Conception et Programmation objet Avancées

Proxy et Itérateur

Sylvie Coste

IUT de Lens

2021–2022

UNIVERSITÉ D'ARTOIS

Plan

Proxy

Itérateur

Un problème d'accès à la conduite d'une voiture

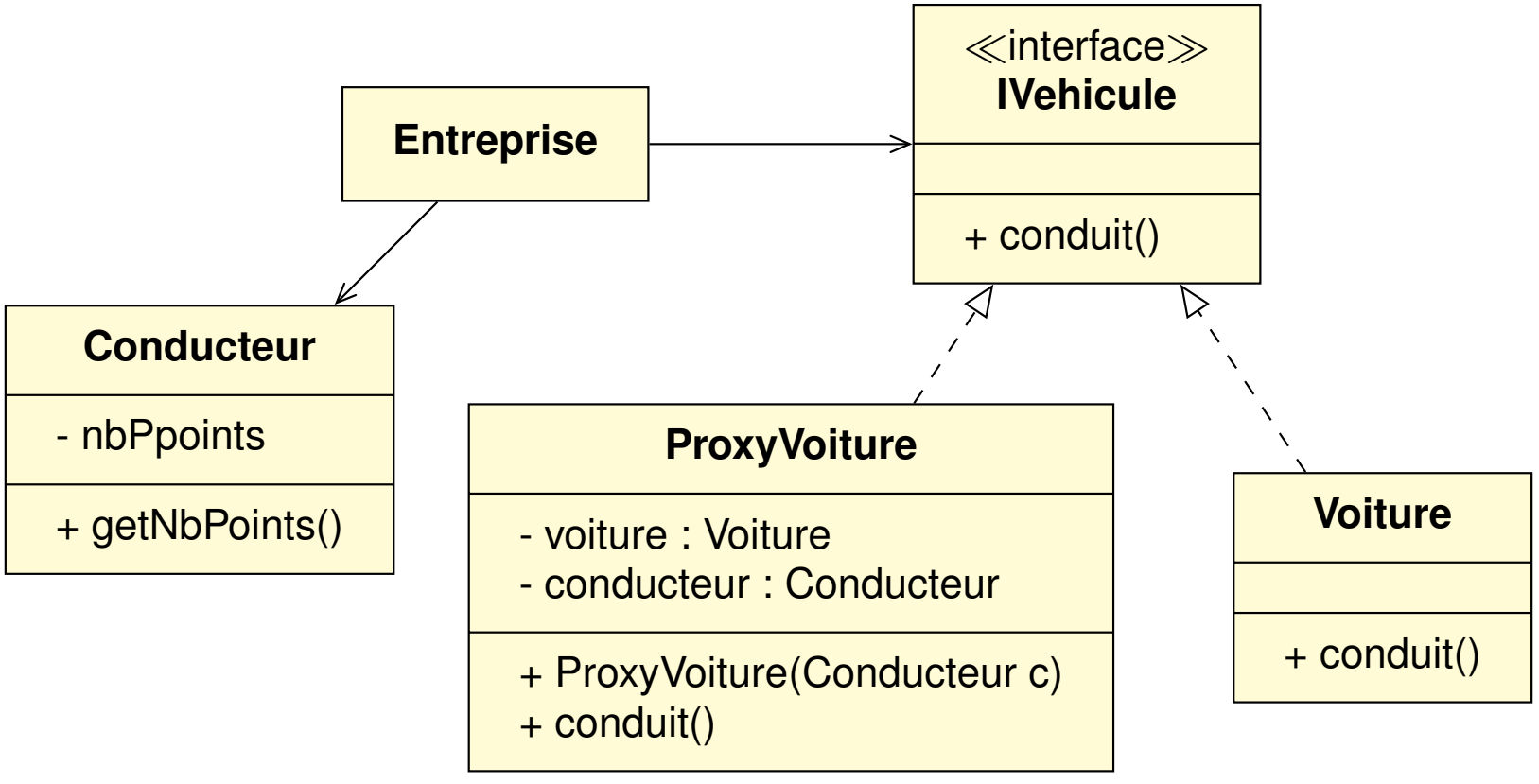
On souhaite écrire une application qui gère le parc de voitures d'une entreprise

Quand un employé doit utiliser une voiture de l'entreprise, l'entreprise commence par vérifier si son permis est encore valide (le nombre de points est strictement positif)

Si c'est le cas, l'application crée l'instance de la voiture et l'employé peut la conduire

- Quelle modélisation proposez-vous ?

