POO-IG

Programmation Orientée Objet et Interfaces Graphiques

Cristina Sirangelo
IRIF, Université Paris Cité
cristina@irif.fr

Exemples et materiel empruntés :

* Core Java - C.Horstmann - Prentice Hall Ed.
* POO en Java - L.Nigro & C.Nigro - Pitagora Ed.

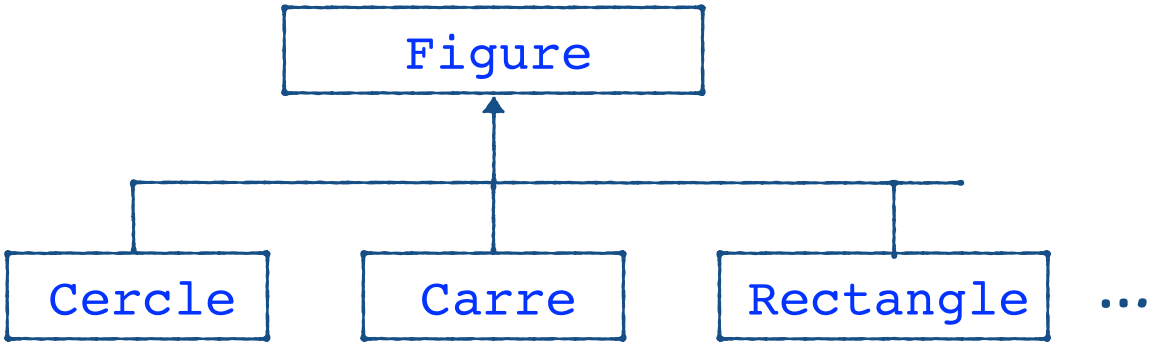
Classes abstraites, interfaces

Classes abstraites

- Classe abstraite : classe dont certaines méthodes n'ont pas définition (seulement la signature est fournie)
- Impossible de créer des objets d'une classe abstraite
- Sert comme base pour l'héritage
- Une méthode sans définition doit avoir le modificateur `abstract`
- Une classe qui contient des méthodes `abstract` doit avoir elle même le modificateur `abstract`

Classes abstraites : exemple 1

- But : définir une hiérarchie de classes pour les figures planes :



Classes abstraites : exemple 1

```
public abstract class Figure {

  //toutes le figures planes ont au moins une dimension
  //(coté d'un carré, rayon d'un cercle …)
   protected double dimension;
   public Figure (double dim) {
     if (dim <= 0) throw new IllegalArgumentException();
     dimension = dim;
    }

   //toutes les figures planes ont un périmètre et une
   //aire, mais la façon de la calculer dépend de la figure
   public abstract double perimetre();
   public abstract double aire();
   //ces méthodes peuvent être définies uniquement dans les
   //sous-classes

}
```

Classes abstraites : exemple 1

- code/poo/figures

Classes abstraites : exemple 2

```
public abstract class Benchmark{
    abstract void benchmark(); //un programme a tester
    public final long repeat(int c){ //le test
        long start = System.nanoTime();
        for(int i=0;i < c;i++)
            benchmark();
        return (System.nanoTime() - start);
    }
}
```

Classes abstraites : exemple 2 - suite

```
public class MonBenchmark extends Benchmark {
    void benchmark(){
        …
        //Mon super programme à tester
    }

    public static void main (String args[]) {
       System.out.println("temps="+
              new MonBenchmark().repeat(1000000));
    }
}

Résultat:
temps=2667238
```

Interfaces

- Une interface est une collection de signatures de méthodes
- (depuis Java 8 il est admis que certaines méthodes de l'interface aient une implémentation par défaut)
- Elle peut contenir également des champs constants et des classes internes, mais pas de variables d'instance
- Elle peut être vue comme une classe abstraite sans champs d'instance où toutes les (ou, depuis Java 8, la plupart des) méthodes sont abstraites

```
//exemple : interface présente dans java.lang
public interface Comparable {
 /**
 * @param x : objet à comparer
 * @return : < 0, ==0, >0 si l'objet est "inférieur",
   "égal" ou "supérieur" à obj */
 int compareTo( Object obj);
}
```

  - Mais il y a une différence fondamentale : une sous-classe peut hériter d'une seule autre classe, alors qu'elle peut implémenter plusieurs interfaces

