Université Paul Sabatier – Toulouse III
IUT A - Toulouse Rangueil

# Programmation Orientée Objet

Semestre 3

Toulouse, le 17 décembre 2011

# Chapitre 1

# Les classes et les objets

**Clef moodle** $noj11b$
2 facteurs de qualité en génie logiciel

**Extensibilité** Capacité pour un logiciel à intégrer de nouvelles fonctionnalités

**Réutilisabilité** Capacité pour un logiciel à être réexploité, tout en partie, pour de nouvelles applications

## 1.1 L'objet

**Définition** Structure de données présente à l'exécution formée de champs et d'opérations applicable à ces champs.
Un objet est une identité, un état et un comportement

**État** Ensemble des valeurs des champs de l'objet

**Comportement** L'ensemble de ces opérations

## 1.2 La classe

**Définition** Description d'une famille d'objets ayant même comportement.
Deux composantes dans une classe :
- La description des données appelés **attributs**
- La description des opérations appelés **méthodes**

### 1.2.1 Exemple

La classe *article* associée aux des objets *laLacoste* et *laBadoit*

```
class Article
{
  private String designation ;
  private float prixHT;


  public float prixDeVent() {
    return (prixHT * 1.196);
  }

  public float coutLivraison() {
    return(prixHT * 0.05);
  }

  public String getDesignation(){
    return (designation);
  }

  public float getPrixHT(){
    return (prixHT);
  }

  public void setPrixHT (float p){
    prixHT = p;
  }
}
```

### 1.2.2 Remarques

**1** Les méthodes de la classe Article (getPrixHT, setPrixHT, ...) ne possèdent pas en paramètre un article

**2 Principe de l'encapsulation** une classe n'exporte (mot-clé **public**) que ses services (méthode applicable à un article). Tout ce qui n'est pas exporté est considéré privé (mot-clé **private**)

**3** L'ensemble des services d'une classe (ses méthodes publiques) sont spécifiés par une interface.

## 1.3 Les objets instances de classe

Classe = modèle
Objet = Représentant de ce modèle (appelé l'**instance**)

### 1.3.1 Relation d'instanciation

Relation entre la classe et l'objet.
écriture d'une méthode spécifique à la classe appelée **constructeur** (création d'une instance à partir du modèle)

**Convention** le nom du constructeur est le nom de la classe et ne renvoit aucun résultat.

```
class Article
{
  // constructeur  initialise les champs d'un article
  public Article (String d, float p) {
    designation = d;
    prixHt = p;
  }

  //autres méthodes
}
```

### 1.3.2 Création d'objet

Pour créer un objet, on invoque le constructeur par l'opération **new**

**Exemple**

```
// déclaration de l'instance laLacoste
Article laLacoste
// création de l'instance laLacoste
laLacoste = new Article("Cehmise", 50.0);

// déclaration de l'instance laBadoit
Article laBadoit
//création de l'instance laBadoit
laBadoit = new Article ("EauGazeuse", 0.70);
```

**Remarques**

- Classe = Type
- Classe = Entité statique et Objet = Entité statique (créé par l'opérateur **new**)
- Une classe comprend en général les méthodes suivantes :
  - Un (ou plusieurs) constructeur
  - Des destructeurs (pour récupérer la mémoire des objets)
  - Des sélecteurs ou opération de consultation pour accéder aux champs de l'objet (souvent préfixés par get)
  - Des modificateurs pour modifier l'état d'un objet (préfixés par set)
  - Des itérateurs permettant de balayer une collection d'attributs
- L'interface de la classe Article s'enrichit du constructeur

# Chapitre 2

# L'héritage de classes et la composition d'objets

Deux relations entre classes
- La relation "être"(héritage) m
- La relation "avoir" ou "posseder" (⇒ composition d'objets)

## 2.1 L'héritage la relation "être"

**Définition** Une classe partage ses propriétés (attributs + méthodes) avec une autre classe apelée super-classe. Relation entre la classe et sa super-classe.

**Exemple** Les objets *laLacoste* et *laBadoit* partagent des propriétés communes (désignation, prixHT, getDesignation, ...) mais avec des spécifités (couleur et taille pour *laLacoste*, prixDeVente pour *laBadoit* (TVA=5.5%), ...)

### 2.1.1 Deux modes d'héritages

- Le mode par **enrichissement** : ajout d'attributs et/ou méthodes à la super-classe (appelée aussi classe ancêtre)

```
1  class Chemise extends Article
2  {
3    private int taille;
4    private Couleur coloris;
5
6    //constructeur de la classe Chemise
7    public Chemise (String d, float p, int t, Couleur c){
8      super (d, p);
9      taille = t;
10     coloris = c;
11   }
12 }
```

– Le mode par substitution redéfinition dans une sous-classe d'une méthode héritée

```
class EauGazeuse extends Article
{
  // Constructeur de la classe EauGazeuse
  public EauGazeuse (String d, float p){
    super(d,p);
  }
  // redéfinition du calcul du prix de vente de la classe
     Article
  // TVA = 5.5%
  public float prixDeVente() {
    return(getPrixHT() * 1.055);
  }
}
```

