

Programmation objet – Java

Classes abstraites & Interfaces

1. Classes abstraites
2. Interfaces
3. Implémentation d'une interface
4. Une interface définit un type
5. Étendre une interface
6. L'interface *Comparable*
7. Interface de marquage

1. Classes abstraites

Contrat

Une classe définit un contrat : l'ensemble des services qu'elle s'engage à rendre

Chaque méthode remplit une part du contrat :

- la signature (et la documentation) de la méthode décrit le service rendu. C'est un service abstrait qui répond à la question : quoi ?
- les instructions de la méthode établissent la façon dont ce service est rendu. C'est un service concret (implémentation) qui répond à la question : comment ?

Méthode abstraite

Une méthode abstraite ne possède pas d'instructions
(d'implémentation)

c'est une signature définissant le service que doit rendre la méthode, sans préciser la façon dont elle le rendra. Elle définit donc un service abstrait

Exemple :

// vérifie que le coup arrivant sur la case (lig, col) est valide
public abstract boolean coupOk(int lig, int col) ;

(une méthode **static** ne peut être abstraite)

Exemple

```
public abstract class Piece
{
    private int ligne ;
    private int colonne ;
```

```
    public Piece(int lig, int col )
    {
        this.ligne = lig ;
        this.colonne = col ;
    }
    ...
```

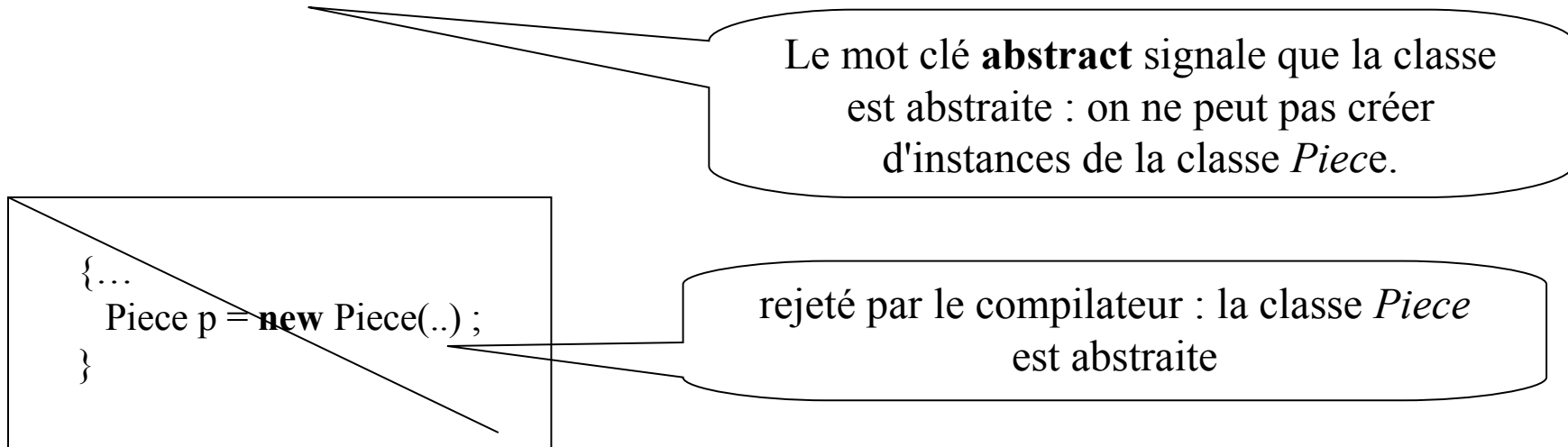
```
    public abstract boolean coupOk(int lig, int col) ;
```

```
} // fin classe Piece
```

Le mot clé **abstract** signale que la méthode *coupOk* est abstraite

Classe abstraite

Une classe est abstraite lorsqu'on lui interdit d'avoir des instances



Une classe contenant au moins une méthode abstraite est nécessairement abstraite.

Une classe ne contenant que des méthodes concrètes (avec un corps d'instructions) peut être abstraite.