Diagramme de classes



Design pattern

Objectif

- Contrôler l'accès à un objet

Problème

- Il faut vérifier certaines conditions avant d'accéder à l'objet, soit de droit, soit d'utilisation de ressources

Solution

- On ajoute une objet Proxy qui sert de substitut à l'objet sujet et implémente les fonctionnalités additionnelles nécessaires

Conséquence

- L'utilisation du Proxy à la place du SujetRéel est transparente pour la classe cliente

Les constituants du DP

InterfaceSujet

- Définit les méthodes communes au proxy et au sujet réel

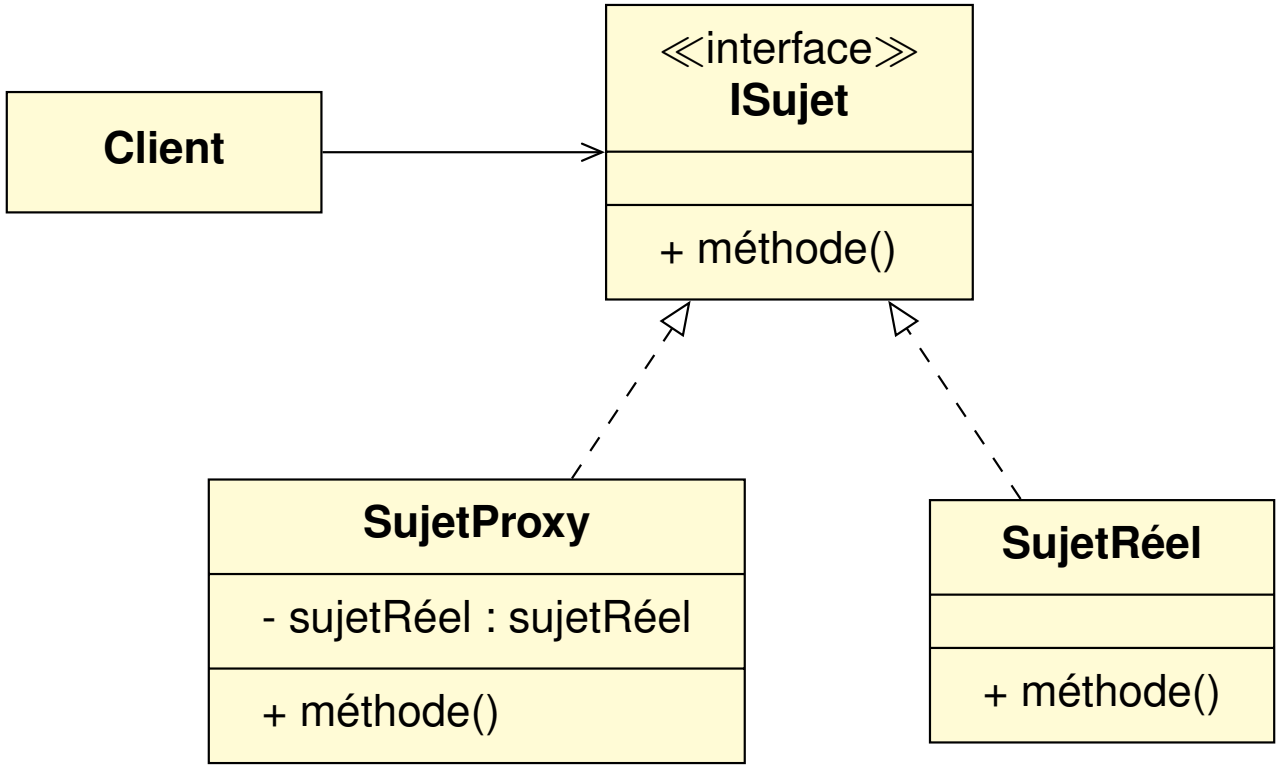
Proxy

- Se substitue au sujet réel : il est chargé de créer (et détruire) celui-ci et lui délègue ses messages

SujetRéel

- Objet que le *Proxy* contrôle et représente

Diagramme de classes : cas général



Indications d'utilisation du DP proxy

Rappel

Le DP *Proxy* crée un objet qui implémente la même interface que le Sujet, contient une référence à l'objet Sujet et sert à contrôler l'accès à ce dernier.

Proxy de protection

- permet de sécuriser l'accès à un objet

Proxy virtuel

- permet de créer un objet de taille importante au meilleur moment

Proxy distant

- permet d'accéder à un objet qui s'exécute dans une autre environnement

Proxy : savez-vous le programmer ?

