Programmation Par Objets et Langage Java
Partie III. La Programmation Objets en Java

Najib Tounsi

Lien permanent : http://www.mescours.ma/Java/PooJavaPart-3-tdm.html

Table des matières

Les Concepts de la POO

- ✓ Classe d'objets
  - Concrétisation d'un TAD
  - Objets, Méthodes, Envoie de Message, Instanciation ...
- ✓ Relations entre Classes
  - Composition (*use*) et Généralisation Hiérarchique (*isa*).
- ✓ Classes et Sous Classes
  - Héritage, Liaison Dynamique, Polymorphisme
  - Classes Abstraites et Interfaces
- ✓ Généricité
  - Paramétrage des Classes,
  - Collection et Itérateurs

Objets et Classe d'Objets

✓ Objets et Classes d'Objets 

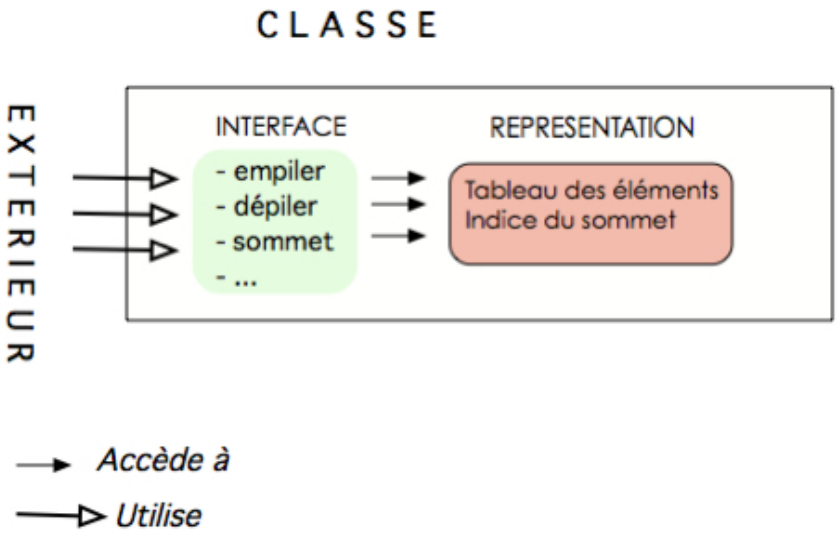
Une classe, est une implantation d'un TAD. Un TAD définit un ensemble d'objets caractérisé par les opérations qui leur sont applicables.

- La classe est l'unité modulaire de structuration d'un programme.
- Elle ***encapsule***

- la description des données représentant un objet (état d'un objet)
- la description des opération de manipulation
  (information sur l'état, ou changement d'état d'un objet)

- Un objet est une ***instance*** directe d'une classe. (cf. valeur et son type)

Champ, Méthode, Envoie de Message



Encapsulation des données et des traitements
dans une classe, ici une *pile*

Champ : Donnée faisant partie de la *représentation* d'un objet

Méthode : Opération de *manipulation* d'un objet

Envoie de message : *demande d'exécution* d'une méthode sur un objet

Note: Champs et méthodes sont appelés membres (*field member, method member*) en Java et C++.

Exemple

Une classe Article de commerce.

```
Classe Article
champs
    numéro;
    nom;
    prixHT;
    qte;
méthodes
    prixTTC () { return prixHT * 1.14; }
    ajouter (int q) { qte = qte + q; }
    retirer (int q) { qte = qte - q; }
fin
```

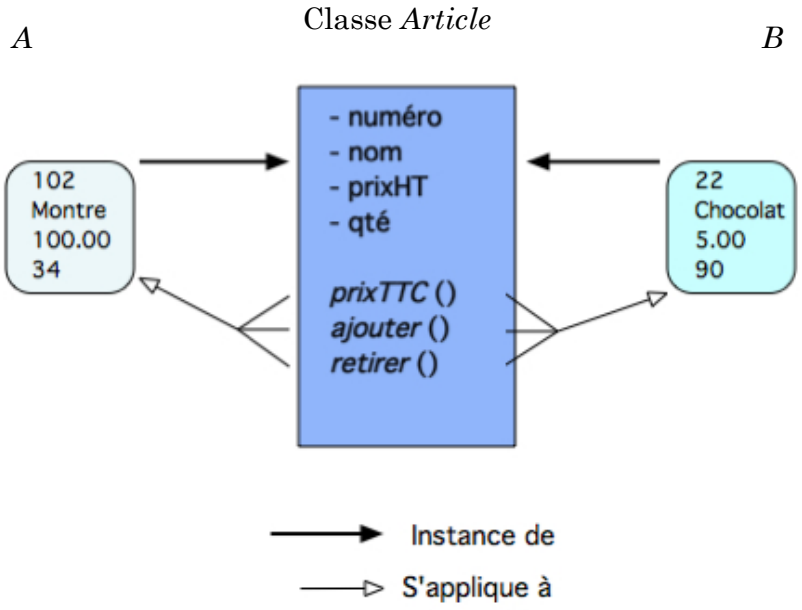
Champs et Méthodes ⇨ *caractéristiques* (*features*).

Identificateurs ⇨ *sélecteurs*

Champs dits aussi: *attributs*, *propriétés*

Instanciation d'objet

- C'est la création en mémoire d'un objet d'une classe.
- Avec initialisation par constructeur (voir plus loin).
- Une instance est référencée par une variable.



Une classe, deux instances (objets) *A* et *B*

En Java, on parle de *variables d'instance* (les champs) et de *méthodes d'instance* (les opérations). Voir plus loin, plus à ce sujet.

Instanciation d'objet

Dans une classe utilisatrice

- Exemple de déclaration (initialisation par défaut)

```
Article a = new Article();
```

- Un déclaration associe un nom de variable à un objet
- Exemples d'envois de messages

```
a.ajouter (5);
x = b.prixTTC();
```

- Un *constructeur* peut servir à l'initialisation (Voir plus loin).

Notation

Envoi de message à un objet (appel d'une méthode d'instance). Trois notations:

1. ***Envoyer_message*** (*Instance*, *Méthode*, *Paramètres*).

```
envoyer_message (a, ajouter, 5);
```

(cf. CALL, anciens langages)

2. *Méthodes* (*Instance*, *Paramètres*)

```
ajouter (a, q);
```

3. *Instance.Méthode* (*Paramètres*)

```
a.ajouter (5);
```

Intérêt de 3.

i. Forme contractée programmation méthodes
   Avec notations 1 et 2 on programme la méthode `{a.qte = a.qte + q;}`
ii. Mise en évidence envoie de message à un objet `a`

```
a.ajouter (5);
```

iii. Avec notation 3 on programme la méthode `{ qte = qte + q;}`

Référence `this`

- Appelé aussi: *self*, *current*
- Dans les méthodes, les champs se rapportent à l'instance à laquelle le message est envoyé

Appel opération	Méthode associée
a.ajouter(q);	Ajouter (entier q) {qte = qte + q;}

- Il existe un pointeur *implicite*, (**this**) vers l'instance d'appel.

qte = qte + q;	⟺	this.qte = this.qte + q;

Accès au données d'un objet

Consulter un attribut de l'objet

✓ Accès direct: *instance.champ*

```
Article art = new Article();
p = art.prixHT;
n = art.nom;
```

✓ Méthode explicite qui retourne la valeur d'un champ

```
p = art.getPrixHT();
n = art.getNom();
```

✓ Avec

```
classe Article
     ...
     getPrixHT() { return prixHT;}
     getNom() { return nom;}
fin
```

✓ méthodes dites ***fonctions d'accès***.

appelée conventionnellement `getPrix()`, `getNom()` ou `aPrixHT()`, `aNom()` en français...