Exemple

Le mot clé **abstract** signale que la classe est abstraite : on ne peut pas créer d'instances de la classe *Piece*.

```
public abstract class Piece
{
    private int ligne ;
    private int colonne ;

    public Piece(int lig, int col )
    {
        this.ligne = lig ;
        this.colonne = col ;
    }

    public abstract boolean coupOk(int lig, int col) ;
} // fin classe Piece
```

```
public int getLigne()
{
    return this.ligne ;
}

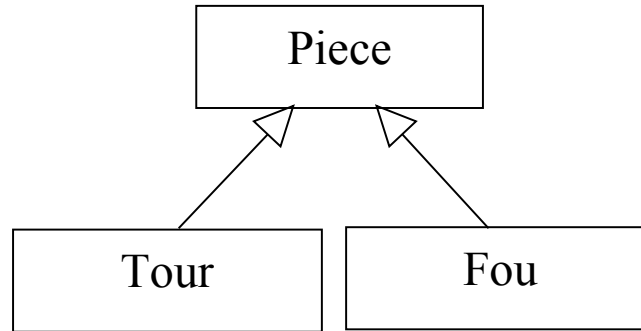
public int getColonne()
{
    return this.ligne ;
}
```

Utilisation d'une classe abstraite

Une classe abstraite sert de modèle pour créer des classes concrètes (non abstraites) qui en hériteront

Une classe abstraite définit une implémentation partielle (les méthodes concrètes ont des instructions, les méthodes abstraites n'en ont pas) que les classes qui en hériteront devront compléter

Exemple (1/2)



```
public class Tour extends Piece
{
```

```
    public Tour(int lig, int col)
    {
        super(lig, col);
    }
```

```
    public boolean coupOk(int lig, int col)
```

```
    {
        return (
            this.ligne == lig ||
            this.colonne == col
        );
    }
```

```
} //fin classe Tour
```

```
public class Fou extends Piece
{
```

```
    public Fou(int lig, int col)
    {
        super(lig, col);
    }
```

```
    public boolean coupOk(int lig, int col)
```

```
    {
        return (
            Math.abs(this.ligne - lig) ==
            Math.abs(this.colonne - col )
        );
    }
```

```
} //fin classe Fou
```

la méthode *coupOk* est
implémentée (possède
un corps
d'instructions).

Exemple (2/2)

```
public class TestPiece
{
    public static void main(String[] args)
    {
        Piece[] lesPieces = { new Tour(4,2), new Fou(5,6), ...}

        for (int i = 0 ; i < lesPieces.length ; i++)
        {
            int lig = Lire.jint("ligne ?") ;
            int col = Lire.jint("colonne ?") ;

            if (lesPieces[i].coupOk(lig, col))
                System.out.println("ok") ;
            else
                System.out.println("pas ok") ;

        } // fin for
    } // fin main
} // fin classe TestPiece
```

A la compilation :
le compilateur accepte car il constate qu'une méthode de signature *coupOk(int, int)* est définie dans *Piece* (le fait qu'elle soit abstraite ne change rien)

A l'exécution :
Polymorphisme, la classe de l'objet référencé détermine la méthode *coupOk* invoquée :
celle de la classe *Tour* pour *i = 0*
celle de la classe *Fou* pour *i = 1*

2. Interface

Motivation

Une classe concrète définit un contrat concret :

chaque méthode rend un service concret (implémenté par des instructions)

Une classe abstraite contenant des méthodes abstraites définit un contrat mixte (concret et abstrait) :

- chaque méthode concrète définit un service concret
- chaque méthode abstraite définit un service abstrait (sans implémentation)

Comment définir un contrat purement abstrait où rien ne doit prédéterminer la façon dont le contrat sera implémenté ?

Interface

Une interface définit un contrat purement abstrait,
elle ne contient que :

- des constantes de classe publiques
- des méthodes abstraites publiques

Une interface répond à la question : quoi ?

