Types interfaces

HAI8403I
Modélisation et Programmation par Objets

Université de Montpellier

Plan

Motivations

- Types plus abstraits que les classes
  - plus réutilisables
- Technique pour masquer l'implémentation
  - découplage public/privé : type/implémentation
- Favorise l'écriture de code plus général
  - écrit sur des types plus abstraits
- Relations de spécialisation multiple
  - entre les interfaces (avec `extends`)
  - entre les classes et les interfaces (avec `implements`)
- Meilleure organisation des types

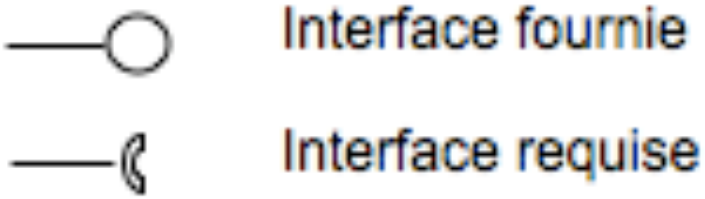
Plan

La notion d'interface dans le cadre de la modélisation

- zone de contact
- façade pour une implémentation (classe, programme)
- décrit par exemple pour une classe :
  - ce qu'elle peut offrir comme services à un programme
  - ce qu'elle requiert comme services de son environnement ou du programme
  - l'un des rôles qu'elle peut jouer

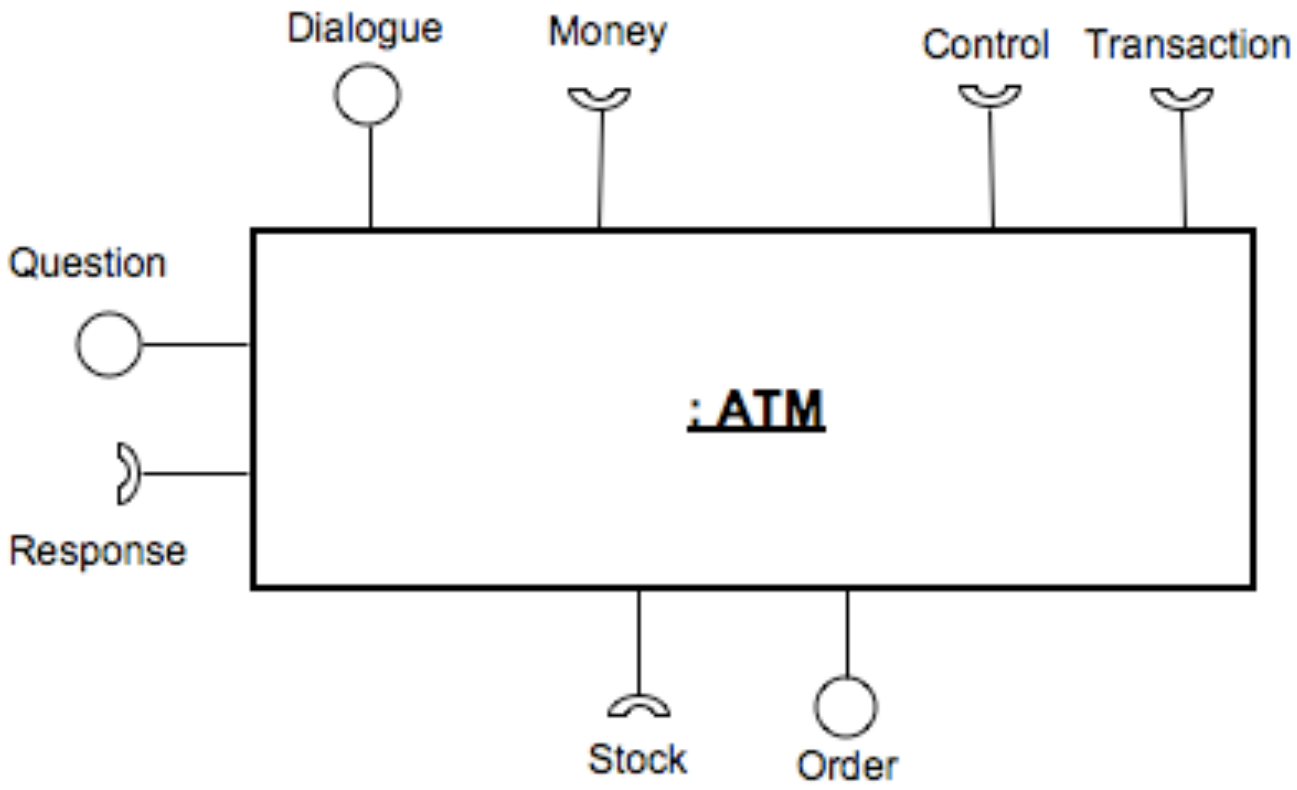
Interface en UML

- Ensemble nommé de propriétés publiques
- Constituant un service cohérent offert par une classe
- Ne spécifie pas la manière dont ce service est implémenté

