardien medor = new Gardien();  
    nourrir(fidjie);  
    nourrir(medor);  
}
```

Constructeurs

Le mystère de l'instanciation d'objet (enfin résolu!)



Constructeurs: cycle de vie d'un objet

- Un programme crée des objets, les manipule et à un moment ou à un autre, doit détruire ces objets pour restituer la mémoire au système.
- À la création d'un objet, il faut réserver la mémoire nécessaire pour stocker ses données.
- L'objet sera ensuite manipulé dans le programme
- Puis nous n'aurons plus besoin de l'objet, il faudra restituer au système les ressources utilisées par l'objet.

C++

C'est la responsabilité du programmeur, qui peut détruire les objets via un mot clé *delete*. La mémoire est alors rendue au système qui pourra l'utiliser pour d'autres objets.

Java

Un mécanisme appelé ramasse-miettes (garbage collector) s'occupe de détruire les objets quand ceux ci ne sont plus référencés nulle part.

Instanciación



Manipulation




Destruction



Constructeurs: comment ça fonctionne

- A la création d'un objet, une méthode appelée **constructeur** est invoquée juste après la réservation de mémoire (**new**).
- En Java, cette méthode porte le nom de la classe :

```
public class Animal {  
    String nom;  
  
    ...  
    public Animal() {  
        // je suis le constructeur  
    }  
    public static int main(String[] args){  
        Animal a = new Animal();  
    }  
}
```



- C'est la seule méthode qui commence par une majuscule.
- Elle permet d'effectuer des tâches lors de l'instanciation de l'objet.

Constructeurs: initialisation des attributs

- Il sert généralement à initialiser les attributs de l'objet.

```
public class Animal {  
    String nom;  
  
    ...  
    public Animal() {  
        this.nom = « je n'ai pas encore de nom » ;  
        this.poids = 6;  
    }  
    ....  
}
```

- Si on execute:

```
Animal a = new Animal() ;  
System.out.println(« Je m'appelle : » + a.nom) ;
```

- On affichera :

Je m'appelle : je n'ai pas encore de nom

Constructeurs: initialisation des attributs

- Le constructeur est une méthode, il peut donc prendre des arguments.
- Généralement, ce sont des valeurs à donner aux attributs :

```
public class Animal {  
    String nom;  
  
    ...  
    //Constructor  
    public Animal(String nomA, int poidsA) {  
        this.nom = nomA;  
        this.poids = poidsA;  
    }  
  
    public static int main(String[] args){  
        Animal a = new Animal(« medor »,25);  
        System.out.println(« je m'appelle » + a.nom + « , je pèse » + a.poids + « kilos.»);  
    }  
}
```

- ce qui donnera à l'exécution :
- « je m'appelle medor, je pèse 25 kilos»

Constructeurs: constructeur par défaut

- Règle 1 : Tout objet a un constructeur, même s'il n'est pas défini explicitement.
- Ce sera alors le **constructeur par défaut** qui ne prend pas de paramètre et qui ne fait qu'appeler le constructeur par défaut de son ancêtre* :

```
public class Veterinaire {  
    void soigne(Animal a) {  
        System.out.println(a.nom + " est  
guéri !");  
    }  
}
```

=

```
public class Veterinaire {  
    public Veterinaire() {}  
    void soigne(Animal a) {  
        System.out.println(a.nom + " est  
guéri !");  
    }  
}
```

- * tout objet est une classe fille de la classe Object

Constructeurs: constructeur par défaut

- Règle 2 : Si un constructeur est définie (avec ou sans paramètres), il n'existe plus de constructeur par défaut.
- Le code suivant ne compile donc pas :



```
public class Animal {  
    String nom;  
    public Animal(String n) {  
        nom = n ;  
    }  
  
    public static int main(String[] args){  
        Animal a = new Animal();  
    }  
}
```

Constructeurs: constructeur et héritage

- Règle 3: Lorsqu'une classe hérite, le constructeur appelle implicitement le constructeur de sa superclasse AVANT d'exécuter son code :

```
class Gardien extends Chien {  
    public Gardien() {  
        System.out.println("Constructeur de Gardien");  
    }  
}
```

```
class Chien extends Animal {  
    public Chien() {  
        System.out.println("Constructeur de Chien");  
    }  
}  
  
public class Animal {  
    public Animal() {  
        System.out.println("Constructeur d'Animal");  
    }  
}
```

