ST
Dpt de Formation et de Recherche Sciences et Technologies

Programmation Objet - Cours 4
Bonnes pratiques

Licence 3 Informatique

Université de Guyane

Cours de Vincent Bonnemains
vincent.bonnemains@hotmail.fr

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 définie une classe avec le mot clé ***class***
  - ***Class*** *Chien {*
    String nom;
    void aboie{…};
    }
- On ***instancie*** (i.e. on créé) un ***objet*** (une variable ayant pour type une classe) avec le mot clé ***new:***
  - *Chien variableChien =* ***new*** *Chien();*
- On définie les caractéristiques d'une classe sous forme d'***attributs*** (variables internes) et de ***méthodes*** (fonctions), dont l'union est appelée ***membres***. On manipulera 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);
}
```

Université de Guyane

Résumé des cours précédents

- Il existe des méthodes classiques que l'on retrouve dans pratiquement toutes les classes :
  - ***Le constructeur :*** méthode portant le nom de la classe qui permet d'instancier un objet. Il peut prendre des paramètres. Il est possible d'avoir plusieurs constructeurs.
    - Règle 1 : Tout objet a un constructeur. S'il n'est pas défini explicitement, c'est un ***constructeur par défaut***, vide.
    - 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

```
Class Chien extends Animal{
        String race;
        public Chien(String nom, int poids, String race){
                super(nom,poids);
                this.race = race;
        }
}
```

  - ***Le destructeur :*** est une méthode appelée par le ramasse-miette pour libérer de l'espace mémoire

Résumé des cours précédents

- ***Les accesseurs :*** on distingue deux types d'accesseurs:
  - Les getters permettent d'accéder à la valeur d'un attribut
  - Les setters permettent de modifier la valeur d'un attribut

```
class Chien extends Animal{
        private String race;
        /* getter */
        public String getRace(){
            return this.race;
        }
         /* setter */
        public void setRace(String race){
            this.race = race;
        }
}
```

- ***Les classes abstraites***
  - Les classes abstraites sont des classes qui contiennent des méthodes abstraites.
  - Une méthode abstraite est une méthode qu'on déclare mais qu'on ne définie pas.
  - Toute classe fille héritant d'une classe abstraite doit définir les méthodes abstraites de cette classe.
  - Les classes et méthodes abstraites se définissent avec le mot clé ***abstract***.

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

Résumé des cours précédents

- ***Les membres statiques***
  - Les ***attributs statiques*** appartiennent à la classe, et non à l'objet. Leur valeur est donc commune à tous les objets de la classe
  - De même, les ***méthodes statiques*** appartiennent à la classe. On peut donc les appeler depuis un objet, mais aussi et surtout directement depuis la classe.
  - On définie un membre statique par le mot clé ***static***.

```
class Chien extends Animal{
        public static int nb_chiens = 0;
        public Chien(){
            nb_chiens = nb_chiens + 1;
        }
        public static identification(){
            System.out.println(« tatouage »);
        }
        public static int main(String[] args){
                Chien.identificiation();
        }
}
```

Résumé des cours précédents

- ***Les interfaces***
  - Un équivalent de classe abstraite, mais qui n'est pas rattaché au système d'héritage
  - Se définit avec le mot clé ***interface***
  - Une classe implémente une interface avec le mot clé ***implements***

```
interface Mangeur {
        public void manger();
}
```

```
classe Veterinaire implements Mangeur {
        public void manger(){
            System.out.println(« manger une pizza »);
        }
}
```

- ***Les tableaux***
  - Soit statiques, avec une taille fixe : *String[] listeNoms = new String[10];*
  - Soit dynamiques, grâce à la bibliothèque collections, comme par exemple la classe *ArrayList*

Résumé des cours précédents

- ***Surcharge de méthodes***
  - On peut surcharger une méthode soit avec une signature différente
  - Soit avec une même signature, dans une classe fille. On veillera dans ce cas à ajouter l'annotation *@override*