A partir des deux classes Chemise et EauGazeuse, on peut créer les objets laLacoste et laBadoit.

```
Chemise laLacoste;
laLacoste = new Chemise("Chemise", 50.0, 1, rouge);

EauGazeuse laBadoit;
laBadoit = new EauGazeuse ("EauGazeuse", 0.70);
```

**Remarques**

– Dans une même classe on peut hériter par enrichissement et par substitution
– Pour un eau gazeuse, une seule définition de la méthode prixDeVente, celle redéfinie dans la classe chemise (elle masque la méthode héritée)
– Dans la classe Chemise l'accès au prixHT par le selecteur getPrixHT d'Article (rincipe d'encapsulation)

### 2.1.2 Sémantique de l'héritage

– Pas de sémantique formelle ! En général hériter = "être une sorte de" **Attention** : héritage ≠ partage de code

**Point de vue extension**

Une classe est un ensemble d'objet
Si E designe l'ensemble des objets d'une classe alors si B hérite de A (mot clé extends)

$$E(B) \in E(A)$$

**Remarque** L'ensemble des chemises est un inclus dans l'ensemble des articles

**Point de vue intention**

Une classe est un ensemble de proriétés (attributs + méthodes)
si I désigne l'intention d'une classe alors si B hérité de A on a $I(A) \in I(B)$

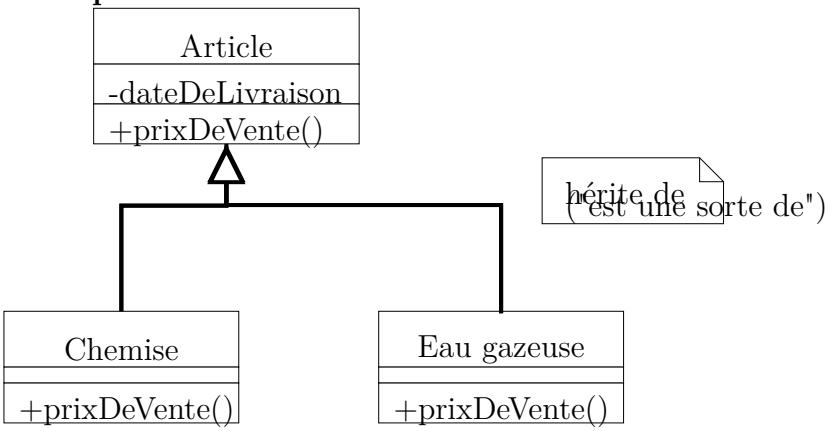
**Remarque** Une chemise possède les propriétés d'un article

**Remarque** Avec ces deux points de vue complémentaire :
Si B hérite de A, alors B est considéré sous-type de A (d'après classe = type)
Conséquence : Tout objet déclaré de type A pourra mémoriser dynamiquement un objet de type B (polymorphisme) :

### 2.1.3 Le graphe d'héritage

Représentation graphique de la relation d'héritage entre classe (cf diagramme de classe en UML par exemple)

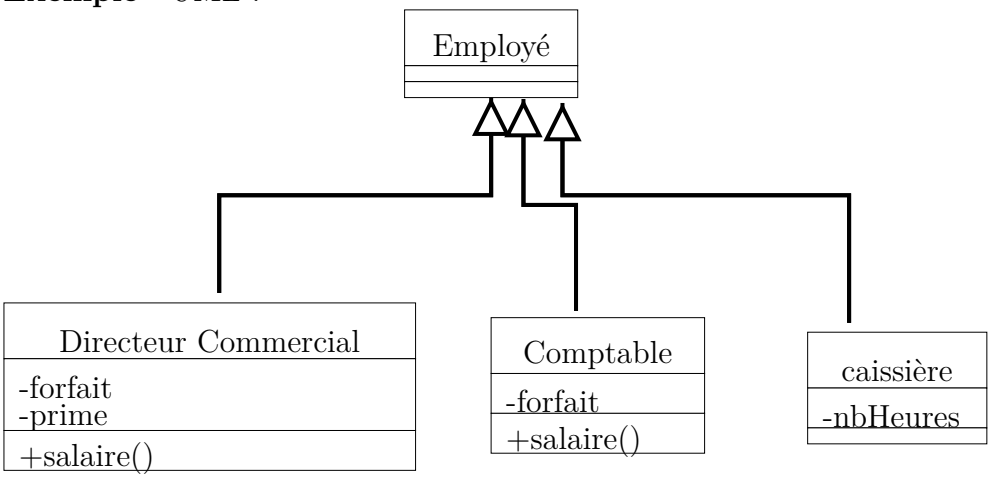
**Exemple** Classes Article et ses dérivés




### 2.1.4 Méthodes retardées et classes abstraites

Une méthode est dite retardée lorsque l'ensemble des sous-classes d'une classe donnée en proposent une définition.

