Licence Pro DevOps

Java :

(encore) quelques rappels

Interfaces

- Peuvent être vues comme des classes “totalement abstraites” :
  - Aucun attribut
  - Toutes les méthodes abstraites
- Syntaxe :

```
public interface Drawable {

   public void draw();

   public int getNumberofChildren();

}
```

```
public class Losange implements Drawable {
    @Override
    public void draw() {
        ...
    }

    @Override
    public int getNumberofChildren()  {
        ...
    }
}
```

Interfaces

- Le nom d'une interface finit généralement par *-able
  - Donne une "capacité" aux classes qui l'implémentent (dans le sens "être capable de")
- Une classe peut implémenter plusieurs interfaces :

```
public class Losange implements Drawable, Mesurable {
```

- La classe doit alors déclarer les méthodes définies dans toutes les interfaces (pas de problème en cas de signatures identiques dans des interfaces différentes)

Interfaces

- Possibilité d’héritage entre interfaces

```
public interface Fooable extends Drawable {
```

- Dans ce cas, on hérite de toutes les méthodes de l’interface mère...

Interfaces

- Depuis Java 8, possibilité de fournir une implémentation “par défaut” à certaines méthodes d’une interface

```
public default int methode(int x) {
   return x*x;
}
```

- Pour une classe implémentant une telle interface, nul besoin d’implémenter les méthodes ayant une implémentation par défaut. Mais si la méthode est implémentée dans la classe, c’est cette implémentation qui sera appelée en priorité

Interfaces

- Une classe peut implémenter plusieurs interfaces
  + les interfaces peuvent implémenter certaines méthodes

  = Héritage multiple !

- Que se passe-t-il si :
  - L'interface Aable a une implémentation d'une méthode m
  - L'interface Bable a une implémentation d'une méthode m
  - La classe C implémente Aable et Bable

  Réponse : erreur de compilation !

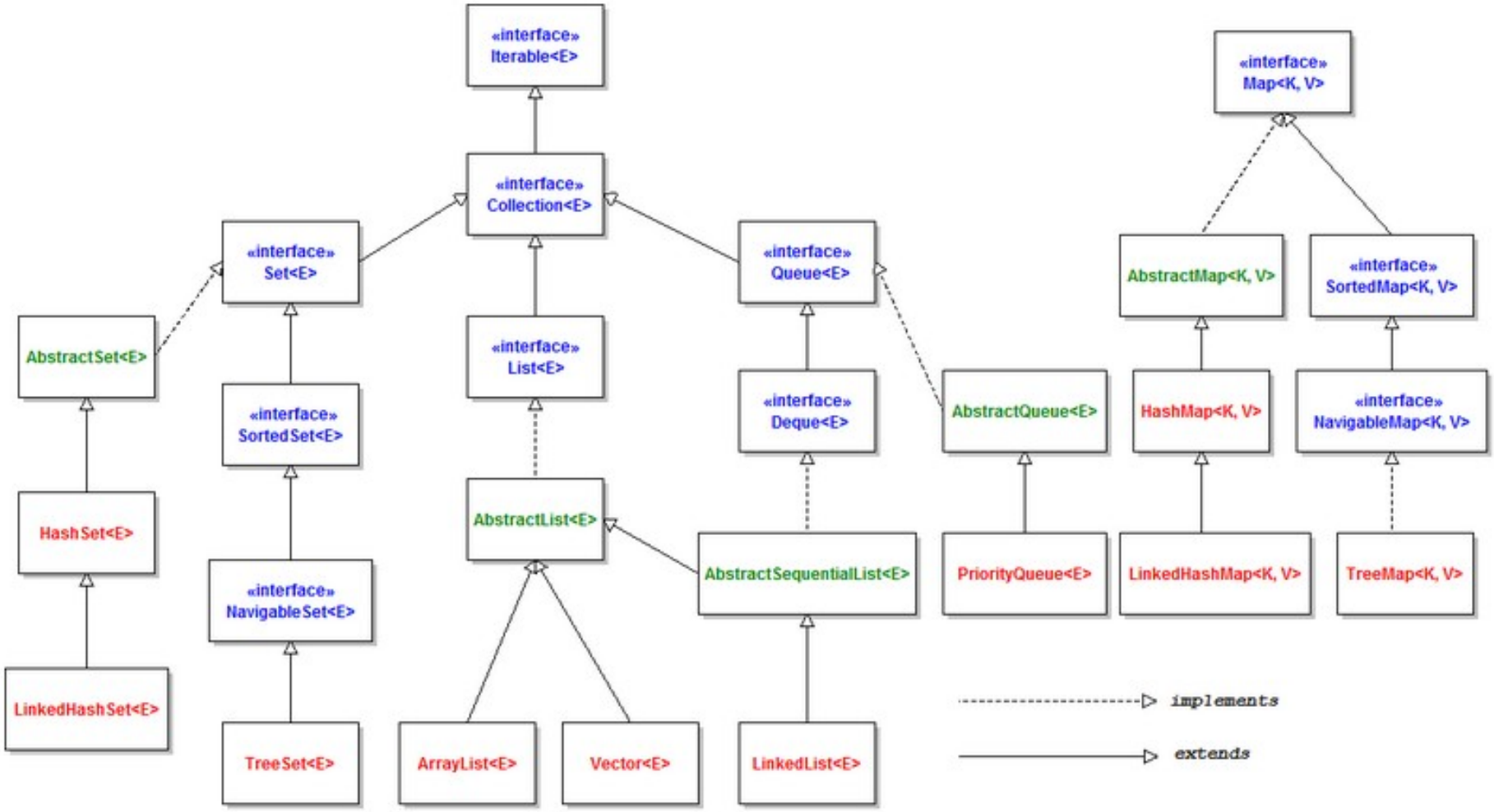
  Solution : redéfinir m dans la classe C