```
classe Oiseau extends Animal{
    @override
    public void manger(){
        System.out.println(this.nom + « picore »);
    }
}
```

- ***final***
  - Le mot clé ***final*** interdit un membre ou une classe d'être modifiés (une classe ne pourra pas être dérivée, une méthode ne pourra pas être surchargée, et une variable modifiée)

Résumé des cours précédents

- ***Programmation générique***
  - Consiste à pouvoir paramétrer des Objets lors de leur future utilisation

```
public class Cage<T> {
    private T enCage = null;
    public void enferme(T a) {
        enCage = a ;
    }
    public T libere() {
        T animalLibere = enCage ;
        enCage = null ;
        return animalLibere ;
    }
}
```

```
public static int main(String[] args){
    Chat minou = new Chat("Felix");
    Cage<Chat> maCage = new Cage<Chat>();
    /* On enferme minou */
    maCage.enferme(minou);
    /* On libère minou)
    minou = maCage.libere();
}
```

- ***Introspection***
  - Consiste à questionner la nature d'un objet. Pour cela, on utilise par exemple :
    - Le mot clé « ***instanceof*** » qui permet de vérifier qu'un objet est de la bonne classe
    - La classe « ***class*** » qui récolte toutes les informations d'une classe

Exceptions

Et si ça ne marchait quand même pas ?

Université de Guyane

Exceptions : définition

- Un système de gestion d'exceptions permet de gérer les conditions exceptionnelles pendant l'exécution du programme. Lorsqu'une exception se produit, l'exécution normale du programme est interrompue et l'exception est traitée.

- Exemples d'exceptions :
    - division par zéro
    - accès à une zone mémoire indéfinie (pointeur null)
    - fichier inexistant
    - pas assez d'espace disque

Exceptions : définition

- En Java les exceptions sont des objets de classes filles héritant de la classe standard *Exception*.

```
public class MonException extends Exception {};
```

- On peut « levée » une *Exception* (càd signaler qu'une situation d'exception est arrivée) via le mot clé ***throw*** :

```
throw new MonException() ;
```

- Une méthode susceptible de remonter une exception DOIT le déclarer dans sa signature via le mot clé ***throws*** :

```
public void maMethode() throws  MonException {
        // code
        throw new MonException() ;
}
```

- Les seules exceptions qu'il n'est pas nécessaire de déclarer dans la signature des méthodes sont les exceptions sous-classes de *RunTimeException*.

Exceptions : le bloc try catch

- Les exceptions peuvent s'intercepter et se traiter avec un un bloc *try/catch* :