**Exemple** UML :



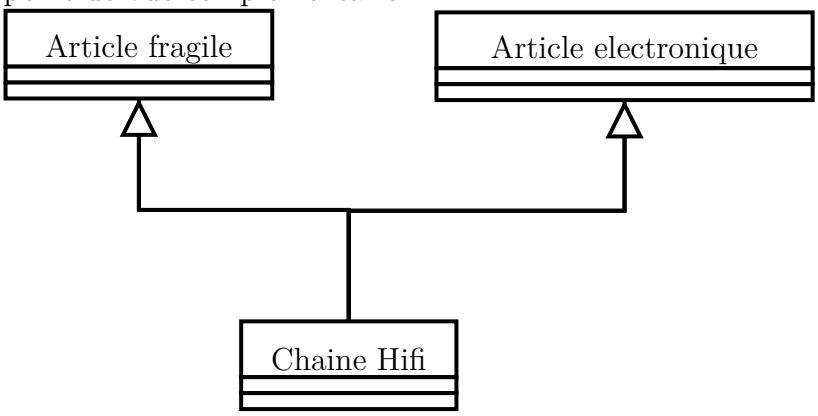

**Remarques**

- Appliquer la méthode salaire à un employé n'a pas de sens. A l'exécution, l'employé devra référencer(dans le code) un comptable, une caissière ou un directeur commercial.
- Une classe qui hérite d'une classe abstraite reste abstraite si elle ne définit pas la méthode retardée.
- On ne peut pas créer d'objet à partir d'une classe abstraite Par contre une classe abstraite peut avoir un constructeur pour ses classes descendantes (**super**)

### 2.1.5 Héritage simple et héritage multiple

**Héritage simple** une classe ne peut hériter directement que d'une seule classe ancêtre

**Héritage multiple** Sa classe peut hériter directement de plusieurs classe intérêt partager plusieurs point de vue complémentaire




```
1 class ChaineHifi extends ArticleFragile, ArticleElectronique
2 {
3
4 }
```

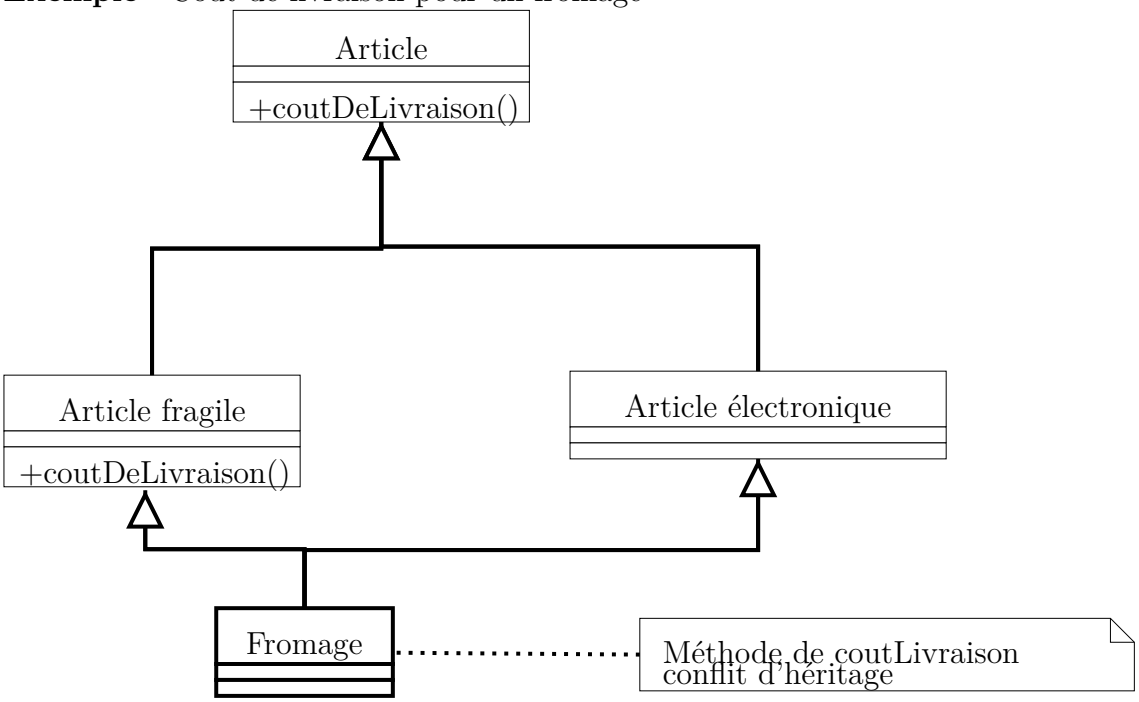
**Remarques**
- En héritage simple le graphie d'héritage est un arbre.
- Tous les langages à objets ne possèdent pas l'héritage multiple car :
  - Sémantique peut claire
  - Pue d'exemple pertinent exploitant l'héritage multiple
  - Des problèmes théorique et pratiques que subsistent

### 2.1.6 Conflits d'héritage

En héritage multiple, une même méthode peut être héritée par une sous classe plusieurs fois !

**Problème** pour la sous-classe quelle méthode retenir parmi l'ensemble des méthodes.

**Exemple** Coût de livraison pour un fromage




Dans les langages pas de relation universelle. Deux approches :
- Le choix est établi par le langage en considérant un parcours de graphe d'héritage
- Le programmeur résout (dans son code) le conflit

**1ère approche** Différents parcours possibles du graphe : en largeur d'abord, en profondeur d'abord, stratagème mixe...

