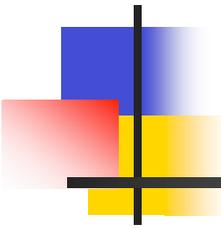


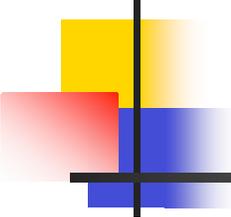
# Cours de programmation orientée objets



---

Salima Hassas

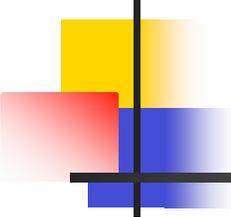
Université Claude Bernard-Lyon 1



# Plan du cours (1/2)

---

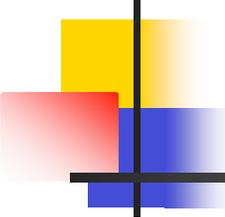
- Introduction
- Pourquoi la programmation orientée objets?
  - Quelques bons concepts
- Des Types de Données Abstraites (TDA) au Modèle Orienté Objets (MOO)
  - Exemples illustratifs
  - Que permet la programmation orientée objets de plus/ à la programmation par type de données abstraits
- Concepts de base de la programmation orientée objets



# Plan du cours (2/2)

---

- Concepts de base
  - Objets: classes et instances
  - Encapsulation
    - Protection des données et du code
  - Polymorphisme
    - surcharge
  - Héritage
    - Héritage simple et multiple
    - Réutilisation de code (spécialisation)
  - Relations entre classes
    - Association, agrégation, composition



# Les langages objets

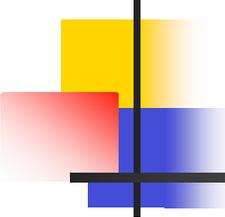
---

- Simula

- Premier langage objet, Oslo 1967
- Extension d'Algol 60 aux types abstraits
- Programmation évènementielle

- Eiffel

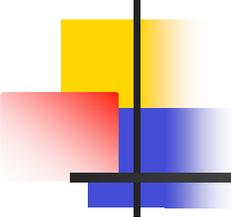
- Inspiré de Simula 67
- Préoccupations Génie-Logiciel
  - (Protections, pré-post conditions, héritage multiples, etc)



# Les langages objets

---

- Smalltalk
  - Envoi de messages(72), notion de meta-classe
  - Notion de tout objet
- C++ (86-87)
  - Extension de C à l'objet
- Java (94)
  - Compromis entre C++ et Smalltalk
  - Machine virtuelle



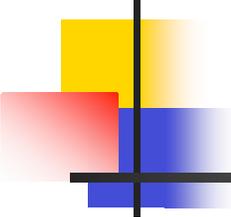
# Introduction

---

## Problèmes de génie logiciel

*« software is not enough »*

- Spécification
- Développement
- Protection
- Mise au point
- Maintenance
- Evolution



# Introduction

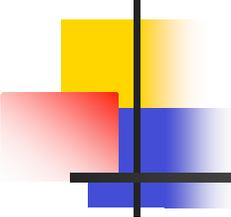
---

## Problèmes de génie logiciel

« *software is not enough* »

- Spécification
- Développement
- Protection
- Mise au point
- Maintenance
- Evolution

75% du coût du logiciel sont des coûts de maintenance

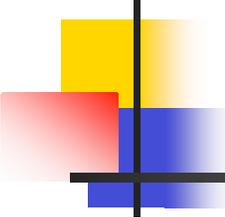


# Introduction

---

## Bons Concepts

- Modularité: diviser pour régner
- Protection: des données
- Généricité: partage et réutilisation
- Abstraction (de l'implantation): lisibilité



# Introduction

---

En programmation classique

Dualité des programmes /données

→ Priorité aux traitements

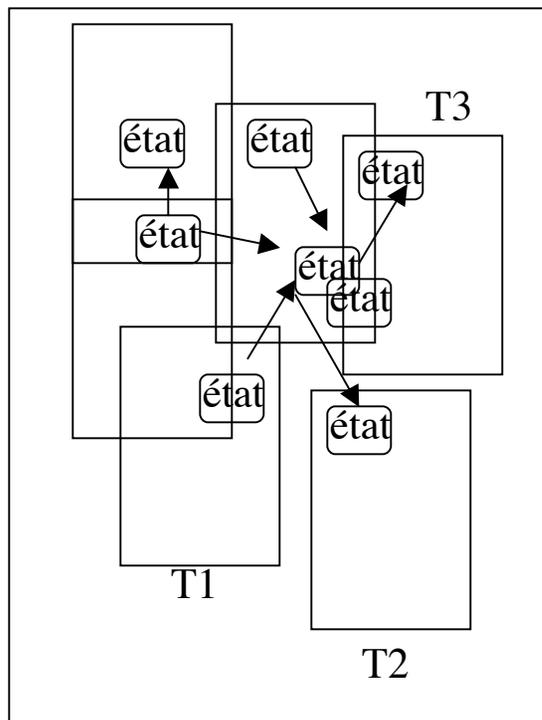
Or

Les données sont plus stables que les traitements

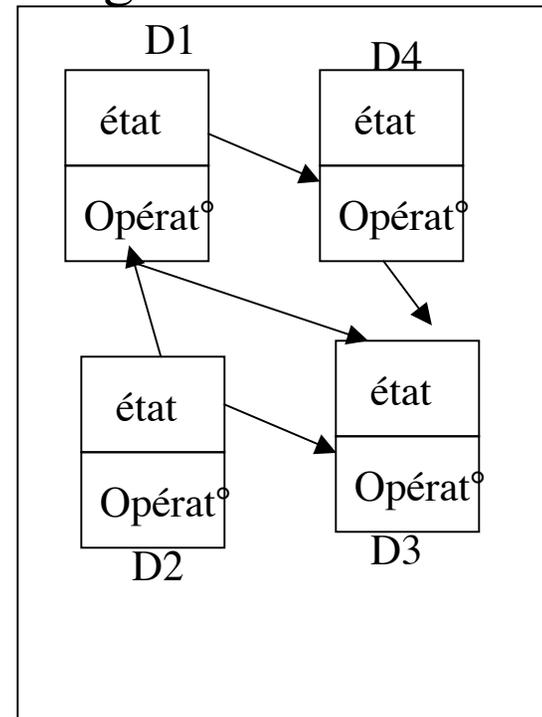
→ Procéder de manière inverse

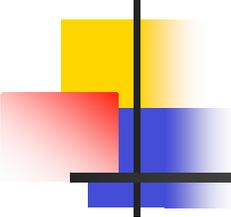
# Introduction

## Organisation/traitements



## Organisation/données

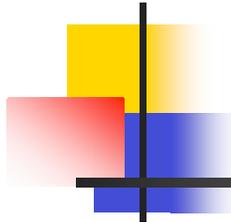




# Introduction

---

- Spécifier les données
  - Caractéristiques définissant la nature la données : état et opérations permettant la manipulation de la donnée
- Organiser l'application (traitements) autour des données
  - Approche guidée par les données (types abstraits de données)

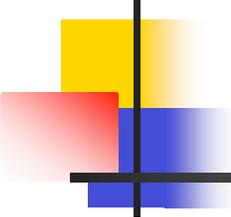


# Introduction

---

Premier pas vers la programmation orientée  
objets

les Types Abstrais de Données?



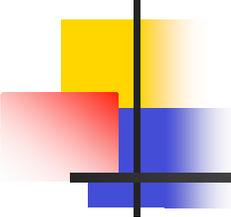
# Introduction

---

## Première solution: les TDA

*Ils apportent une réponse à*

- La modularité
- Protection: des données (+ ou -)
- Généricité: partage et réutilisation
- Abstraction (de l'implantation): lisibilité



# Introduction

---

## Première solution: les TDA

*Par exemple*

- TDA Liste, Pile, File, Arbre, ..etc
  - Définitions propres, modulaires, réutilisables, ...
  
- Mais !!!
  - Opérations: introduisant même concepts, noms différents
    - Insertion/suppression, affichage dans une liste, dans un arbre, ..etc
  - Données: pas de protection (en dur) : pas d'encapsulation
    - Règles de bonne programmation

# Introduction

## *Liste*

### *Inserer*

Liste x Element x Position -> Liste

### *Supprimer*

Liste x Position -> Liste

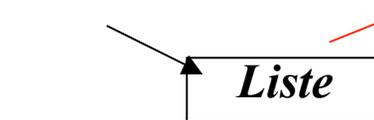
### *Afficher\_liste:*

Liste -> Liste

...

### *Inserer*

*Inserer dans une liste*



### *Supprimer*

## *Pile*

### *Empiler*

Pile x Element -> Pile

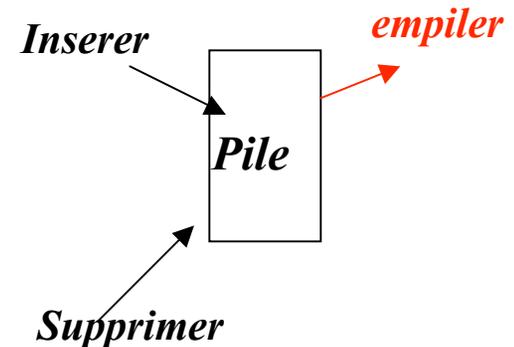
### *Dépiler*

Pile -> Pile

### *Afficher\_Pile:*

Pile -> Pile

...



*Opérations introduisant le même concept, devraient porter le même nom : plus de lisibilité, éviter la complexité lexicale*

# Introduction

- *Liste*

*Données*

*Tete: Cellule \**

*Suite: Cellule \**

*Opérations:*

*Inserer ...*

*supprimer:*

...

*Inserer(L, e, 1)*

*Liste*



- *Pour modifier la tête de liste : Inserer (L, e, 1) ou supprimer (L, 1)*  
*==> règle de conduite du bon programmeur !!*

# Introduction

- *Liste*

*Données*

*Tete: Cellule \**

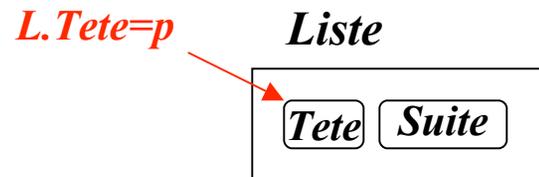
*Suite: Cellule \**

*Opérations:*

*Inserer ...*

*supprimer:*

...