Une interface ne répond pas à la question : comment ?
(elle ne définit aucune implémentation)

Exemple

```
public interface Mediatheque  
{
```

le mot clé **interface** signale qu'on déclare une interface et non une classe

```
    int CAPACITE = 5000 ; // nombre maximal de médias
```

```
    // retourne le media de numéro num
```

implicitement :
public static final int CAPACITE = 5000 ;

```
    Media getMedia(int num) throws Exception ;
```

```
    // retire le media de numéro num de la Médiathèque courante et le retourne
```

```
    Media retirerMedia(int num) throws Exception ;
```

```
    // ajoute le Media m a la Mediatheque courante
```

```
    void ajouterMedia(Media m) throws Exception, MediaDejaPresent ;
```

```
} // fin interface Mediatheque
```

chaque méthode est implicitement **public abstract**

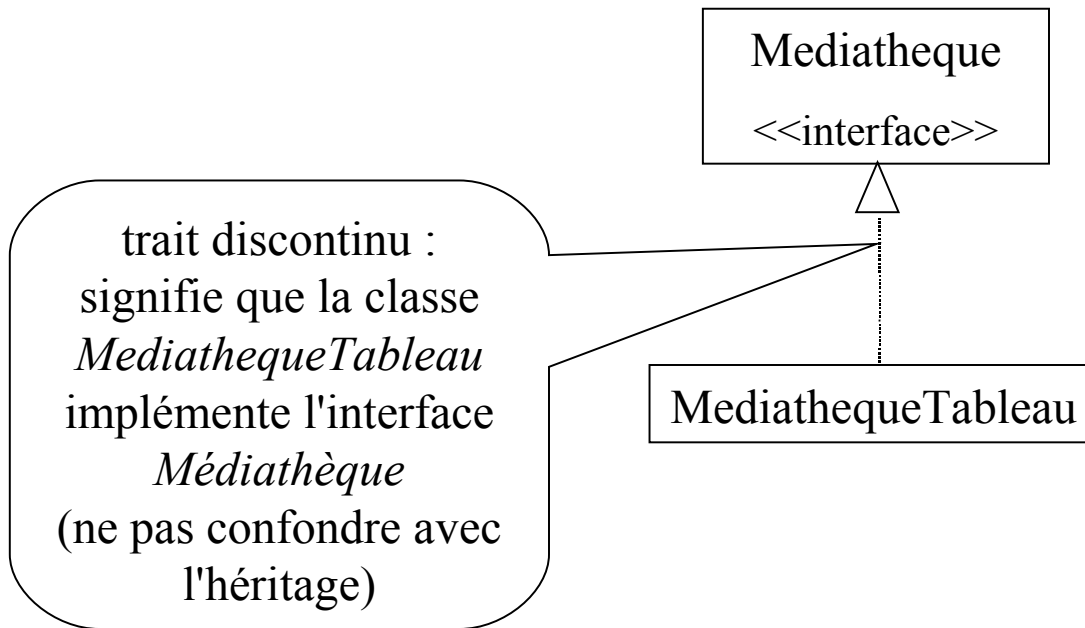
3. Implémentation d'une interface

Principes

Une interface est un modèle abstrait :

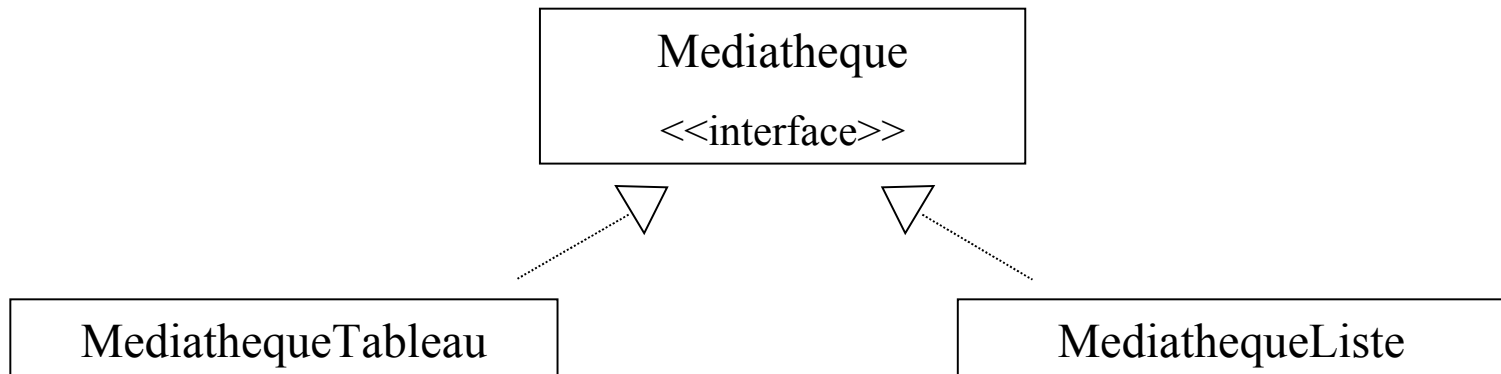
Le contrat abstrait qu'elle définit doit être implémenté par une classe pour être utilisé