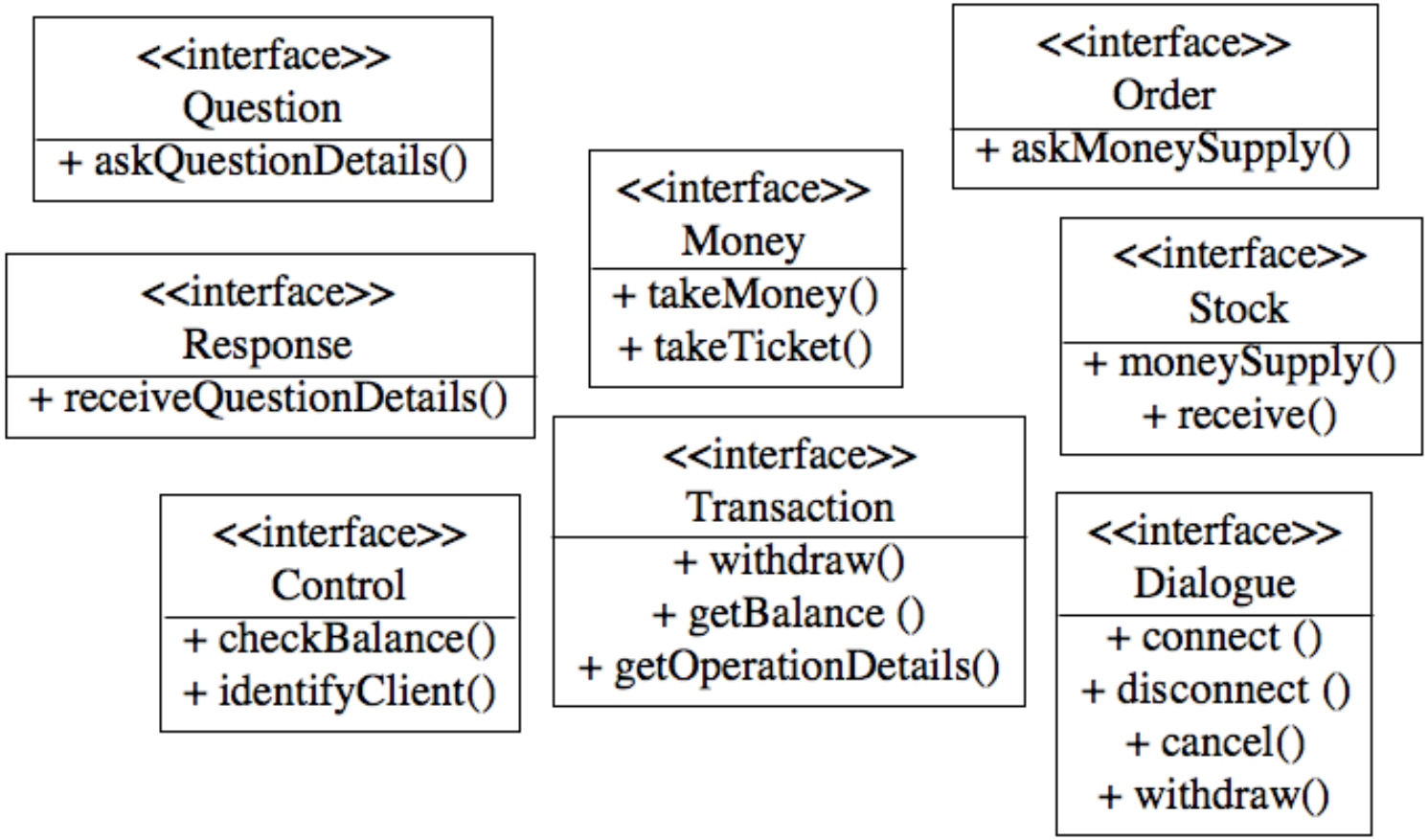


Un composant en UML

Brique logicielle de haut niveau décrite par des interfaces

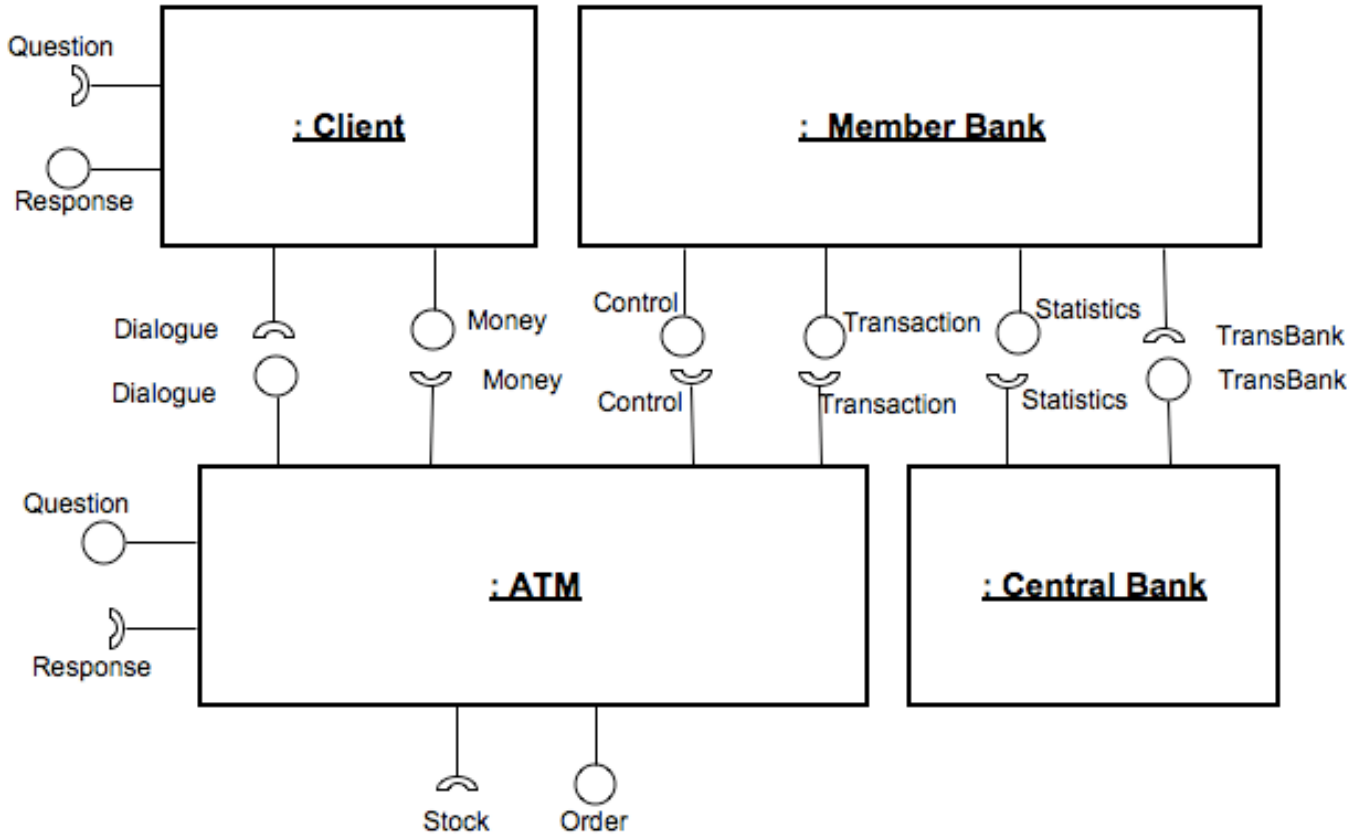


Interfaces en UML



Un assemblage de composants en UML

Architecture de haut niveau d'un logiciel



Plan

Interfaces en Java

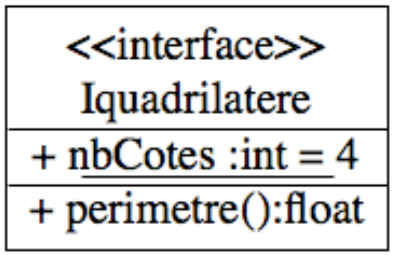
- Dans les versions <= Java 1.7
  - des méthodes d'instance publiques abstraites avec les modifieurs `public abstract`
  - des attributs de classe constants publics avec les modifieurs `public final static`
- À partir de Java 1.8
  - des méthodes d'instance publiques présentant des comportements par défaut (avec les modifieurs `public` et `default`)
  - des méthodes statiques (avec les modifieurs `public` et `static`)
  - des types internes
- Spécialisation entre une interface et une ou plusieurs interfaces par `extends`
- Implémentation entre une classe et une ou plusieurs interfaces par `implements`

Une interface quadrilatère

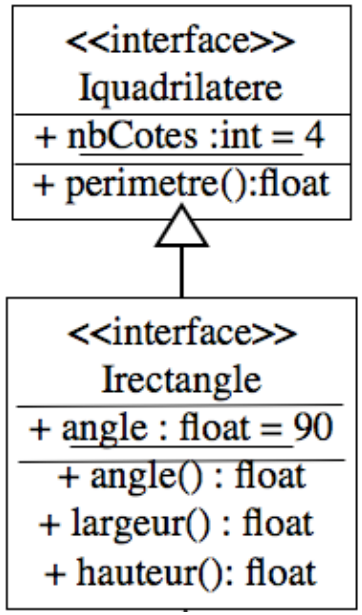
```
1 public interface Iquadrilatere {
2   public static final int nbCotes = 4;
3   abstract public float perimetre();
4 }
```

ou (en enlevant les mots-clefs obligatoires)

```
1 public interface Iquadrilatere {
2   int nbCotes = 4;
3   float perimetre();
4 }
```

