

# **Cours M3105**

# **Conception et programmation objet avancées**

**Cours 7 - Héritage et polymorphisme**

# Langages à Objets

## ■ **Toute chose est un objet**

Les classes peuvent être vues comme des objets



Avec des méthodes et des attributs (ex : `String.class`)

Sont instances de la meta-classe :  
`java.lang.Class<T>`

# Langages à Objets

## API réflexive



```
List l = ArrayList.newInstance();
```

Les méthodes et les attributs aussi  
sont des objets

```
Method[] mths = "".getClass()  
                .getMethods();
```

```
Field[] atts = "".getClass()  
                .getDeclaredFields();
```

java.lang.reflect.Method

java.lang.reflect.Field

# Visibilité

## ■ Quatre niveaux

Public – accessible à toutes les classes



Protected – accessible aux classes du même paquetage et aux sous-classes



Default – accessible aux classes du même paquetage



Private – accessible seulement à la classe elle-même



# Visibilité

## ■ Mécanisme d'encapsulation

Pensez votre code comme une bibliothèque que d'autres utiliseront

Quelle est l'API ? (`public`)

Qu'est-ce qui doit rester caché ? (`protected`)

## ■ Attention à `private`

Une méthode `private` ne peut pas être appelée dans une sous-classe

Plus difficile de surcharger le comportement  
(patron *Template Method* !)

# Encapsulation

## ■ Principe de masquage d'information dans une classe

Tous les attributs

Certaines méthodes

# Encapsulation

## ■ Principe de masquage d'information dans une classe

Tous les attributs

Certaines méthodes

## ■ Attributs masqués, pourquoi ?

Modifications passent par les méthodes

Facilite l'intégrité des données

Modifier la structure interne de la classe sans impacter ses clients

# Encapsulation

## ■ Principe de masquage d'information dans une classe

Tous les attributs

Certaines méthodes

## ■ Méthodes masquées, pourquoi ?

Cacher les détails d'utilisation aux utilisateurs  
(code vue comme une bibliothèque)

Minimiser l'impact sur les utilisateurs des  
évolutions de la classe



# Visibilité

## ■ Attention au private

Complicque la redéfinition du comportement dans les sous-classes (*Template Method*)

Dans le doute, utiliser protected

## ■ Règles de base

Les attributs sont protected (sauf les constantes : « public static final »)



Les classes sont (généralement) public



Les méthodes sont public ou protected

# Héritage

## ■ Substituabilité

On peut remplacer une classe par une sous-classe



Toute méthode définie dans une classe devrait-  
être appellable sur les instances des sous-  
classes

# Héritage

## ■ Substituabilité

Les sous-classes peuvent surcharger les méthodes pour adapter le comportement

Appel à la méthode de la superclasse avec envoi de message a super



Une surcharge de méthode devrait normalement faire appel à super

# Héritage

## ■ Encapsulation

Principe de connaissance minimale

Loi de Déméter (*Law of Demeter* – LoD) 

« Ne parlez qu'à vos amis immédiats »

Règle de conception OO inventée à l'université de Boston vers 1987

Éviter de faire (et dépendre sur) des hypothèses sur la structure des autres classes que celle qu'on définit

# Héritage

## ■ Taxonomie

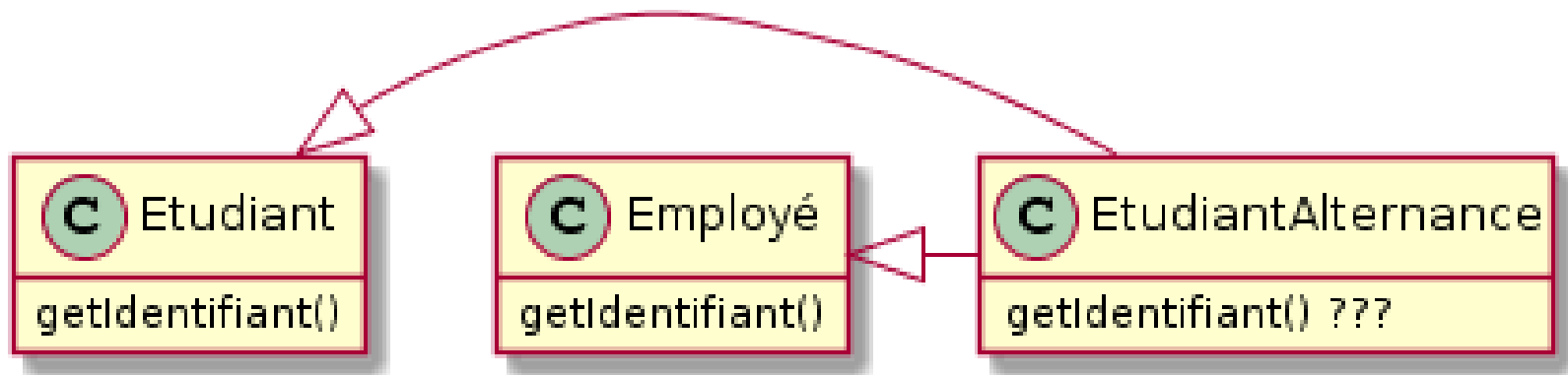
Représente l'arbre d'évolution des espèces  
→ Héritage simple (arborescence)