```
public void maMethode() {
     // code qui précède
    try  {
        // code qui risque de lever une exception (via throw par
exemple)
    } catch (MonException e) {
        // code de gestion des exceptions de type MonException
    } catch  (AutreTypeException e) {
       // code de gestion de l'autre type d'exception
    }
    // poursuite de l'execution
}
```

- Si l'exception est une instance de *MonException*, le code du premier *catch* sera exécuté et la méthode reprendra son cours au niveau du code qui suit le bloc *try/catch*.
- Si l'exception est une instance de *AutreTypeException*, le code du deuxième *catch* sera exécuté et la méthode reprendra après le bloc *try/catch*.
- Si l'exception n'est une instance d'aucun des deux types cités, elle sera remontée à la méthode appelante de *maMethode()*.

Exceptions : exemple courant

- Exemple :

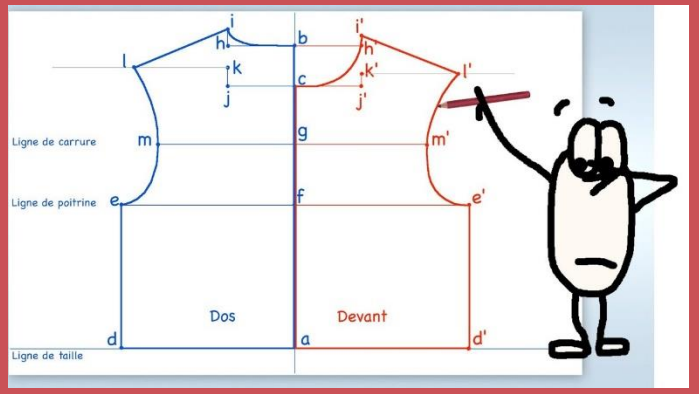
```
try {
        FileInputStream fs = new FileInputStream("toto");
} catch (FileNotFoundException e1) {
        System.out.println(«Fichier inconnu ») ;
}
```

- On veut ouvrir un flux en lecture sur un fichier « toto ». Le constructeur de *FileInputStream* est susceptible de remonter une exception *FileNotFoundException* (i.e. de faire « *throw new FileNotFoundException() »*)qui sera alors gérée par l'affichage d'un message.

Design pattern

Des outils communs

Université de Guyane



Design pattern : définition

- En informatique, un patron de conception (design pattern) est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels.
- Un patron de conception est issu de l'expérience des concepteurs de logiciels. Il décrit un arrangement récurrent de rôles et d'actions joués par les modules d'un logiciel.

- En conséquence :
- Un patron de conception n’est pas dépendant d’un langage, contrairement à sa mise en œuvre.
- Il existe probablement un ou plusieurs patrons de conception connus qui répondront à tout ou partie de votre problème.
- Il existe souvent plusieurs manières de mettre en œuvre un design pattern.

Design pattern: définition

- Il décrivent à la fois une problématique classique et une proposition de solution à cette problématique.
- Par exemple, lorsqu'on souhaite coder une application avec données stockées en ligne, on aura tendance à exploiter un design pattern de type MVC (ModelViewController)
- On a un problème de modélisation : il existe généralement une manière de le découper en sous problèmes correspondants chacun à un pattern connu.
- C'est une bonne pratique d'essayer de répondre à des problèmes en s'appuyant sur des design patterns car ils simplifient la compréhension de votre code et architecture logicielle.
- même s'il existe une infinité de solutions à un problème, celle que tout le monde adopte est généralement issue d'années d'essais de solutions plus originales mais moins performantes.

Singleton

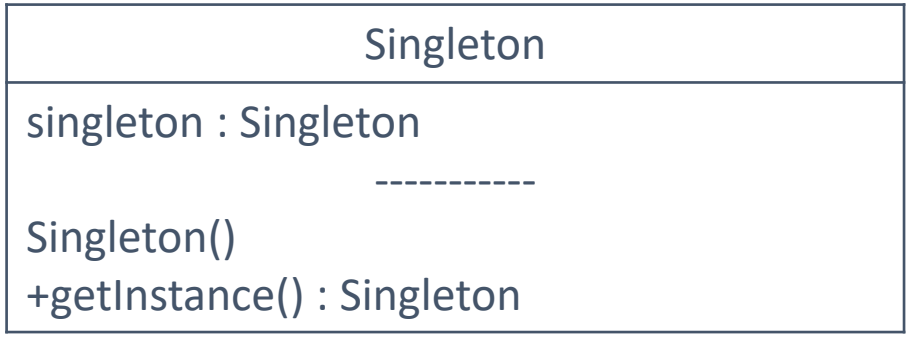
#LaSolitude

Université de Guyane

Singleton : définition

- Le singleton est un patron de conception (design pattern) dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet. Il est utilisé lorsqu'on a besoin exactement d'un objet pour coordonner des opérations dans un système.

- Usages typiques :
- Le monde dans un jeu s'il est unique (Minecraft, Wow)
- La configuration globale du programme
- La modélisation d'une ressource unique
- partager un état entre plusieurs classes

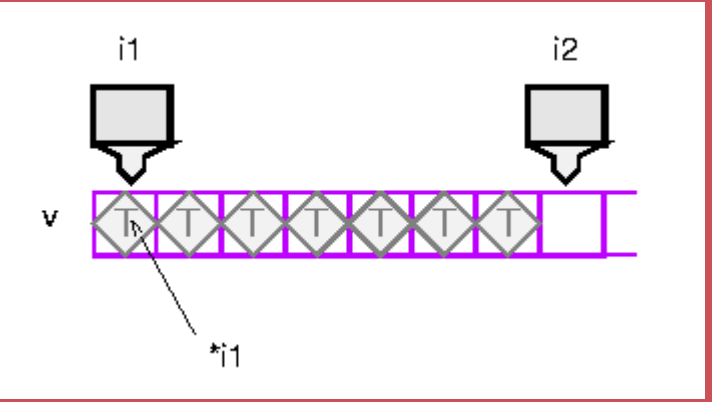


Singleton : implémentation (exemple)