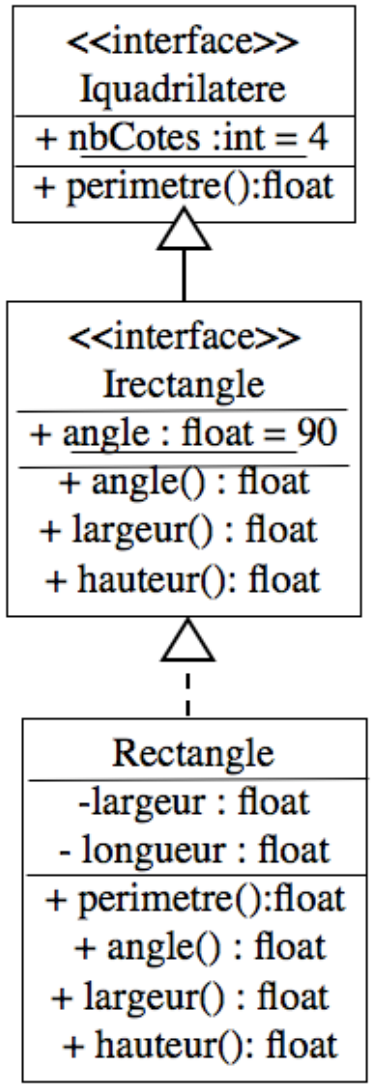


Spécialisation d'une interface




```
public interface Irectangle extends Iquadrilatere{
    float angle = 90;
    float angle();
    float largeur();
    float hauteur();
}
```

Interface et réalisation par une classe en UML



Interface et réalisation par une classe en Java / version 1

L'implémentation utilise deux attributs de type réel

```
public class Rectangle2floats implements Irectangle{

    private float largeur,hauteur;

    public Rectangle2floats(){}
    public Rectangle2floats(float l, float h)
            {largeur=l; hauteur=h;}

    public float perimetre()
            {return 2*largeur()+2*hauteur();}

    public float angle(){return Irectangle.angle;}
    public float largeur(){return largeur;}
    public float hauteur(){return hauteur;}
}
```

Instanciation

On instancie les classes mais pas les interfaces

```
public class MainRectangle {
    public static void main(String[] arg){

            // une classe est un type et elle peut ↩
                etre instanciee
            Rectangle r1 = new Rectangle();

            // une interface est un type MAIS elle ne↩
                 peut etre instanciee
            Irectangle r2 = new Rectangle();

            // mais on ne peut ecrire : new ↩
                Irectangle();

    }
}
```

Interface et réalisation par une classe en Java / version 2

L'implémentation uilise un tableau de deux réels

```
class RectangleTab implements Irectangle{

    private float tab[]=new float[2];

    public RectangleTab(){}
    public RectangleTab(float l, float h)
            {tab[0]=l; tab[1]=h;}

    public float perimetre()
            {return 2*largeur()+2*hauteur();}

    public float angle(){return Irectangle.angle;}
    public float largeur(){return tab[0];}
    public float hauteur(){return tab[1];}
}
```

Limite des interfaces avant Java 8

On ne pouvait pas factoriser de code commun dans une interface
L'usage était d'introduire une classe (abstraite) pour cette factorisation

```
public abstract  class RectangleAbs
                       implements Irectangle{
    public RectangleAbs(){}
    public float perimetre(){
        return 2*largeur()+2*hauteur();}
    public float angle(){return Irectangle.angle;}
}
```

Abstraite car :

- Elle ne propose pas de modèle d'implémentation en mémoire (pas d'attributs)
- Elle n'implémente pas `largeur` ni `hauteur`

Interface et réalisation par une classe en Java
Version 1 modifiée
pour qu'elle spécialise la classe RectangleAbs

```
public class Rectangle2floats extends RectangleAbs
{
    private float largeur,hauteur;
    public Rectangle2floats(){}
    public Rectangle2floats(float l, float h){
        largeur=l; hauteur=h;}

    public float largeur(){return largeur;}
    public float hauteur(){return hauteur;}
}
```

Interface et réalisation par une classe en Java
Version 2 modifiée
pour qu'elle spécialise la classe RectangleAbs

```
public class RectangleTab extends RectangleAbs
{
    private float tab[]=new float[2];
    public RectangleTab(){}
    public RectangleTab(float l, float h){
        tab[0]=l; tab[1]=h;}

    public float largeur(){return tab[0];}
    public float hauteur(){return tab[1];}
}
```

Les méthodes par défaut en Java 8

Elles permettent dans le cas des rectangles de factoriser les méthodes communes qui étaient dans RectangleAbs

```
public interface Irectangle extends Iquadrilatere
{
    float angle = 90; float angle();
    float largeur(); float hauteur();

    default float perimetre()
          {return 2*largeur()+2*hauteur();}

    default float angle(){return Irectangle.angle;}
....
}
```

Les méthodes par défaut en Java 8

Mais on ne pourra pas redéfinir les methodes d'Object comme toString

```
public interface Irectangle extends Iquadrilatere
{
    float angle = 90; float angle();
    float largeur(); float hauteur();

....

    default String description()
     // on ne peut pas la nommer "toString"
     {return "largeur␣="+this.largeur()
        +"␣hauteur␣="+this.hauteur();}
....
}
```

Les méthodes static dans les interfaces en Java 8

```
public interface Irectangle extends Iquadrilatere
{
    float angle = 90; float angle();
    float largeur(); float hauteur();

    .....

   static boolean egal(Irectangle r1, Irectangle r2){
     return r1.largeur()==r2.largeur()
              && r1.hauteur()==r2.hauteur();
   }

}
```

Plan

Description de comportements génériques

Noter : seules des interfaces et méthodes abstraites apparaissent

```
public class StockRectangle{

  ArrayList<Irectangle> listeRectangle
      = new ArrayList<Irectangle>();

  public void ajoute(Irectangle r){
      listeRectangle.add(r);}

  public float sommePerimetres(){
    float sp=0;
    for (int i=0; i<listeRectangle.size(); i++)
      {sp+=listeRectangle.get(i).perimetre();}
    return sp;
  }
}
```

Description de comportements génériques

Noter : on peut mettre des rectangles de toutes sortes dans le stock de rectangles

```
public class TestStockRectangle
{
   public static void main (String [] arg)
   {
          StockRectangle st = new StockRectangle();
          st.ajoute(new RectangleTab(3,8));
          st.ajoute(new Rectangle2floats(2,9));
          .....
   }
}
```

Plan

Spécialisation multiple

- Plus naturelle pour exprimer des relations de classification quelconques
- Permet une factorisation maximale dans tous les cas
- Spécialisation de plusieurs interfaces
- Permet de pallier partiellement l'absence d'héritage multiple en Java

Interfaces Carré et Losange

```
public interface Ilosange extends Iquadrilatere{
  float cote();
}
```

Un carré est à la fois un rectangle et un losange :

```
public interface Icarre extends Irectangle, Ilosange
{
}
```

Discussion

- Héritage simple en Java
  - Avec seulement des classes : les carrés ne pourraient pas être à la fois des rectangles et des losanges
  - Donc : une méthode admettant des losanges en paramètre ne pourrait admettre des carrés !
  - Toutes les propriétés ne pourraient pas être factorisées, certaines seraient redondantes, comme `cote()` ou `angle()`.
- La multi-spécialisation que nous avons pu faire sur les interfaces limite ces problèmes.