- *Pour modifier la tête de liste : Inserer (L, e, 1) ou supprimer (L, 1)*

- *Mais :*

*L.Tete= p : tout a fait possible !!!*

- *On ne peut pas contrôler la modification des valeurs des données internes aux TDA,*
- *Un programmeur ne respecte pas forcément les règles de bonne programmation*

# Introduction

- *Liste*

*Données*

*Tete: Cellule \**

*Suite: Cellule \**

*Opérations:*

*Inserer ...*

*supprimer:*

...

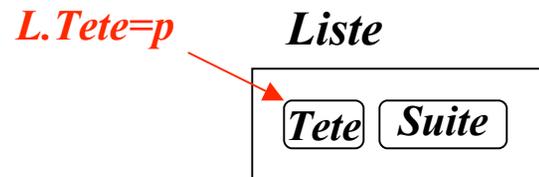
- *Pour modifier la tête de liste : Inserer (L, e, 1) ou supprimer (L, 1)*

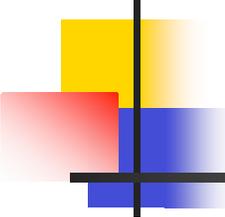
- *Mais :*

*L.Tete= p : tout a fait possible !!!*

- *On ne peut pas contrôler la modification des valeurs des données internes aux TDA,*
- *Un programmeur ne respecte pas forcément les règles de bonne programmation*

*Pas de protection effective des données : manque d'encapsulation*



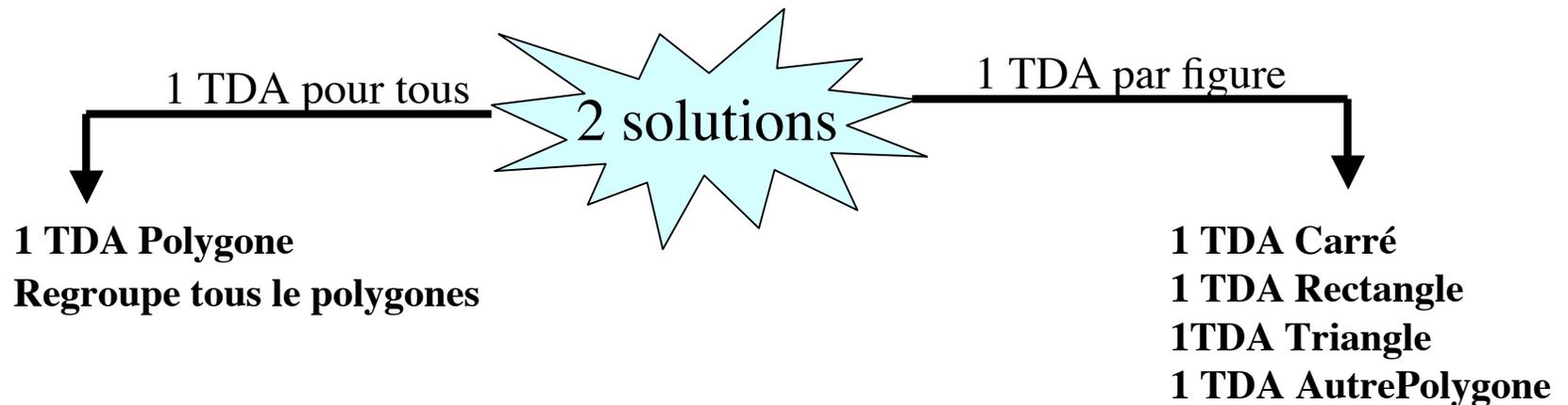


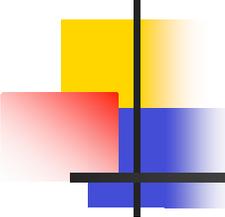
# Introduction

---

Première solution: les TDA

Programmer une bibliothèque de manipulation de figures géométriques: les polygones



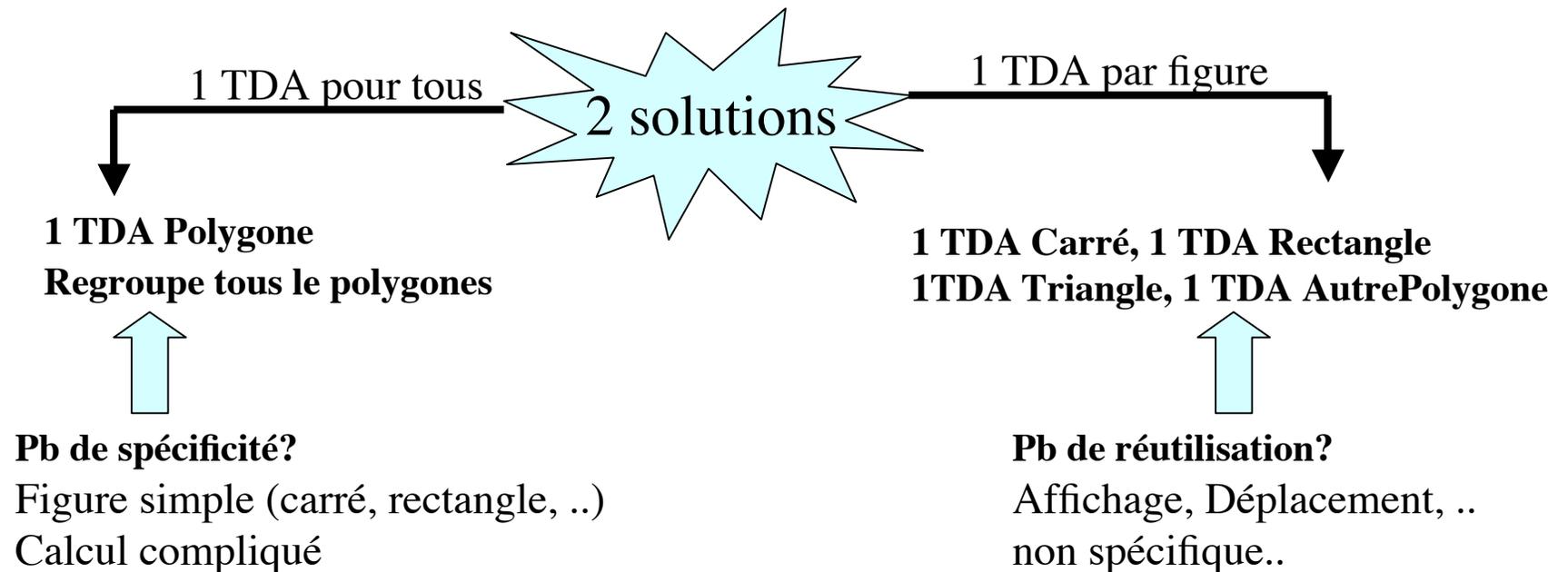


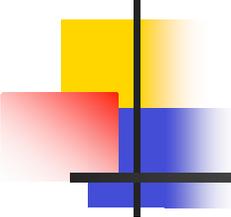
# Introduction

---

Première solution: les TDA

Programmer une bibliothèque de manipulation de figures géométriques: les polygones





# Introduction

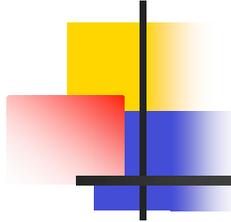
---

## Première solution: les TDA

*Il aurait été intéressant de pouvoir*

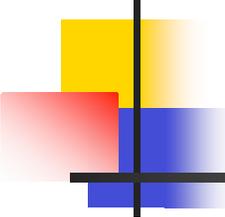
- Généraliser : représenter ensemble les points communs entre les différents TDA pour plus de possibilités de réutilisation
- Spécialiser : garder la spécificité de chaque TDA

 Le modèle orienté objets



# Le modèle orienté objets

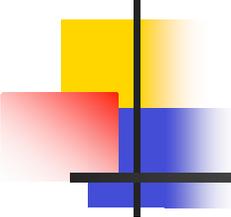
---



# Concepts de base

---

- Qu'est-ce qu'un objet?
  - Attributs et méthodes, interfaces et messages,
- Classes et instances
- Héritage : spécialisation/généralisation
- Polymorphisme et surcharge



# Concepts de base

## Qu'est ce qu'un objet?

- Un objet est un modèle de données avec
  - une partie déclarative: contenant des variables internes à l'objet, exprimant son état, appelés : *champs* ou *attributs*
  - une partie fonctionnelle, contenant un ensemble d'opérations appelées *méthodes*, exprimant le comportement de l'objet.

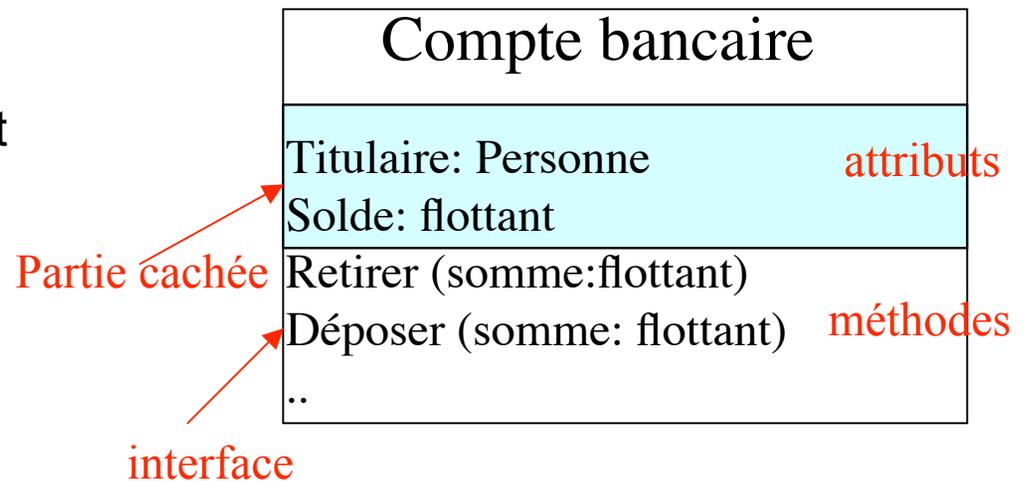
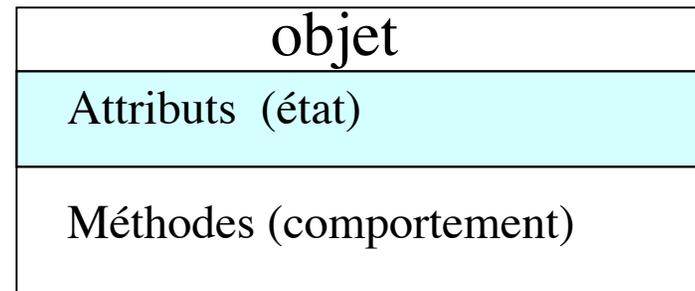
objet
Attributs (état)
Méthodes (comportement)

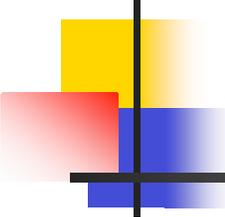
Compte bancaire	
Titulaire: Personne	<i>attributs</i>
Solde: flottant	
Retirer (somme:flottant)	
Déposer (somme: flottant)	<i>méthodes</i>
..	