Accès au données d'un objet

Caractéristique publiques / privées d'un objet

Bonne pratique 👍

- Méthodes *publiques*
- Champs *privés*

⇒ Abstraction, protection des informations ...

*Il est utile de ne laisser visibles à l'extérieur que certains sélecteurs (***publiques***) et de cacher les autres (***privés***). La bonne pratique c'est : seules les méthodes sont publiques.*

La Classe *Article* en Java

Sans constructeurs (voir plus loin)

```
public class Article {

// Champs
    private int numero = 0;
    private String designation = new String("spécimen");
    private double prixHT = 1.;
    private int qte = 0;

// Méthodes

    // Méthodes spécifiques
    public double prixTTC () { return prixHT * 1.10; }
    public void ajouter (int q) { qte =  qte + q; }
    public void retirer (int q) { qte = qte - q; }

    // Fonctions d'accès...
    public int getNumero(){return numero;}
    public double getPrixHT(){return prixHT;}
    public String getDesignation(){return designation;
    public int getQte(){return qte;}

} // Fin classe
```

Source complet.

- Ici, les affectations sur les champs membres donnent une valeur initiale à chaque instance (objet) créé.
- Noter que ces derniers sont déclarés privés, mot clé `private`, comme il sied à "*information hiding*".
- NB. Il y a d'autres moyens d'effectuer les initialisations: bloc d'instructions, constructeurs...

La Classe *Article* en Java

- Un objet *Article* est créé par l'opérateur *new*. Exemple dans *main*():

```
Article art = new Article();
```

- L'Article `art` est créé contenant `<0, "spécimen", 1.0, 0>` les valeurs fournies aux initialisations
- Exemple d'utilisation

```
System.out.println (art.getDesignation());     // spécimen
System.out.println (art.getPrixHT());          // 1.0
System.out.println (art.prixTTC());            // 1.10
art.ajouter(5);
System.out.println (art.getQte());            // 5
```

- Usage de `this`. (voir classe page précédente)

```
public void ajouter (int qte) { this.qte =  this.qte + qte; }
```

- Ici, le paramètre `qte` masque le champ `qte`. On peut alors le qualifier avec `this`.

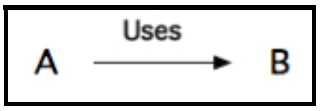
Relations Entre Classes

La Programmation Orientée Objets est la construction d'un système logiciel comme une collection structurée de classes. -- B. Meyer --

Première Relation Entre Classe

Relation Utilise «*Uses* »

- Une première relation entre classes est la relation: une classe en utile une autre.
- Une classe A *utilise* une autre classe B si elle déclare en son sein une ou plusieurs entités (champ, variable locale) de la classe B et en appelle les méthodes.



- On dit aussi que la classe A est ***cliente*** de la classe B (cf. langage Eiffel).
- A un certain point, si l'objet *A* vérifie la *précondition* pour lancer un traitement fournie une une méthode de l'objet *B*, alors *B* garantit à *A* la *postcondition* après exécution de cette méthode. C'est un ***contrat*** entre *A* et *B*.

Relation Utilise «*Uses*»

Relation utilise exemple

```
classe Commande
champs
   Article e;
   ...
méthodes
   facturer(){
   ...
   e.prixTTC();
   }
   ...
fin
```

↗

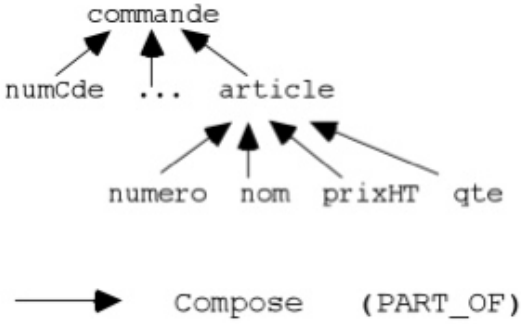
```
classe Article
  ...
  prixTTC () {...}
  ...
fin
```

- La classe `Commande` utilise la classe `Article`, car
  - dans la Commande il y a un Article commandé
  - la méthode `facturer` d'une commande a besoin de connaître le `prixTTC` d'un article

Relation Utilise «Uses »

Considérations sémantiques

- Un objet de classe *A* *a* parmi ses caractéristiques un objet de classe *B*.
- *B* est un composant de *A*.
- La composition, dite aussi agrégation (*aggregation*), est un puissant concept sémantique. (cf. notions de record, tuple)



- cf. relation ◊—— UML

NB. UML nuance un peu composition et agrégation...

La classe Article En Java

- Classe Article
- Source java

La Relation Utilise En Java

- En Java, le relation utilise est exprimée par la notion de **`import`**.
- Les classes sont généralement organisées en **packages** (répertoire de classes).
- Les classes s'utilisent les unes les autres en important le package contenant une classe.

```
import java.lang.*;           // import du package lang
```

- ou la classe spécifique

```
import java.lang.String;          // import de la classe String
```

- Voir plus loin la notion de *package*.

Les Constructeurs

- Un constructeur, est une méthode spéciale qui, quand elle est appelée, crée une instance d'une classe.
- En Java, le constructeur est une méthode qui a le même nom que la classe.
- Une Classe a intérêt à avoir, au moins, un constructeur.

- Initialiser les composants (champs) primitifs
- allouer l'espace aux composants objets et les initialiser

Les Constructeurs `Article`

Constructeurs de classe Article

- Constructeur explicite sans paramètres (dit aussi par défaut)

```
public Article(){
     numero = 0;                                  // ou au autre numéro
     designation = new String("@@@");    // idem
     prixHT = 1.0 ;                     // idem
     qte = 0;                              // idem
}
```

- Constructeur qui initialise par des valeurs choisies.
- Usage: Appel par l'opérateur `new` .

```
Article art = new Article();
```

- Un constructeur n'est jamais appelé explicitement comme une méthode normale.