```
public class MonSingleton {
        private static MonSingleton instance = new MonSingleton() ;
        private String monAttribut ;
        private MonSingleton() { }

        public String getMonAttribut() {
                return monAttribut ;
        }

        public void setMonAttribut(String a) {
                monAttribut = a ;
        }

        public static MonSingleton getInstance() {
                return instance ;
        }

}
```

Itérateur (Iterator)

Un pas après l'autre



Université de Guyane

Itérateur : définition

- Un itérateur (iterator en anglais) est un objet qui permet de parcourir tous les éléments contenus dans un autre objet (généralement un conteneur : collection, liste, arbre…).

- L'itérateur permet d’accéder aux éléments du conteneur en masquant son implémentation.

- Généralement l'itérateur a à minima deux méthodes :
  - récupération de l'objet courant
  - passer à l'objet suivant dans la collection

Itérateur : exemple

- En java, un iterateur implémente généralement l'interface Iterator<E> :

```
public class Chenil {
        public class IteratorPerso implements Iterator<Animal> {
                private int index = 0;
                public boolean hasNext() { return index < liste.length;  }
                public Animal next() { return liste[index++]; }
                public void remove() {        System.err.println("fonction non implémentée");          }
        }

        private Animal liste[];
        private int tailleChenil = 0;

        public Chenil(int t) {
                 tailleChenil = t;
                liste = new Animal[t];
        }
        public IteratorPerso iterator() { return new IteratorPerso();         }
        public Animal getAt(int i) {   return liste[i]; }
        public void setAt(int i, Animal v) { liste[i] = v; }
}
```

Itérateur : utilisation

- On peut alors l'utiliser comme suit :