Une classe implémente une interface lorsqu'elle donne une implémentation (un corps d'instructions) aux méthodes abstraites définies dans l'interface



Une interface peut avoir plusieurs implémentations

Chaque implémentation est une façon différente de réaliser le contrat abstrait défini par l'interface



Exemple (1/2)

```
public class MediathequeTableau implements Mediatheque
```

```
{ private String nomMediatheque ;  
  private Media[] medias          ; // - contient les médias  
  private int prochainIdentifiant ; // - prochain numéro de media a attribuer  
  private int prochainIndice      ; // - indice ou placer le prochain Media
```

variables d'instances décrivant
l'implémentation du point de vue
des données

```
public MediathequeTableau( ...) {this.medias = new Media[Methiateque.CAPACITE] ;...}
```

```
// retourne le media de numéro num
```

```
Media getMedia(int num) throws Exception {...// instructions }
```

```
// retire le media de numéro num de la Médiathèque courante et le retourne
```

```
Media retirerMedia(int num) throws Exception {...//instructions }
```

méthodes concrètes,
utilisant les données
et décrivant
l'implémentation du
point de vue
fonctionnel

```
void ajouterMedia(Media m) throws Exception, MediaDejaPresent {...//instructions }
```

chaque méthode abstraite de l'interface *Mediatheque* a été
implémentée dans la classe *MediathequeTab*

Exemple (2/2)

```
public classe MediathequeListe implements Mediatheque
```

```
{ private String nomMediatheque ;
```

```
  private ArrayList medias          ; // - contient les medias
```

```
  private int prochainIdentifiant    ; // - prochain numero de media a attribuer
```

les médias sont stockés dans une liste et non dans un tableau

```
public MediathequeListe( ... ) {this.medias = new ArrayList( ) }
```

```
// retourne le media de numéro num
```

```
Media getMedia(int num) throws Exception { ...// instructions }
```

```
// retire le media de numéro num de la Médiathèque courante et le retourne
```

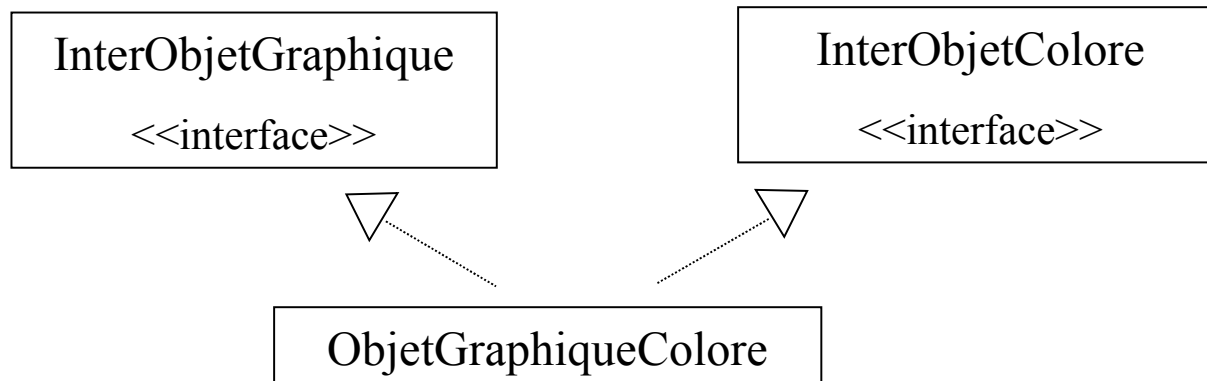
```
Media retirerMedia(int num) throws Exception { ...//instructions }
```

Les méthodes gèrent désormais une liste

```
void ajouterMedia(Media m) throws Exception, MediaDejaPresent { ...//instructions }
```

Une classe peut implémenter plusieurs interfaces

Elle doit donner un corps aux méthodes abstraites des interfaces qu'elle implémente



Exemple

// gestion d'une objet graphique

public interface InterObjetGraphique

{

void afficher() ;

void fermer() ;