Les Constructeurs `Article`

- Constructeur à partir d'objets de base

```
public Article(int n, String m, double p, int  q){
    numero = n;
    designation = new String(m);
    prixHT = p;
    qte = q;
}
```

- Usage :

```
Article art = new Article(10, "chemise", 249., 100 );
```

- Si *aucun* constructeur n'était déclaré pour la classe `Article`, i.e. aucune méthode `public Article()`, alors l'instruction `new Article()` créerait une instance dont les champs sont initialisé à `0`, `false` ou `null`.
- En fait, le compilateur cherche un constructeur par défaut sans argument hérité d'une super classe et s'il n'en trouvera pas il va appliquer le constructeur défaut de la classe `Object`.

Constructeur par Copie

- On peut vouloir créer un nouvel objet à partir d'un objet de même type déjà existant

```
Article art2 = new Article(art),
```

- Le constructeur suivant permet cela

```
public Article(Article a){
    numero = a.getNumero();
    designation = new String(a.getDesignation());
    prixHT = a.getPrixHT();
    qte = a.getQte();
}
```

- où les champs membres sont affectés un à un à partir des champs de l'instance en paramètre. (Remarquer l'affectation du champ `designation` qui est un objet `String`; on instancie une nouvelle chaîne en y mettant la valeur du champ correspondant de `a`).
- Ce genre de constructeur peut servir à créer une copie en profondeur d'un objet, i.e. dupliquer toute la structure de l'instance source.
- Voir plus loin: le *clonage*.
- ...

Destruction d'objets (le ramasse-miettes)

- Un objet créé en java est détruit (sa mémoire récupérée) dès qu'il n'est plus accessible.
- Un objet n'est plus accessible si il n'y a aucune référence qui le désigne.
- La référence à un objet est détenue par une variable qui disparaît à la sortie du bloc qui la déclare.
- Exemple simple.

```
...
{    Article a = new Article ();
     // l'article désigné par a est accessible
     ...
}
    // a n'existe plus.
    // La mémoire occupée par l'objet doit être libérée
```

- Tâche fastidieuse pour le programmeur. On aimerait bien que cela se fasse automatiquement
- Un programme spécial de la machine virtuelle, **ramasse-miettes** (*garbage collector*) est exécuté périodiquement et récupère la mémoire inutilisée.
- Le ramasse-miettes s'exécute automatiquement à un moment jugé utile pour récupérer de la mémoire.

Destruction d'objets (le ramasse-miettes)

- Une référence à un objet peut aussi disparaître quand on affecte la valeur spéciale `null` à une variable.
- Cependant, il peut exister plusieurs références à un même objet. Un objet est candidat au ramasse-miette quand la dernière référence à cet objet aura disparu.

```
Article mien = new Article ();
     ...
  {
       Article tien = mien;
       // l'article est accessible par la référence tien
       ...
  }
       // la référence tien à disparu, mais l'article existe toujours!
       // la référence mien est valable.
```

Méthode `finalize`

- Chaque objet Java possède une méthode appelée `finalize()`, héritée de la classe `Object`, mais qui ne fait rien.
- Cette méthode est prévue pour être redéfinie dans une classe utilisateur.
- Elle est destinée à être automatiquement appelée pour un objet quand il est candidat au ramasse-miettes.

- Elle peut servir à nettoyer les ressource utilisées par un objet comme par exemple fermer les fichiers ouverts.

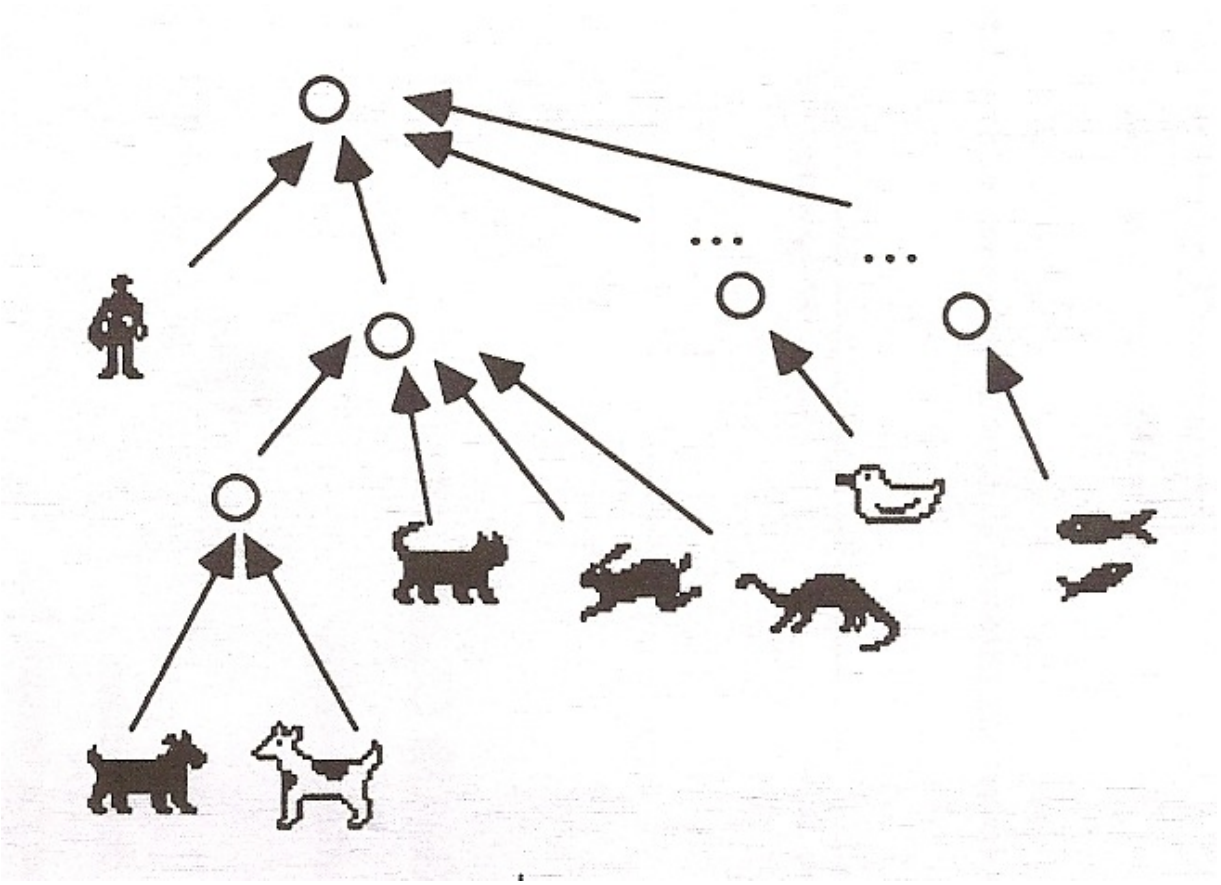
2e Relation Entre Classes: Héritage de Classe

Relation Hiérarchique «*ISA*»

✓ Deuxième relation fondamentale entre classes.
✓ Très structurante au niveau conceptuelle
✓ Concept sémantique de généralisation (Généralisation/Spécialisation).
✓ Très avantageuse au niveau pratique (héritage et réutilisation de code).
✓ ...