```
Chenil monChenil = new Chenil(5);
 monChenil.setAt(0, new Chien("Medor"));
 monChenil.setAt(1, new Chien("Rex"));
 monChenil.setAt(2, new Chien("Milou"));
Chenil.IteratorPerso ite = t.iterator();
while(ite.hasNext()) {
        Animal a = ite.next();
        if(a != null) {
                System.out.println(a.nom);
        }
}
```

qui affichera :

« *Medor*
Rex
Milou »

Factory (fabrique)

Fabriquer vos fabriques

Université de Guyane

Factory : définition

- D'après wikipedia :

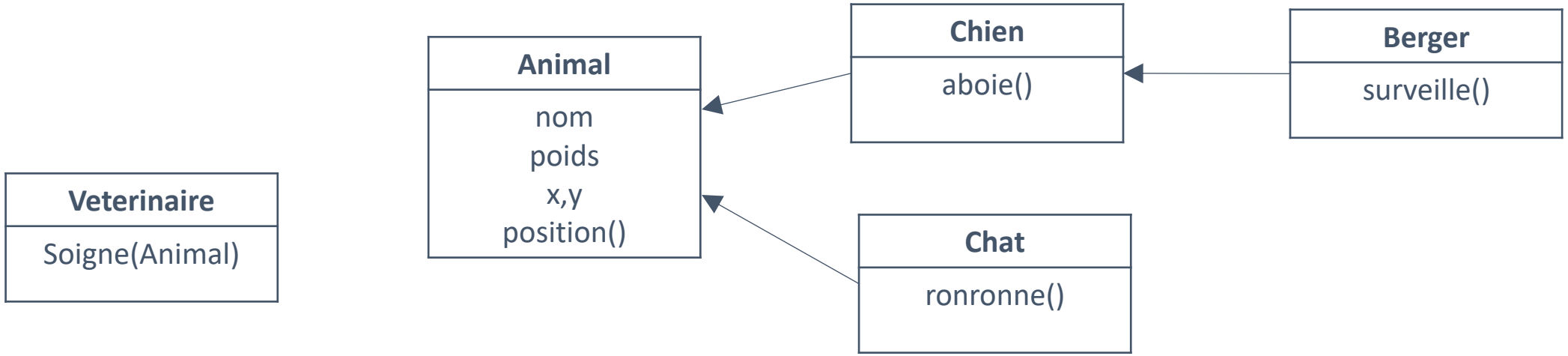
- La fabrique (factory) est un patron de conception permettant d'instancier des objets dont le type est dérivé d'un type abstrait. La classe exacte de l'objet n'est donc pas connue par l'appelant.

- Plusieurs fabriques peuvent être regroupées en une fabrique abstraite permettant d'instancier des objets dérivant de plusieurs types abstraits différents.

- Les fabriques sont généralement uniques dans un programme.

Factory : définition

- On utilise donc une fabrique pour instancier des objets de nature éventuellement différentes mais qui ont un ancêtre ou une interface commune.
- La fabrique utilisera les paramètres ou le contexte pour déterminer le bon type d'objet à instancier
- Si l'on reprend notre cabinet vétérinaire :



Factory : exemple

- Nous voudrions pouvoir créer l'*Animal* à son arrivée au cabinet vétérinaire pour que notre *Veterinaire* puisse le soigner.
- Le *Veterinaire* ne s'occupe pas du fait que ce soit un *Chien*, un *Chat* ou un *Berger* : il soigne des *Animal*. Toutefois, nous aimerions créer le bon type d'*Animal* en fonction des informations que nous avons à la fiche d'enregistrement et qui sont ses préférences :
- L'*Animal* aime
  - les souris => c'est un *Chat*
  - les moutons => un *Berger*
  - *Etc.*

- Nous aimerions aussi nous réserver la possibilité d'instancier d'autre type d'animaux dans le futur.
- Enfin, nous voudrions pouvoir faire ça depuis n'importe où dans le programme.
- Vu qu'on veut pouvoir évoluer, il n'est pas question de répéter un fragment de code partout où l'on veut créer ces animaux : il faut passer par une fonction qui centralisera ces instanciations : ***une fabrique***.

Factory : exemple

- Imaginons la méthode statique suivante dans *Animal* :

```
public static Animal creeAnimal(String nom, String preference) {
        Animal result = null;
        switch(preference) {
                case "souris":
                        result = new Chat(nom);
                        break;
                case "mouton":
                        result = new Berger(nom);
                        break;
                case "gibier" :
                        result = new Chasseur(nom);
                        break;
                case "territoire" :
                        result = new Gardien(nom);
                        break;
                default:  System.out.println("animal inconnu");
        }
        return ret;
}
```

Factory : utilisation

- Nous pouvons maintenant créer aisément tous les animaux désirés via un ensemble d'informations contextuelles :

```
Animal a = Animal.creeAnimal("Rex", "mouton") ; // crée un Berger
Animal b = Animal.creeAnimal("Felix",  "souris") ; // crée un chat
```

- Et si on veut rajouter des oiseaux qui aiment les graines, il suffit d'ajouter un cas dans le switch de la fabrique (et une classe *Oiseau* fille d'*Animal*, bien sur) :

```
case "graine":
        ret = new Oiseau(nom);
        break;
```

- L'avantage est que la seule partie du code impactée par cette modification est la fabrique, le code existant utilisateur de la fabrique fonctionne toujours.

MERCI !