} *// fin interface Fenetre*

// gestion d'un objet colore

public interface InterObjetCouleur

{

int ROUGE = 0 ; int VERT = 1 ; int JAUNE = 2 ;

void changerCouleur(int couleur) ;

} *// fin interface ObjetCouleur*

public class ObjetGraphiqueCouleur **implements** InterObjetGraphique, InterObjetCouleur

{

public ObjetGraphiqueCouleur {...}

void afficher {...// instructions }

void fermer {...//instructions }

void changerCouleur(int couleur) {...//instructions }

} *// fin classe FenetreCouleur*

Ici les méthodes abstraites des
2 interfaces ont été
implémentées

Mixer héritage et implémentation

Une classe ne peut hériter que d'une unique classe
mais peut implémenter plusieurs interfaces

```
public class classeX extends classeY implements interfaceZ, interfaceT
```

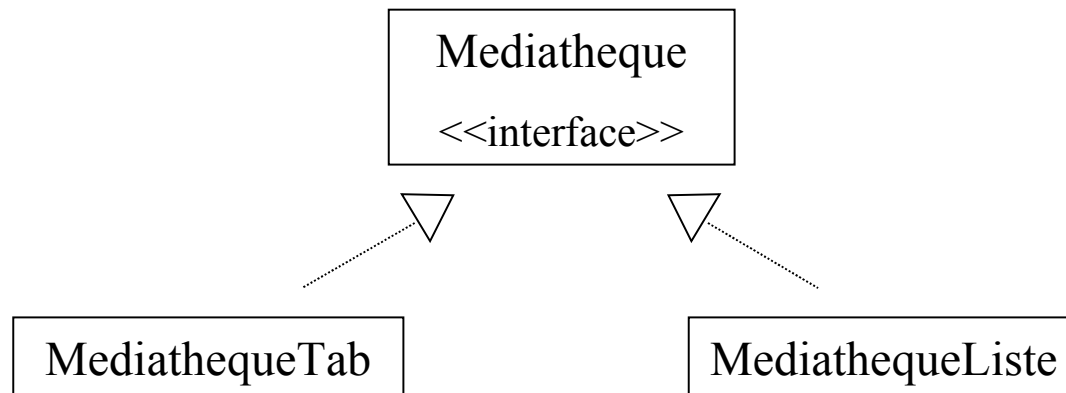
4. Une interface définit un type

Une interface définit un type abstrait

Une classe définit un type concret,

une interface définit un type abstrait (sans implémentation)

Le type d'une interface est plus général que le type de ses implémentations



On peut donc utiliser le type défini par une interface (ex : *Mediatheque*) à la place du type défini par une implémentation de cette interface (ex : *MediathequeTab*)

ex) Mediatheque mdt = new MediathequeTab();

Exemple d'utilisation du type de l'interface

```
public class Test
```

```
{  
    public static void afficheMediatheques(Mediatheque[] lesMediatheques)  
    {  
        for (int i = 0 ; i < lesMediatheques( ).length ; i++)  
        {  
            System.out.println(lesMediatheques[i]) ;  
        }  
    }  
}
```