Programmer le DP proxy

- Coder l'interface *IVehicule*
- Coder la classe ProxyVoiture
- Créer deux conducteurs Castor et Pollux qui possèdent respectivement 12 et 0 points sur leur permis de conduire, créer deux voitures v1 et v2 et faites conduire v1 à Castor et v2 à Pollux.
- Prévoir le résultat affiché. Cela correspond-il à ce que vous souhaitez ?

Le code du proxy (1)

L'interface IVehicule

```
public interface IVehicule {
    public void conduitVoiture();
}
```

La classe ProxyVoiture

```
public class ProxyVoiture implements IVoiture{
    private Conducteur conducteur;
    private Voiture voiture;
    public ProxyVoiture(Conducteur conducteur) {
        this.conducteur = conducteur;
        this.voiture = new Voiture();
    }
    public void conduitVoiture() {
        if (conducteur.getNbPoints()>0)
            voiture.conduitVoiture();
        else
            System.out.println("Désolé, permis non valide");
    }
}
```

Le code du proxy (2)

Utilisation par la classe cliente

```
Conducteur castor,pollux;
castor = new Conducteur(12);
pollux = new Conducteur(0);
IVehicule v1 = new ProxyVoiture(castor);
v1.conduitVoiture();
IVehicule v2 = new ProxyVoiture(pollux);
v2.conduitVoiture();
```

Résultat

```
Vroum-vroum
# castor a un nombre de points strictement positif
# il peut conduire la voiture
Désolé, permis non valide
# pollux n'a plus de point, il ne peut pas conduire
```

Adaptateur/décorateur/proxy : des design pattern « proches »

DP structurels

- Étudient la façon de composer des classes ou des objets pour réaliser des structures plus importantes
- Les modèles structurels de classe utilisent l'héritage pour composer interfaces et implémentations
- Les modèles structurels objets décrivent les moyens de composer des objets pour réaliser de nouvelles fonctionnalités
- Possibilité de modifier la composition à l'exécution

Plan

Proxy

Itérateur

Un problème de parcours de collections

On souhaite donner un accès séquentiel aux livres composant une bibliothèque.

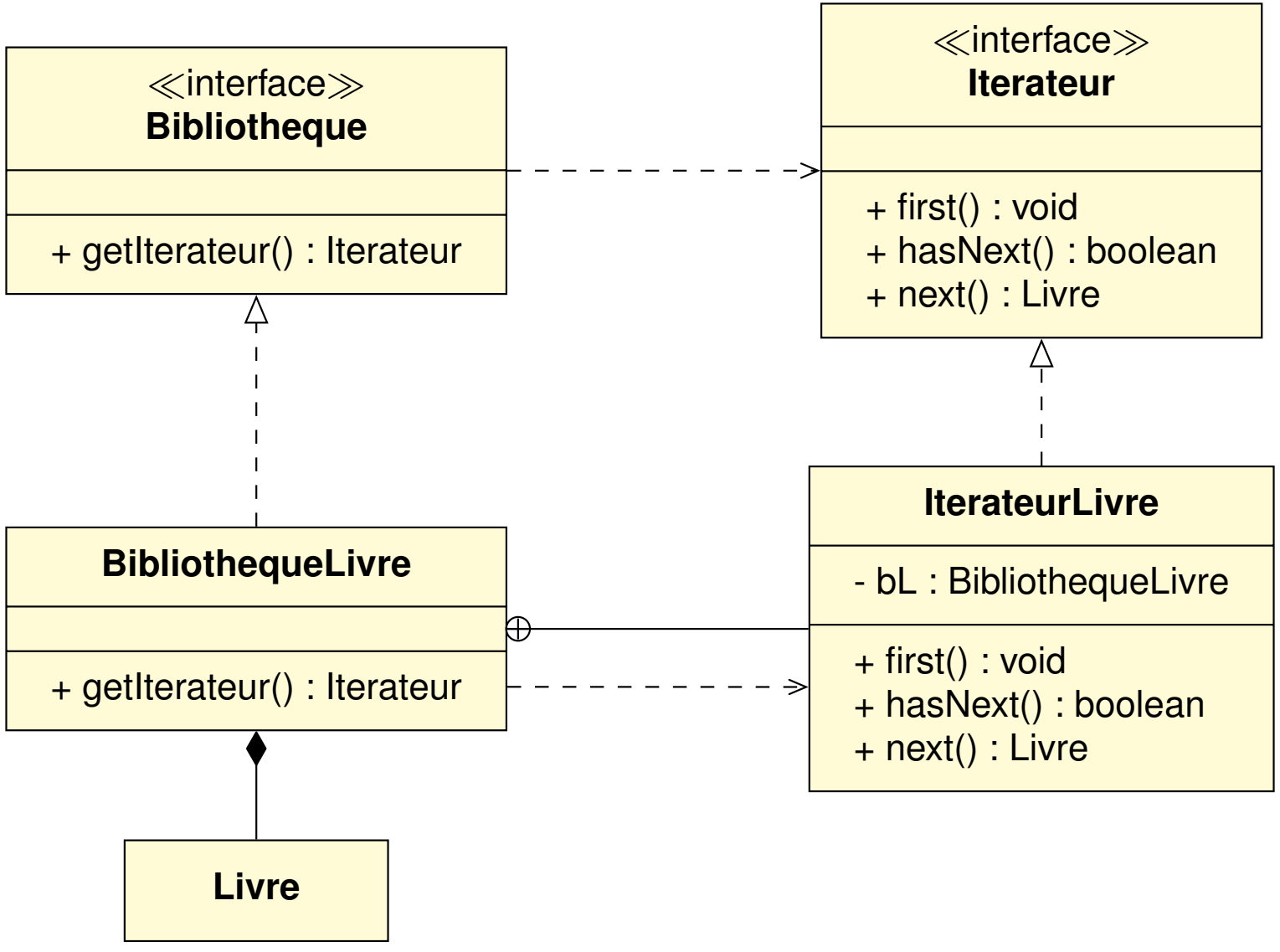
Pour cela, nous devons implanter les méthodes suivantes