**exemple : Parcours 1 :** Fromage, Article fragile, Article Périssable, Article ⇒ CoutLivraison de la classe Fragile
**Parcours 2 :** Résolution du conflit par le programmeur par exemple :
- En forçant la redéfinition
- En obligeant à renommer la méthode dans la sous-classe

## 2.2 La composition d'objets. La relation *Avoir*

Deux manières d'utiliser un classe :
- L'héritage (relation *Être*)
- La composition relation (*Avoir*)

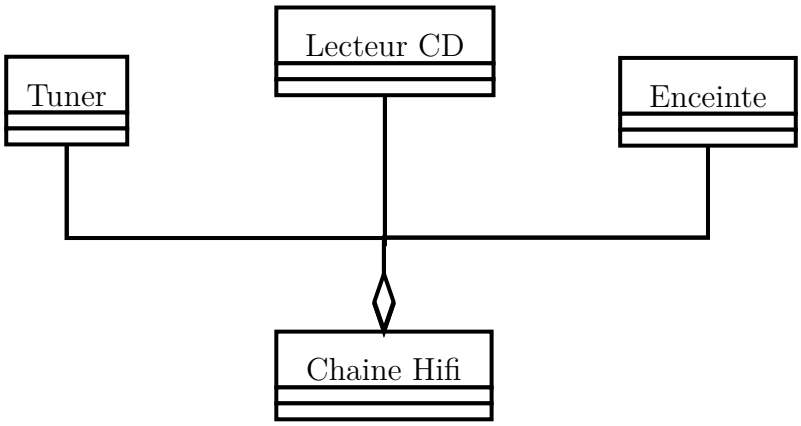
### 2.2.1 Définition

Un objet composite (ou agrégation d'objets) est un objet formé de l'assemblage de plusieurs objets.

**Avantage** Construire des objets complexes à partir d'objets existant. Les constituants d'un objet composite ne sont accessible que via leurs interfaces (principe d'encapsulation)

### 2.2.2 Exemple

**La chaîne hifi**




```
class ChaineHifi
{
  private Tuner t;
  private LecteurCD l;
  private Tableau <Enceinte> e;

  // Constructeur
  public ChaineHifi (int nb) {
    t = new Tuner();
    l = new LecteurCD();
    e = new Tableau<Enceinte> (nb);
  }
}

interface Tableau <T>
{
  Tableau <T>(int n);
  int longueur();
  T getIeme (int i);
  void setIeme(int i, T x);

}
```

**Remarques**

Avec un langage à objet, définition d'une bibliothèque de classes pour réutiliser ce qui existe déjà pour la classe Tableau

La communication entre un objet composite et ses constituants est réutilisée via l'interface des constituants ⇒ relation client-fournisseur entre le composite (client) et ses constituants (fournisseur)

Une classe A est cliente d'une classe B (et B est fournisseur de A) si
- A contient un attribut b de type B
- A possède une méthode avec un paramètre d'entrée ou de retour de type B
- A utilise une variable locale de type B pour une de ses méthodes

Le premier cas correspond à une **dépendance structurelle** les deux autres cas à une dépendance **non structurelle**

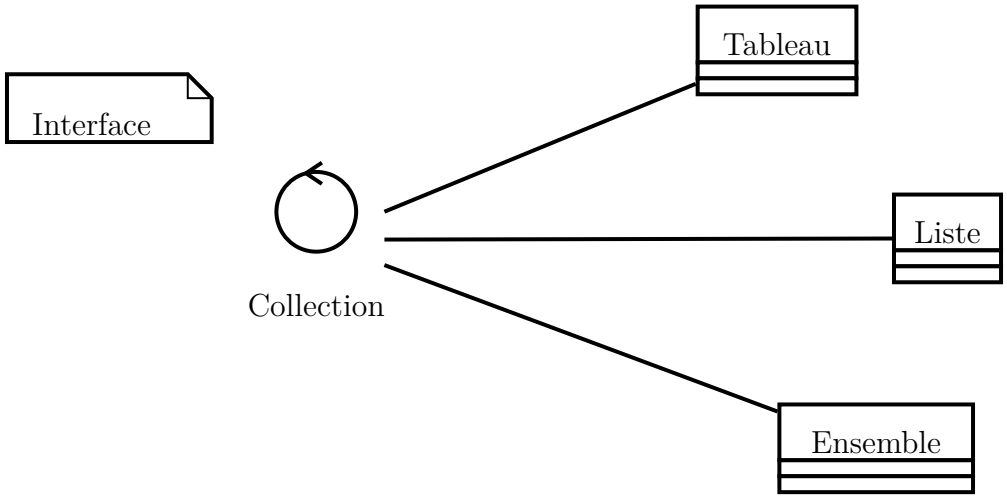
## 2.3 Mise en œuvre des mécanismes de réutilisation

### 2.3.1 Héritage d'interface

Mode d'héritage dans lequel toutes les sous-classes peuvent répondre aux services d'une classe ancêtre

**Point de vue** Une classe est assimilé à un type donc une sous classe est assimilé à un sous type. En principe la classe ancêtre est souvent abstraite.