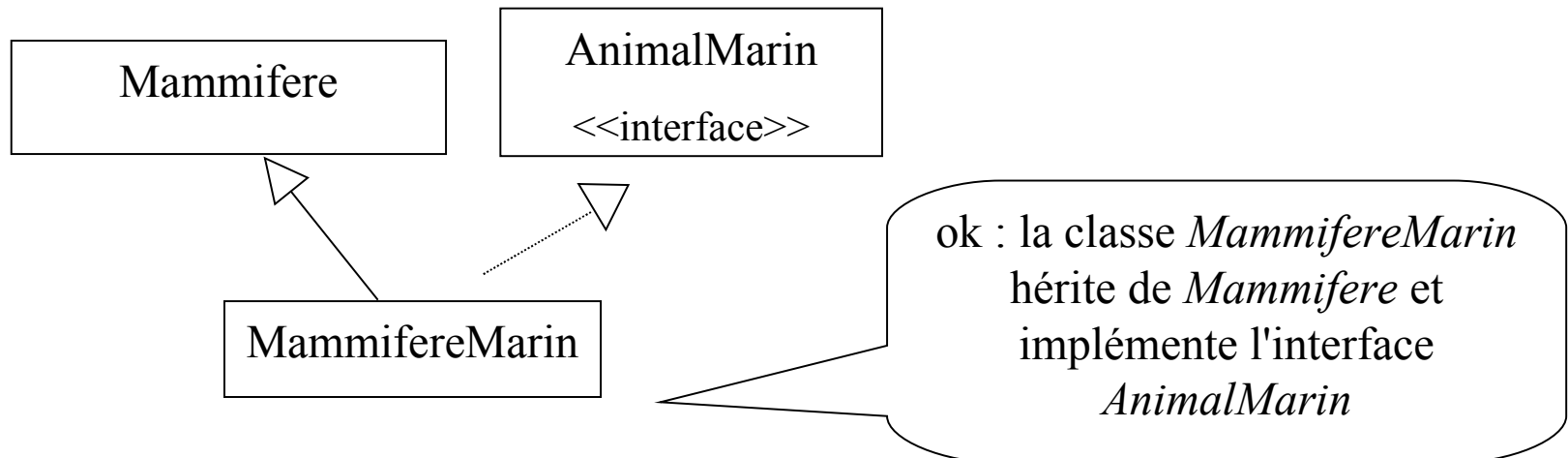
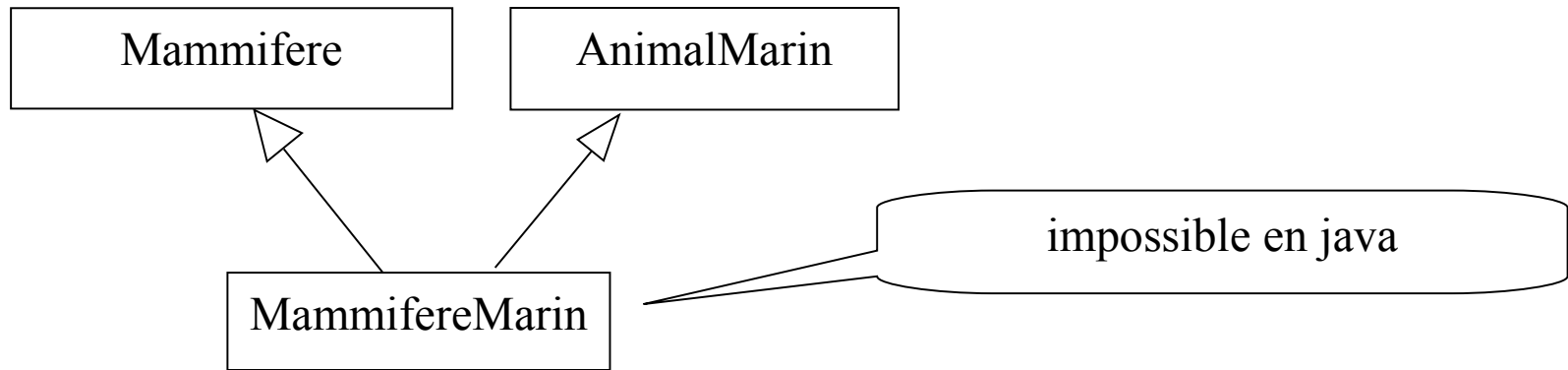
méthode indépendante des
implémentations de l'interface
Mediatheque utilisées

```
public static void main(String[] args)
```

```
{  
    Mediatheque[] lesMediathequesDeParis = new Mediatheque[50] ;  
  
    for (int i = 0 ; i < lesMediathequesDeParis.length ; i++)  
    {  
        if ((i %2) == 0) lesMediathequeDeParis[i] = new MediathequeTableau( ) ;  
        else  
            lesMediathequesDeParis[i] = new MediathequeListe( )  
        }  
    }  
    Test.afficheMediatheques(lesMediathequesDeParis) ;  
}
```

```
} // fin classe Test
```

Les interfaces permettent de palier (partiellement) l'absence d'héritage multiple en java