Interfaces en Java

Un exemple de spécialisation multiple entre interfaces et d'implémentation avec une classe

```
1  public interface IobjetColore {
2          Color couleurDefaut = Color.white;
3          Color getCouleur();
4  }
5  // specialisation multiple
6  public interface IrectangleColore extends IobjetColore,↩
       Irectangle{
7          void repeindre(Color c);
8  }
9
10 // implementation par une classe
11 public class RectangleColore implements IrectangleColore{
12   ... // ecriture de toutes les methodes des interfaces
13   ... // ou bien la classe est abstraite
14 }
```

Interfaces en Java

Un exemple d'implémentation multiple entre une classe et des interfaces

```
1  // dans l'API Java, une interface pour indiquer que les ↩
       objets de
2  // la classe peuvent etre clones
3  public interface Cloneable{
4  }
5
6  public class RectangleColore
7                              implements IrectangleColore, ↩
                                   Cloneable{
8    ... // ecriture de toutes les methodes des interfaces
9    ... // ou bien la classe est abstraite
10 }
```

Conflits et leur résolution technique en Java

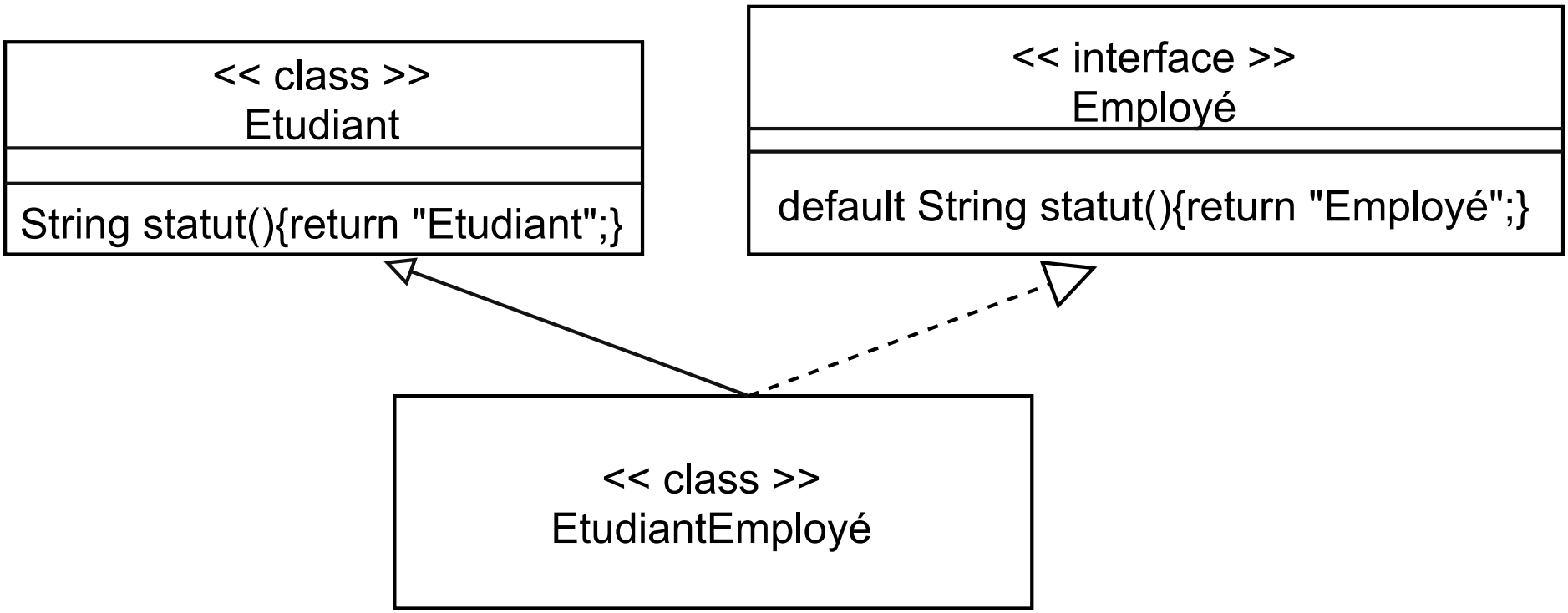
Depuis l'arrivée en Java des méthodes `default` dans les interfaces, des conflits peuvent se produire lorsqu'un type hérite de plusieurs méthodes avec la même signature. Principes de résolution :

- une méthode d'instance prend le dessus sur une méthode `default`
- une méthode déjà redéfinie est ignorée
- si un conflit persiste, on redéfinit la méthode concernée
- T.super.m() permet d'invoquer la méthode m d'un super-type direct T

Conflits et leur résolution technique en Java

Principe de résolution :