    - Si on souhaite appeler l'implémentation de Bable, on écrira

```
Bable.super.m();
```

Collections, généricité

private HashSet<Maladie> lesMaladies



Class diagram of Java Collections framework

Collections, généricité

- Framework unifié pour stocker des collections d'objets
  → généralise la notion de tableau
- Plusieurs interfaces différentes
  - List : liste ordonnée d'éléments
  - Set : ne contient pas de doublons
  - Queue : file (FIFO : First In First Out)
  - Map : stocke des couples (clé, valeur)
- Pour chacune de ces interfaces, plusieurs implémentations (=classes) disponibles :
  - List : ArrayList, LinkedList, Vector
  - Set : HashSet, TreeSet, …
  - Queue : PriorityQueue, LinkedList
  - Map : Hashtable, ...

Collections, généricité

- Généricité : permet de définir et d'utiliser une classe avec un type générique. Exemples :
  - LinkedList<String> ex1 = new LinkedList<String>()
  - HashSet<Maladies> ex2 = new HashSet<Maladies>()
  - List<Voiture> ex3 = new ArrayList<Voiture>()
- Toutes ces classes implémentent (directement ou par héritage) de Collection :
  - ex1.add("toto");
  - ex2.remove(m) ; //où m est une maladie
  - ex3.isEmpty();

Définition d'une classe

Attribut static :
valeur commune à toutes les instances

Attributs de type primitif

Attributs de type "composé"
(instances d'autres classes)

```
public class Patient {

    private static int nbMalade = 0;

    private double vie;
    private double age;

    private String nom;
    private String prenom;
    private HashSet<Maladie> lesMaladies;

    public Patient(String n, String p, double v, double a) {
        nom = n;
        prenom = p;
        vie = v;
        age = a;
        lesMaladies = new HashSet<Maladie>();
    }

    public void ajoutMaladie(Maladie m) {
        //si c'est la première maladie, on incrémente le nombre de patients malades
        if (lesMaladies.size() == 0) {
            nbMalade++;
        }
        lesMaladies.add(m);
    }
```