- *first()* initialise le parcours de la bibliothèque
- *hasNext()* renvoie vrai si le livre courant existe
- *next()* renvoie le livre courant et avance sur le suivant

On souhaite éventuellement être capable de faire plusieurs parcours à la fois

- Comment procéder pour réaliser cela ?
- Qui doit fournir l'itérateur ?

Diagramme de classes



Design pattern Itérateur

Objectif

Découpler l'algorithme d'itération du type de conteneur

Problème

Accéder de manière unifiée à une collection d'objets

Solution

Un objet itérateur chargé de l'accès aux objets de la collection

Conséquence

Le parcours est indépendant de l'implémentation de la collection et il est paramétrable

Les constituants du DP itérateur

Iterateur

- Interface qui définit les méthodes *first*, *hasNext* et *next*

Collection

- Interface qui définit des méthodes de création, initialisation et retourne une instance Iterateur

CollectionConcrète

- Classe qui décrit la collection à parcourir

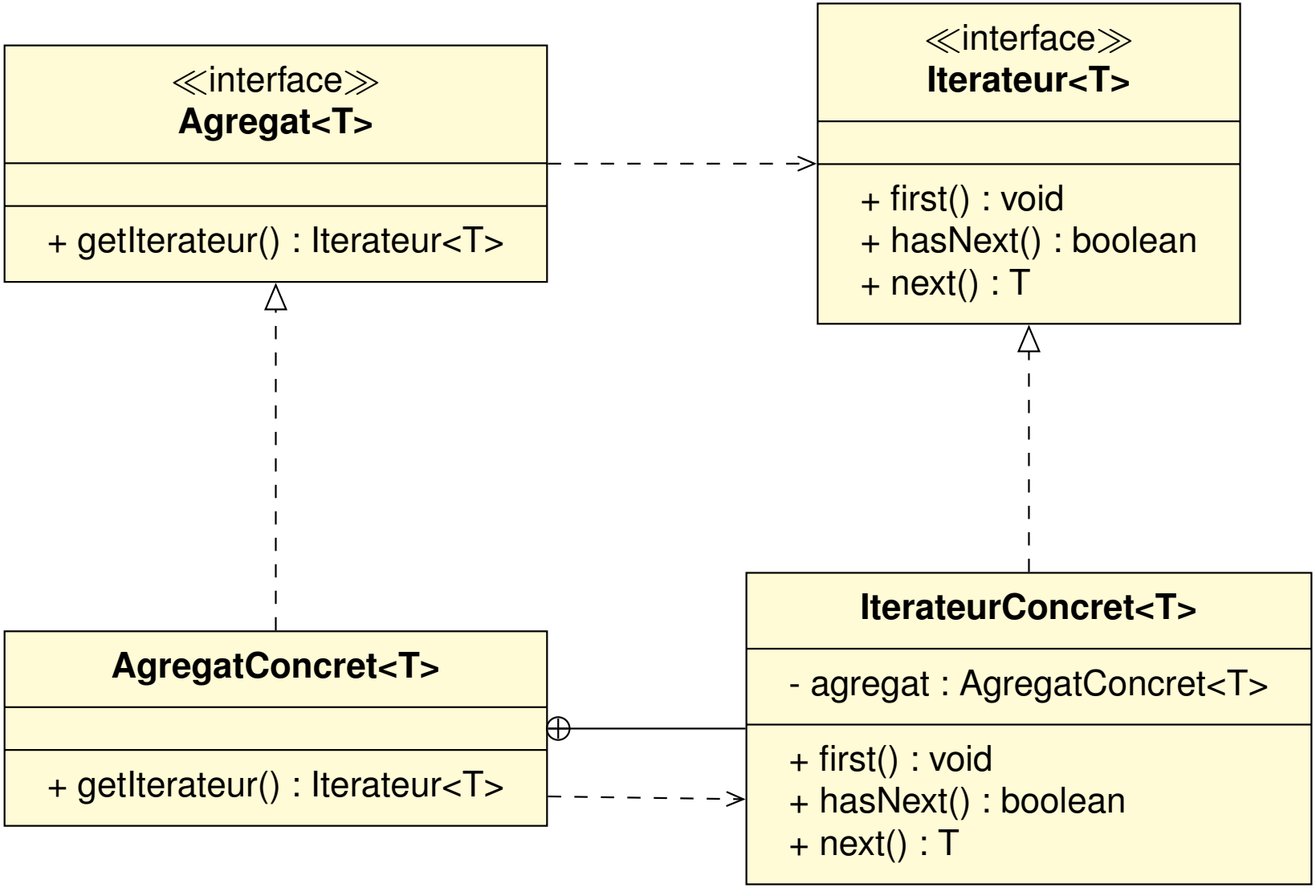
IterateurConcret

- Classe (souvent interne) à la *CollectionConcrète* qui implémente le parcours

Indications d'utilisation du DP itérateur