```
class Gardien extends Chien {  
    public Gardien() {  
        System.out.println("Constructeur  
d'Animal");  
        System.out.println("Constructeur de  
Chien");  
        System.out.println("Constructeur de  
Gardien");  
    }  
}  
  
class Chien extends Animal {  
    public Chien() {  
        System.out.println("Constructeur  
d'Animal");  
        System.out.println("Constructeur de  
Chien");  
    }  
}
```

Constructeurs: exemple d'héritage

- Quand on instanciera un Gardien :

Gardien medor = new Gardien();

- Il sera alors affiché :

« Constructeur d'Animal

Constructeur de Chien

Constructeur de Gardien »

- Il est aussi possible d'appeler explicitement le constructeur de la superclasse avec le mot clé **super** (cf. slide 15). Cela peut être utile, notamment en cas de multiple constructeurs de la superclasse.

Constructeurs: constructeur par défaut

- Si on reprend nos trois règles :
- Règle 1 : Tout objet a un constructeur, même s'il n'est pas défini explicitement.
- Règle 2 : Si un constructeur est définie (avec ou sans paramètres), il n'existe plus de constructeur par défaut.
- Règle 3: Lorsqu'une classe hérite, le constructeur appelle implicitement le constructeur de sa superclasse AVANT d'exécuter son code
- Le code suivant ne compile donc pas :

```
public class Animal {  
    String nom;  
    public Animal(String n, int p) {  
        nom = n ;  
        poids = p;  
    }  
}  
  
public class Chat extends Animal{  
    public void ronronne(){  
        System.out.println(« ronron »);  
    }  
}
```

Constructeurs: constructeur par défaut

- Il faut donc définir explicitement un constructeur pour Chat qui appelle le constructeur de la superclasse (classe mère) en utilisant le mot clé *super*.

```
public class Chat extends Animal {  
    public Chat() {  
        super(« minou », 5) ;  
    }  
    ....  
}
```

- Mais tous mes chats s'appelleront « *minou* » tant que je n'aurais pas changé leur nom. On peut aussi proposer un constructeur personnalisé:

```
public class Chat extends Animal {  
    public Chat(String nom, int poids) {  
        super(nom, poids) ;  
    }  
}
```

Constructeurs: constructeur par défaut

- Ou bien définir explicitement dans la classe Animal le constructeur vide

```
public class Animal {  
    String nom;  
    public Animal() {  
    }  
    public Animal(String n, int p) {  
        nom = n ;  
        poids = p;  
    }  
}
```


Constructeurs: multiples constructeurs

- Une classe peut avoir plusieurs constructeurs :
- Ce qui me permettra d'utiliser celui que je souhaite :

```
public class Animal {  
    String nom;  
    private int x;  
    private int y;  
  
    public Animal() {  
    }  
  
    public Animal(String nom, int x, int y) {  
        this.nom = nom;  
        this.x = x ;  
        this.y = y ;  
    }  
}
```

```
Animal a = new Animal();  
Animal b = new Animal("Gérard", 20,  
30);
```

Destructeurs

Les objets de la non-infinité



Destructeurs : fin de vie d'un objet

- Quand on sait qu'on ne se servira plus d'un objet, alors il faut restituer au système les ressources qu'on lui a emprunté comme la mémoire, par exemple.
- Cette opération s'appelle la destruction d'un objet.
- Dans de nombreux langages, la destruction est explicite et à la charge du programmeur.
- En C++ par exemple, elle se fait via le mot clé *delete*.

```
MonObjet o = new MonObjet() ;
```

```
    ...  
delete o ;
```

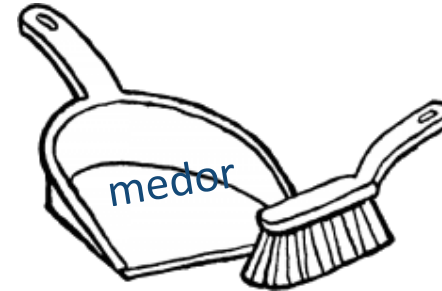
Destructeurs : nettoyage du système

- Lors de la destruction d'un objet, une méthode spéciale est exécutée : c'est le destructeur.
- Le destructeur permet d'exécuter les actions importantes liées à la fin de vie de l'objet :
 - libération de la mémoire dynamiquement allouée
 - fermeture des fichiers ouverts
 - enregistrement des données
 - etc.. ;

Destructeurs : le ramasse-miettes

- Java est un langage qui prend en charge la gestion de la mémoire :
 - Un mécanisme appelé ramasse-miettes (garbage collector) se charge de récupérer la mémoire des objets qui ne seront plus utilisés.
 - Un objet ne sera plus utilisé s'il n'est plus référencé par aucune variable.