# Concepts de base

## Qu'est ce qu'un objet?

- L'ensemble des attributs de l'objet ne sont accessibles que via les méthodes de celui-ci
  - Encapsulation et protection de données
- L'ensemble des méthodes définit **l'interface** à travers laquelle on peut agir sur l'objet (ou le manipuler)

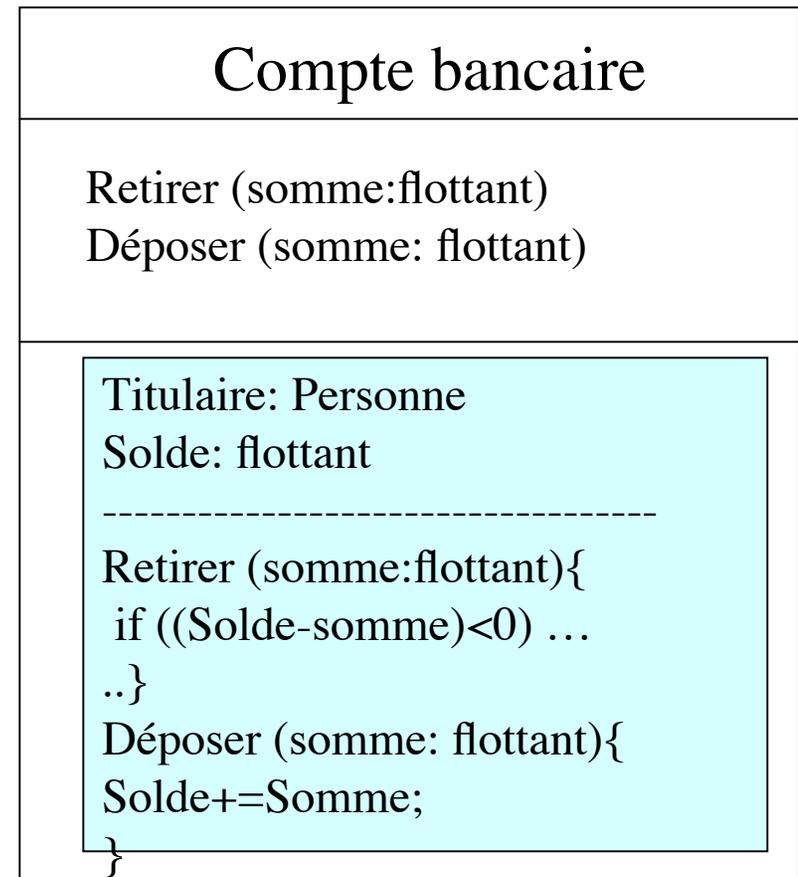




# Concepts de base

## Encapsulation

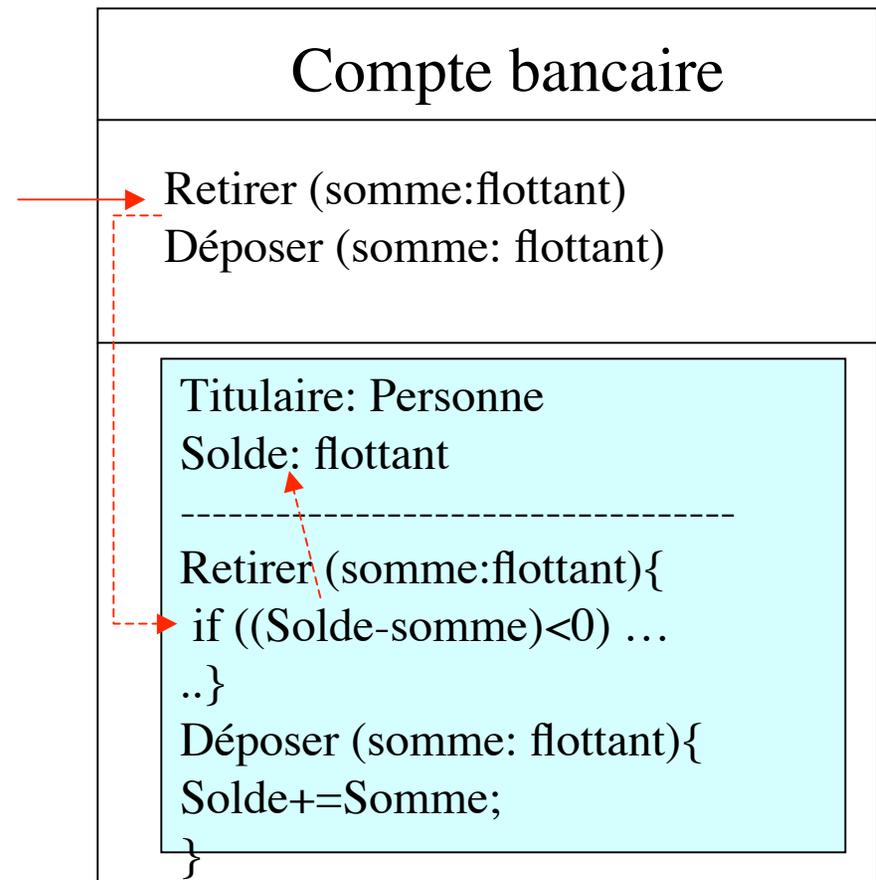
- Partie visible de l'extérieur d'un objet
  - opérations
- Partie invisible de l'extérieur
  - Données (Attributs)
  - Description du comportement des opérations



# Concepts de base

## Encapsulation

- Partie visible de l'extérieur d'un objet
  - opérations
- Partie invisible de l'extérieur
  - Données (Attributs)
  - Description du comportement des opérations



# Concepts de base

## Mode de fonctionnement?

- Envoi de messages entre objets:
  - Un message contient le nom d'une méthode, et la liste des arguments
  - La réponse à la réception d'un message consiste à exécuter la méthode associée

Compte bancaire	
Titulaire: Personne Solde: flottant	attributs
Retirer (somme:flottant) Déposer (somme: flottant)	méthodes
..	

CC

Titulaire: Dupont  
Solde: 2534

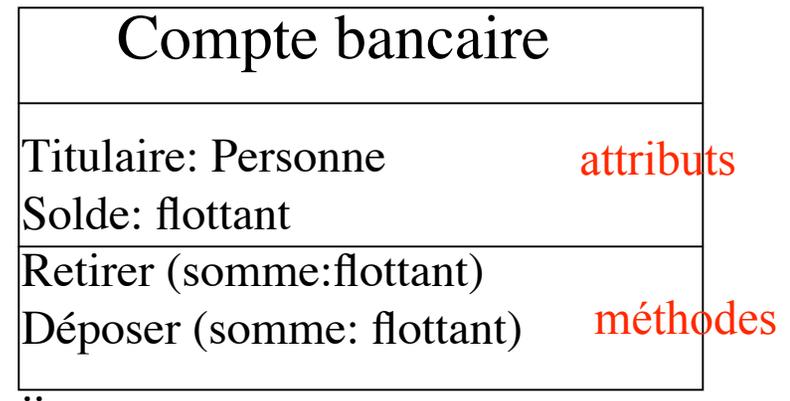
Envoi de message: Déposer (30)

# Concepts de base

## Mode de fonctionnement?

- Envoi de messages entre objets:
  - Un message contient le nom d'une méthode, et la liste des arguments
  - La réponse à la réception d'un message consiste à exécuter la méthode associée

Envoi de message: Déposer (30)



CC

Titulaire: Dupont  
Solde: 2534

↓ exécute

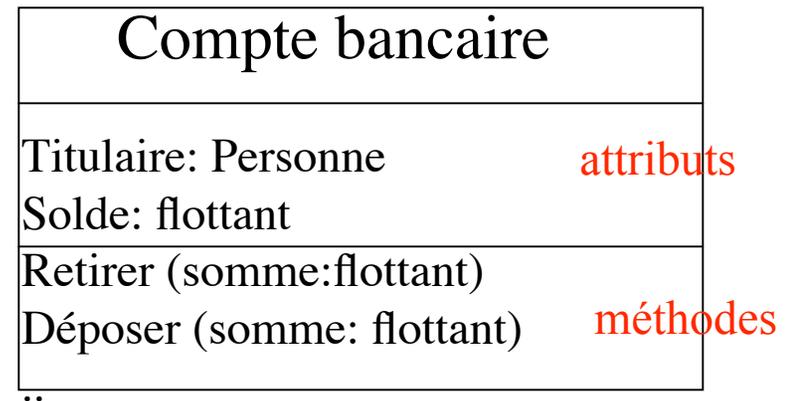
Déposer(somme<sup>30</sup>)  
{solde=solde+somme}

# Concepts de base

## Mode de fonctionnement?

- Envoi de messages entre objets:
  - Un message contient le nom d'une méthode, et la liste des arguments
  - La réponse à la réception d'un message consiste à exécuter la méthode associée

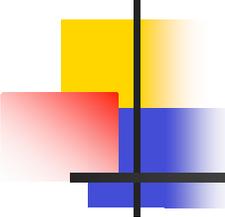
Envoi de message: Déposer (30)



CC



exécute  
Déposer(somme) {solde=solde+somme}



# Concepts de base

---

## Classes et instances

- Une classe définit la description d'une famille d'objets ayant la même structure (partie déclarative) et le même comportement (partie procédurale)
  - Description d'un concept

Compte bancaire	
Titulaire: Personne	attributs
Solde: flottant	
Retirer (somme:flottant)	
Déposer (somme: flottant)	méthodes

..