Définition d'une classe

Rôle du constructeur : initialiser tous les attributs

Avec un constructeur vide/pas de constructeur

```
public Patient() {

}
```

```
public class Patient {


    private static int nbMalades = 0;

    private double vie; ← 0
    private double age; ← 0
    private String nom; ←null
    private String prenom; ←null
    private Set<Maladie> lesMaladies;←null
```

```
Patient p = new Patient();
```

Définition d'une classe

Rôle du constructeur : initialiser tous les attributs

```
public class Patient {


    private static int nbMalades = 0;

    private double vie; ← 100
    private double age; ← 30
    private String nom; ← "Watrigant"
    private String prenom; ← "Rémi"
    private Set<Maladie> lesMaladies; ← un objet Set<Maladie>
```

Avec un constructeur qui initialise tout :

```
public Patient(String nom, String prenom, double vie, double age) {
    this.nom = nom;
    this.prenom = prenom;
    this.vie = vie;
    this.age = age;
    lesMaladies = new HashSet<Maladie>();
}
```

```
Patient p = new Patient("Watrigant", "Rémi", 100, 30);
```

```
Exception in thread "main" java.lang.NullPointerException
```

```java
public class Patient {

    private static int nbMalades = 0;

    private double vie;
    private double age;
    private String nom;
    private String prenom;
    private Set<Maladie> lesMaladies;

    /**
     * Constructeur
     * @param nom nom du patient
     * @param prenom prenom du patient
     * @param vie points de vie du patient
     * @param age age du patient
     */
    public Patient(String nom, String prenom, double vie, double age) {
        this.nom = nom;
        this.prenom = prenom;
        this.vie = vie;
        this.age = age;
        //lesMaladies = new HashSet<Maladie>();
    }

    /**
     * ajoute une maladie au patient. Clone la maladie recue en parmètre (relation de composition)
     * @param m la maladie à ajouter
     */
    public void ajoutMaladie(Maladie m) {
        try {
            //si c'est la première maladie, on incrémente le nombre de patients malades
            if (lesMaladies.size() == 0) {
                nbMalades++;
            }

            Maladie newM = (Maladie)m.clone();
            lesMaladies.add(newM);
        } catch (CloneNotSupportedException ex) {
            Logger.getLogger(Patient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

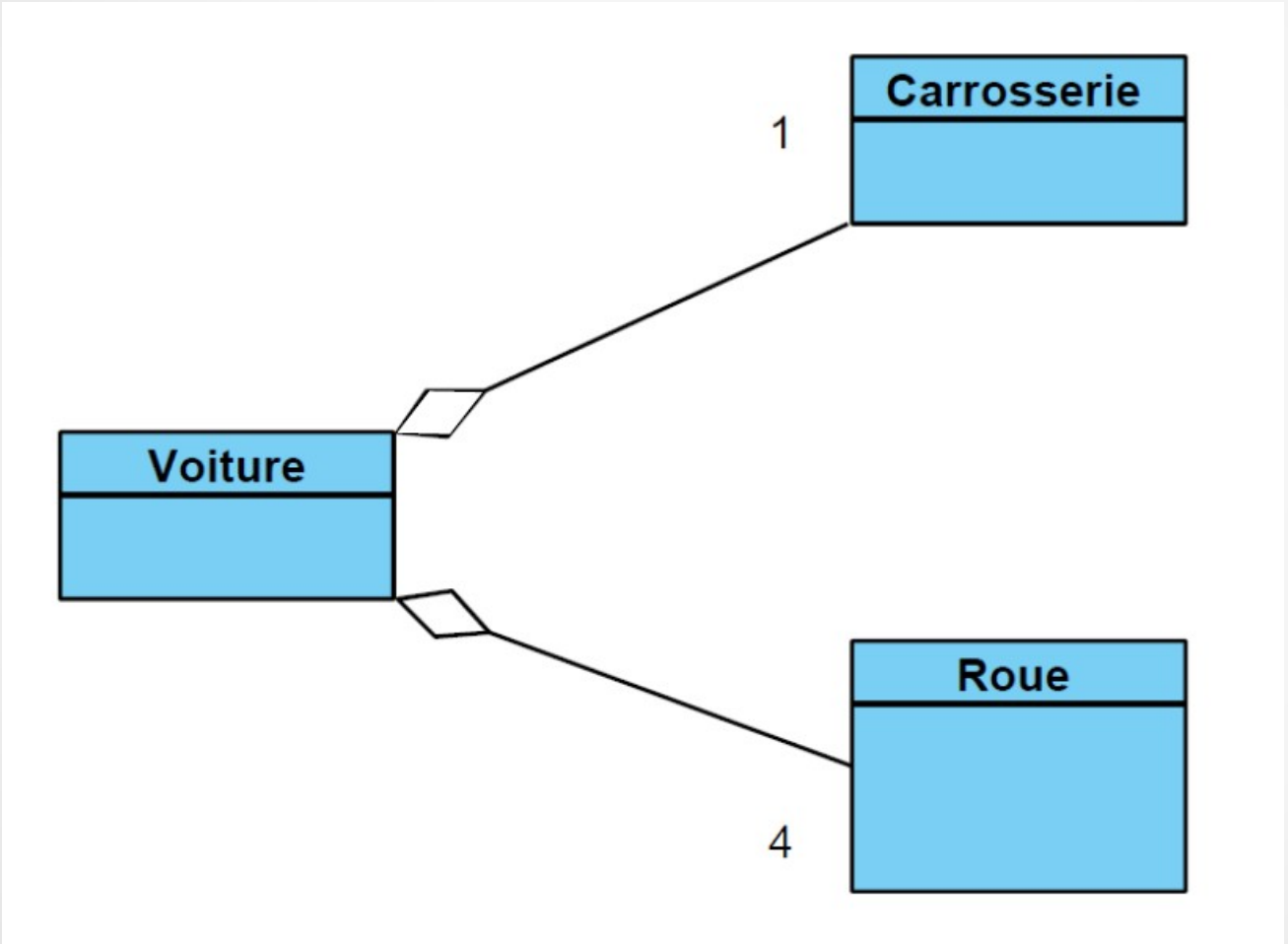
Compositions/agrégations

- Au cœur de la programmation orientée objet (POO)
- Modélise la relation d’appartenance « *has a* » ou « *is part of* »

Agrégation

- Description d'une voiture :
  - Une carrosserie `Classe Carrosserie`
  - 4 roues `Classe Roue`
  - …
- Description d'une roue :
  - Diamètre
  - Largeur
  - …

Agrégation



Agrégation

