ST
Dpt de Formation et de Recherche Sciences et Technologies

UNIVERSITÉ GUYANE

Programmation Objet - Cours 3
Outils avancés

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);
}
```

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();
        }
}
```

Interfaces

Comme des classes abstraites, mais pas pareil

Université de Guyane

Interface : définition

- ***Une interface*** est un type abstrait permettant de définir un comportement, pour l'affecter à des classes, indépendamment de leur position dans une hiérarchie de classe.
- Les classes implémentant une interface devront définir le comportement décrit par l'interface.

- Concrètement, une interface, en Java permet de déclarer un ensemble de méthodes afin d'imposer leur présence dans des classes qui devront les implémenter (à la manière des méthodes d'une classe abstraite).

- Exemples d'interfaces existant en java:

serializable
Une classe implémentant serializable doit définir les fonctions ***read***(*flux*) | ***write***(*flux*)

Comparable
Une classe implémentant comparable doit définir la fonction int ***compareTo***(*autreElementDeLaClasse*)

Interface : définition

- En java, une *interface* ressemble à une classe abstraite. Cependant, une classe peut implémenter plusieurs interfaces alors qu'elle ne peut hériter que d'une seule classe.

- Le mot clé pour définir une interface est ***interface***. Globalement, on écrira une interface de la même manière qu'une classe, en remplaçant ***class*** par ***interface***.
- Une interface peut hériter de plusieurs autres interfaces
- Une classe implémente une interface par le mot clé ***implements***. Une classe peut implémenter plusieurs interfaces, alors qu'elle ne peut hériter que d'une seule classe.

Interface : exemple

- Exemple: On définie l'interface « Num ». Ce qui garanti que toute classe qui implémentera cette interface définira les fonctions de multiplication, addition, et égalité entre ces objets.

```
public interface Num {
        public Num multiplication(Num a) ;
        public Num addition(Num a) ;
        public boolean egal(Num b) ;
}
```

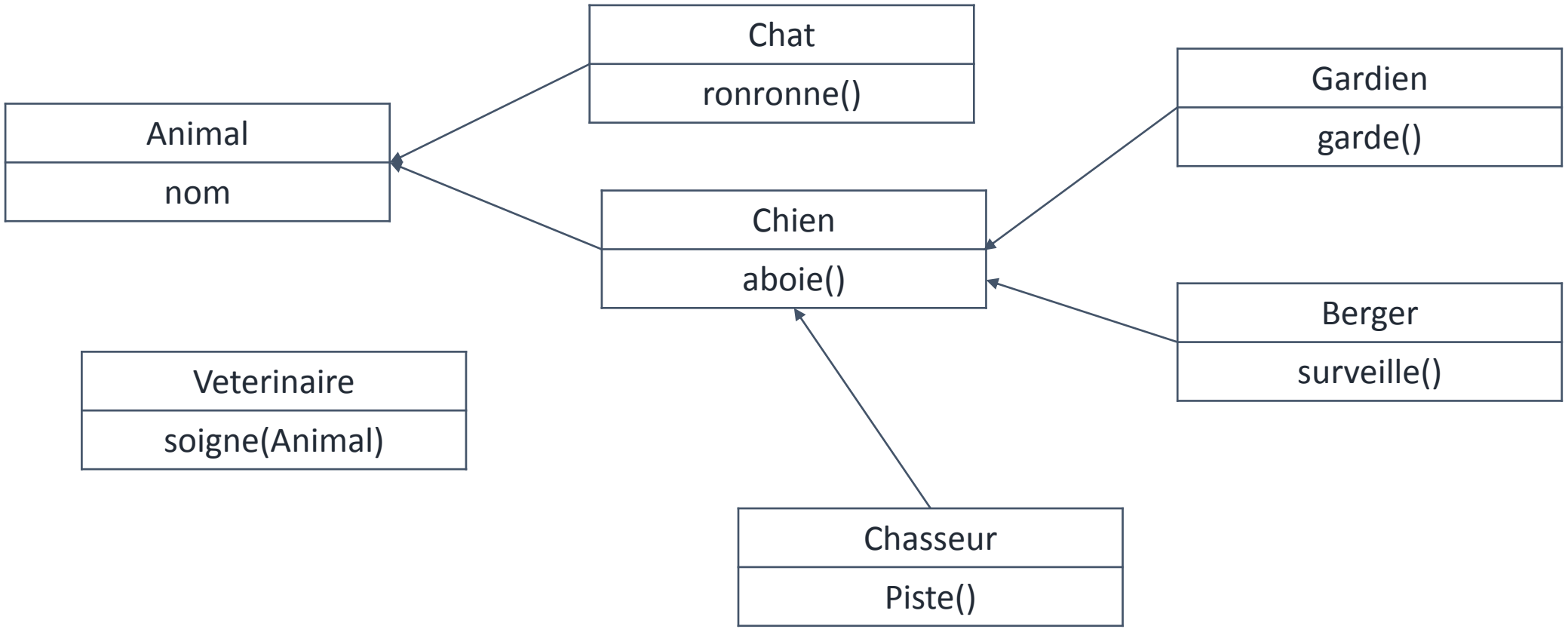
```
public class Entier implements Num {
    int chiffre;
    public Entier(int a){this.chiffre = a;}
    public Num multiplication(Num a) {
        return Entier(this.int*a);
    }

    public Num addition(Num a) {
        return Entier(this.int + a)
    }

    public boolean egal(Num b) {
        return this.int == b;
    }
}
```