Héritage de Classes: Relation «*ISA*»

Who's who

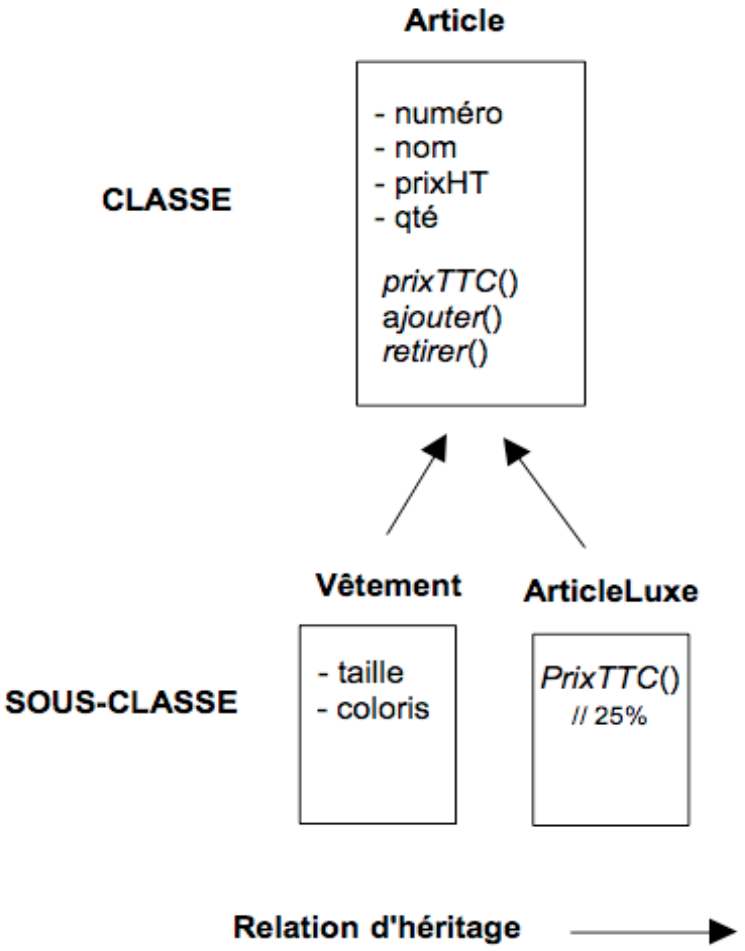


Héritage de Classes: Relation «*ISA*»

Un objet de classe *B* **est un** objet de classe *A* (aussi) s'il possède les caractéristiques de *A* plus d'autres qui lui sont propres.

- Dans un même ensemble, certains objets sont plus... nuancés.
- Une classe *A* pour les caractéristiques communes: **super classe**.
- Des sous classes *B*, *B*',... pour nuancer :
  - Rajout de nouvelles caractéristiques (plus spécifiques)
  - Modification d'une caractéristiques commune
- Les classes *B*, *B*', dites **sous-classes** de *A*...
- ... héritent ainsi de la classe *A* les propriétés communes.

Héritage de Classes: Relation «ISA»

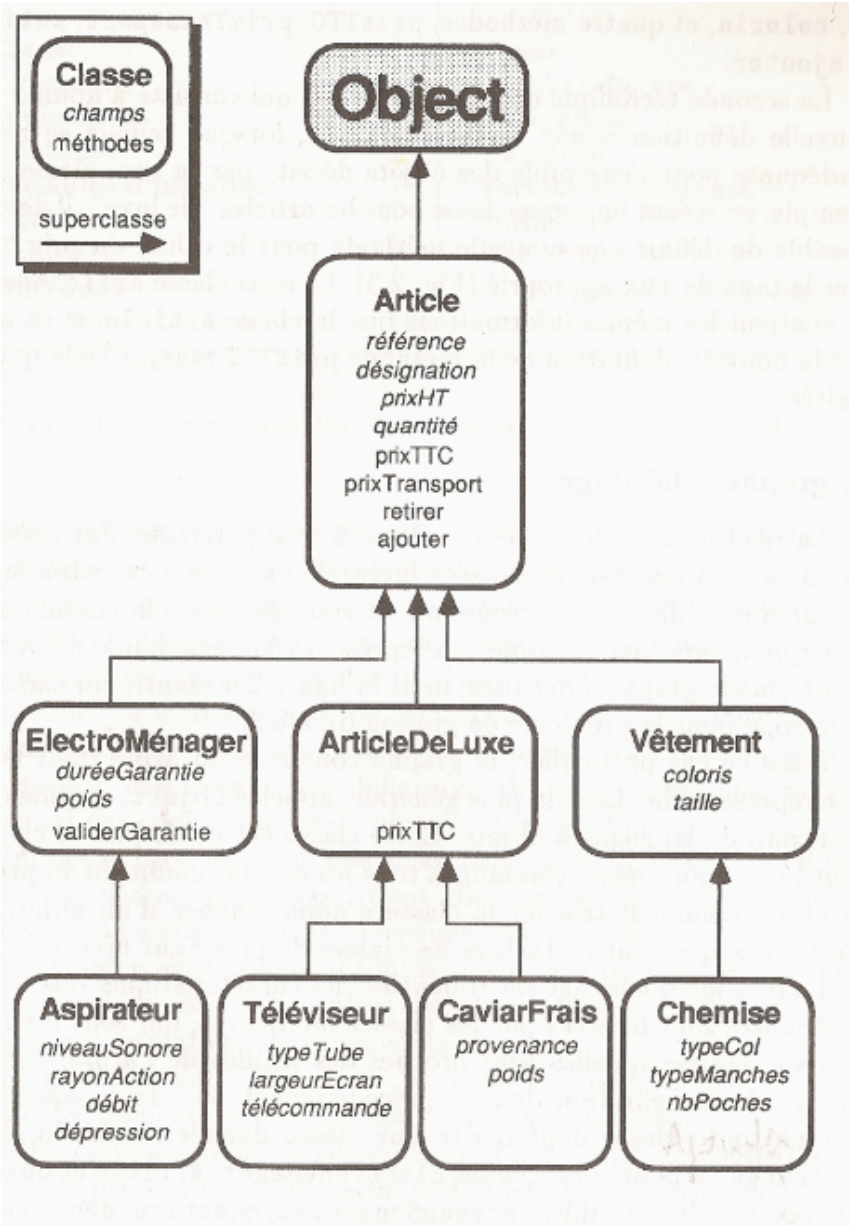


Exercice : Quelles sont toutes les caractéristiques de *Vêtement* et de *ArticleLuxe*?

Héritage de Classes: Relation «ISA»

- *Partage* de description
- *Réutilisation* du travail fait
- *Enrichissement* et complément de connaissances
- Développement *incrémental*
- Arbre d'héritage

Arbre d'héritage



Exemple tiré du livre "Les Langages à Objets", G. Masini et al.

NB. *Article* classe mère ***directe*** de *Vêtement*. Classe mère aussi de *Chemise* etc.