- une méthode d'instance prend le dessus sur une méthode `default`

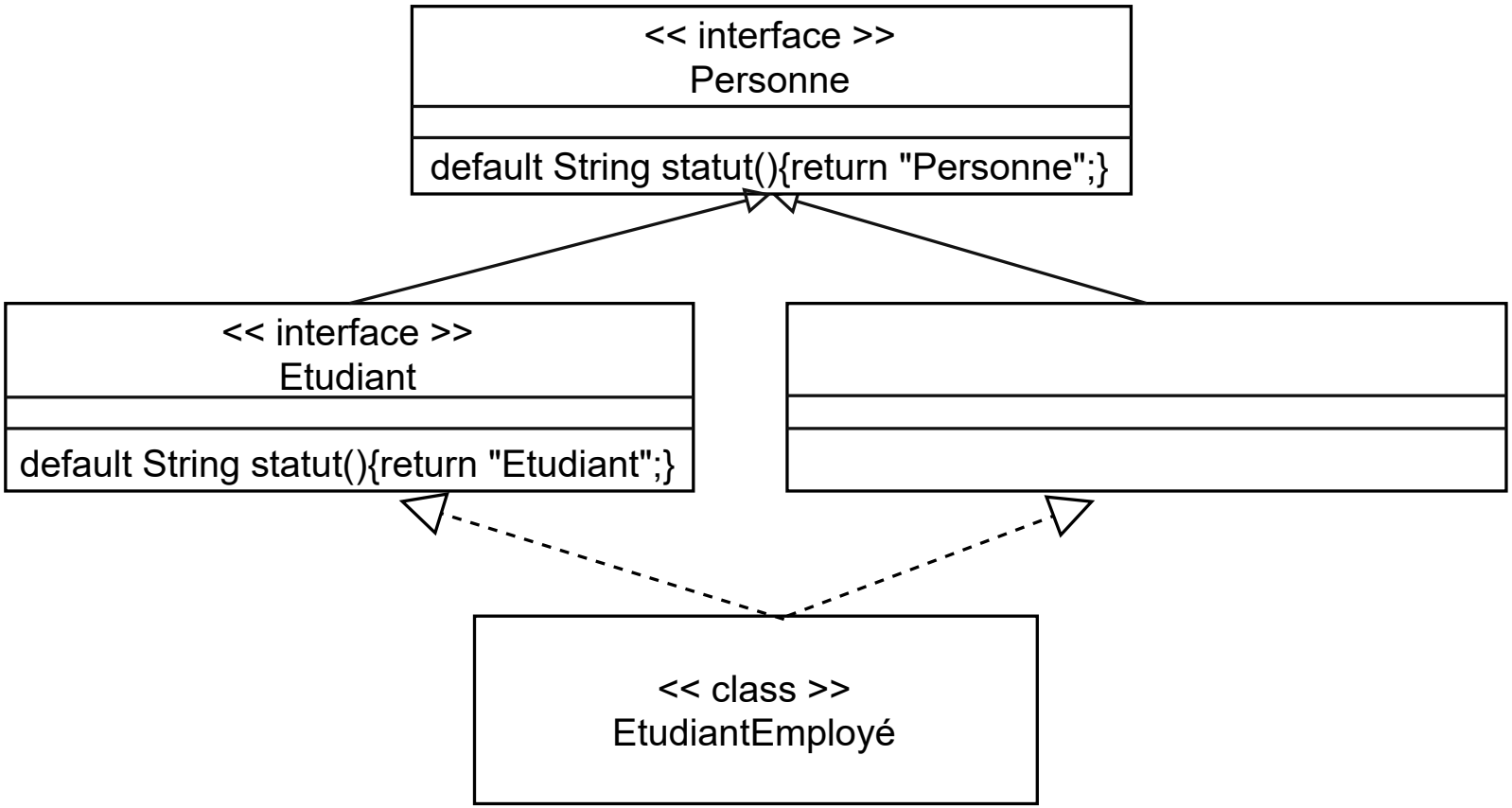



```
EtudiantEmployé e = new EtudiantEmployé();
System.out.println(e.statut()); // "Etudiant"
```

Conflits et leur résolution technique en Java

Principe de résolution :

- une méthode déjà redéfinie est ignorée

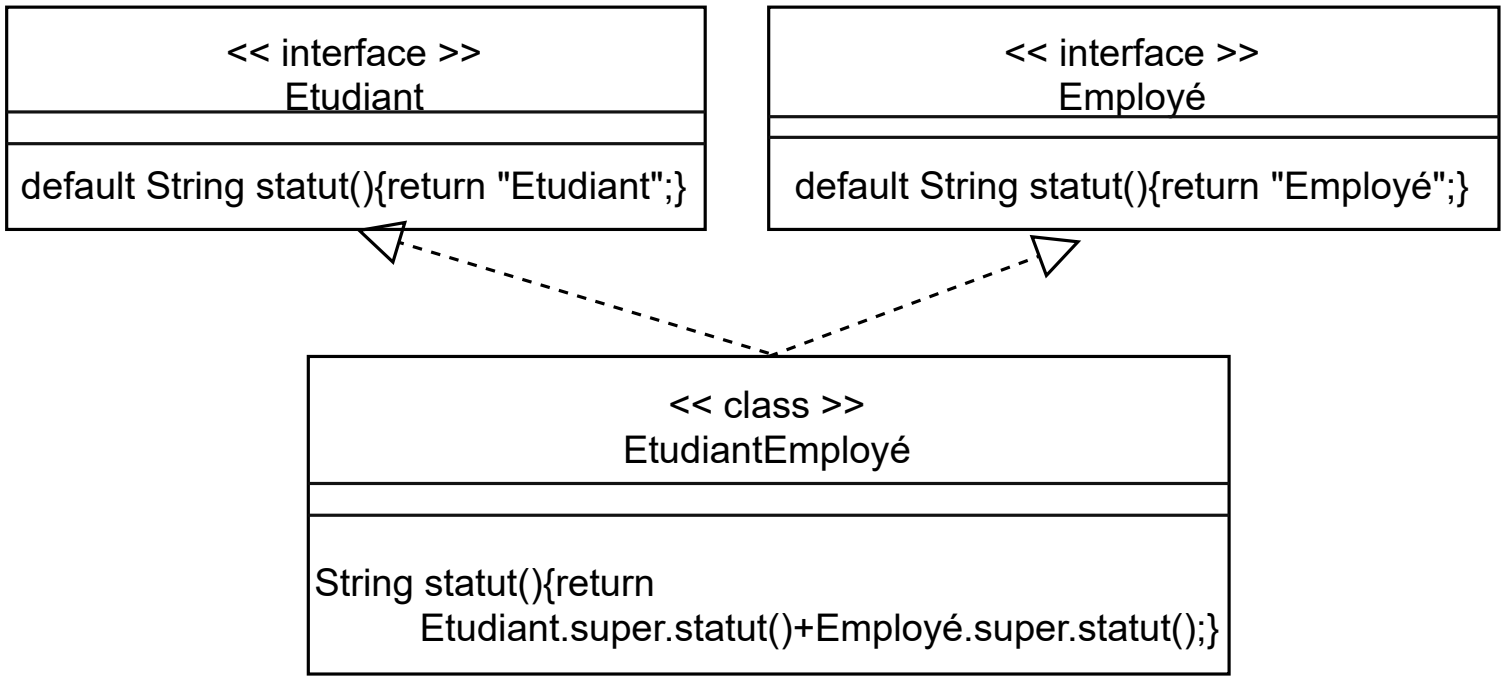



```
EtudiantEmployé e = new EtudiantEmployé();
System.out.println(e.statut()); // "Etudiant"
```

Conflits et leur résolution technique en Java

Principe de résolution :

- si un conflit persiste, on redéfinit la méthode concernée
- T.super.m() permet d'invoquer la méthode m d'un super-type direct T




```
EtudiantEmployé e = new EtudiantEmployé();
System.out.println(e.statut()); // "EtudiantEmployé"
```

Interfaces en Java : Résolution de conflits

Exemple : obligation de redéfinir `getNomCouleur` dans `CarteAJouer`

Forme de l'appel `NomInterface.super.nomMethodeEnConflit`

```
1  public interface IObjetColore {
2   Color getCouleur();
3   default String getNomCouleur() {return ""+getCouleur();}
4  }
5  interface ICarteAJouer{
6   enum Couleur{trefle, carreau, pique, coeur}
7   Object getCouleur();
8   default String getNomCouleur() {return ""+getCouleur();}
9  }
10 interface CarteAJouer extends IObjetColore, ICarteAJouer{
11    @Override
12    default String getNomCouleur() {
13      return "Couleurs :"+IObjetColore.super.getNomCouleur()
14             +" "+ICarteAJouer.super.getNomCouleur();
15     }
16 }
```