- Fournir un parcours à une collection de manière unifiée et cohérente indépendamment de la structure de données choisie
- Implanter des restrictions additionnelles dans le parcours
- Paramétrer plusieurs types de parcours selon des critères

Diagramme de classes : cas général



Classe paramétrée en JAVA

Utilisation

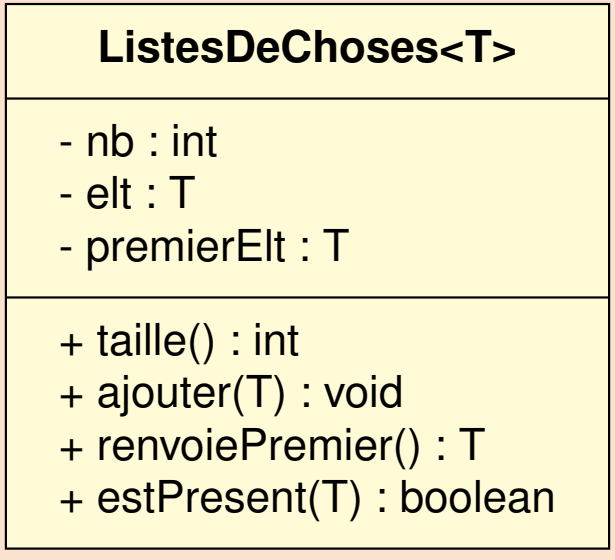
- Classe associée à la classe des objets qu'elle manipule
- Exemple : on veut manipuler une liste de livres ou de chaînes de caractères
- (appelée «générique» en Java)

Notation

- La classe associée est indiquée entre < >
- `ArrayList<String> = new ArrayList<String>();`
- Type à préciser à la déclaration de la référence et à la création de l'objet

Classe paramétrée : utilisation

en UML



Utilisation

- T est le type paramètre
- peut apparaître dans les attributs
- et dans les méthodes

en Java

```
public class ListeDeChoses<T>{
  private int nb;
  private T elt;
  ...
  public void ajouter(T elt){
  ...
  }
}
```

Rappel : les classes internes (simples)

Classe interne simple (*Inner class*)

- Déclarée à l'intérieur d'une autre classe
- Visibilité modifiée
- Déroge à la règle 1 classe $\leftrightarrow$ 1 fichier