Interface: exemple de problématique

- Si on reprend la hiérarchie de classes suivante :



Interface: exemple de problématique

- Nous aimerions programmer un distributeur qui permet de nourrir tout ce petit monde à la clinique vétérinaire (animaux et vétérinaires).

- On peut imaginer rajouter une classe comme suit :

```
public class Distributeur {

        public void nourrit(XXX quelqueChose) {
                quelqueChose.mange() ;
        }
}
```

Problème : Quelle classe mettre?

- Problème : on veut que nourrit puisse nourrir les animaux et les vétérinaires. On ne peut donc mettre ni Animal ni Veterinaire. On va donc utiliser une ***interface***!

Interface : garantir des propriétés communes entre classes

- On va donc définir une interface pour le comportement « mange » :

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

- On va désormais pouvoir adapter nos classes pour faire d'elles des classes ayant un comportement de mangeur.

Interface : adaptation d'une classe

- On commence par dire que *Veterinaire* implémente l'interface *Mangeur* :

```
public class Veterinaire implements Mangeur {
        /* code */

        /* méthode à implémenter pour correspondre à l'interface Mangeur*/
        public void mange() {
                System.out.println("Le vétérinaire mange une pizza en salle de repos");
        }
}
```

- Dans le cas précis du *Veterinaire*, on sait comment et ce que mange un vétérinaire.

Interface: adaptation d'une classe abstraite

- On indique que la classe *Animal* est un *Mangeur* :

```
public abstract class Animal implements Mangeur {
        /* code : attributs, constructeurs et autres… */
        public abstract void mange();
}
```

- Chaque classe fille d'animal mangera de manière spécifique. On laisse donc les classes filles définir leur propre méthode mange

```
public class Chat extends Animal {
    /* code : attributs, constructeurs et autres…*/

    public void mange() {
        ronronne();
        System.out.println(nom + " mange dans son assiette");
    }
}
```

Interface : utilisation

- On peut maintenant définir notre *Distributeur* grâce à notre interface mangeur:

```
public class Distributeur {
        public void nourrit(Mangeur m ) {
                m.mange();
        }
}
```

- On peut alors l'utiliser:

```
Public static int main(String[] args){
    Distributeur d = new Distributeur() ;
    Chat c = new Chat() ;
    Veterinaire v = new Veterinaire() ;

    d.nourrit(c) ;
    d.nourrit(v) ;
}
```

Interfaces : principe général

- On peut donc déclarer autant de comportements (i. e. interfaces) que nous voulons et les appliquer à des classes sans se préoccuper de leur position dans la hiérarchie de classes (Dormeur, Renomable, etc.).

- La philosophie : définir des comportements génériques sans se préoccuper des classes.

- Par exemple, les classes suivantes implémentent toutes l'interface standard List :
  - - ArrayList : un tableau
  - - LinkedList : une liste chaînée
  - - Stack : une pile

- Si on veut utiliser une List, on pourra utiliser n'importe quelle implémentation de List.