**Exemple : les classes Tableau, Liste, Ensemble d'interface collection**




```
1 class ChaineHifi {
2   private Collection <Enceinte> e;
3
4   public chainehifi (int nb){
5     //...
6     e = new Tableau <Enceinte> (nb);
7   }
8
9 }
```

### 2.3.2 Héritage versus composition

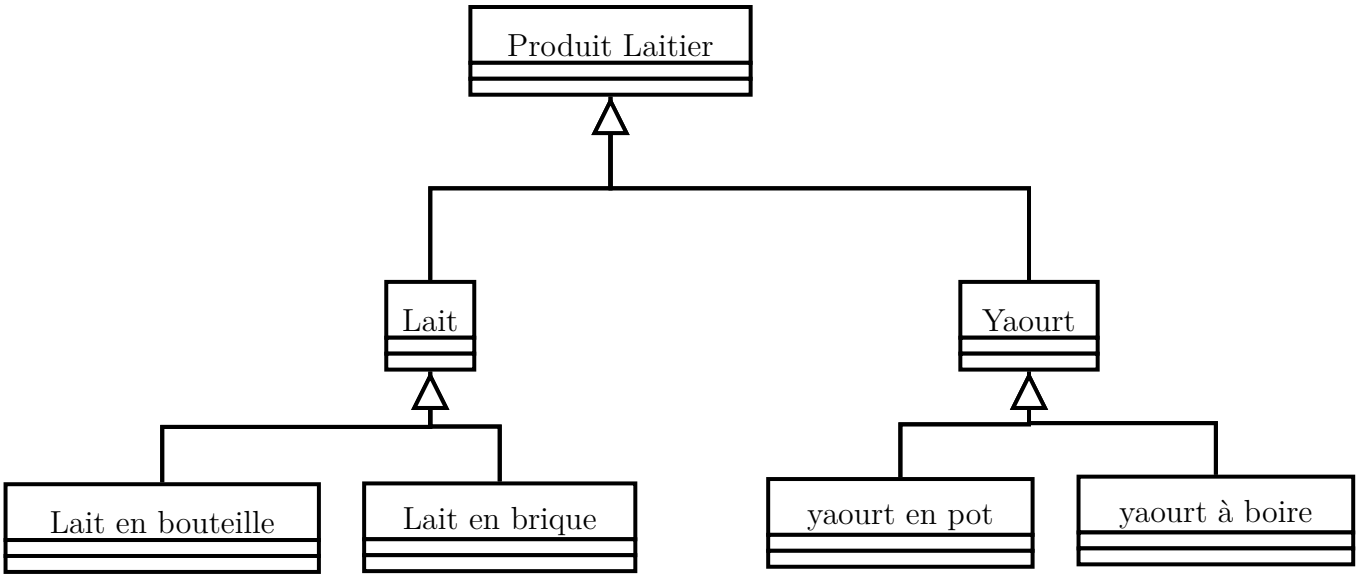
**héritage** statique (définit à la compilation).

**Composition** dynamique (à l'exécution)

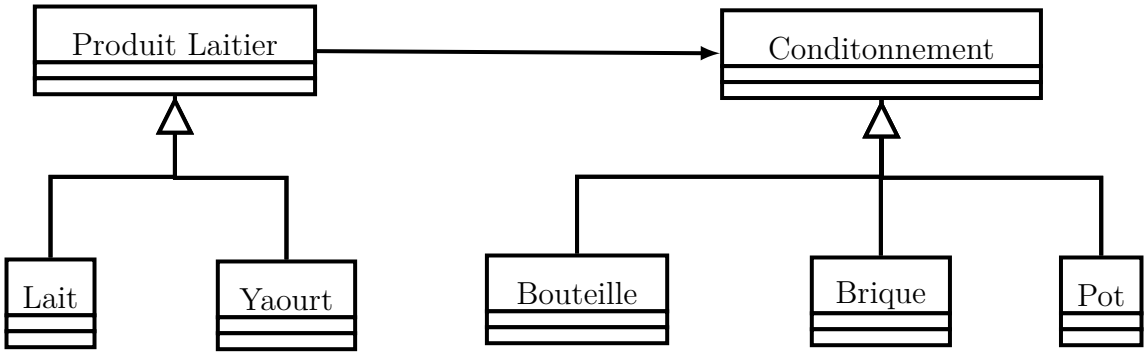
Avec la composition possibilité de combiner dynamiquement des objets entre eux.

**Exemple**

Les produits laitiers avec uniquement l'héritage




Les produits laitiers avec deux hiérarchies d'héritage : le produit et son conditionnement.




### 2.3.3 Héritage versus état d'un objet

Ne pas confondre héritage (classification) et état d'un objet (valeurs des champs de l'objet)

# Chapitre 3

# L'envoi de message et le polymorphisme

Envoie de message demandé par un objet de l'exécution d'un service de la classe.
C'est l'unique de manière de dialoguer entre objets.