Interfaces : utilité

- Une interface est une sorte d'"étiquette" qu'on peut attacher à une classe qui garantit que la classe implémente certaines méthodes
  - Une même classe peut avoir plusieurs de ces "etiquettes"
- Exemple :

```
public class Point implements Comparable {
 double x, y;
 …
 public int compareTo (Object o) {
    Point p = (Point) o;
      //soulève une exception si o n'est pas de
      // classe Point : objet incomparable
    return distance() - p.distance();
 }
}
```

- L'"étiquette" `Comparable` garantit que la classe définit `compareTo` et donc que ses objets peuvent être comparés

Interfaces : utilité

- N'importe quelle classe (même sous-classe) peut être étiquetée Comparable à condition qu'elle définit `compareTo`

```
public class CompteurModulo
             extends Compteur implements Comparable {
  public int compareTo (Object o) {
    CompteurModulo c = (CompteurModulo) o;
    return val - c.val;
  }
}
```

- Cela ne serait pas possible si Comparable était une classe abstraite

Interfaces : utilité

- On peut écrire du code générique qui travaille avec des objets comparables, sans savoir à quelle classe ils appartiennent

- Exemple : pour trier des objets on n'a pas besoin de savoir à quelle classe ils appartiennent, on requiert juste qu'il soient comparables !

```
public class Utilite {
 public static void selectionSort (Comparable [] v) {
    for (int j = v.length - 1; j > 0; j--) {
      //on met dans v[j] les max de v[0..j]
      int indMax = 0;
      for (int i = 1; i <= j; i++)
        {if (v[i].compareTo(v[indMax]) > 0) indMax = i;}
      Comparable tmp = v[j]; v[j] = v[indMax];
      v[indMax] = tmp; //échange v[j] <-> v[indMax]
    }
 } // selectionSort
 …
}
```

Interfaces : utilité

```
public class Utilite {
 public static void selectionSort (Comparable [] v) {…}

 public static void main (String[] args) {
    Point [] pv = new Point[3];
    pv[0] = new Point(2,2); pv[1] = new Point(1,1);
    pv[2] = new Point(0,0);
    CompteurModulo [] cm = new CompteurModulo[3];
    cm[0] = new CompteurModulo(2);
    cm[1] = new CompteurModulo(1);
    cm[2] = new CompteurModulo(0);
    // on peut utiliser selectionSort pour trier tout
    // tableau d'elements comparables
    selectionSort (pv);
    System.out.println(java.util.Arrays.toString(pv));
    selectionSort (cm);
    System.out.println(java.util.Arrays.toString(cm));
 }
}
```

Remarque sur l'interface Comparable

- En réalité Java fournit aussi une variante générique `Comparable<T>`
- Permet d'étiqueter une classe comme "comparable à une autre classe T"
- Il est donc mieux d'écrire :

```
public class Point implements Comparable<Point>
     …
     //un Point est comparable à des Points
     public int compareTo (Point p) {
       return distance() - p.distance();
     }
}
```

Remarque : Cloneable

- On a déjà vu que pour redéfinir la méthode `clone()` de `Object` il est nécessaire d'implémenter l'interface `Cloneable`

```
classe Point implements Cloneable, Comparable<Point> {
   …
   public Point clone() {…}
   public int compareTo( Point p ) {…}
   …
}
```

- `Cloneable` est une interface vide, elle sert uniquement à étiqueter une classe comme pouvant redéfinir `clone()`

- Remarquer la nécessité de pouvoir implémenter plusieurs interfaces

Interfaces et polymorphisme

- Une interface définit un type, tout comme une classe
- On peut déclarer des variables / paramètres d'un type interface,
- Mais on ne peut pas créer des objets de ce type

```
Comparable p; //variable
public void selectionSort (Comparable [] v) {…} //parametre
p = new Comparable(); //ERREUR
```

- En ce qui concerne le polymorphisme, implémenter une interface a le même effet que hériter d'une classe
- C'est-à-dire on peut utiliser un objet d'une classe qui implémente une interface I, partout ou on s'attend une variable de type I

```
p = new CompteurModulo(0);
v[0] = new CompteurModulo(2); v[1] = new CompteurModulo(3);
selectionSort(v);
```

un objet de classe CompteurModulo est aussi de type Comparable !

Interfaces : syntaxe

- Les méthodes d'une interface sont automatiquement publiques, pas besoin de spécifier `public`
  - en revanche les classes qui implémentent l'interface doivent utiliser le modificateur `public` pour les méthodes redéfinies de l'interface

```
Interface I {
 void f();
}

class C implements I {
   public void f(){…}
   …
}
```

- Les méthodes d'une interface peuvent être static (pas possible avant Java 8). Dans ce cas elles **doivent avoir une définition**.