Interfaces : quelques règles à retenir

- Une classe ne peut hériter que d'une seule classe, mais peut implémenter plusieurs interfaces
- Une interface ne peut pas avoir de constructeur (contrairement à une classe abstraite).
- Les méthodes d'une interface sont toutes considérées comme abstraites et publiques (***public abstract***).
- Les attributs d'une interface sont tous considérés comme ***public static final***.
- Une classe abstraite peut implémenter une interface. Le cas échéant, il n'est pas obligatoire de définir toutes les méthodes de l'interface.
- Une interface peut hériter d'une ou plusieurs autres interfaces (mais pas d'une classe ou classe abstraite)

Ligne vers colonne	Classe	Interface
Classe	extends (une seule)	implements (plusieurs)
Interface		implements (plusieurs)

Interfaces : nouveauté Java 8

- On peut désormais définir des méthodes dans les interfaces! Mais pas n'importe comment.
- Avec des méthodes statiques comme dans n'importe quelle classe (voir cours 2)
- Ou avec des ***méthodes par défaut*** : des méthodes qui, si elles ne sont pas définies dans les classes filles, proposeront malgré tout un comportement par défaut:

```
interface mangeur{
    default void mange() {
        System.out.println(« Le mangeur mange »);
    }
}
```

- Attention, dans ce cas, il est possible d'avoir des problèmes lors de la compilation, si deux interfaces implémentées par une même classe proposent toutes les deux une fonction par défaut de même nom.

Les tableaux

Université de Guyane

Les tableaux statiques

- Cas classique : on définie un tableau de manière statique, (presque) comme en C

```
class Chenil{
    Chien[] listeChiens = new
Chien[10];
}
```

- Notre tableau a une taille fixe qu'on ne pourra modifier.
- On peut initialiser un tableau statique lors de sa création (cela fixera sa taille)

```
class Refuge{
    String[] animauxAcceptes = {« Chien », « Chat »,
« Oiseaux »};
}
```

Les tableaux dynamiques : ArrayList

- La bibliothèque « Collections » (import java.util.*) proposent des classes particulières permettant de manipuler des collections d'objets.
- Les différentes classes (ArrayList, Stack, HashMap, etc.) ont toutes des particularités différentes. Pour des tableaux dynamiques classiques, on utilisera généralement « ArrayList ».

```
class Chenil{
    ArrayList<Chien> listeChiens = new ArrayList<Chien>();
```

Classe des objets contenus dans le tableau

```
    public static int main(String args[]){
    Chenil c = new Chenil();
    c.listeChiens.add(new Chien(« médor »,26));
    Chien monChien = new Chien(« fidjie »,19);
    c.listeChiens.add(monChien);
   listeChiens.remove(monChien);}
}
```

Les tableaux : parcours

- On peut parcourir avec une boucle standard.

```
for(int i = 0; i <= tableauStatique.length; ++i){
    System.out.println(tableau[i]);
}

for(int i = 0; i <= tableauDynamique.size(); ++i){
    System.out.println(tableau.get(i));
}
```

- Ou avec une boucle for each

```
for(int elem : tableau){
    System.out.println(elem);
}
```

Redéfinition de méthode

Aussi appelé « surchage »

Université de Guyane

Surcharge de méthodes

- Cas classique : on redéfinie une méthode avec une signature différente, dans ce cas, tout va bien!

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

Surcharge de méthodes

- Redéfinition par héritage : on redéfinie une méthode *dans une classe fille* avec une signature similaire.

```
class Animal{
    String nom;
    int poids;
    public Animal(){
    }
    public void manger(){
        System.out.println(this.nom + « mange.»);
}
}
```

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

- Dans ce cas, on annote notre méthode avec « ***@override*** ». Ce n'est pas obligatoire, c'est une bonne pratique.
- On peut appeler la méthode de la classe mère dans la méthode de la classe fille avec le mot clé ***super.nom_methode()***

Les membres « *final* »

« Le changement, c’est pas maintenant »

Université de Guyane

Mot clé final

- Il sert à restreindre les possibilités de manipulation d'un membre ou d'une classe:
  - Pour une classe : elle ne peut être dérivée (une classe ne peut en hériter)
  - Pour une méthode : elle ne pourra pas être redéfinie (notamment dans une classe fille)
  - Pour une variable : une fois initialisée, sa valeur ne pourra pas être modifier (cette variable devient une constante)
    - Un attribut final devra être initialisé au plus tard dans le constructeur
    - Pour une variable qui fait référence à un objet, les attributs de l'objet pourront être changés, mais pas la référence (ce sera toujours le même objet)
    - Un paramètre peut-être final : on ne pourra modifier la valeur de ce paramètre dans la méthode