## ■ Conception orientée objet

Les objets ont plusieurs natures :  
enseignant intervenant extérieur ; étudiant en  
alternance  
→ Héritage multiple

# Héritage

## ■ Héritage multiple pose des problèmes



## ■ Java n'autorise que l'héritage simple

Mais que faire si une méthode exige en paramètre un Étudiant, un Employé ?

# Interface

## ■ Interface

Permettent de contourner l'héritage simple

Une classe peut honorer plusieurs contrats  
(implémente plusieurs interfaces)

Mettent l'accent sur l'aspect conceptuel  
(taxonomie) plutôt que de programmation  
(héritage)

# Interface

## Interface

Déclare un contrat (signatures de méthodes)



Toutes les méthodes que les classes devront implémenter

Notion d'API (*Application Programming Interface*) : ce qui peut-être utilisé

Ses méthodes sont `public` ou `default`

Exemple : Iterator

```
boolean hasNext();  
Object next();  
void remove();
```



# Interface

## ■ On parle aussi d'interface pour désigner :

Interface d'une classe (signatures de ses méthodes `public/default`)

Interface d'un paquetage (interfaces de ses classes `public`)

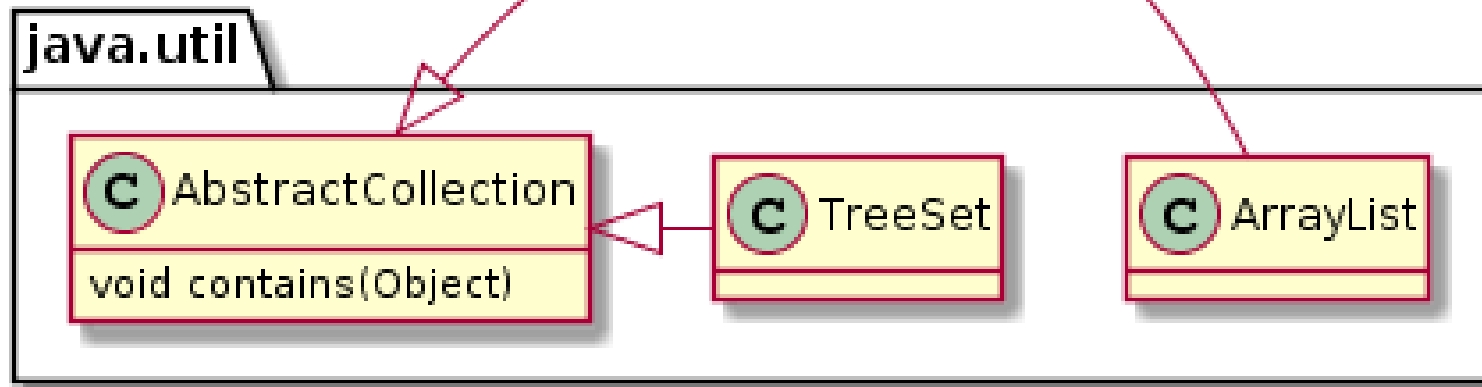
# Polymorphisme

## ■ Conséquence directe de l'héritage

L'émetteur du message ne se soucie pas de la nature réelle du récepteur



Le message s'exécute selon la classe de l'objet récepteur



\* Diagrammes : [plantuml.com](http://plantuml.com)