Interfaces : méthodes private

- Depuis Java 9 il est possible d'introduire des méthodes `private` d'interface
  - sont visibles uniquement dans l'interface, donc peuvent être utilisées uniquement par les autres méthodes de l'interface qui ont une définition
    - — méthodes "helper"
  - **doivent être définies** (avoir un corps)
  - statiques ou pas

```
Interface I {
 private void g(){…}; //corps obligatoire
 void p();
}
```

Interfaces : méthodes par défaut

- Depuis Java 8 une interface peut fournir une définition par défaut des méthodes
- Les méthodes avec une définition doivent porter le modificateur `default` (sauf si `private` ou `static`)

```
public interface  Collection {
  int taille(); // méthode sans définition (abstraite)
  default boolean estVide() //méthode avec déf. par défaut
  { return taille() == 0; }
  ...
}
```

- Les classes qui implémentent cette interface héritent alors les définitions des méthodes par défaut, et peuvent ne pas les redéfinir
  - mais la redéfinition est toujours possible (overriding)

Interfaces : méthodes par défaut

- Une méthode par défaut peut bien sûr utiliser d'autres méthodes avec définition

```
public interface I {
  default void f() {…}
  private void g() {…}
  default void h() {…f();…g();…}
}
```

- Mais également des méthodes abstraites !

```
public interface  Collection {
   int taille(); // méthode sans définition (abstraite)
   default boolean estVide() //méthode avec déf. par défaut
   { return taille() == 0; }
   ...
}
```

Interfaces : champs

- Les champs d'une interface sont automatiquement des constantes statiques publiques : `public static final` implicite
  - une interface ne peut pas avoir des champs d'instance

```
public interface I {
 int i =1; //public static final implicite
 default int f(){ return i; }
}
```

- Ils sont hérités par les classes qui implémentent l'interface

```
class D implements I {
}
//D.i est un champ public static final de la classe D
//I.i fait référence au même champ
```

Interfaces : overriding

- Une classe qui implémente une interface doit en redéfinir toutes les méthodes abstraites (c-à-d sans corps)
  - Sinon la classe doit être déclarée abstraite

```
interface I {
 private void g(){…};
 default void h() {… g();…}
 void p();
}

class C implements I {
 public void h() {…} // redefinition non obligatoire
 public void p() {…} // redefinition obligatoire
}
```

Interfaces : overriding

- Une classe qui implémente une interface doit en redéfinir toutes les méthodes abstraites (c-à-d sans corps)
  - Sinon la classe doit être déclarée abstraite

```
interface I {
 private void g(){…};
 default void h() {… g();…}
 void p();
}
```

```
abstract class C implements I {//si p() n'est pas redéfinie
 public void h() {…}
}
```

Interfaces : syntaxe et exemple

```
//Rotatable.java
package poo.geometry;
public interface Rotatable {
 double FullAngle = 360; //constante statique
 void rotate(double angle);
 void reset();
}
//Compas.java
package poo.geometry;
public class Compas implements Rotatable {
 private Point center;
 private double rayon, angle;
 public Compas() { center = new Point(); reset();}
 public void rotate (double angle) {
    this.angle = (this.angle + angle) % FullAngle; //herité
 }
 public void reset () { center.deplace (0,0); angle = 0;
     rayon = 1;
 }
 //…
```

Interfaces : syntaxe et exemple

```
public double arc () {
   return 2 * Math.PI * rayon * angle / FullAngle;
}
public double getAngle() { return angle;}
public double getRayon() {return rayon;}
public void setRayon(double r) { rayon = r;}
public Point  getCenter () { return new Point(center);}
public void moveCenter (Point p) {
   center.deplace (p.getX(), p.getY());
}
public String toString() {
  return String.format("Compas \n:
    Angle : %1.2f, Rayon: %1.2f, centre: %s, arc: %1.2f",
                                angle, rayon, center, arc());
  }

} //Compas
```