```
public static int main(String[] args){  
    Animal a = new Animal(« medor », 25);  
    Animal b = new Animal(« fidjie », 19);  
    a = b;  
}
```



- Régulièrement, le ramasse-miette parcourt les objets marqués comme non référencés pour les détruire.
- Lors de cette destruction une méthode particulière permettant d'exécuter les tâches liées à la destruction de l'objet est appelée : la méthode *finalize*.

```
protected void finalize() throws Throwable {  
    super.finalize();  
}
```

Destructeurs : fonction *finalize()*

- Généralement, on trouvera dans *finalize* les actions liées à la fin de vie de l'objet :
 - envoi d'un message réseau indiquant la fin de connexion sur un objet représentant une connexion
 - enregistrement sur disque des attributs dans un objet Configuration
 - libération d'une ressource matérielle
- Ce qu'il faut retenir c'est qu'on ne sait pas quand cette méthode sera appelée. On sait juste qu'elle sera appelée par le ramasse miette à un moment ou à un autre.

Accesseurs

Le retour du masquage

Accesseurs: définition

- Un accesseur est une méthode permettant de **lire** ou **modifier** la valeur d'un attribut.
- Généralement, on utilise des accesseurs en combinaison avec le masquage (*private*) des attributs. Ainsi, on oblige les utilisateurs de la classe à utiliser les accesseurs pour lire et modifier les attributs.
- On appelle parfois la méthode de modification mutateur.
- On appelle aussi souvent les accesseurs :
 - getter(get = avoir) pour lire un attribut
 - Setter(set = affecter) pour modifier/écrire une valeur dans un attribut.

Accesseurs : exemple

```
class Chien {
    String nom ;
    int poids;
    private int x,y;

    public void Chien(){
        this.nom = « je n'ai pas de nom »;
        this.poids = 25;
        this.x = 15;
        this.y = 35;
    };
    /*getters*/
    public int getX() { return x; }
    public int getY() { return y; }

    /*setters*/
    public void setX(int x) {
        if (x > 10 && x < 30){this.x = x;}
        else {system.out.println(« le chien ne doit
pas sortir! »)} }

    public void setY(int y) {
        if (y > 20 && y < 50){this.y = y;}
        else {system.out.println(« le chien ne doit
pas sortir! »)} }
}
```

- Ainsi, aucune instance de chien ne pourra sortir du jardin

```
Class Proprio{
    Chien toutou;

    public static int main(String[] args){
        Proprio p = new Proprio();
        p.toutou = new Chien();
        p.toutou.setX(40);

        System.out.println(p.toutou.getX());
    }
}
```

« le chien ne doit pas sortir »

- Ce code renvoi bien : 15 »

Méthodes et Classes abstraites

Déclarer sans définir

Méthodes et Classes abstraites : définition

- Une classe abstraite est une classe qu'on ne peut pas instancier, et qui peut comporter une ou plusieurs méthodes abstraites.
- Une méthode abstraite est une méthode déclarée mais non définie.
- Une classe qui contient des méthodes abstraites doit obligatoirement être abstraite.
- En Java, les méthodes abstraites sont déclarées via le mot clé **abstract**.
- De même, la classe abstraite doit être aussi explicitement déclarée avec le mot clé **abstract**.

```
public abstract class maClasse {  
    public abstract void  
    maFonction();  
}
```

- **Une classe abstraite ne peut pas être instanciée**



Méthodes et Classes abstraites : problématique

- On veut que notre Vétérinaire puisse nourrir un Animal, qu'il soit Chat, Chien, Oiseau ou tout autre Animal futur.
- Nos animaux doivent tous manger, mais ils ne mangent pas de la même façon : la méthode est spécifique au type d'Animal :
 - le Chien mange ses croquettes
 - le Chat mange sa patée

Méthodes et Classes abstraites : solution basique

- On pourrait faire de la surcharge dans Veterinaire pour répondre au besoin :

```
public class Veterinaire {  
    void nourrit(Chat c) {  
        c.ronronne();  
        c.mange();  
    }  
  
    void nourrit(Chien c) {  
        c.aboie(true);  
        c.mange();  
    }  
}
```

Méthodes et Classes abstraites : solution basique

- L'ennui c'est que nous aimerions aussi nourrir des Oiseaux qui picorent des graines et des Poissons gobent des mouches.
- A chaque nouvelle classe fille d'Animal il va nous falloir créer une méthode nourrit : ce n'est pas satisfaisant.
- Le meilleur moyen est donc d'utiliser des méthodes abstraites.
- Ainsi, on **déclare** que tout animal mange, mais on ne **définie** cette fonction que dans les classes filles.