Plan

Interfaces marqueurs

- vides d'opérations et de constantes de classes
- précisent la sémantique et indiquent dans quel contexte leurs instances peuvent être utilisées
- implémenter ces interfaces marqueurs n'est pas toujours suffisant pour obtenir le comportement attendu, mais c'est nécessaire

Deux exemples :

- **Cloneable**
- **Serializable**

Interface Cloneable

Principe

- La classe `Object` dispose d'une méthode `protected Object clone()`
- Elle doit être redéfinie sous une forme publique si on veut l'utiliser
- Elle effectue une copie superficielle (shallow) : la méthode crée une nouvelle instance et initialise les attributs avec les valeurs des attributs de l'instance clonée. Ces valeurs ne sont pas elles-même clonées
- Pour effectuer une copie profonde (deep), il faut redéfinir la méthode `clone`

Interface Cloneable

```
1  public interface Iquadrilatere extends Cloneable{
2  ...
3      Object clone()throws CloneNotSupportedException;
4  }
5
6  public interface Irectangle extends Iquadrilatere{
7  }
8
9  public class Rectangle2floats implements Irectangle{
10 .....
11     @Override
12     public Rectangle2floats clone()
13                 throws CloneNotSupportedException {
14         return (Rectangle2floats) super.clone();
15     }
16 }
```

Interface Cloneable

```
1  public class StockRectangle implements Cloneable{
2  ....
3      @Override
4      public StockRectangle clone()
5                              throws CloneNotSupportedException {
6          // Plusieurs solutions :
7          // (1) copie superficielle (shallow)
8          // (2) intermediaire
9          // (3) profonde (deep)
10     }
11 }
```

Interface Cloneable - copie superficielle (shallow)

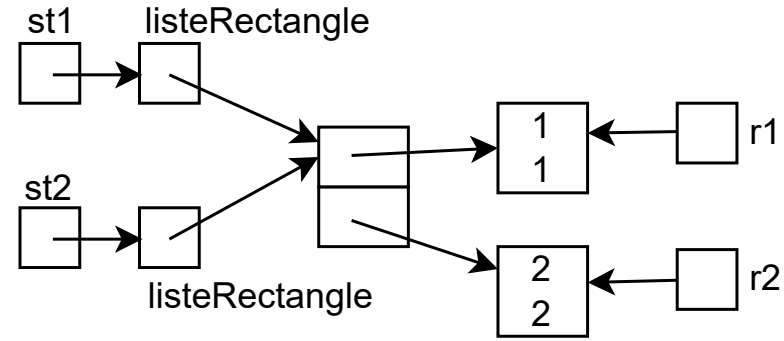
```
1 public class StockRectangle implements Cloneable{
2 ....
3     @Override
4     public StockRectangle clone()
5                     throws CloneNotSupportedException {
6         return (StockRectangle) super.clone();
7     }
8 }
```

Interface Cloneable - Exemple

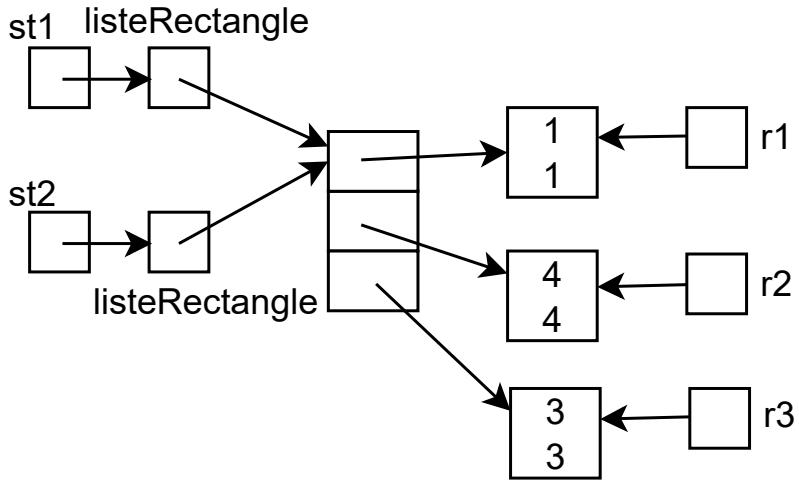
```
1  Rectangle2floats r1 = new Rectangle2floats(1,1);
2  Rectangle2floats r2 = new Rectangle2floats(2,2);
3  Rectangle2floats r3 = new Rectangle2floats(3,3);
4  StockRectangle st1 = new StockRectangle();
5  st1.ajoute(r1); st1.ajoute(r2);
6  StockRectangle st2 = st1.clone();
7  st2.ajoute(r3);
8  r2.setLargeur(4); r2.setHauteur(4);
```

Interface Cloneable - Copie superficielle

Après clonage de st1 en st2



Après ajout de r3 dans st2 et modification de r2 en (4,4)