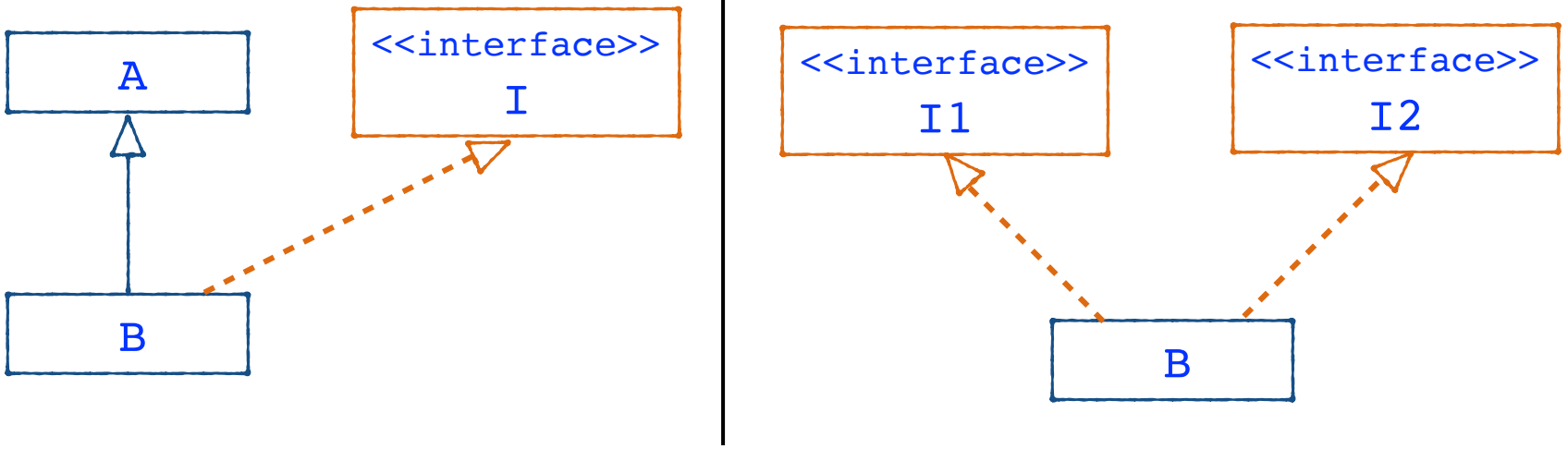
Quelques interfaces…

- `Cloneable`: est une interface vide (!) un objet qui l'implémente peut redéfinir la méthode clone()
- `Comparable`: est une interface qui permet de comparer les éléments (méthode `compareTo`)
- `Runnable`: permet de définir des "threads" (méthode `run`)

- `Serializable`: un objet d'une classe qui l'implémente peut être "sérialisé" = converti en une suite d'octets pour être sauvegardé

Héritage multiple

- Des conflits de définitions peuvent être générés par l'héritage multiple



- Qu'est-ce qui se passe si A et I, ou bien I1 et I2, définissent la même constante, ou la même méthode (i.e. même signature) ?

Héritage multiple : conflit de constantes

- Les constantes conflictuelles sont toutes héritées dans la sous-classe, il faudra enlever l'ambiguïté avec la notation

```
NomClasse.nomConstante
```

(notation habituelle pour le champs static)

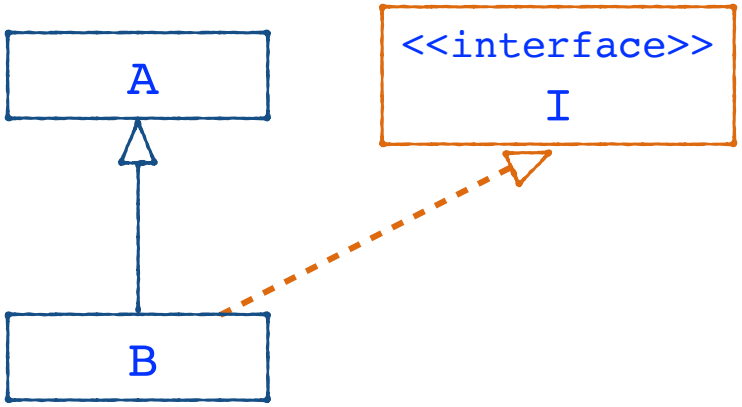
Héritage multiple : conflit de constantes

```
interface I1 {
    int i = 0;
}
interface I2 {
    int i = 1;
}
class A {
    public static final int i = 2;
}

public class B extends A implements I1, I2 {
     public int f() {
       return A.i; //ou I1.i ou I2.i
     }
}
```

Héritage multiple : conflit de méthodes

- Premier cas : conflit entre classe et interface



- Si A et I possèdent une méthode avec la même signature

  B hérite la méthode de la classe mère A, une éventuelle définition de la même signature dans I est ignorée