# Concepts de base

## Classes et instances

- Une classe définit la description d'une famille d'objets ayant la même structure (partie déclarative) et le même comportement (partie procédurale)
  - Description d'un concept
- Une instance correspond à un exemplaire concret de la classe, avec des valeurs d'attributs définis (l'objet)
  - Objet concret créé à partir du moule fourni par la définition de classe (instanciation)

Compte bancaire	
Titulaire: Personne Solde: flottant	attributs
Retirer (somme:flottant) Déposer (somme: flottant)	méthodes

..

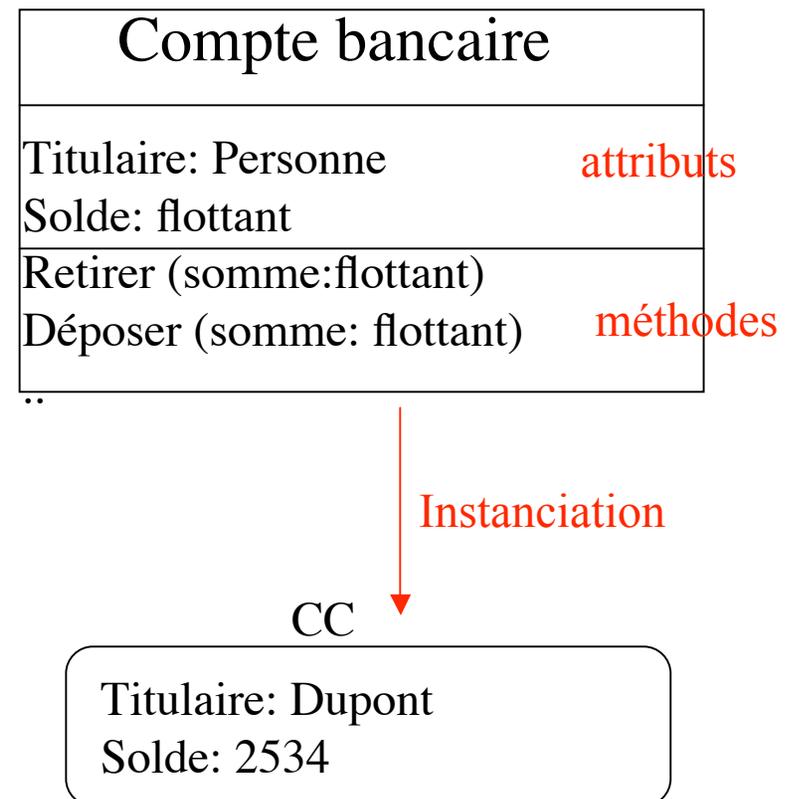
CC

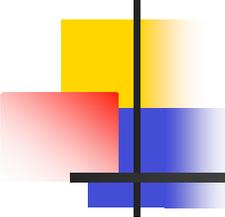
Titulaire: Dupont  
Solde: 2534

# Concepts de base

## Classes et instances

- Une classe définit la description d'une famille d'objets ayant la même structure (partie déclarative) et le même comportement (partie procédurale)
  - Description d'un concept
- Une instance correspond à un exemplaire concret de la classe, avec des valeurs d'attributs définis (l'objet)
  - Objet concret créé à partir du moule fourni par la définition de classe (instanciation)





# Concepts de base

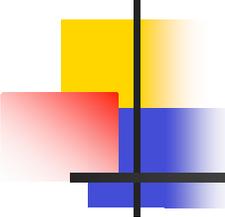
## Classes et instances

- Description du concept : Carré
  - La classe CARRE
- Un carré contient : *attributs*
  - 4 cotés
  - 4 sommets
  - Une liste de sommets
  - Une longueur de côté
- On peut effectuer les opérations suivantes sur un carré *méthodes*
  - Calcul du périmètre
  - Calcul de la surface
  - Affichage, translation, rotation/sommet, rotation/axe

### La classe CARRE

Nombre de coté: 4  
Nombre de sommets: 4  
Liste de sommets  
Longueur cote:

Périmètre:  $4 * \text{Longueur cote}$   
Surface:  $\text{Longueur cote}^2$   
Affichage() ...  
Rotation (sommet)..  
Rotation (Axe) ..



# Concepts de base

## Classes et instances

- Instanciation d'un objet de la classe Carré
  - L'objet Carré1
- Carré1 a des valeurs d'attributs définis :
  - 4 cotés
  - 4 sommets
  - Une liste de sommets définie:  
(0,0),(0,4), (4,0), (4,4)
  - Une longueur de coté: 4
- Les opérations sont celles de la classe Carré
  - Calcul du périmètre
  - Calcul de la surface
  - Affichage, translation, rotation/sommet, rotation/axe

### L'objet Carré1

Nombre de coté: 4

Nombre de sommets: 4

Liste de sommets

Longueur cote: 4

attributs

Périmètre:  $4 * \text{Longueur cote}$

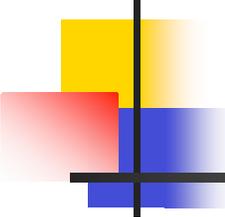
Surface:  $\text{Longueur cote}^2$

Affichage

Rotation (sommet)

Rotation (Axe) ..

méthodes



# Concepts de base

## Classes, sous classes et instances

La classe CARRE
Nombre de coté: 4 Nombre de sommets: 4 Liste de sommets Longueur cote:
Périmètre: $4 * \text{Longueur cote}$ Surface: $\text{Longueur cote}^2$ Affichage Rotation (sommet) Rotation (Axe) ..

La classe Rectangle
Nombre de coté: 4 Nombre de sommets: 4 Liste de sommets Longueur Largeur
Périmètre: $2 * (\text{Longueur} + \text{Largeur})$ Surface: $\text{Longueur} * \text{Largeur}$ Affichage Rotation (sommet) Rotation (Axe) ..

La classe Triangle
Nombre de coté: 3 Nombre de sommets: 3 Liste de sommets Base Hauteur
Périmètre: Surface: $\text{Base} * \text{Hauteur} / 2$ Affichage Rotation (sommet) Rotation (Axe) ..

# Concepts de base

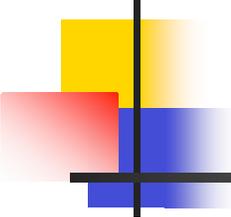
## Classes, sous classes et instances

Redondance

La classe CARRE
Nombre de coté: 4 Nombre de sommets: 4 Liste de sommets Longueur cote:
Périmetre: $4 * \text{Longueur cote}$ Surface: $\text{Longueur cote}^2$
Affichage Rotation (sommets) Rotation (Axe) ..

La classe Rectangle
Nombre de coté: 4 Nombre de sommets: 4 Liste de sommets Longueur Largeur
Périmetre: $2 * (\text{Longueur} + \text{Largeur})$ Surface: $\text{Longueur} * \text{Largeur}$
Affichage Rotation (sommets) Rotation (Axe) ..

La classe Triangle
Nombre de coté: 3 Nombre de sommets: 3 Liste de sommets Base Hauteur
Périmetre: Surface: $\text{Base} * \text{Hauteur} / 2$
Affichage Rotation (sommets) Rotation (Axe) ..

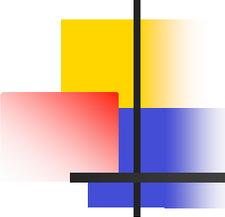


# Concepts de base

---

## Classes, sous classes et instances

- Organisation des objets en hiérarchies
  - A la racine une classe regroupant les propriétés communes à toutes les autres classes
    - ==> classe mère (super classe)
  - Descendre dans la hiérarchie par spécialisation ( plus de propriétés communes)
    - ==> création d'une sous classe
  - Réitérer le processus jusqu'à ce qu'il n'y ait plus de classes ayant des propriétés communes
    - ==> classes concrètes

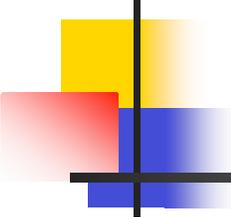


# Concepts de base

---

## **Classes, sous classes et instances**

- La hiérarchie des classes définit le graphe d'héritage.
  - Aux niveaux supérieurs, on dispose des classes abstraites
  - Au niveau le plus bas : les classes concrètes utilisées pour l'instanciation
  
- Récupération des propriétés (attributs, méthodes) au niveau le plus bas grâce à l'héritage
  - Héritage : parcours du graphe d'héritage



# Concepts de base

## Classes, sous classes et instances

- **Définitions**

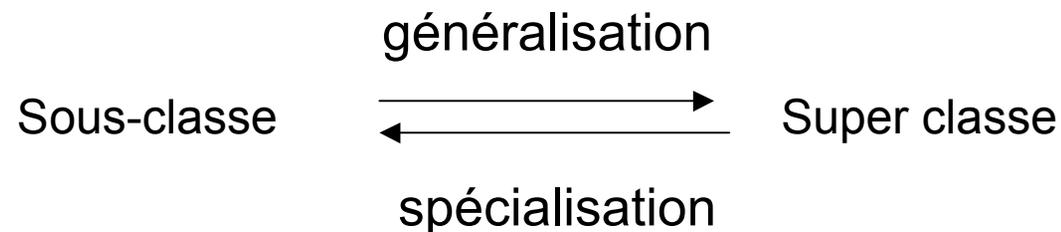
- **Classe/instance**

*La classe* détient la liste des attributs et la liste des méthodes.