# Polymorphisme

## ■ Polymorphisme

Ad hoc – mêmes noms de méthodes dans des classes différentes

Paramétrique – paramètres différents en nombre et type

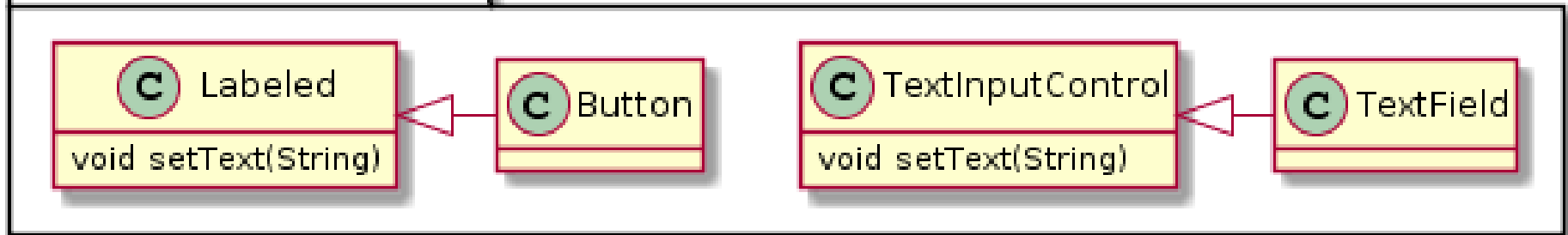
D'héritage – comportement spécialisé

# Polymorphisme

## ■ Polymorphisme

**Ad hoc** – mêmes noms de méthodes dans des classes différentes

`javafx.scene.control`



Paramétrique – paramètres différents en nombre et type

D'héritage – comportement spécialisé

# Polymorphisme

## ■ Polymorphisme

Ad hoc – mêmes noms de méthodes dans des classes différentes

**Paramétrique** – paramètres différents en nombre et type

```
java.io.FilterInputStream  
    int read()  
    int read(byte[])  
    int read(byte[],int,int)
```

D'héritage – comportement spécialisé

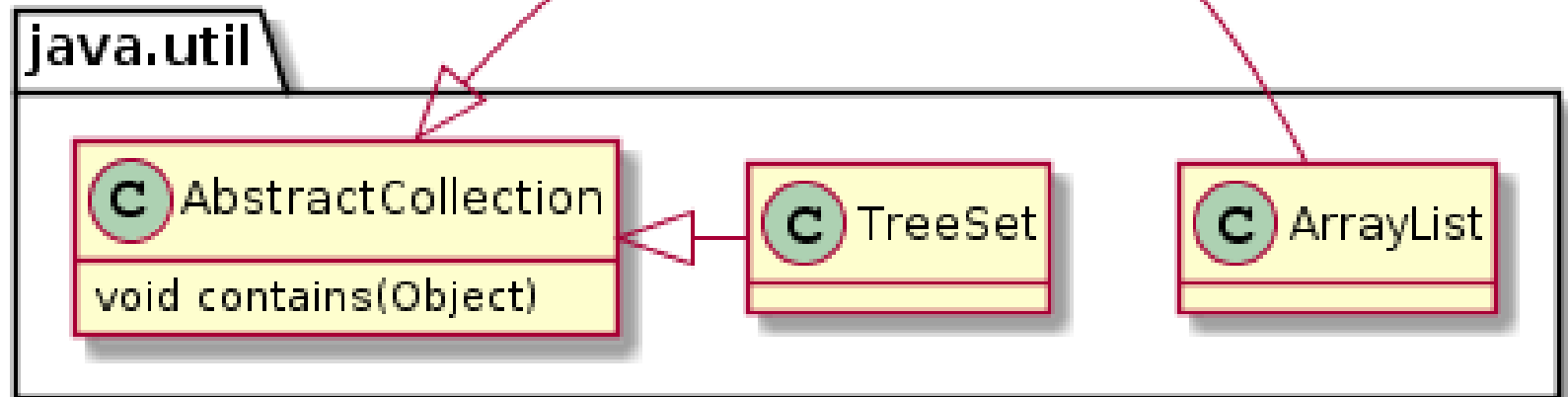
# Polymorphisme

## ■ Polymorphisme

Ad hoc – mêmes noms de méthodes dans des classes différentes

Paramétrique – paramètres différents en nombre et type

D'héritage – comportement spécialisé



# Polymorphisme

## instanceof

Tous les langages à objets offrent la possibilité de vérifier le type réel d'un objet

instanceof en Java

```
AbstractCollection collection;  
...  
if (collection instanceof TreeSet)  
then ((TreeSet)collection).last();
```

# Polymorphisme

## instanceof

L'utilisation de `instanceof` est considérée un *code smell* en POO



On ne devrait pas avoir à connaître le type réel d'un objet



C'est le polymorphisme (et le *late binding*) qui s'occupe(nt) de tout



# Polymorphisme

## instanceof

```
if (figure instanceof Carre)
then figure.drawPolygone();
else if (figure instanceof Cercle)
then figure.drawCurve();
else ...
```

# Polymorphisme

## instanceof

Création d'une méthode `draw()` polymorphique dans toutes les classes

Éventuelle création d'une super-classe abstraite

```
public abstract class Figure {  
    public abstract void draw();  
}  
  
...  
Figure figure;  
figure.draw();
```

# Polymorphisme

## ■ Tests ou polymorphisme

On peut aussi remplacer certains tests sans instanceof par du polymorphisme

```
public double toKelvin(double value) {  
    if (echelle == CELSIUS)  
        then return value+273.15;  
    else if (echelle == NEWTON)  
        then return value/0.33+273.15;  
    else ...  
}
```

# Polymorphisme

## ■ Tests ou polymorphisme

Création de classes Celsius, Fahrenheit, Newton ...