```
public class Voiture{
    private Carrosserie car;
    private Roue avD, avG, arD, arG;
    // Autres attributs éventuels
    private String immat;

    /*
    * Constructeur
    */
    public Voiture(Carrosserie obj1, Roue obj2,
    Roue obj3, Roue obj4, Roue obj5, String im){
        car = obj1;
        avD = obj2;
        avG = obj3;
        arD = obj4;
        arG = obj5;
        immat = im;
    }
}
```

Agrégation

```
public static void main(String[] args){
    // Création des composants
    Carrosserie laCarrosserie = new Carrosserie(...);
    Roue laRoue1 = new Roue(...);
    Roue laRoue2 = new Roue(...);
    Roue laRoue3 = new Roue(...);
    Roue laRoue4 = new Roue(...);

    // Création du composé
    Voiture maVoiture = new Voiture(laCarrosserie, laRoue1, laRoue2, laRoue3, laRoue4);
}
```

Agrégation

- **Agrégation**
  - Le composant existe en dehors de l’agrégé

- Exemple :
  - Les roues et la voiture

Agrégation

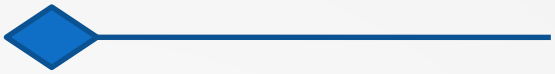
```java
public class A{
    private B b;

    public A(B x){
        b = x;
    }

    public void montrer(){
        System.out.print("mon agrégé : ");
        b.montrer();
    }
}
```