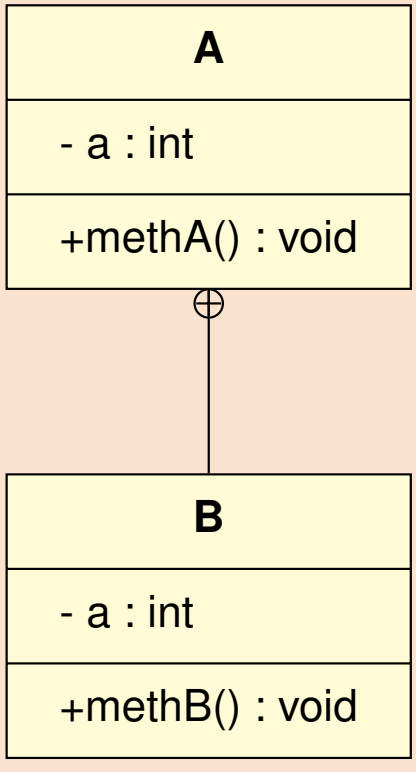
- Est un membre de la classe contenante
- Voit les membres privés de la classe contenante
- Possède une référence (champ caché) sur la classe contenante

Autres types de classes internes

- statique
- locale (interne à une méthode)
- anonyme (locale sans nom)

Classe interne : utilisation

en UML



en Java

```
public class A{
  private int a;
  public class B{
    private int a;
    public void methB(){
      this.a ... ;
      A.this.a ...;
    }
  };
  public void methA(){
    B unB = new B();
    ...
  }
}
```

Code de l'itérateur : les interfaces

L'interface Collection

```
public interface Collection<T> {
    int size();
    void add(T element);
    T getFirst();
    boolean isEmpty();
    Iterator<T> getIterator();
}
```

L'interface Iterator

```
public interface Iterator<T> {
    void first();
    boolean hasNext();
    T next();
}
```

Code de l'itérateur : LinkedList<T> (1)

```
public class LinkedList<T> implements Collection<T> {
  protected class NodeLL {
      protected T elt;
      protected NodeLL next;
      protected NodeLL(T elt,NodeLL next) {
          this.elt = elt;
          this.next = next;
      }
      protected T getElement() {return elt;}
      protected void setNext(NodeLL n) {next = n;}
  }

  protected NodeLL head,end;
  protected int nbElements;
  public LinkedList() {
      head = end = null;
      nbElements=0;
  }
  public int size() {
      return nbElements;
  }
```

Code de l'itérateur : LinkedList<T> (2)

```
    public void add(T element) {
        NodeLL tmp = new NodeLL(element,null);
        if(head==null) {
            head = end = tmp;
        } else {
            end.setNext(tmp);
        }
        nbElements++;
    }
    public T getFirst() {
        return head?head.getElement():null;
    }

    public boolean isEmpty() {
        return size()==0;
    }
    public Iterator<T> getIterator() {
        return new LinkedListIterator<T>(this);
    }
}
```

Code de l'itérateur : LinkedListIterator<T>

```
public class LinkedListIterator<T> implements Iterator<T> {
    protected LinkedList<T> l;
    protected LinkedList.NodeLL current;
    public LinkedListIterator(LinkedList<T> l) {
      this.l = l;
      current = l.head;
    }
    public void first() {
        current = l.head;
    }
    public boolean hasNext() {
        return current!=null;
    }
    public T next() {
        T tmp = (T)current.elt;
        current = current.next;
        return tmp;
    }
}
```

Code de l'itérateur : Utilisation de l'itérateur

Code

```
  public static void main(String[] args) {
    Collection<String> c = new LinkedList<String>();

    c.add("SE3");
    c.add("RX2");
    c.add("APA");
    c.add("CPA");
    c.add("BDA");

    Iterator<String> it = c.getIterator();

    while(it.hasNext())
        System.out.print(it.next());
}
```

Affichage à l'exécution

```
SE3 RX2 APA CPA BDA
```

Stratégie/Observateur/Itérateur : bilan

DP comportementaux

- Traitent des algorithmes et de l'affectation des responsabilités entre les objets
- Les modèles de classe utilisent l'héritage pour répartir les comportements entre les classes
- Les modèles d'objets utilisent la composition d'objets
- Un groupe d'objets se constitue pour accomplir en coopération une tâche
- Encapsulation de comportements à l'intérieur d'un objet et délégation de requêtes à ce dernier