Méthode polymorphique

double toKelvin(double value)

```
public abstract class Echelle {  
    public abstract  
        double toKelvin(double value);  
    public abstract  
        double fromKelvin(double value);  
}
```

# Polymorphisme

## ■ Tests ou polymorphisme

Création de classes Celsius, Fahrenheit, Newton ...

Méthode polymorphique

double toKelvin(double value)

```
public class Celsius extends Echelle {  
    @Override  
    public double toKelvin(double value) {  
        return value+273.15;  
    }  
}
```

# Polymorphisme

## ■ Tests ou polymorphisme

On peut utiliser la méthode `toKelvin()` en ayant un objet de la bonne classe (selon l'échelle)

```
new Celsius().toKelvin(38.2);
```

# *Double dispatch*

## ■ **Problème 1**

Différentes classes organisées dans un arbre d'héritage

- Figures géométriques (quadrilatère, carré, rectangle, triangle, cercle, ellipse, ...)
- Éléments d'un document (chapitre, section, paragraphe, figure, ...)

Toutes acceptent une méthode donnée

- Figures géométriques `draw()`
- Éléments d'un document `format()`

# *Double dispatch*

## ■ **Solution 1**

Utiliser le polymorphisme d'héritage

Méthode définie abstraite dans la super classe

Méthode implémentée dans chaque classe concrète

Le langage à objets s'occupe d'exécuter la bonne méthode pour chaque récepteur

```
new Carre().draw();  
new Cercle().draw();
```



# *Double dispatch*

## ■ **Problème 2**

La méthode a un paramètre qui peut prendre plusieurs types différents

Ex : Différentes surfaces sur lesquelles dessiner les figures géométriques (haute/basse résolution, couleur/N&B, ...)

`figure.drawOn (AbstractSupport support)`

# *Double dispatch*

## ■ **Solution 2**

Utiliser le polymorphisme paramétrique

Créer toutes les méthodes nécessaires (pour chaque type de paramètre) dans les classes

```
public abstract class FigureGeometrique {  
    public abstract  
        void drawOn(SupportBasseResolution s);  
    public abstract  
        void drawOn(SupportHauteResolution s);  
}
```

# *Double dispatch*

## ■ **Solution 2**

Java s'occupe d'exécuter la bonne méthode pour chaque récepteur et paramètre

```
Carre c=new Carre();  
c.drawOn(new SupportBasseResolution());  
c.drawOn(new SupportHauteResolution());
```

# Double dispatch

## ■ Solution 2

Mais le type du paramètre est calculé **statiquement** (à la compilation)

```
AbstractSupport sp;  
sp = new SupportBasseResolution();  
new Carre().drawOn(sp);
```

Même si `sp` est un `SupportBasseResolution`, le compilateur va considérer que c'est un `AbstractSupport` et chercher la méthode `drawOn` correspondante

# Double dispatch

## ■ Solution 2

Le **polymorphisme paramétrique** est calculé statiquement (à la compilation)



C'est le **type déclaré** de la variable qui compte, pas le **type réel** (au contraire du polymorphisme d'héritage)

# *Double dispatch*

## ■ **Solution 3**

Forcer la recherche dynamique de la bonne méthode à exécuter en fonction du paramètre  
Utiliser le polymorphisme d'héritage

## ■ ***Double-Dispatch***

Rajouter dans les classes paramètres une méthode intermédiaire qui va faire le bon *dispatch*


Note : Base du patron de conception Visiteur

# Double dispatch

## Double dispatch

Appel d'une méthode intermédiaire

```
AbstractSupport sp;  
sp = new SupportBasseResolution();  
new Carre().drawOn(sp);  
sp.draw(new Carre());
```



```
class SupportBasseResolution  
    extends AbstractSupport {  
    void draw(Figure f) { f.drawOn(this); }
```

# Double dispatch

## ■ Double-Dispatch

```
class SupportBasseResolution  
    extends AbstractSupport {  
    void draw(Figure f) { f.drawOn(this); }
```

```
class SupportHauteResolution  
    extends AbstractSupport {  
    void draw(Figure f) { f.drawOn(this); }
```

Les méthodes sont les mêmes !!!