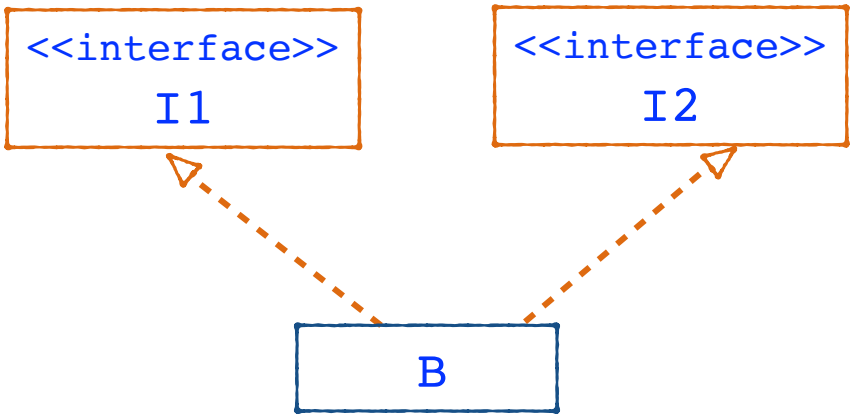
Héritage multiple : conflit de méthodes

```
interface I1 {
   default void f() {System.out.println("f de I1”);}
}
interface I2 {
   default void f() {System.out.println("f de I2”);}
}
class A {
   public void f() {System.out.println("f de A”);}
}

public class B extends A implements I1, I2 {
     public void g() {
       f(); // affiche “f de A”
     }
}
```

Héritage multiple : conflit de méthodes

- Deuxième cas : conflit entre interfaces



- Si deux méthodes ont la même signature en I1 et I2 et au moins une des deux a une définition, la classe B doit redéfinir la méthode (Sinon erreur de compilation)
  - Remarque : même si une des méthodes est abstraite

Héritage multiple : conflit de méthodes

```
interface I1 {
   default void f() {System.out.println("f de I1”);}
}
interface I2 {
   default void f() {System.out.println("f de I2”);}
   // ou void f();
}

public class B implements I1,I2 {
    //redéfinition obligatoire
    public void f() {System.out.println("f de B”)};
     public void g() {
        f(); // affiche “f de B”
     }
}
```

Héritage multiple : conflit de méthodes

```
interface I1 {
   default void f() {System.out.println("f de I1”);}
}
interface I2 {
   default void f() {System.out.println("f de I2”);}
   // ou void f();
}
```

- Dans la classe B on peut faire référence à l'une ou l'autre des deux fonctions conflictuelles, si concrète

```
public class B implements I1,I2 {
    //redéfinition obligatoire
    public void f() {I1.super.f();};
     public void g() {
        f(); // affiche “f de I1”
     }
}
```

Héritage multiple : conflit de méthodes

- Quand les deux méthodes conflictuelles sont abstraites, la redefinition n'est pas obligatoire (mais possible)

```
interface I1 {
    Object f();
}
interface I2 {
    Number f();
}

abstract class B implements I1, I2 {
    boolean g(){ return (f() == null); }
}
```

- En l'absence de redefinition : classe B abstraite
  - toute utilisation de f() dans B doit être compatible avec les deux versions héritées

Héritage multiple : conflit de méthodes

- Quand les deux méthodes conflictuelles sont abstraites, la redefinition n'est pas obligatoire (mais possible)

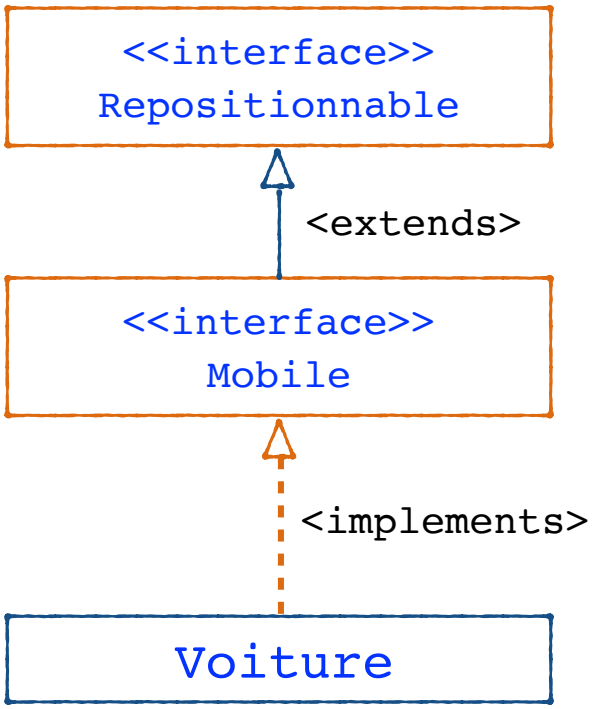
```
interface I1 {
    Object f();
}
interface I2 {
    Number f();
}

abstract class B implements I1, I2 {
    public Integer f(){return 0;}
    boolean g(){ return (f() == null); }
}
```