Exemple (1/2)

```
public class Mammifere
{
    int nbMamelles ;

    public int getNbMamelles( )
    { return this.nbMamelles ; }
}
```

```
public interface AnimalMarin
{
    public abstract void maFaconDeNager( ) ;
}
```

```
public class MammifereMarin extends Mammifere implements AnimalMarin
{
    int dureePlongee ;

    public MammifereMarin(float poids, int nbMamelles, int uneDuree)
    { super(poids, nbMam) ; this.dureePlongee = d ; }

    public void maFaconDeNager( )
    {
        System.out.println("je plonge et remonte apres " + this.dureePlongee + "minutes") ;
    }
}
```

Exemple (2/2)

```
public class Test
{
    public static void main(String[] args)
    {
        MammifereMarin unPhoque = new MammifereMarin(150, 6, 30) ;
        System.out.println(unPhoque.getNbMamelles( )) ;
        unPhoque.maFaconDeNager( ) ;
        TraiteMammiferes.traiteUnMammifere(unPhoque) ;
        TraiteAnimauxMarins.traiteUnAnimalMarin(unPhoque) ;
    }
}
```

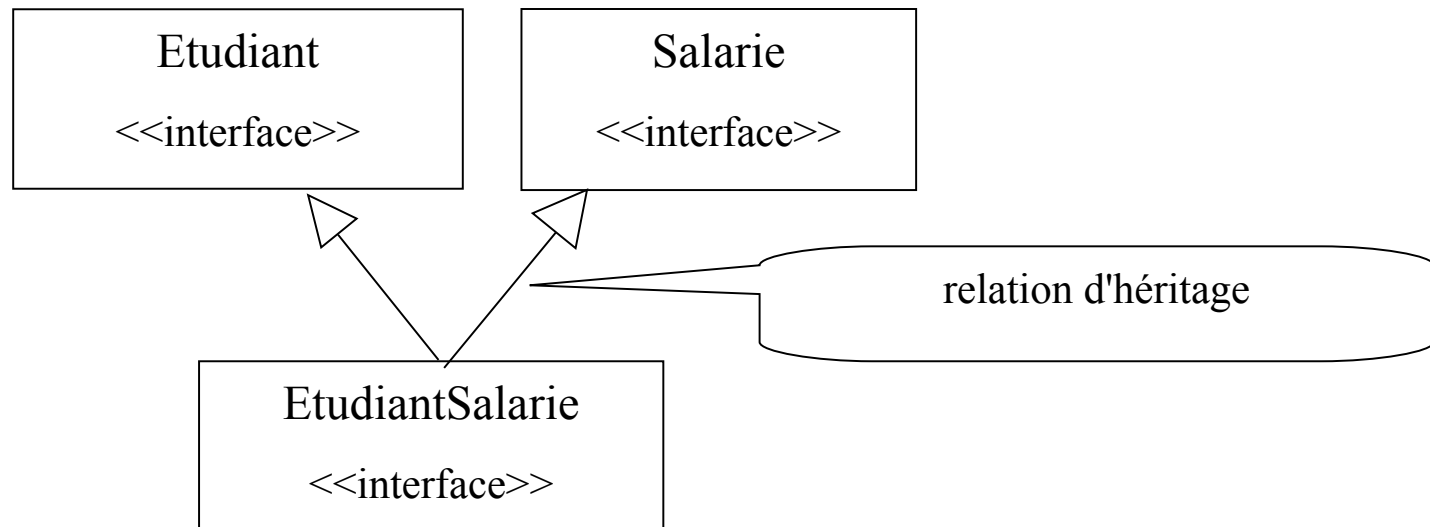
```
public class TraiteMammiferes
{
    public static void traiteUnMammifere
        (Mammifere m)
    {
        ...
    }
}
```

```
public class TraiteAnimauxMarins
{
    public static void traiteUnAnimalMarin
        (AnimalMarin a)
    {
        ...
    }
}
```

5. Étendre une interface

Héritage multiple entre interfaces

Une interface peut hériter d'une ou plusieurs autres interfaces
(elle hérite des constantes et méthodes abstraites des interfaces mères)



```
public interface EtudiantSalarie extends Etudiant, Salarie
```

```
{...
```