```
public abstract class Animal {  
    public abstract void mange();  
}
```

Méthodes et Classes abstraites : méthode

- Toute classe fille qui hérite d'une classe abstraite **doit définir les méthodes abstraites**:

```
public class Chat extends Animal {  
    public void mange() {  
        System.out.println("mange sa patée");  
    }  
}
```

```
public class Chien extends Animal {  
    public void mange() {  
        System.out.println("mange ses croquettes");  
    }  
}
```

Méthodes et Classes abstraites : bénéfiques

- Le code de `Veterinaire.nourrit()` est du coup très simplifié :

```
public class Veterinaire {  
    void nourrit(Animal a) {  
        a.mange();  
    }  
}
```

- Une seule méthode suffit pour nourrir tout type d'animal, on pourra ajouter un nouvel animal sans impacter le code de la classe vétérinaire (encapsulation):

```
public class Oiseau extends Animal {  
    public void mange() {  
        System.out.println(« picore des graines ») ;  
    }  
}
```


Membres statiques

Éléments communs propre à la classe

Membres statiques : méthode statique

- Une méthode statique est une méthode accessible sans avoir besoin d'instance de la classe.
- Elle servent à décrire un comportement lié à un concept et non à une instance en particulier de ce concept.
- en Java, une méthode statique se déclare via le mot clé **static** :

```
public abstract class Animal {  
    public static String identification() {  
        return "tatouage";  
    }  
}
```

Membres statiques : accès

- On accède à cette méthode le plus souvent par le biais de la classe :

```
System.out.println(« Méthode d'identification : » + Animal.identification()) ;
```

- Ce code affiche:

```
Méthode d'identification : tatouage
```

- Mais on aurait obtenu le même résultat à partir d'une instance:

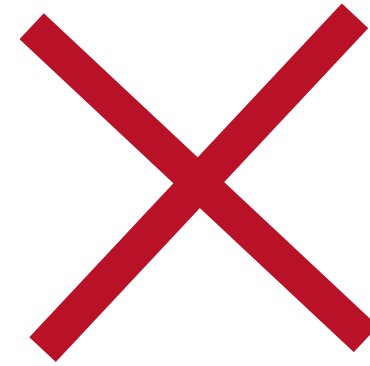
```
Chien c = new Gardien(« Medor ») ;  
System.out.println(« Méthode d'identification : » + c.identification()) ;
```

- On note que quelque soit l'instance, on obtiendra **le même résultat**.

Membres statiques : propriétés et utilité

- Une méthode statique n'est pas surchargeable dans les classes filles. Le code suivant ne compile pas :

```
class Chien extends Animal {  
    public static String identification() {  
        return "tatouage";  
    }  
}
```



Membres statiques : attribut

- Un attribut statique est un attribut dont la valeur est partagée par toutes les instances de la classe.
- Il se déclare en java par le mot clé **static**:

```
public abstract class Animal {  
    private static int nbAnimaux = 0;  
    public Animal{  
        //équivalent à nbAnimaux = nbAnimaux + 1;  
        ++nbAnimaux;  
    }  
}
```

Membres statiques : attribut

- Comme pour les méthodes statiques, on y accède généralement via la classe, mais on peut y accéder via une instance.

```
System.out.println(« il y a » + Animal.nbAnimaux + « animaux ») ;  
Chien a = new Gardien(« Medor ») ;  
Chien b = new Gardien(« Toutou ») ;  
System.out.println(« il y a » + Animal.nbAnimaux + « animaux ») ;  
System.out.println(« Medor dit qu'il y a » + a.nbAnimaux +  
« animaux ») ;
```

- Qui affiche
 - « il y a 0 animaux*
 - il y a 2 animaux*
 - Medor dit qu'il y a 2 animaux »*



MERCI !