Mot clé final

- Il sert à restreindre les possibilités de manipulation d'un membre ou d'une classe:

```
class final Animal{
}
```

```
class Chien extends Animal{
}
```

```
class Animal{
    String nom;
    public final mange(){
        System.out.println(this.nom +
« mange.»)
    }
}
```

```
class Oiseau extends Animal{
    public mange(){
        System.out.println(this.nom + « picore.»
    }
}
```

```
class Animal{
    int final nbPattes = 4;
    public static int main(String[] args){
        Animal a = new Animal;
        this.nbPattes = 2;
    }}
```

Programmation générique

La guerre des classes

Université de Guyane

Programmation générique : définition

- La ***programmation générique*** (aussi appelée « ***généricité*** »), consiste à définir des outils génériques pouvant opérer sur différents types de données.

- C'est une forme de polymorphisme : le ***polymorphisme de type*** (appelé aussi ***paramétrage de type***) : en effet, le type de donnée apparaît comme un paramètre des algorithmes, structures de données ou autre.

- La méthode : utiliser dans un bloc de code (une classe ou une méthode) un type inconnu (un peu comme une variable). Quand on utilise la classe ou la méthode, on passe en paramètre le type qu'on va réellement utiliser. Ce type sera substitué au type inconnu dans le code.

Tableau<typeInconnu>

typeInconnu	typeInconnu	…			

Tableau<Animal>

minou	medor	titi	…		

Classe générique

- On peut donc définir une classe générique qui représente la cage d'un animal :

```
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 ;
        }
}
```

- *T* est un type qui sera passé en paramètre lors de l'utilisation du type.

Classe générique : contraintes

- On peut ajouter des contraintes. Par exemple pour garantir que le type passé en paramètre est une classe fille de la classe Animal :

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

Classe générique : contraintes

- On a donc une classe *Cage* qui prend en paramètre un type *T* qui **doit** dériver d'*Animal* et qui comportera :
  - un attribut privé *enCage* qui contiendra l'*Animal* de type *T*
  - une méthode enferme(T ani) qui prend un paramètre de type *T* et qui le met *enCage*
  - une méthode *libere()* qui libère l'*Animal* de type *T* et vide la cage.

- Nous pouvons donc l'utiliser avec n'importe quel type héritant d'*Animal. Ici, on fera une cage à chat* :

```
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();
}
```

Classe générique : contraintes

- Notre classe *Cage<Chat>* est alors utilisée comme si elle avait été écrite comme suit :

```
public class Cage {
        private Chat enCage;
        public void enferme(Chat a) {
                enCage = a ;
        }

        public Chat libere() {
                Chat ret = enCage ;
                enCage = null ;
                return ret;
        }
}
```

- C'est ce qui fait que la valeur de retour de *libere()* est de type *Chat* :

```
Chat autreChat = maCage.libere();
```

Classe générique : plusieurs types

- On peut aussi avoir plusieurs types génériques comme paramètres :

```
class MaClasse<T, U> {
        T monAttribut ;
        U autreAttribut ;

        /* code */
}
```

- Exemple de la bibliothèque standard : le tableau associatif

```
interface Map<K,V>
```

Classe générique : tableaux associatifs

- Le tableau associatif permet d'associer un élément à un autre. Comme un dictionnaire qui associe un mot à une définition par exemple.

```
class Chenil{
        HashMap<int,Chien> listeChiens = new HashMap<int,Chien>();
        public HashMap init(){
            listeChiens.put(123456,new Chien(« médor »,26));
            listeChiens.put(789654,new Chien(« fidjie »,19));
        }
                        …
}
```

Généricité : le principe

- On utilise généralement les classes génériques quand on a besoin d'appliquer des algorithmes génériques qui peuvent/qu'on veut (s')appliquer sur différents type de données.

- Cas typiques : les conteneurs (liste, collection, tableau associatif, etc.)
- Les actions réalisées (ajout, suppression, modification de la liste) ne sont pas dépendantes des objets contenus mais l'utilisateur veut pouvoir manipuler les objets contenus sans jouer du transtypage.