*L'instance*, créée à partir de la classe, ne détient que la liste des valeurs d'attributs

- **Classe/sous-classe**

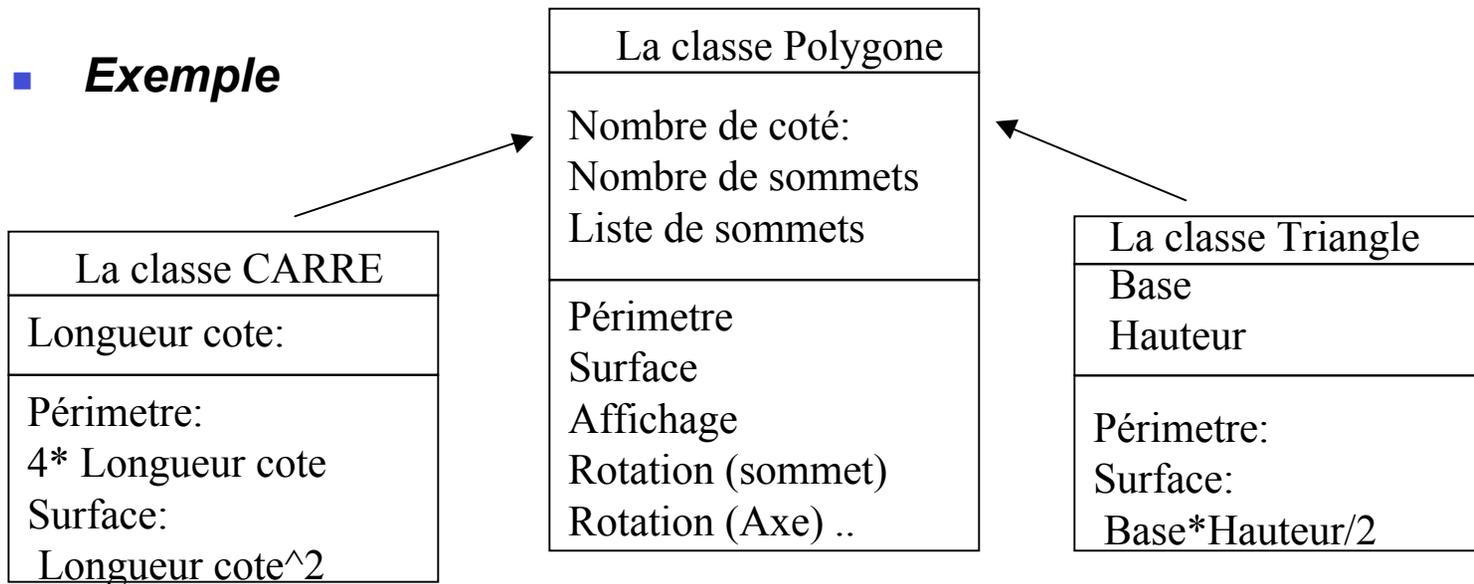
*Une classe* contient un ensemble de propriétés (attributs, méthodes) à partir desquelles on peut définir des *sous-classes* plus spécifiques, complétant les propriétés de la *classe mère*.



# Concepts de base

## Classes, sous classes et instances

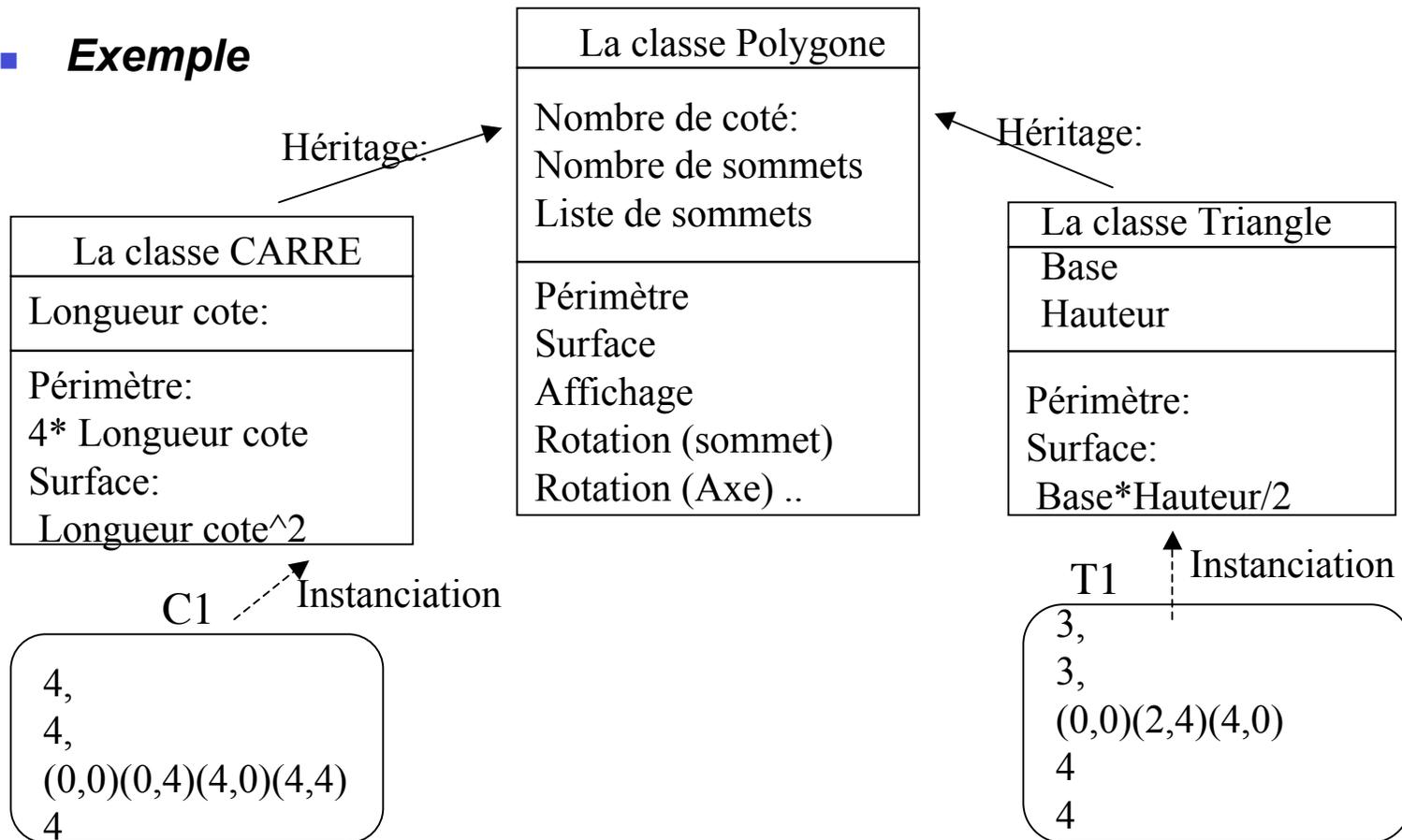
- **Exemple**

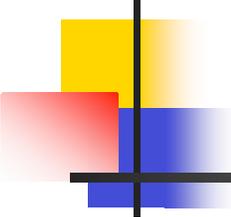


# Concepts de base

## Classes, sous classes et instances

### ■ Exemple





# Concepts de base

## Classes, sous classes et instances

- **Définitions**

### Spécialisation d'une classe



#### Par enrichissement

- *Définition d'attributs et/ou de méthodes supplémentaires*

*Ex: base, hauteur dans la classe Triangle*

#### Par substitution

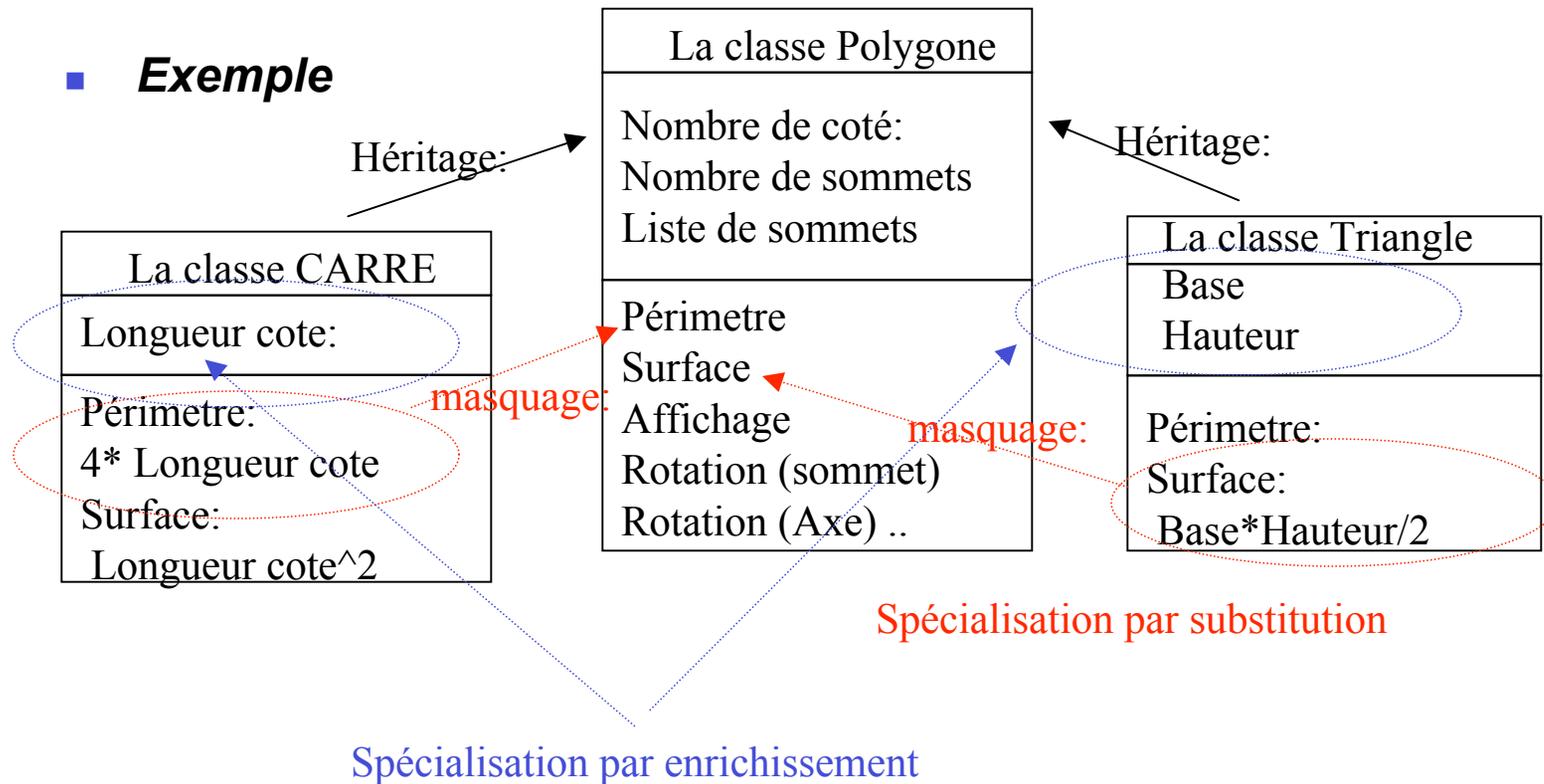
- *Définition d'une nouvelle méthode héritée*  
- *Masquage de la méthode héritée*