Sous Classes en Java - Exemple1

- Classe *Vêtement* sous classe de *Article* (version 1)

```
class Vetement extends Article {
    private byte taille;
    private String coloris;
};
```

- Le mot clé **`extends`** indique la super classe de la classe en cours de définition.
  - Donc une seule classe mère directe possible. Héritage simple. (voir plus tard, "héritage multiple").
- La nouvelle classe `Vetement` *rajoute* deux champs (`taille`, `coloris`) à ceux déjà hérités de la classe mère `Article` et qui sont (`numéro`, `designation`, `prixHT`, `qte`).

Sous Classes en Java - Exemple1

- Usage:

```
Vetement v = new Vetement();
v.ajouter(5);        // La méthode héritée de la classe Article s'applique.
```

- Ici, l'instance `v` est crée par l'instruction `new Vetement()`.
- Le constructeur défaut n'étant pas encore fournis, il est fait appel au constructeur défaut `Article()` hérité.
- Ce constructeur hérité de la super classe initialise donc uniquement la "partie article" d'un vêtement.
- Il faut donc définir un constructeur pour la sous classe

Sous Classes en Java - Exemple1 (Constructeurs)

- Sous classe `Vetement` (version 2) avec constructeur qui initialise la partie propre à la sous classe.

```
class Vetement extends Article {
    private byte taille;
    private String coloris;

    public Vetement(){

        super();

        taille = 6;
        coloris = new String("blanc");
    }
};
```

- Le mot clé `super` permet de faire appel explicite au constructeur hérité de la classe mère *directe*. Ici, le constructeur défaut `Article()`.
- Noter donc l'initialisation de la partie vêtement d'un objet de la sous classe.
- Sans `super`, l'appel au constructeur défaut hérité est *implicite* dans la méthode `Vetement()`. Il faut bien initialiser les champs hérités. L'absence de ce constructeur dans la classe `Article` provoque une erreur. On n'appellera pas les constructeurs défauts ancêtres.
- En cas de chaîne de sous-classes, il y aura une chaîne d'appels constructeurs.
- Le mot clé `super` permet entre autre de choisir le constructeur parent à appeler. on pourrait écrire `super(1, "t-shirt", 10, 5)` pour appeler le constructeur *Article* () avec paramètres.

Sous Classes en Java - Exemple2 (notion de *override*)

- Classe *ArticleLuxe* sous classe de *Article* (version 1)

```
class ArticleLuxe extends Article {
       // ...
    public double prixTTC () { return prixHT * 1.25; }

};
```

- Cette sous classe contient une redéfinition (même profile) de la méthode d'instance héritée *prixTTC*(). (Changement de taux de TVA pour le calcul du prix TTC d'un article).
- En Java on dit *override*, i.e. la nouvelle méthode surclasse celle héritée.
- Rque: surcharge particulière ici. C'est l'objet auquel le message est envoyé qui determine la méthode à appeler...

Sous Classes en Java (notion de `protected`)

- Le champ *PrixHT* hérité est utilisé directement dans la nouvelle sous-classe. Or il a été déclaré `private` dans la classe mère `Article`.

- Pour que cela soit possible il faut changer le statut de ces champs, sans les rendre publiques (`public`) pour autant.
- Il faut déclarer `protected` les champs de la classe mère `Article`.

```
protected int numero = 0;
protected String nom = new String("spécimen");
protected double prixHT = 1.;
protected int qte;
```

- Le statut `protected` signifie donc inaccessible pour les classes clientes (relation *uses*), accessible pour les sous classes (relation *isa*).

Les Sous Classes en Java

Dans une sous classe on peut donc: (http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html)

- Utiliser directement les champs hérités (`protected` dans la classe mère)
- Déclarer de nouveaux champs ou de nouvelles méthodes, pour enrichir la sous classe.
- Si on déclare un champ avec le même nom que dans la classe mère, ce dernier sera caché. Mauvaise pratique.
- Si une méthode (d'instance) a la même signature qu'une méthode héritée de la classe mère, elle redéfinit (*override*) la méthode héritée.
- On peut déclarer un constructeur pour la sous-classe qui fait appel au constructeur de la superclasse directe, implicitement ou avec le mot clé `super`.

Polymorphisme

- Les sous-classes définissent leurs propres comportements tout en gardant des caractéristiques héritées.
- Une particularité de l'héritage, c'est qu'un objet d'une sous classe *EST aussi* un objet de sa classe mère et de ses super classes.
- Un vêtement (ou un article de luxe), sont avant tout des articles. Tous ce qui s'applique à un article, s'applique aussi à un vêtement
- ... ou à des sous classes de vêtement (e.g. chaussure, vêtement-sport...)

Polymorphisme

- Cela veut dire que, malgré le typage fort de Java, une variable peut désigner non seulement un objet de sa propre classe, mais aussi un objet de ses sous-classes. C'est le ***polymorphisme***.

```
Article  a     = new Article(1, "Pomme", 10, 100 );
ArticleLuxe al = new ArticleLuxe(2, "iPhone", 200, 100);

a = al ;         // affectation juste!
```

- Par contre, l'inverse

```
al = a ;         // incompatible types
                 // found   : Article required: ArticleLuxe
```

- est une erreur de compilation: types incompatibles.
- Astuce : dans l'affectation, respecter le sens de la flèche *ISA*: `a` ← `al`, article reçoit article de luxe

Polymorphisme

- Le polymorphisme implique une conséquence importante.
- Dans l'exemple précédent, la classe `ArticleLuxe` a redéfini la méthode `prixTTC()`. Le calcul de la TVA est différent. Un article de luxe est taxé à 25% au lieu de 10% pour un article normal.

```
System.out.println( a.prixTTC() );      //  donne 11     ( 10 * 1.10)
System.out.println( al.prixTTC() );     //  donne 250    (200 * 1.25)
```

- Que se passe t-il maintenant si une variable `Article` (`a` ici) contient une instance de `ArticleLuxe`?

```
a = al;
System.out.println( a.prixTTC() );       //   donne 250 maintenant!
```

- A la compilation, Java vérifie que la méthode (redéfinie) `prixTTC()` s'applique à la variable `a`. Mais ne décide pas quel sera le code à exécuter pour ça.
- C'est à l'exécution, et en fonction de l'objet référencé par `a`, que sera cherché quelle méthode à appliquer.
- L'édition de lien est dynamique.
- C'est là tout l'avantage de l'héritage et de la programmation par objets.