Mais le type de « this » est différent

Calcul statique (à la compilation) de la méthode à appeler



# *Double dispatch*

## ■ **Double dispatch**

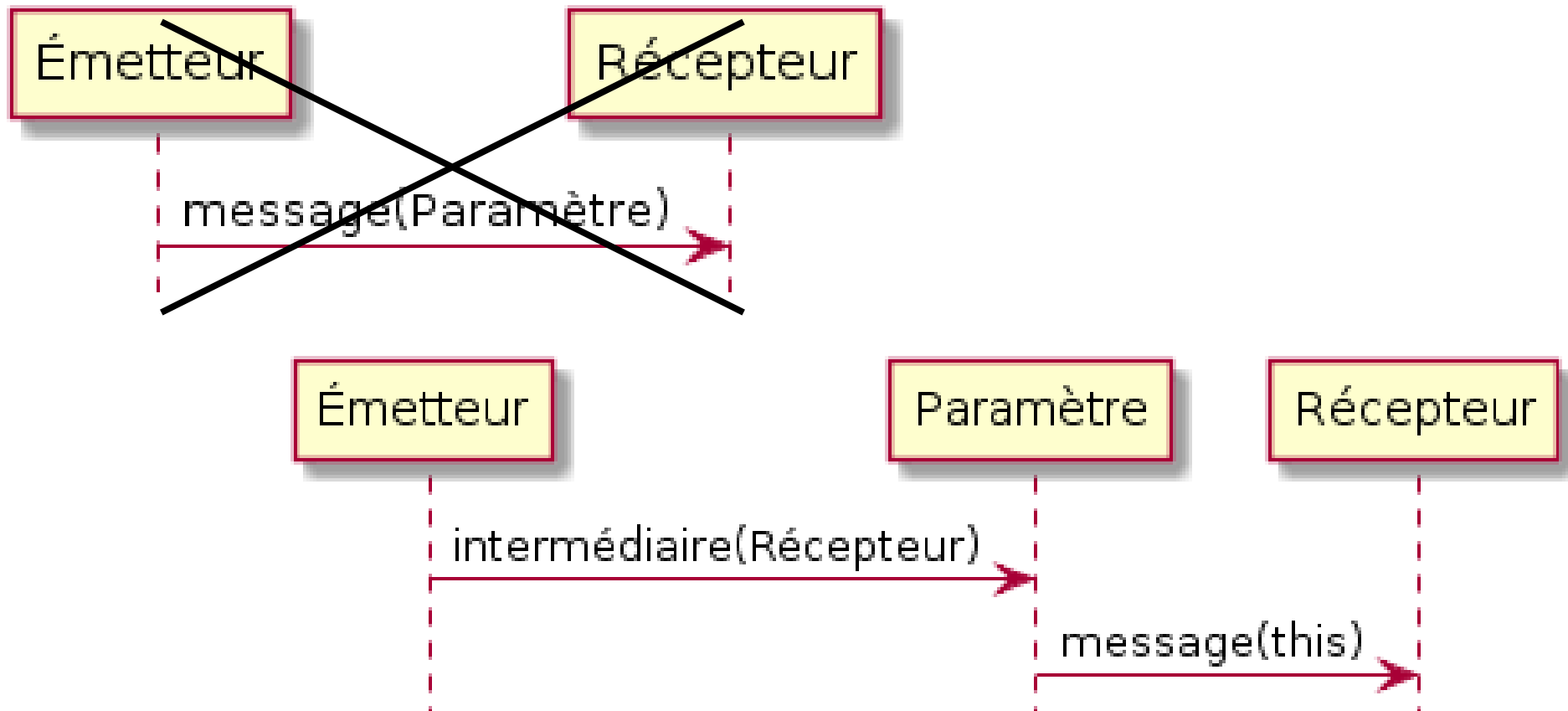
`sp.draw(...)` appelle la bonne méthode dans la sous-classe de `AbstractSupport` par polymorphisme d'héritage

Dans `SupportBasseResolution`,  
`f.drawOn(this)` appelle la bonne méthode de `FigureGeometrique` par polymorphisme paramétrique

Car, dans `SupportBasseResolution`, on sait statiquement quel est le type de `this`

# Double dispatch

## Double dispatch



# A retenir

- **Visibilité et Encapsulation**

- **Substituabilité**

Une sous-classe peut remplacer sa super-classe

- **Polymorphisme**

L'émetteur du message ne se soucie pas de la nature réelle du récepteur

`instanceof` est un *code smell*

- **Double dispatch**