*Ex: surface dans la classe Triangle*

# Concepts de base

## Classes, sous classes et instances

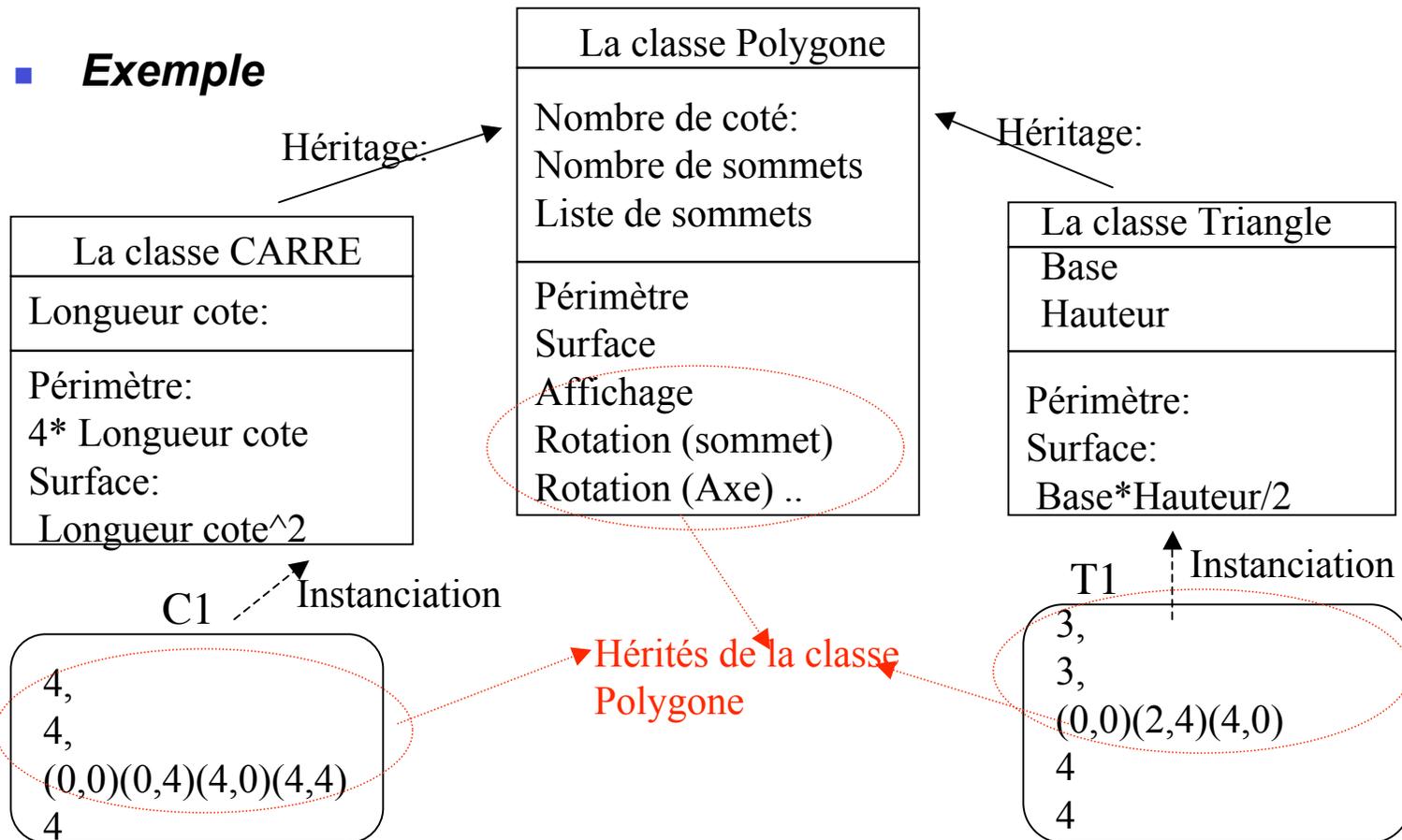
### ■ Exemple



# Concepts de base

## Classes, sous classes et instances

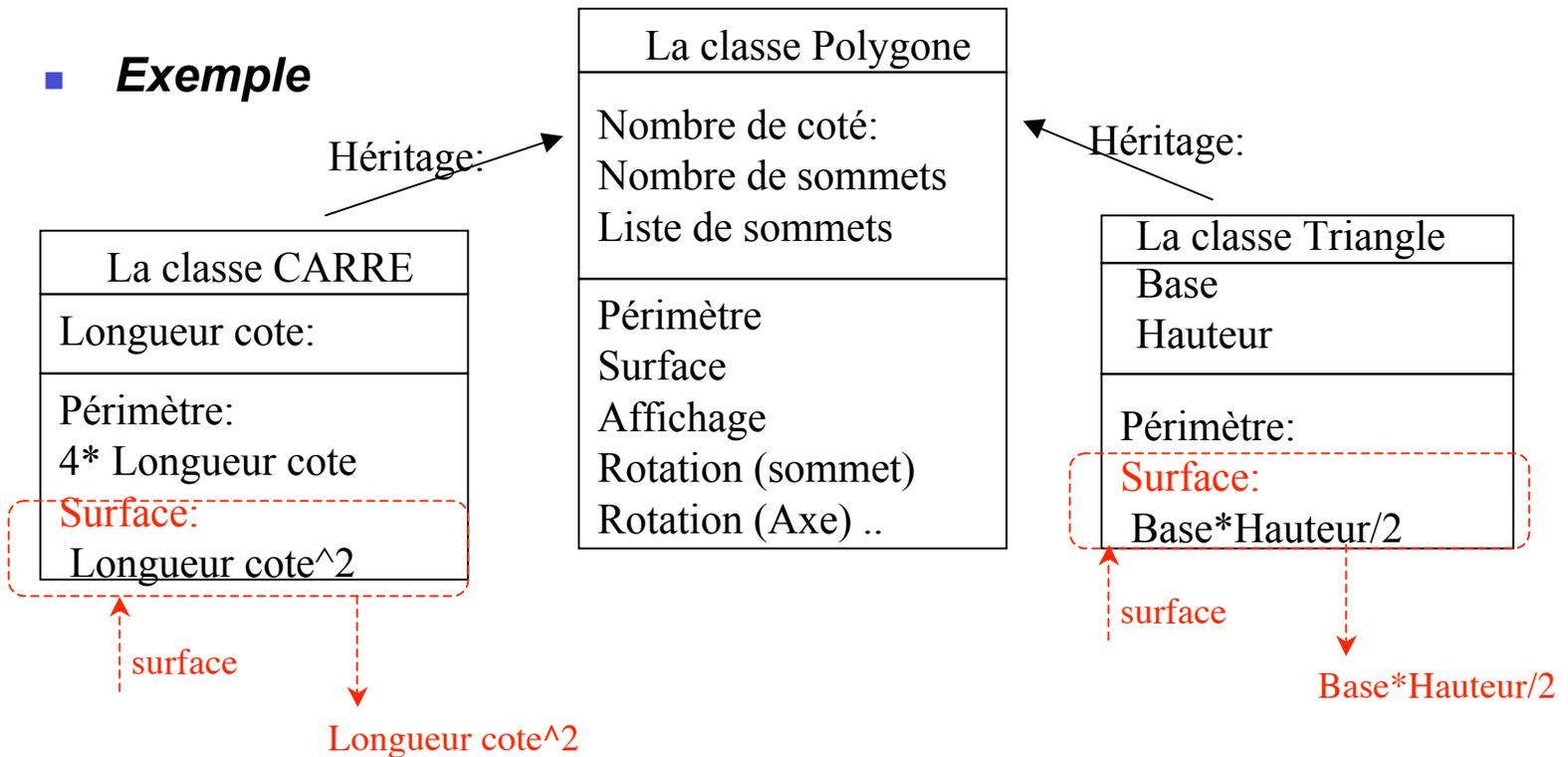
### ■ Exemple



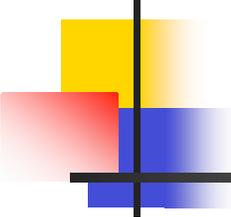
# Concepts de base

## Polymorphisme

### ■ Exemple



Même nom de méthode (surface), mais code différent selon la classe qui reçoit le message

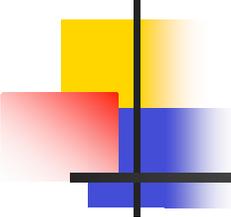


# Concepts de base

---

## Généricité

- **Méthode générique**
  - Une méthode dont le code est indépendant du type de ses arguments
  - Ex: insertion dans une pile, parcours d'un arbre, etc
- **Classe Générique**
  - Classe dont toutes les méthodes sont génériques
  - Exemple:
    - Les classes conteneurs : classe Liste, classe arbre, classe graphe, etc
    - Les opérations de manipulation de ces données est indépendant du type des éléments qu'elles contiennent
  - Notation (C++) : Nom-classe <T1, T2, ..>
  - Ex: Arbre<T>, Arbre<entier>, Liste<Arbre<caractère>>