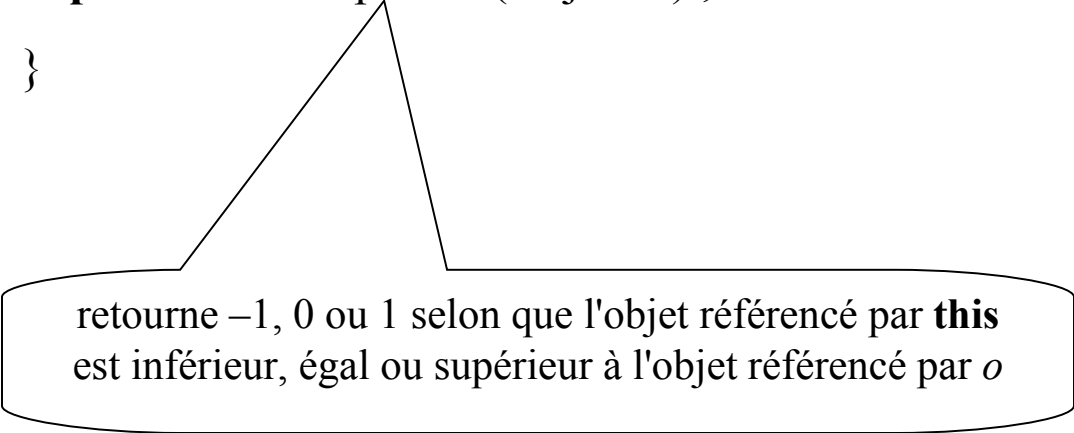
```
}
```

6. L'interface *Comparable*

L'interface *Comparable*

Contient une méthode abstraite définissant un ordre entre objets

```
public interface Comparable  
{  
    public int compareTo(Object o) ;  
}
```



retourne -1, 0 ou 1 selon que l'objet référencé par **this** est inférieur, égal ou supérieur à l'objet référencé par *o*

Toute classe implémentant l'interface *Comparable* définit un ordre total sur les objets qu'elle crée

Exemple (1/2)

```
public class Point implements Comparable
{
    private double abscisse ;

    // retourne -1, 0 ou 1 selon que
    // le Point courant est d'abscisse inférieure,
    // égale ou supérieure à l'abscisse du
    // Point référencé par o

    public int compareTo(Object o)
    {
        double x1 = this.abscisse ;
        double x2 = ((Point) o).abscisse ;
        if (x1 < x2) return -1 ;
        if (x1 == x2) return 0 ;
        return 1 ;
    }
} // fin classe Point
```

```
public class Rectangle implements Comparable
{
    private float longueur ;    private float largeur ;

    // retourne -1, 0 ou 1 selon que le Rectangle courant
    // a une surface inférieure, égale ou supérieure
    // au Rectangle référencé par o

    public int compareTo(Object o)
    {
        Rectangle r = (Rectangle) o ;
        s1 = this.longueur * this.largeur ;
        s2 = r.longueur * r.largeur ;
        if (s1 < s2) return -1 ;
        if (s1 == s2) return 0 ;
        return 1 ;
    }
} // fin classe Rectangle
```

une exception non contrôlée sera levée par la machine virtuelle si $o == \text{null}$ ou si l'instance référencée par o n'est pas une instance de *Point*

Example (2/2)

```
public class Comparer
{
    public static boolean plusGrand(Comparable c1, Comparable c2)
    {
        return (c1.compareTo(c2) == 1);
    }
} // fin classe Comparer
```

```
{
    Point p1 = new Point(15);
    Point p2 = new Point(10);
    If (Comparer.plusGrand(p1, p2))
        System.out.println(p1 + " est plus grand que " + p2);

    Rectangle r1 = new Rectangle(10, 15);
    Rectangle r2 = new Rectangle (5, 8);
    If (Comparer.plusGrand(r1, r2))
        System.out.println(r1 + " est plus grand que " + r2);
}
```

7. Interface de marquage

Interface de marquage

Certaines interfaces ne contiennent rien et sont utilisées pour désigner (marquer) les classes qui les implémentent à la machine virtuelle

ex) *Cloneable*

Une interface de marquage est assortie d'une documentation précisant les contraintes que toute classe qui l'implémente doit satisfaire.

ex) une classe *ClasseX* implémentant *Cloneable* autorise le clonage (copie) de ses instances avec la méthode *clone()* de la classe *Object*

```
{  
    ClasseX x = new ClasseX();  
    try  
    {  
        Object y = x.clone();  
    }  
    catch (CloneNotSupportedException e) { System.out.println(e + "clonage interdit"); }  
}
```