```java
public class B{
    private String nom;

    public B(String x){
        nom = x;
    }

    public void montrer(){
        System.out.println(nom);
    }
}
```

```java
public static void main(String[] args){
    B b = new B("unB");
    b.montrer();
    A a = new A(b);
    a.montrer();
}
```

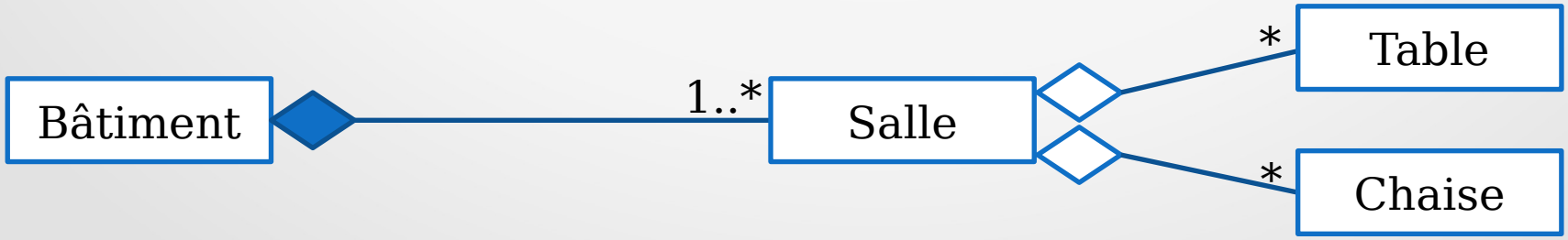
Composition

- **Composition**



  - *« Agrégation forte »*
  - Tous les composants sont détruits quand on détruit le composé

- Exemple :
  - Un bâtiment de différents étages comporte des salles, qui contiennent des chaises et des tables



Composition

```java
public class A {

    B b;

    public A(){
        b = new B("mon B");
    }

    public A(String nomB) {
        b = new B(nomB);
    }

    public A(B autreB) {
        b = new B(autreB);
    }
```

```java
public class B {

    String unAttribut;

    public B(String unAttribut) {
        this.unAttribut = unAttribut;
    }

    public B(B autreB) {
        this.unAttribut = autreB.unAttribut;
    }
```

Constructeur
par clonage

Association

- Une classe A est en association avec une classe B si B apparaît dans la classe A, par exemple :
  - En paramètre de méthode
  - En type de retour de méthode
  - Dans le corps d’une méthode

Retour sur le TP

```
Maladie grippe = new Virus("Grippe", 50, 10, "etat grippal");

Patient patientToto = new Patient("To", "To", 100, 30);
Patient patientTata = new Patient("Ta", "Ta", 100, 50);

patientToto.ajoutMaladie(grippe);
patientToto.ajoutMaladie(grippe);
```

Agrégation ou composition ?

Encore 2 courts rappels...

Rappels en vrac 1

- Comparaison d'objets

```
String s1 = new String("test");
if (s1 == "test")
    System.out.println("trop facile les tests !");
else
    System.out.println("en fait non");
```

- → affiche "en fait non"
- == compare les références (pointeurs)
  Utiliser plutôt : `if (s1.equals("test"))`

  Ou encore mieux : `if ("test".equals(s1))`

  (car on est sûr que **"test"** ne sera jamais **null**)

Rappels en vrac 2

- Écrire une méthode equals robuste
  Exemple de la classe Maladie :

```
@Override
public boolean equals(Object other) {
    if (other == null) return false;
    if (this == other) return true;
    if ((other instanceof Maladie) == false) return false;
    Maladie otherMaladie = (Maladie)other;

    //puis les "vrais" tests de comparaison
    return nom.equals(otherMaladie.nom);
}
```