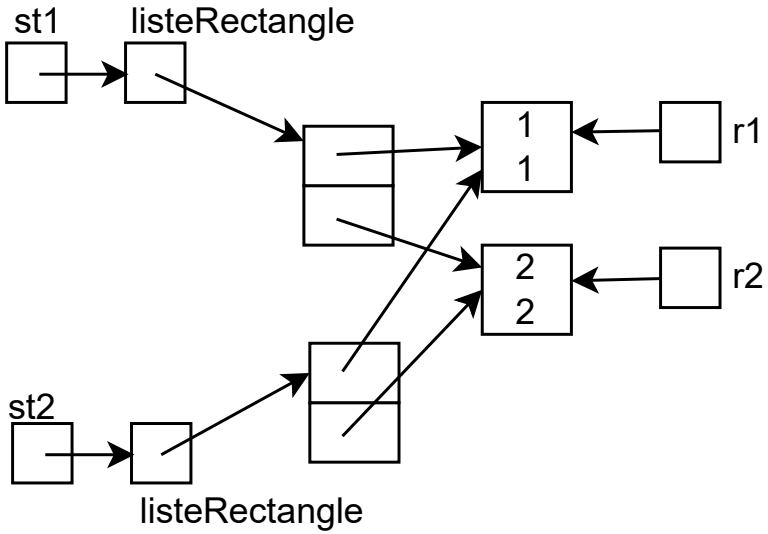


Interface Cloneable - copie intermédiaire

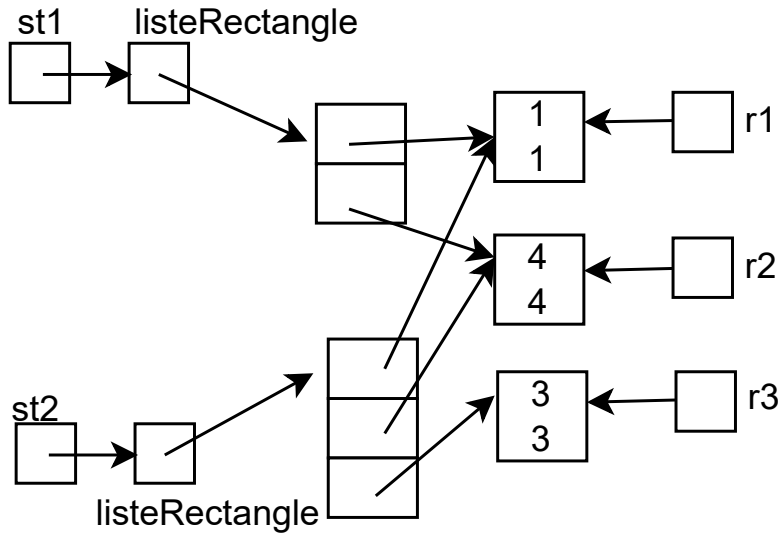
```
1  public class StockRectangle implements Cloneable{
2  ....
3      @Override
4      public StockRectangle clone()
5                      throws CloneNotSupportedException {
6        StockRectangle copy =  (StockRectangle) super.clone();
7        copy.listeRectangle = (ArrayList<Irectangle>)
8                                this.listeRectangle.clone();
9        return copy;
10     }
11   }
```

Interface Cloneable - Copie intermédiaire

Après clonage de st1 en st2



Après ajout de r3 dans st2 et modification de r2 en (4,4)

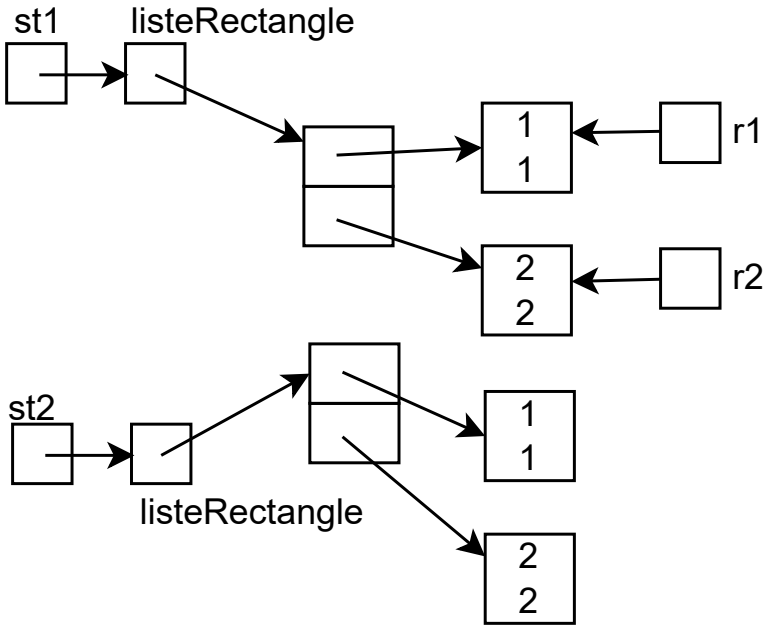


Interface Cloneable - Copie profonde (deep)

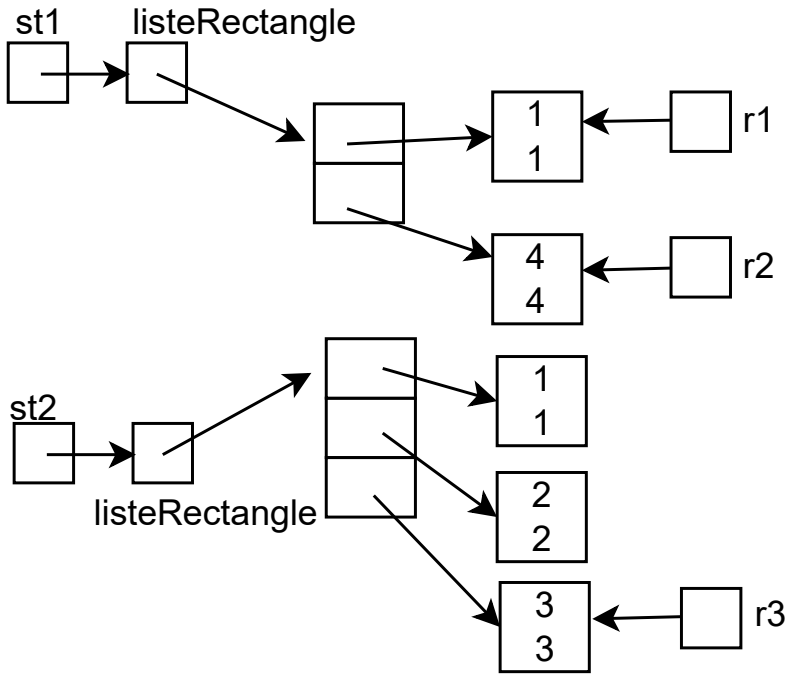
```
1  public class StockRectangle implements Cloneable{
2  ....
3      @Override
4      public StockRectangle clone()
5                      throws CloneNotSupportedException {
6        StockRectangle copy =  (StockRectangle) super.clone();
7        copy.listeRectangle = new ArrayList<>();
8        for (Irectangle r : this.listeRectangle)
9              copy.listeRectangle.add((Irectangle) r.clone());
10       return copy;
11     }
12 }
```

Interface Cloneable - Copie profonde

Après clonage de st1 en st2



Après ajout de r3 dans st2 et modification de r2 en (4,4)



Interface Serializable

Principe

- Les objets peuvent être « sérialisés » c'est-à-dire écrits (sauvegardés) dans un flux de données
- Pour cela leur classe doit implémenter `Serializable`
- `readObject` et `writeObject` sont réécrites si on désire une sérialisation particulière
- Les attributs doivent être primitifs ou sérialisables eux aussi.

Approfondissement laissé en exercice

Interfaces introduisant des méthodes

Quelques exemples

- L'interface `Comparable<T>`
- Les collections
- Les maps

Comparaison d'objets et tris

L'API définit une interface `Comparable` dont le code est le suivant.

```
1 public interface Comparable<T>
2 {
3         int compareTo(T o);
4 }
```

`compareTo` retourne -1, 0, ou 1 suivant si l'objet receveur est plus petit, égal ou plus grand que le paramètre. Cette opération est notamment utilisée pour les opérations de tri de la classe `Collections`

Comparaison d'objets et tris

On modifie l'interface `Iquadrilatere` afin qu'elle implémente `Comparable`

```
1 interface Iquadrilatere extends Comparable<Iquadrilatere>{
2   int nbCotes = 4;
3   float perimetre();
4   default int compareTo(Iquadrilatere autre) {
5     return (int)(this.perimetre()-autre.perimetre());
6   }
7 }
```