Polymorphisme

Autre Exemple

- Soit la variable *chariot* déclarée comme un tableau de trois articles contenant un article normal, un article de luxe et un vêtement.

```
 Article[] chariot = new Article[3];
 chariot [0] = new Article(1, "Pomme", 10, 100 );
 chariot [1] = new ArticleLuxe(2, "iPhone", 200, 100);
 chariot [2] = new Vetement(3, "Chemise", 30, 100, "Vert", XL );
```

- Considérer une méthode *caisse* qui calcule le montant à payer pour un chariot plein d'article. Il suffit d'appeler la méthode `prixTTC()` pour chaque article du chariot, et de faire la somme:

```
double montant = 0;
for (Article p:chariot) {
     montant +=  p.prixTTC();
}
System.out.println(montant);              // 294
```

- soit dans ce cas: 294 , c'est à dire 11 + 250 + 33

Polymorphisme

Le polymorphisme peut intervenir dans plusieurs autres endroits :

- Paramètre de fonction

```
static void f(Article a){
   System.out.println(a.prixTTC());
}
```

f () peut recevoir en paramètre effectif n'importe quel objet de la hiérarchie d'articles.

- Composant d'un objet

```
class Commande {
    Article a;
     // ...
}
```

  N'importe quel article peut faire l'objet d'une commande.

- Elément d'une liste, comme dans le cas d'un tableau (cf. exemple précédent)
- etc...

Polymorphisme

Le polymorphisme est très important car il permet à une classe de déléguer à ses classes descendantes le fait de définir elles-même et selon leur besoin le code des méthodes héritées.

La classe `Article` peut se contenter de définir `prixTTC()` avec un code par défaut et laisser les sous classes `Vetement`, `ArticleLuxe`, ... le soin de fournir elles-même le code qui leur est adapté.

Plus encore, la classe `Article` peut se contenter de déclarer `prixTTC()` sans en donner un code. Méthode abstraite. (Voir plus loin les classes abstraites.)

A propos des méthodes `static`

- Une sous classe hérite des caractéristiques de ses superclasses.
- Une sous-classe peut définir une méthodes ayant le même profile qu'une méthodes d'*instance* héritée. C'est une redéfinition (*override*) de méthodes. C'est la discussion précédente sur le polymorphisme.
- Une sous-classe peut aussi définir une méthode ayant le même profile qu'une méthode *static* héritée. C'est un masquage. La méthode de la sous classe masque celle héritée. (Il n'y a pas de polymorphisme.)

La Superclasse `Object`

- C'est la classe racine de l'arbre (d'héritage) des classes Java.
- La classe `Object` est définie dans le package `Java.lang`.
- Chaque classe Java est sous-classe directe (absence de clause `extends`) ou indirecte de `Object`.
- Ainsi, les classes Java héritent d'un ensemble de méthodes d'instances de `Object`.
- On *doit* donc réécrire (*override*) ces méthodes pour les *adapter* à chaque cas.

Les méthodes héritées de `Object` ...

Trois méthodes particulièrement intéressantes sont:

- `protected Object` **`clone`**`() throws CloneNotSupportedException`
  - ☞ Crée un nouvel objet qui est une copie de l'objet (`this`) qui exécute.
- `public boolean` **`equals`**`(Object obj)`
  - ☞ Teste si l'objet en paramètre est "égale" à l'objet (`this`) qui l'exécute.
- `public String` **`toString`**`()`
  - ☞ Retourne une chaîne représentant l'objet. Utile par exemple pour faire *println*() d'un objet.

Les méthodes héritées de `Object`

Et aussi

- `protected void` **`finalize`**`() throws Throwable`
  ☞ déjà mentionnée et qui est appelée lors du ramasse-miettes.
- `public final Class` **`getClass`**`()`
  ☞ Retourne la classe à l'exécution de l'objet qui l'appelle.
- `public int` **`hashCode`**`()`
  ☞ Retourne le hashcode de l'objet qui l'exécute (entier distinct (en pratique) pour chaque objet).

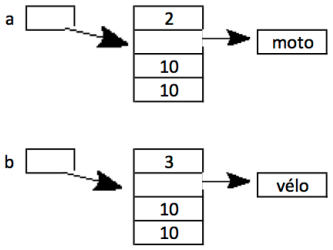
Ainsi que d'autres méthodes liées au parallélisme en Java (programmation concurrente).

Méthode *equals* () ...

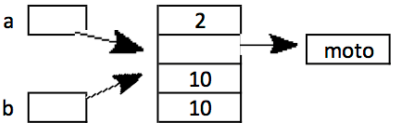
- Deux objets primitifs sont égaux, s'ils ont la même valeur. L'opérateur `==` sert à faire ce test.
- Pour des objets appartenant à des classes, le test `==` donnera vrai si deux variables sont la même référence.

```
Article a = new Article(2, "moto", 10, 10);
Article b = new Article(3, "vélo", 10, 10);
if (a==b) ...    // toujours faux, car deux objets différents
                 // même si b peut être initialisé avec les mêmes valeurs

a = b;
if (a==b) ...     // toujours vrai, exactement le même objet ("vélo" ici)
```



a et b deux objets différents
`a == b` *faux*



a et b désignent un même objet
`a == b` *vrai*

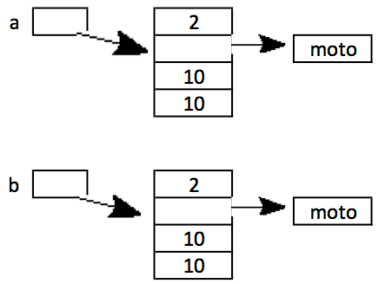
- Or, la méthode `equals()`, hérité de la classe `Object`, fait le test `==` si elle est appliquée entre deux objets.

```
if ( a.equals(b) ) ...  // même résultat que ci-dessus
```

- Ce n'est peut-être pas ce que souhaite l'utilisateur.

Méthode *equals* () ...

- Il faut parfois donner un sens à la comparaison de deux objets utilisateurs. Par exemple, ils contiennent exactement les mêmes informations. Ou bien ils ont le même numéro dans le cas de deux articles, etc.
- La solution c'est de ***redéfinir*** pour la classe `Article` la méthode `equals()` héritée.



a et b deux objets (distincts) de même valeurs
(égalité en profondeur)

Méthode *equals* () ...

- On pourra par exemple décider que deux articles sont égaux, s'ils ont exactement les mêmes valeurs, ou tout simplement le même numéro, comme dans une base de donnée.
- Dans la classe `Article` on redéfinit alors la méthode héritée `public boolean` **`equals`**`(Object obj)`, de paramètre `Object`.

```
class Article {
...
    public boolean equals (Object x){
        // deux articles sont égaux s'ils ont le même numéro
         return (numero == ((Article)x).getNumero());
    }
}
```

- On aura alors

```
Article a = new Article(2, "moto", 10, 10);
Article b = new Article(2, "moto", 10, 10);
System.out.println (a.equals(b));              // donnera true
```

- Noter (1): le profile de la méthode avec comme paramètre `Object`. Sinon, ce ne serait pas un Override (cf. exercice ci-après)
- Noter (2): la conversion explicite de `x` vers `Article` avant d'appliquer la méthode `getNumero()`.

Le choix de programmation de la méthode `equals()` pour une classe utilisateur donnée dépend donc de l'application: égalité des clés, égalité de toutes les valeurs, etc.