Envoi de message = (Objet receveur, sélecteur de méthode, une liste de paramètre)

## 3.1 L'envoi de message

### 3.1.1 Cas général

L'envoi de message est réalisé par la primitive **send** ou la notation pointée

#### Exemple

Soit l'objet laLacoste, instance de la classe chemise.
prix = laLacoste.prixDeVente() ;
Avec un prixHT de 50€, prix = 59.8euros (car TVA = 19.6%)

Soit a de type Article et prix = a.prixDeVente() Si a à l'exécution désigne une chemise (instance de la classe Chemise).
prix = 59.8 euros (TVA = 19.6)

```
1  class HyperMarché
2  {
3    private Tableau <Article> stock;
4
5    public void editerStock(){
6      for(int i = 0; i < stock.longueur; i++){
7        out.println(stock.getIeme(i).getDesignation());
8      }
9    }
10
11     // autres méthodes
12 }
```

### 3.1.2 Désignation de l'objet receveur (objet self)

Dans les exemples de la section 3-1-1, on connait l'identité de l'objet receveur. L'objet laLacoste, l'objet a, l'objet out...
lorsque l'objet receveur n'est pas connu, on le désigne par l'objet courant receveur du message :

l'objet **self** (en Java, self = **this**).
Cas code d'un code d'une classe = code de l'instance (l'objet) courant

**Exemple** Dans la classe Article, ajout d'une méthode prixAvecLivraison = prixDeVente + coutLIvraison
Le même code doit s'appliquer à une chemise avec un taux de TVA = 19.6% et une eau gazeuse avec TVA = 5.5%

```
class Article
{
  // ...

  // Calcule le prix de vente avec livraison d'un article
  public float prixAvecLivraison(){
    return(this.prixDeVente() + this.coutLIvraison());
  }
}
```

**Remarque** Cet objet courant **this** est implicite dans la première version de la classe Article (chapitre 1 section 2).

### 3.1.3 Accès à la superméthode (mot-clé super)

Héritage par substitution ⇒ la méthode héritée est masquée par la redéfinition.
La méthode masquée est appelée superméthode.

```
class HyperMarché
{
  private Tableau <Article> stock;

  public void editerStock(){
    for(int i = 0; i < stock.longueur; i++){
      out.println(stock.getIeme(i).getDesignation());
    }
  }

    // autres méthodes
}
```

**pb** Comment réutiliser le code de la superméthode ?

On utilise la superméthode par un envoi de message en utilisant le mot clé **super** (qui n'est pas un objet)

**Exemple** Dans la classe Chaine-hifi, redéfinition de la méthode prixDeVente définie par :

$$prixDeVente_{materielHifi} = prixDeVente_{Article} + coutGarantie_{MaterielHifi}$$

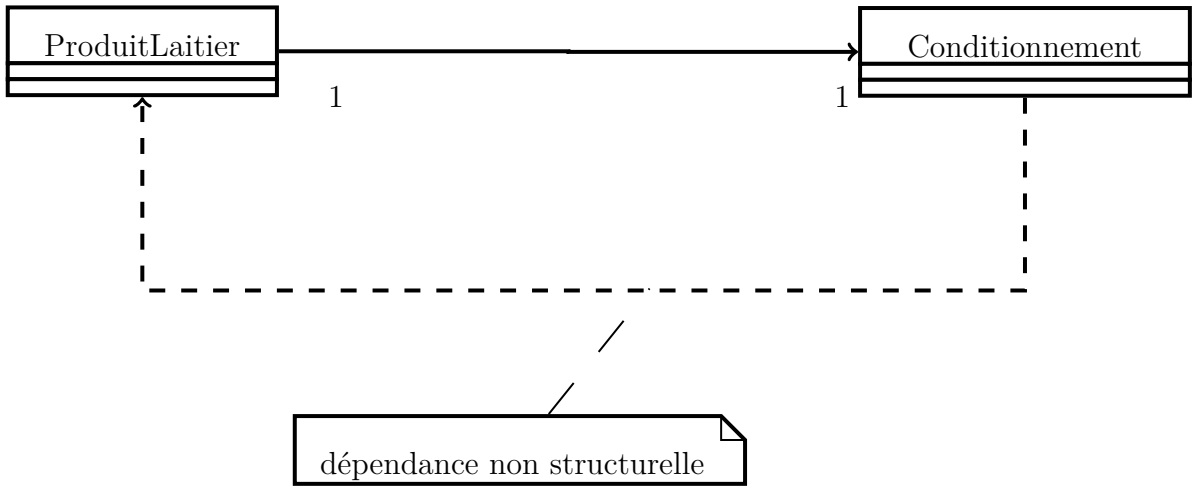
**Remarque** Attention à l'usage de **super** ! Un envoi de message avec **super** ne doit s'appliquer pour l'accès d'une superméthode masquée par l'héritage. Dans les autres cas utiliser **this**.

```
class ChaineHifi extends Article
{
  private float coutGarantie;

  public float prixDeVente() {
    return (super.prixDeVente() + this.coutGarantie());
  }

  //Autres méthodes
}
```

### 3.1.4 La délégation

Mécanisme de communication entre objets (par envoi de message) qui permet à un objet composite de déléguer à un de ses composant la requête à satisfaire.