Comparaison d'objets et tris

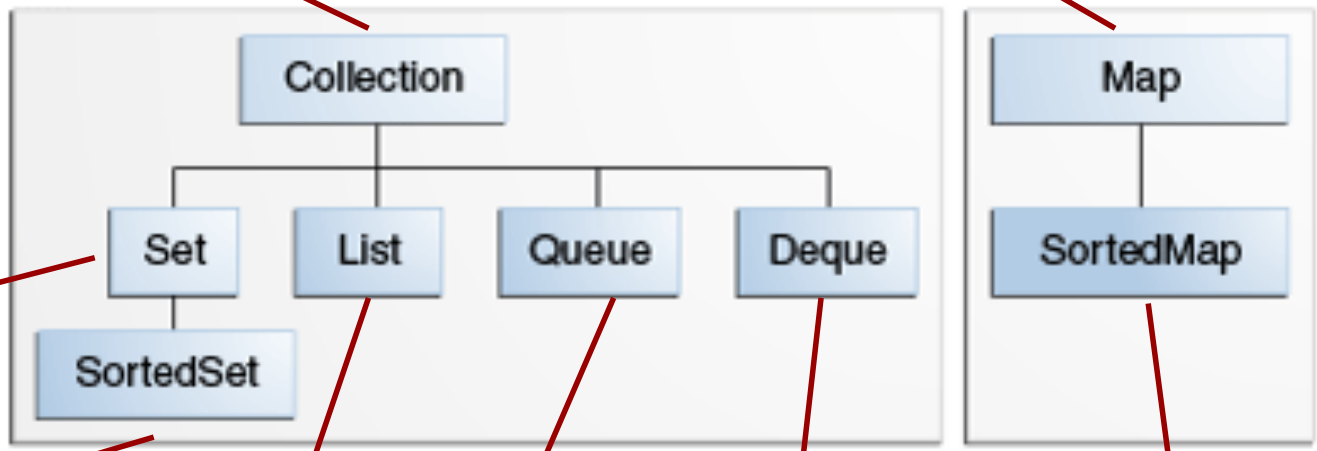
On peut alors trier les rectangles d'un stock d'après leur périmètre grâce à la méthode `sort` de l'API

```
1 class StockRectangle{
2   ArrayList<Irectangle> listeRectangle = new ArrayList<>();
3   ...
4    public void triStockSelonPerimetre() {
5       Collections.sort(this.listeRectangle);
6    }
7 }
```

Interfaces des collections et des maps

isEmpty, size
add, remove
contains, iterator

associe des clefs à des valeurs,
pas de doublon de clefs
put



sans doublons

garde ses éléments triés
grâce à un comparateur

organisation
séquentielle

File (FIFO)

Files (FIFO+FILO+LIFO)

garde ses clefs triées
grâce à un comparateur

Interfaces des collections et des maps

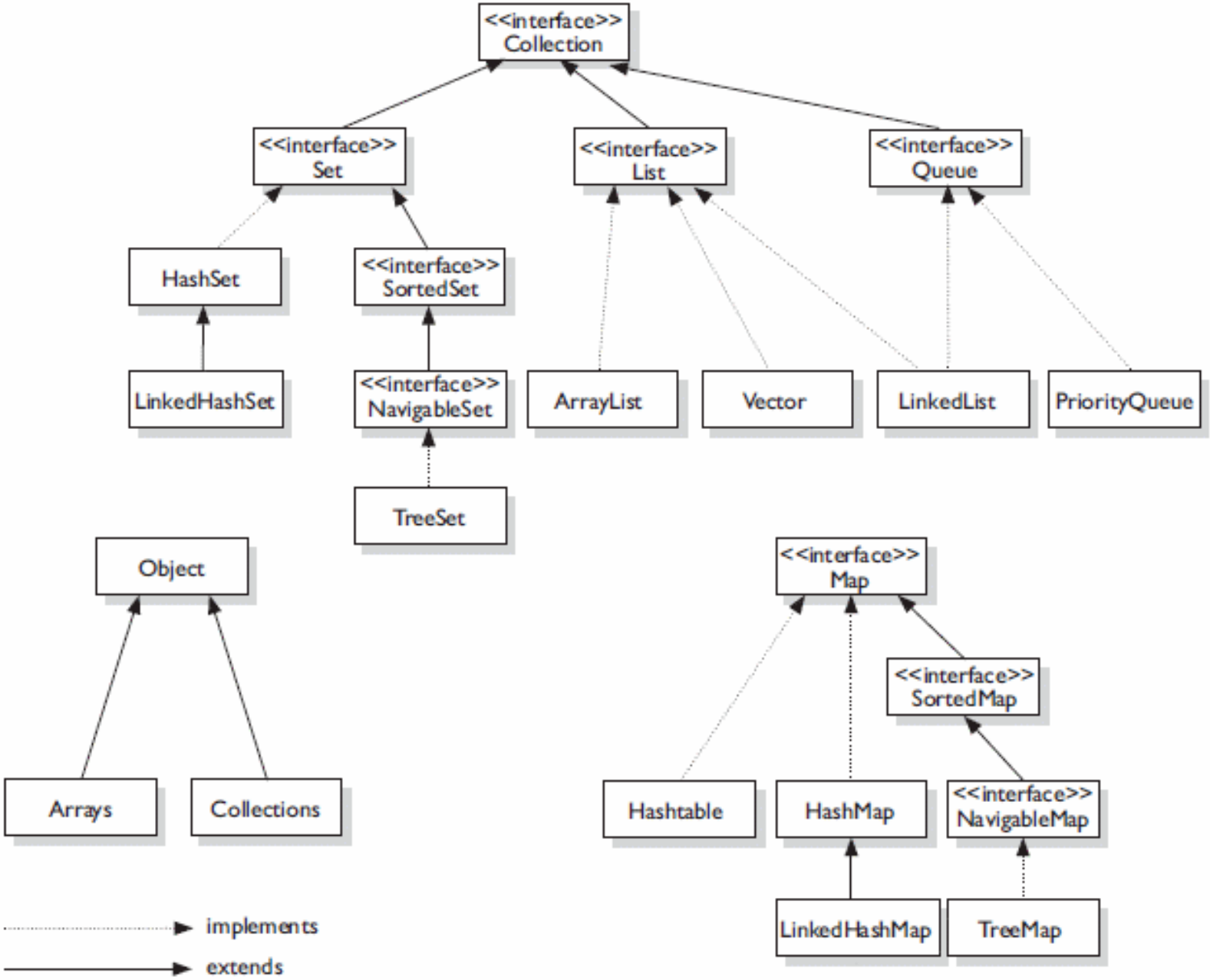
Implémentation par hashtable

Implémentation par arbres équilibrés

Implémentation par tableau

Implémentation par liste chaînée

Set	List	Queue	Deque	Map
HashSet				**HashMap**
	ArrayList	ArrayDeque	**ArrayDeque**	
TreeSet		PriorityQueue		TreeMap
	LinkedList	**LinkedList**	LinkedList	
LinkedHashSet				LinkedHashMap



https ://blog.jaaj.dev/2020/11/12/Framework-collections-java-intro.html

Plan

Synthèse

- Type interface
- Méthodes public et : abstraites, default, ou static
- Attributs public static final
- Spécialisation entre interfaces extends (multiple)
- Implémentation entre une classe et des interfaces implements (multiple)
- Résolution de conflits, notamment grâce à la forme `NomInterface.super.nomMethodeEnConflit`