# Concepts de base

---

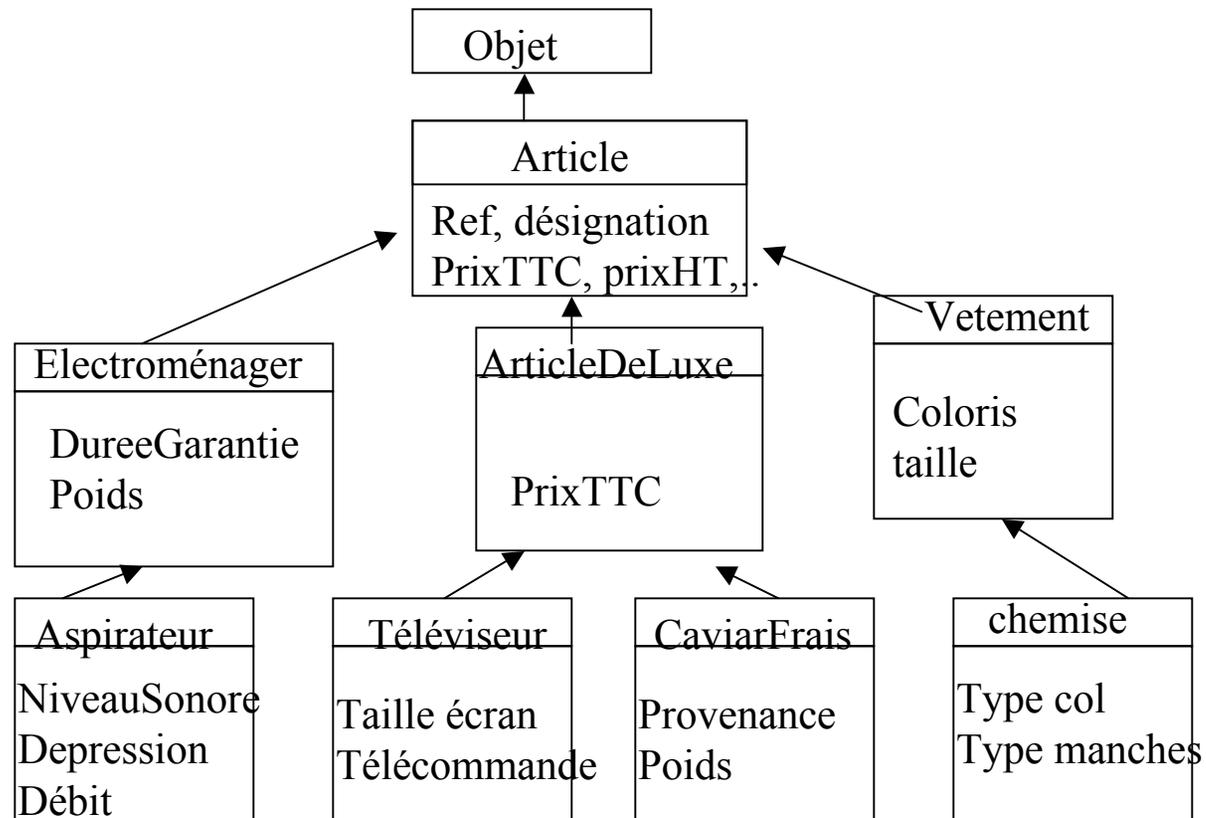
## Partage d'information: Héritage

La représentation graphique de la hiérarchie des classes définit le graphe d'héritage

- ***Héritage simple***
  - Chaque sous-classe admet une et une seule classe mère (super classe)
  - Graphe d'héritage = arbre dont la racine est la super classe la plus générale
  - Relation d'héritage = relation d'ordre total => pas de conflits

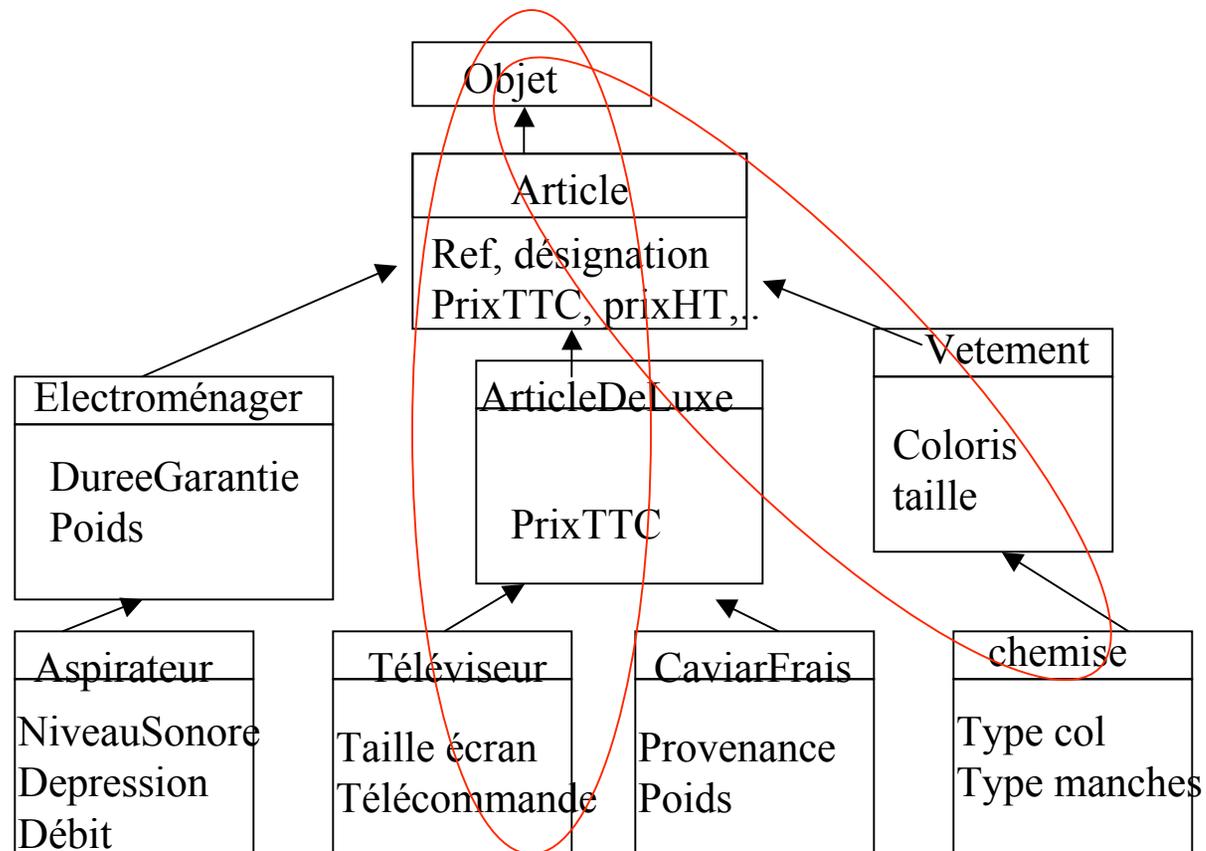
# Concepts de base

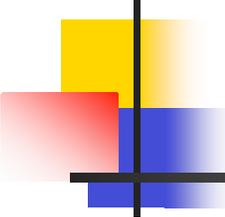
## Héritage simple



# Concepts de base

## Héritage simple





# Concepts de base

---

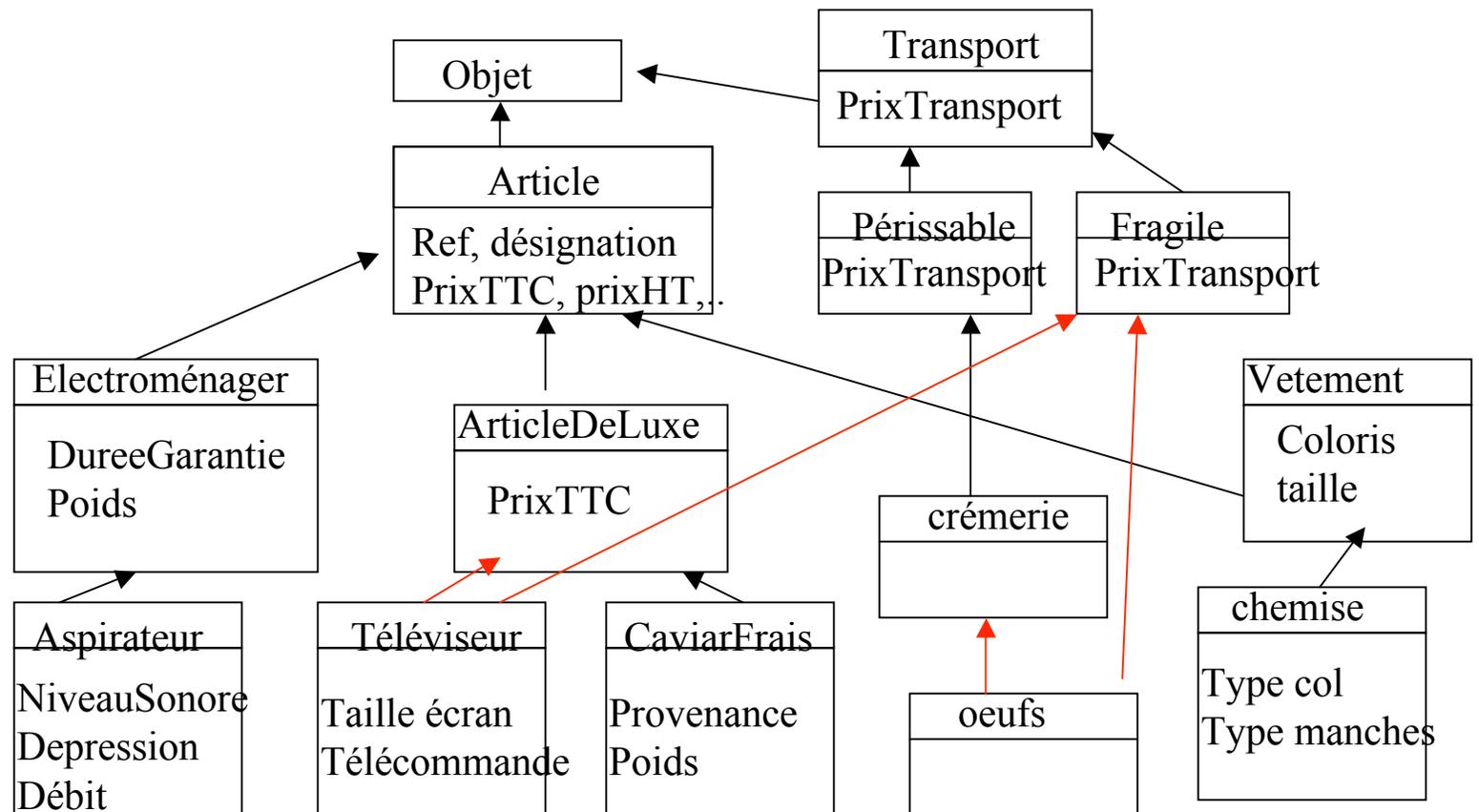
## Partage d'information: Héritage

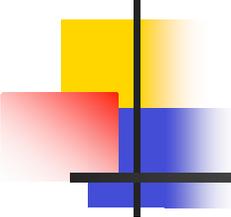
La représentation graphique de la hiérarchie des classes définit le graphe d'héritage