Méthode *equals*()

- *Exercice* : Quel est la différence avec la méthode suivante, où on teste l'égalité de deux objets `Article`:

```
public boolean equals (Article x){
     return (numero == x.getNumero());
}
```

- *Réponse* : Le polymorphisme ne joue pas! Le profile est différent.

```
 Object o = new Article(2, "moto", 10, 10);
                        // Object instancié avec un article
 System.out.println (a.equals(o));          // false
                   // pourtant a et o ont le même n° article
```

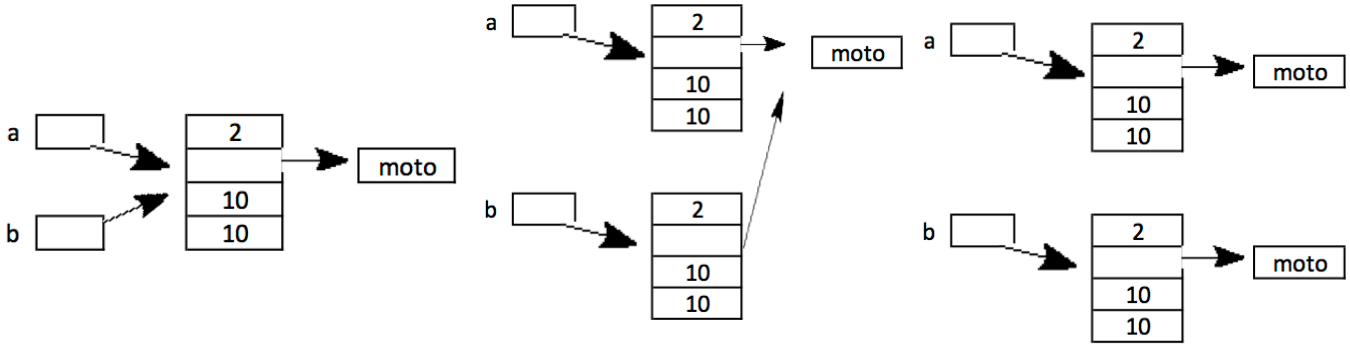
- `a` et `o` ont les mêmes valeurs, et pourtant le test répond faux.
- La méthode `boolean equals(Article)` juste ci-dessus, *ne redéfinit pas* la méthode héritée de `Object`, qui a le profile `boolean equals(Object)` et n'est donc pas appelée.
- Le message `a.equals(o)` fera donc appel à la méthode *héritée* `boolean equals(Object)` qui va

donc tester les références si (`a == o`), ce qui est faux.

- Autre remarque : égalité superficielle *vs*. égalité en profondeur.

Méthode *clone* (), Copie d'Objets

- Soit `Article a = new Article(2, "moto", 10, 10);`
- Que signifie l'affectation `b = a;` ? (`b` étant de classe `Article`)
- Il y a lieu de distinguer



(1) *Simple Copie* (2) *Copie superficielle* (3) *Copie en profondeur*

- Le cas (1) correspond à l'affectation des références. C'est la cas de Java.

```
Article a = new Article(2, "moto", 10, 10);
Article b = new Article();
b = a;
```

- Pour les cas (2) et (3) il faut dupliquer les valeurs de *a* dans *b*.
  - (2) est une copie bit à bit des champs du premier niveau (objet pointé),
  - (3) représente tous les autres cas, i.e. une duplication complète de tout l'objet, (du moins jusqu'au deuxième niveau).
- Comment faire (2) et (3) en Java?

Méthode *clone* (), Copie d'Objets

- Une première façon de faire une copie d'objet c'est d'utiliser un constructeur copie. (cf. Constructeur par Copie de la classe `Article`).
- Exemple :

```
Article b = new Article (a);
```

- pour instancier et *initialiser* un nouvel objet `b` à partir d'un objet `a`.
- Ce pourrait être aussi une *affectation* en cours de programme :

```
...       // b déjà déclarée et utilisée
b = new Article (a);
```

Méthode *clone* (), Copie d'Objets

Clonage superficiel.

- On peut utiliser la méthode `clone()` héritée de la classe `Object`, à condition d'implémenter l'interface `Cloneable`, et qui soulève une exception.

```
class Article implements Cloneable {... corps de Article ...}
```

- et rajouter (redéfinir en fait) la méthode simple :

```
public Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

  Ici on s'est contenté de faire usage de la méthode `clone()` héritée.

- On peut alors copier (cloner) un article avec le code

```
try {
     b = (Article) a.clone();
}
catch (CloneNotSupportedException e){...}
```

- Ici `b` a reçu une copie de `a`. Ce sont deux objets différents mais de même valeur.
- La copie a été réalisée par la méthode `public Object clone()` redéfinie par `Article`.
- Cette redéfinition s'est en fait contenté, avec `super.clone();`, de faire appel à la méthode `clone()` héritée de `Object`, qui crée un objet de la même classe que l'original et initialise ses champs avec les mêmes valeurs que l'original.
- Cette méthode fait donc une *copie superficielle*. En effet, ici on copie tous les champs y compris la référence `nom` d'un article.

Méthode *clone*(), Copie d'Objets

Copie en profondeur

- La copie en profondeur devra être réalisée "à la main".
- Une implémentation pourrait être

```
public Object clone()  throws CloneNotSupportedException {
      Article w = new Article();
      w.numero = this.numero;
      w.designation = new String(this.designation);
      w.prixHT = this.prixHT;
      w.qte = this.qte;
      return w;
}
```

- Un nouvel objet `Article w` est créé, ses champs initialisés à partir de ceux de ceux de `this` et est ensuite retourné.

Classes Abstraites

- Nous avons vu qu'une classe pourrait ne pas définir une méthode et laisser le soins aux sous-classes de fournir le code approprié.
- La classe `Article` pourrait ne pas donner de méthode de calcul du prix TTC, et laisser chaque sous classe le faire à sa façon.
- Il faut alors déclarer la classe et la méthode ***abstraites***.

```
abstract class Article {
    // ...
abstract public double prixTTC ();
  // ...
}
```

- Noter la méthode sans corps.
- La classe `Article` ne sera plus instanciable. A juste titre. On ne peut envoyer le message `prixTTC()` à une éventuelle instance.
- Les classes abstraites ont leur utilité pour définir des classes d'objets généraux qui seront

spécialisés plus tard par des sous classes.

Classes Abstraites

- Un exemple est une classe d'objets *figures géométriques*, avec des méthodes *desiner*(), *surface*(), *périmètre*(), etc.

```
abstract class FigureGeometrique {
    int x, y;
        // ...
    void deplacer(int newX, int newY) {
        // ...
    }
    abstract void dessiner();
    abstract int surface();
 }
```

- La classe sera spécialsée par des sous-classes *carré*, *rectangle*, *cercle*, qui, elles, peuvent fournir les méthodes appropriées à leur cas.