**Exemple** Un produit laitier confie à un de ses composants conditionnement le soi d'emballer le produit. Pour mettre en place la relation non structurelle, on communique à l'emballage la référence **this** du produit laitier.




```
class ProduitLaitier
{
  private Conditionnement emballage;

  public ProduitLaitier(Conditionnement c) {
    this.emballage = c;
  }

  public void condtionner (){
    this.emballage condtionner(this);
  }

  // autre méthodes
}

```

```
16 class Conditionnement
17 {
18   public void conditionner (ProduitLaitier p){
19     //...
20   }
21
22   // autre méthodes
23
24 }
```

**Remarque**

1. Le code de méthodes conditionnement de ProduitLaitier et conditionnement est valide, quels que soient les produits laitiers et le conditionnement (ch chapitre 2 section 3.2)
2. Une classe A dépend d'une classe B (relation non structurelle).
   - Lorsque une méthode de A possède un paramètre de type ou un retour de type B
   - Lorsque une méthode de A possède un objet local de type B
3. On peut coder la classe Chemise du chapitre 2 par par un mécanisme de délégation. On indique alors qu'une chemise possède (relation avoir) la caractéristique d'un Article.

## 3.2 Le polymorphisme

**Polymorphisme** Plusieurs (poly) formes (morphes) possible d'une même entité. Deux types de molymorphisme lié :
- Le polymorphisme d'inclusions
- Le polymorphisme de redéfinition(ou polymorphisme) d'héritage

### 3.2.1 L'attachement polymorphe

Tout objet de classe d'appartenance (de type) A peut désigner à l'exécution un objet, sous-classe (sous type) de A.
D'après le principe d'inclusion des extensions (chapitre 2, section 1)

**Exemple** La classe Article avec Chemise et EauGazeuse.

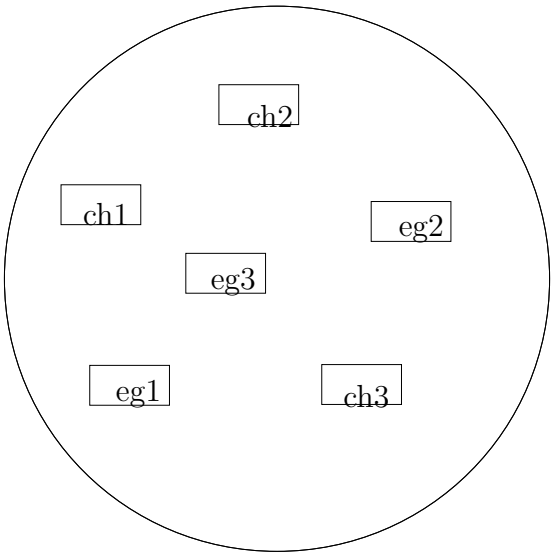
```
1     Article a;
2     Chemise c;
3     c = new Chemise("Chemise", 50.0, 1, rouge);
4
5     // Ecriture valide:
6     a = c;
7     a = new EauGazeuse("Eau Gazeuse", 0.70);
```

**Remarque**

1. Les attachements polymorphes (cf exemple avec a et c) interviennent lors des relations paramètre effectif. Paramètre formel- pour les paramètres de retour, pour les objets locaux et les objets courant **this**
2. L'attachement est autorisé dans le sens $objet_{ancetre} = objet_{descendant}$ L'affectation inverse n'a en général pas de sens !

### 3.2.2 Le polymorphisme d'inclusions

Ce polymorphisme résulte de l'héritage.
À l'exécution, un objet d'une classe peut désigner un objet d'une sous classe. Traiter une famille d'ojet en ignorant pour chacun sa classe d'appartenance. (ici des chemises ch et des eaux gazeuses eg)
Polymorphisme à condition de définir un héritage d'interface.




**Exemple**

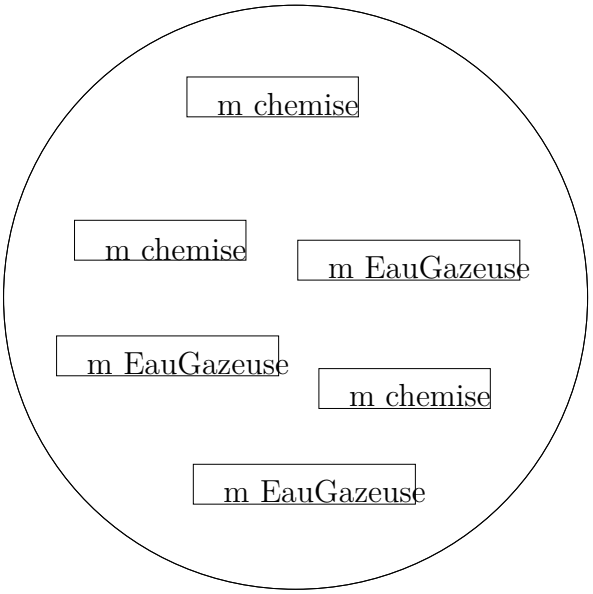
**1** Dans la classe HyperMarché, définition d'un attribut de type Tableau<Article>. Le table stock est un tableau polymorphe car il peut recevoir tout type d'article (des chemises, des eaux gazeuses,...)