- Si f() est redéfini : type de retour et modificateur de f() compatibles avec les deux versions

Héritage d'interfaces

- Tout comme les classes, une interface peut étendre une autre interface



Héritage d'interfaces

```
interface Repositionnable {
 void deplace (double x, double y);
}

interface Mobile extends Repositionnable {
 double vitesse();
}

class Voiture implements Mobile {
 private double posX, posY;
 private double vitesse; …
 public void deplace (double x, double y)
 { posX = x; posY = y; }
 public double vitesse() {return vitesse;}
 public void setVitesse (double v) { vitesse = v; }
 …
}
```

Héritage d'interfaces

- Une interface peut étendre plusieurs interfaces.
- Exemple : une interface décrivant des objets qui peuvent être à la fois sérialisés et exécutés :

```
public interface SerializableRunnable
        extends Serializable, Runnable {
    …
}
```

- En cas de conflit de définition de constantes / méthodes : mêmes règles que pour l'implémentation de plusieurs interfaces

Interfaces et types abstraits de données

- Les interfaces peuvent être utilisées comme mécanisme de spécification d'un type abstrait de données (ADT) en Java
- Rappel : un ADT décrit un type de données par des fonctions et leurs propriétés (sémantique des fonctions)
- Exemple (rappel)

NOM

Pile<T>

FONCTIONS

vide : Pile<T> -> Boolean

nouvelle : -> Pile<T>

empiler : T × Pile<T> -> Pile<T>

dépiler : Pile<T> -> T × Pile<T>

PRECONDITIONS

dépiler(s: Pile<T>) <=> (not vide(s))

AXIOMES

pour tout e in T, pour tout s in Pile<T>

vide(nouvelle())

not vide(empiler(e,s))

dépiler(empiler(e,s))=(e,s)

Interfaces et types abstraits de données

- Le type abstrait Pile<T> comme une interface :

```
package poo.pile;
public interface Pile<T> {
        /**
        * teste si la pile est vide
        * @return  renvoie vrai si la pile ne contient aucun element,
       faux autrement
        */
        boolean estVide();
        /**
        * empile un element
        * ajoute un nouvel element en haut de la Pile
        * @param l'element à empiler
        * @return l'element lui même
        */
        T empiler (T item);
        /**
        * depile un element, la pile doit être non-vide
        * enlève et renvoie l'element en haut de la Pile
        * @return l'element depilé
        */
        T depiler ();
}
```

Interfaces et types abstraits de données

- Une classe qui implémente une pile par une liste chainée

```
package poo.pile;
import java.util.LinkedList;

public class PileLL<T> implements Pile<T>{
    private LinkedList<T> items;
    public PileLL(){
        items = new LinkedList<T>();
    }
    public boolean estVide(){
        return items.isEmpty();
    }
    public T empiler(T item){
        items.addFirst(item);
        return item;
    }
    public T depiler(){
        return items.removeFirst();
    }
}
```

Interfaces et types abstraits de données

- Une pile peut être implémentée de plusieurs façons, tout en respectant la même interface :

```
package poo.pile;
import java.util.*;
public class PileAL<T> implements Pile<T>{
    private ArrayList<T> items;
    public PileAL() { items =new ArrayList<T>(10);}
    public boolean estVide(){ return items.size()==0;}
    public T empiler(T item){
        items.add(item);
        return item;
    }
    public T depiler(){
        int top = items.size()-1;
        if (top < 0) throw new EmptyStackException();
        T item = items.get(top);
        items.remove(top);
        return item;
    }
}
```

Interfaces et types abstraits de données

- Remarque : À différence de la notion formelle de ADT une interface ne fournit pas un mécanisme pour spécifier la sémantique des fonctions

- On utilise les annotations de documentation à tel fin