```
class Cercle extends FigureGeometrique
 {
    void dessiner() {
        ...
    }
    int surface() {
        ...
    }
  }

 class Rectangle extends FigureGeometrique
 {
    void dessiner() {
        ...
    }
    int surface() {
        ...
    }
 }
```

- Le polymorphisme permet alors à chaque figure géométrique de se déssiner par exemple comme il faut.
- Ce mécanisme peut s'étendre à plusieurs niveaux. Une sous-classe de classe abstraite peut se déclarer aussi abstraite et déléguer à une des ses propres sous classe l'implantation des méthodes abstraites.
- Quand une sous-classe implémente une méthode abstraite et veut interdire à ses descendants de la redéfinir, elle déclare la méthode `final`.

```
final int surface(){
 ...
}
```

- La classe `Object` fait cela pour certaines de ses méthodes (e.g. *getClass*, *notify* ...).

Les Interfaces Java

- Une interface est une collection nommée de déclarations de méthodes (sans les corps). Une interface peut aussi déclarer des constantes.

```
interface Pile {
    final int MAX = 8;
    public void empiler(char c);
    public char sommet();
```

```
        // ...
};
```

- L'interface a la forme d'une classe.
- Ici, on a une interface correspondant à une pile de caractères.
- On a juste déclaré une taille de pile et les profiles des opérations sur une pile.
- En effet, on peut laisser à la classe qui *réalise l'interface* le soin de choisir elle-même quelle représentation donner à une pile et donc définir les méthodes. On dit `implements` (en français *réaliser*, *mettre en oeuvre*).

Implémentation des Interfaces

- Une mise en oeuvre de l'interface précédente pourra être :

```
class PileTableau implements Pile {

   char[] t = new char[MAX];
   int top=-1;

  public void empiler(char c) {
    t[++top] = c;
  }

  public char sommet() {
    return t[top];
  }
};
```

- Ici on a réalisé une (`class PileTableau`) avec un tableau de caractères.
- On pourrait en définir d'autres avec d'autres structures de données, e.g. chaîne de caractère, liste, etc.

Héritage "Multiple"

- Un autre usage des interfaces, c'est la possibilité de faire l'héritage multiple

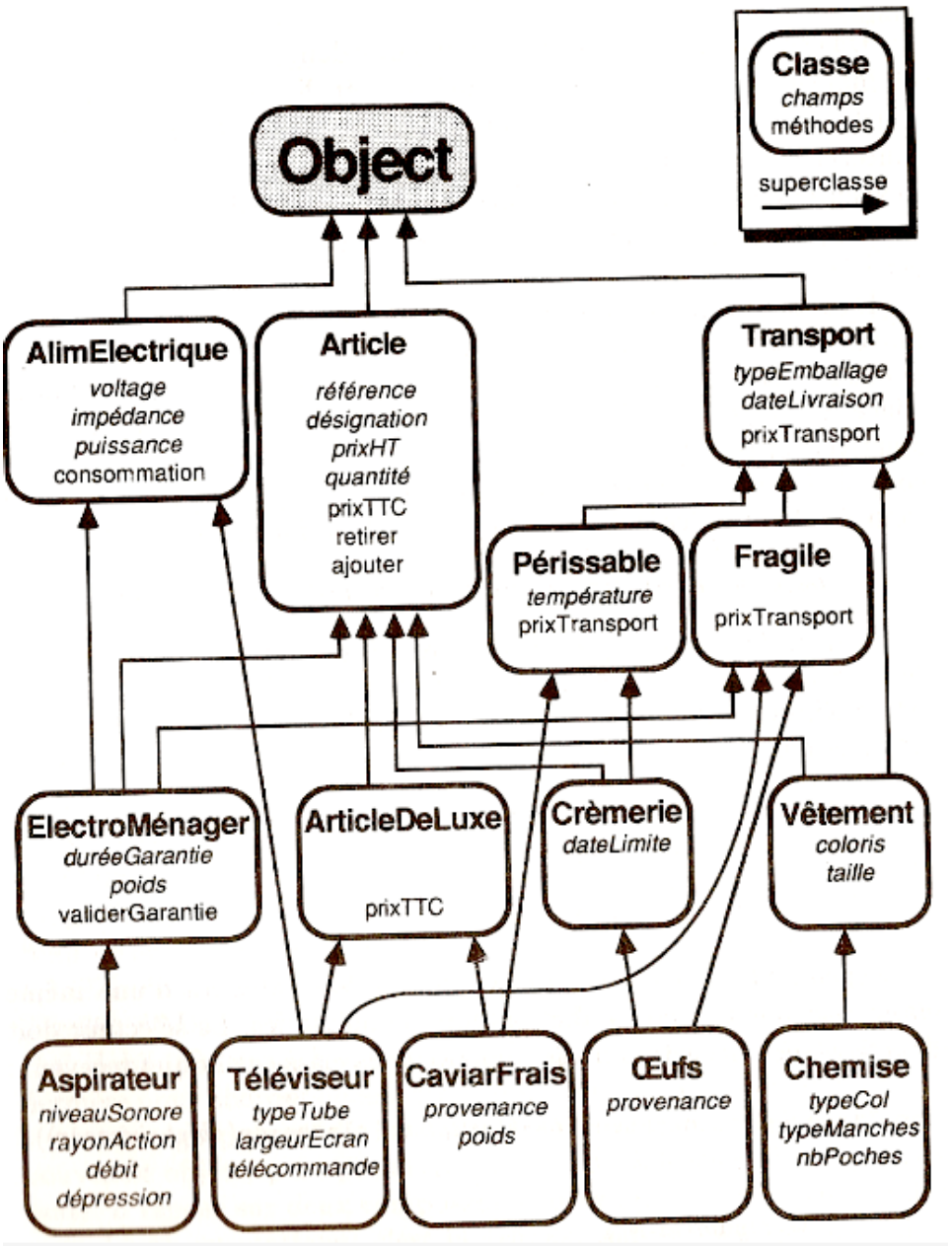


figure tirée du livre "Les Langages à Objets", G. Masini et al.

Héritage "Multiple"

- La hiérarchie des unités serait :

```
interface Transport {
    public int prixTransport();
    ...
}

interface Fragile extends Transport {    // héritage d'interface!
    public double prixTransport();
    ...
}

class Article {...}

class ArticleLuxe extends Article {...}

class Televiseur extends ArticleLuxe implements Fragile {
    public double prixTransport() {
    // Calcul effectif du prix transport de téléviseur
    }
```

```
  ...
}
```

- La classe `Televiseur` hérite des caractéristiques `ArticleLuxe` (et donc aussi `Article`), et elle doit réaliser la méthode `prixTransport()` "héritée".
- Plus encore : elle "hérite" aussi de l'interface `AlimElectrique`.

```
class Televiseur extends ArticleLuxe implements Fragile, AlimElectrique {
    public double prixTransport() {
       // Calcul effectif du prix transport televiseur
    }
    public double consommation() {
    // Calcul de la consommation de téléviseur
    }
  ...
}
```

- avec

```
interface AlimElectrique {
    int voltage = 120;
    public double consommation();
    ...
}
```

- `voltage` est ici une constante déclarée `final`.