**2** Autre exemple :

```
1 ProduitLaitier p;
2 Conditionnement c;
3
4 c = new Bouteille();
5 p = new Lait(c);
6 p.conditionner ();
```

### 3.2.3 Polymorphisme de redéfinition ou polymorphisme d'héritage

Polymorphisme qui s'appuie sur la redéfinition de méthode.
À l'exécution, le système choisit la méthode la plus spécialisée (redéfinie) en fonction du type de l'objet. Appliquer une même méthode à des objets en ignorant pour chacun sa classe d'appartenance (ici la méthode m des chemises et des eaux gazeuses)




### 3.2.4 Exemple

**1** Calcul du prixDeVente d'un article, fonction la TVA (19.6%) en général et 5.5% pour les eaux gazeuses)
Ce calcul s'effectue en fonction du ième article du tableau stock de type Table<Article>.

```
public Float prixDeVente(){
  return this.prixHT * 1.196);
}
```

Listing 3.1 – Dans Article

```
public Float prixDeVente(){
  return this.prixHT * 1.055);
}
```

Listing 3.2 – Dans eau gazeuse

```
prix = this.stock.getIème(i).prixDeVente();
```

Listing 3.3 – Dans supermarché

**2** Calcul de la masse salariale de la classe HyperMarché avec un attribut personnel de type Liste<Employé>.
Remarque : La méthode salaire de la classe employée est retardée !

```
public Float masseSalariale(){
  Float s=0.0
  for (int i = 0; i < this.personnel.longueur(); i++)
    s = s + this.personnel.getIeme.salaire();

  return (s);
}
```

Listing 3.4 – Dans supermarché

**3** Recherche séquentielle d'un élément dans un tableau

```
1  abstract class StructureSequentielle<T>{
2    //T est le type des éléments de la structure séquentielle
3
4    // se positionner sur le premier élément de la structure sé
       quentielle
5    abstract public void allerAuPremier();
6
7    // se positionner sur l'élément suivant
8    abstract public void allerAuSuivant();
9
10   // la fin de la structure séquentielle est-elle atteinte?
11   abstract public boolean epuise();
12
13   // test si l'élément courant de la structure séquentielle
14   // coincide avec x
15   abstract public boolean trouve();;
16
17
18   //recherche séquentielle d'un élément x
19   //dans la structure séquentielle
20   public boolean existeElement(T x){
21     this.allerAuPremier();
22     while(!this.epuise() && !this.trouve(x)){
23       this.allerAuSuivant();
24       return (!this.epuise());
25     }
26   }
27 }
28
29
30 class TableauSequentiel <T> extends structureSequentielle <T> {
31   private Tableau <T> elements;
32   private int indice;
33
34   //constructeur
35   public TableauSequentiel<T>(int n){
36     this.elements = new Tableau<T>(n);
37     this.indice = 1;
38   }
39
40   //Aller au premier élément du tableau
41   public void allerAuPremier() {
42     this.indice = 1;
43   }
44
45   //Aller au suivant du tableau séquentiel
46   public void allerAuSuivant () {
47     this.indice = this.indice + 1;
48   }
49
```

```
50    //fin du tableau séquentiel ?
51    public boolean epuise() {
52      return (this.indice > this.elements.longueur();
53    }
54
55    //L'élément courant du tableau séquentiel est-il égal à x
56    public boolean trouve(T x){
57      return (this.elements.getIème(this.indice) == x);
58    }
59  }
60
61  // Même écriture que tableSéquentielle<T> pour le TAD
      ListeSequentielle<T>, FichierSequentiel<T>, ...
```

# Annexe A

# épilogue : Et en java ?

| **Notion** | **Java** | **Commentaire** |
|---|---|---|
| Typage | statique | Typage définie à la compilation |
| Héritage | simple | ⇒ Pas de conflit d'héritage ! |
| Classes génériques | Oui(depuis Java 1.5) | Possibilité de définir des familles de types (avec un paramètre formel de généricité) |
| Liaison dynamique | Notation pointée | objet receveur.selecteur(parametre) |
| Attributs privées | Oui | Y compris en mode héritage (⇒ accesseurs get et set dans les sous classes) |
| Méthodes privées | oui | cf commentaire sur attributs privées |
| Instanciation | **new** | En général, deux étapes pour créer un objet : le déclarer puis le créer (par **new**. Création explicite. |
| Objetr receveur | **this** | |
| Accès à la superméthode | **super** | A réserver pour l'accès à une méthode ancêtre marquée par l'héritage |
| pointeurs visible | non | Par contre, tout est référence attention à = et == |
| Racine du graphe d'héritage | oui | Classe objet (avec ses méthodes classe, equals et string) |
| Algorithme de ramasse-miettes | oui | ⇒ Pas de destructeur à écrire |
| Autre caractéristiques | interface | Ensemble de méthodes retardées indiquant les services que doivent remplir les classes qui implémentent l'interface |