- Les conteneurs suivants sont fournit par la librairie standard Java :
  - ArrayList<T>
  - HashMap<K, V>
  - LinkedList<T>
- Quand on les utilise, on accède à tout un panel de méthodes existantes:
  - isEmpty()
  - contains()
  - Remove(), etc.

Généricité : exemple avancé

- Le ***polymorphisme de type*** (généricité) est parfois plus efficace que le ***polymorphisme par héritage*** pour factoriser du code.
- Exemple : plutôt que de coder une fonction min(a,b) par type de données, on peut passer par une fonction générique s'appuyant sur une classe utilisant la généricité sur des objets implémentant l'interface ***Comparable***.

```
/* On définie une classe comparable avec elle-même grâce à l'interface générique Comparable<T> de Java*/
class ClasseComparable implements Comparable<ClasseComparable> { /* code avec définition de compareTo(T)*/ }
/* On définie une classe qui offre une méthode static de comparaison quel que soit l'objet. Cet objet doit néanmoins implementer l'interface Comparable<T>, i. e. être comparable avec lui-même*/
class OutilsComparaison {
        public static <T extends Comparable<T>> T min(T objetA, T objetB) {
                T objetPetit = objetA;
                if(objetA.compareTo(objetB)> 0)
                        objetPetit = objetB;
                return objetPetit;
        }
}
```

- alors on peut écrire :

```
ClasseComparable toto = new ClasseComparable(…);
ClasseComparable titi = new ClasseComparable(…);
ClasseComparable resultatMin = OutilsComparaison.min(toto, titi) ;
```

Généricité : la bibliothèque standard

- La bibliothèque standard Java (https://openclassrooms.com/fr/courses/1894236-programmez-avec-le-langage-c/1902913-quest-ce-que-la-bibliotheque-standard) utilise intensément les interfaces et la programmation générique pour offrir des fonctionnalités basées sur le comportement de ces interfaces.

- Exemple avec l'interface standard *Comparable :*
  - Toutes les classes implémentant Comparable ont une méthode : *public int compareTo(T arg) ;*
  - Cette méthode doit retourner 0 si l'objet arg est égal à l'instance courante, un entier négatif s'il est plus petit et un entier positif s'il est plus grand.

- L'intérêt est que l'on peut utiliser un tri sur des objet comparables via la méthode static ***sort*** de la classe standard Collections, sans avoir à coder une fonction supplémentaire!

public static <T extends Comparable<? super T>> void sort(List<T> list);

Signifie : lui-même ou un parent implémentant l'interface comparable

Classe générique : exemple final

Créons une liste d'animaux que l'on va pouvoir trier efficacement:

```
public class Animal implements Comparable<Animal> {
    public int poids;
    public Animal(int p){
        this.poids = p;
    }
    public int compareTo(Animal a){
        return this.poids.compareTo(a.poids);
    }
    public static int main(String[] args){
        Chat minou = new Chat(6);
        Chien medor = new Chien(25);
        Oiseau titi = new Oiseau(2);
        ArrayList<Animal> maListe = new ArrayList<Animal>();
        maListe.add(minou);
        maListe.add(titi);
         maListe.add(medor);
        Collections.sort(maListe);
    }
}
```

Introspection

Qui suis-je?

Université de Guyane

Introspection: définition

- On appelle introspection la faculté à examiner des informations et méta données concernant le type de l'objet que l'on examine, pendant l'exécution du programme.

Introspection: les outils

En java, l'introspection passe par plusieurs outils:

- ***instanceof*** qui permet de tester si un objet est instance d'une classe :

```
public static int main(String[] args){
        Chien medor = new Chien(« médor »);
        if(medor instance of Animal){
                System.out.println(medor.name + « est bien un animal! »);
        }
        if(medor instance of Chien){
                System.out.println(medor.name + « est également un chient! »);
        }
}
```

- Affichage : « médor est bien un animal
  médor est également un chien! »

Introspection : la classe « Class »

- La classe ***Class*** qui récupère les renseignements d'un objet. On récupère l'élément de la classe Class sur un objet par la méthode ***getClass()*** :

```
Chat c = new Chat() ;
Class infos = c.getClass();
System.out.println(« je suis un » + infos.getName()) ;
```

- affichera : « *je suis un Chat* »

MERCI !