- ***Héritage multiple***
  - *Une sous classe* peut avoir plus d'une super classe
  - Graphe d'héritage = graphe orienté sans circuit
  - Héritage = union des propriétés (attributs, méthodes) de ses super classes
  - Relation d'héritage = relation d'ordre partiel => classes non comparables

# Concepts de base

## Héritage multiple





# Concepts de base

---

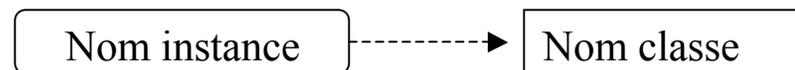
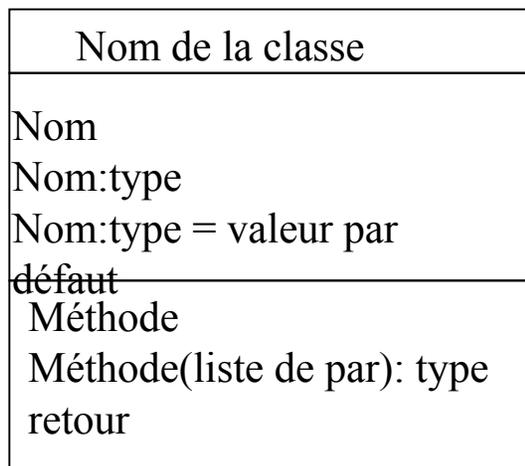
## Partage d'information: Héritage

Héritage multiples et situations conflictuelles

- **Conflit**
  - *Propriétés homonymes appartenant à des classes de hiérarchie d'héritage différentes (classes non comparables)*
  - Problème : trouver un ordre de parcours du graphe d'héritage qui satisfasse au mieux les intentions du concepteur de la hiérarchie
  - Critères: ordre , multiplicité, modularité
  - Dans les langages : l'héritage multiple est géré par l'utilisateur
    - En C++ (explicite lors de la déclaration des classes),
    - En Java: héritage simple + utilisation des interfaces

# Concepts de base

## Notation graphique



ou

Nom instance : nom de la classe

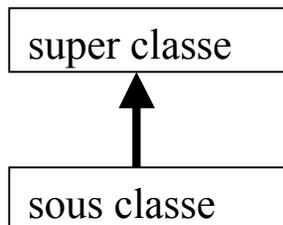
ou

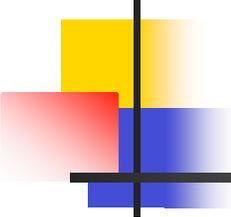
Nom instance : nom de la classe

Attribut1=valeur 1

Attribut2=

Relation d'héritage





# Concepts de base

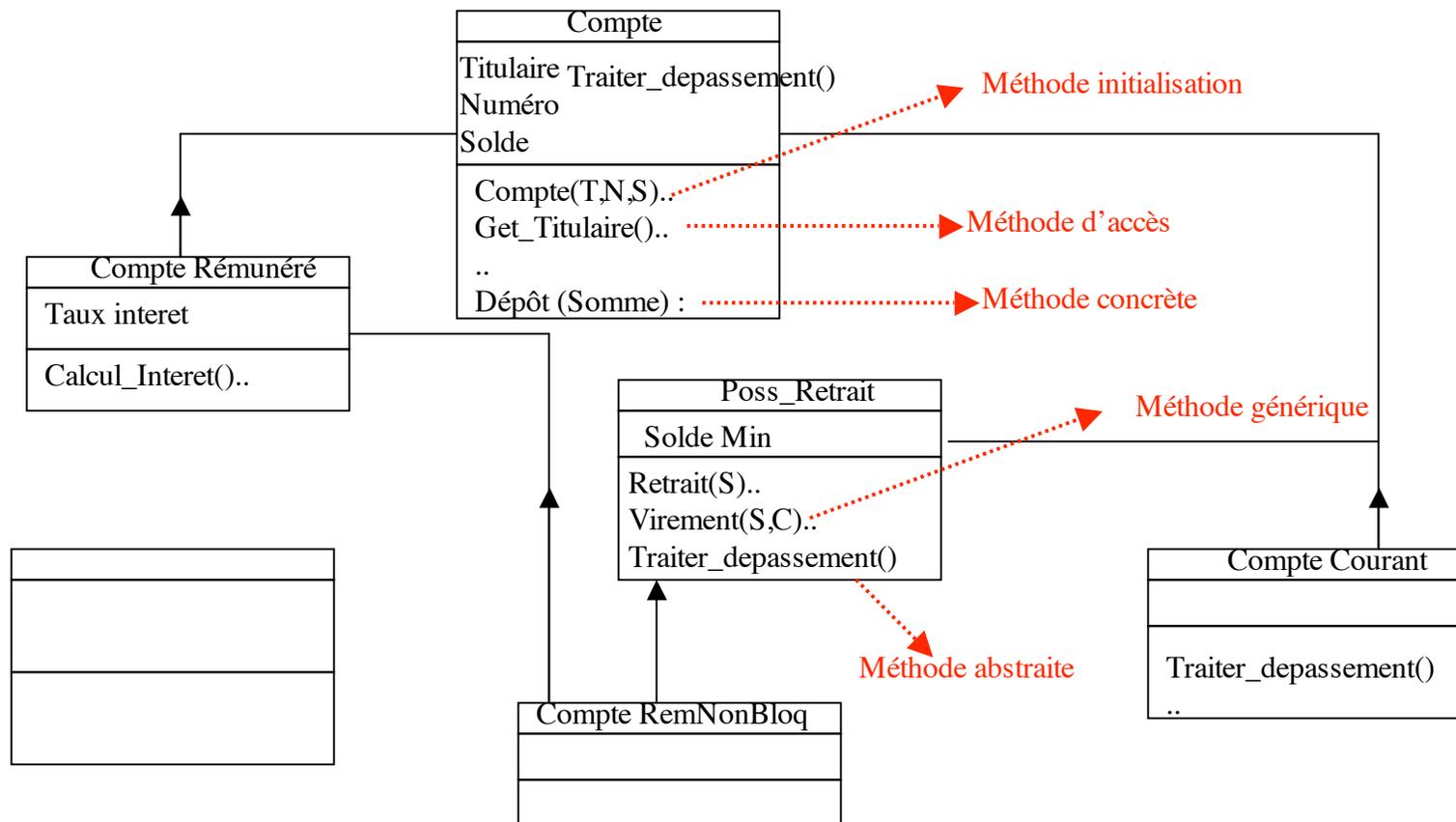
---

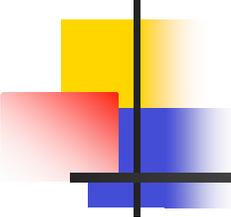
## **Classes concrètes/ Classes abstraites**

- **Classe abstraite**
  - Son rôle est le regroupement de propriétés communes
  - Doit apparaître comme un nœud au niveau de la hiérarchie
  - N'admet pas d'instances concrètes
  - Ex: classe humain, classe figure géométrique
  
- **Classe concrète**
  - Doit apparaître comme feuille au niveau de la hiérarchie
  - Elle est destinée à être instanciée
  - Ex: classe homme, femme, classe rectangle

# Concepts de base

## Exemple



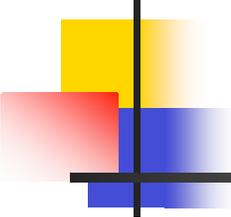


# Concepts de base

---

## Différents types de méthodes

- Méthode d'initialisation
  - Exécutée juste après la création de l'objet pour initialiser les attributs de l'objet (valeurs par défaut par exemple)
- Méthode d'accès: accesseur
  - Permet l'accès en lecture (get) ou en écriture (set) aux attributs de l'objet
- Méthode générique
  - Définie dans une classe abstraite et fait appel à des méthodes spécifiques définies dans les sous classes
- Méthode abstraite (virtuelle)
  - Définie dans une classe abstraite (sans code) et doit être re-définie obligatoirement dans toutes les sous classes directes
- Méthode concrète
  - Toute autre méthode
- Méthode de destruction (optionnelle) doit être exécutée juste avant la disparition de l'objet



# Concepts de base

---

## Variables SELF et SUPER

- Self (This en C++, java)
  - Fait référence à l'objet lui-même, dans le corps des méthodes de l'objet.
  - Ex: `PrixTTC()= Self.PrixNet() + Self. PrixTransport()`
- Super
  - Fait référence à la classe contenant la super méthode recherchée
  - Super méthode=la première méthode (avec le nom recherché) rencontrée dans la hiérarchie de l'objet
  - En java:utilisé juste par les méthodes d'initialisation
  - Ex: `PrixTransport()= Super.PrixTransport ()*20%`

# Concepts de base

## Relations Conceptuelles entre Objets

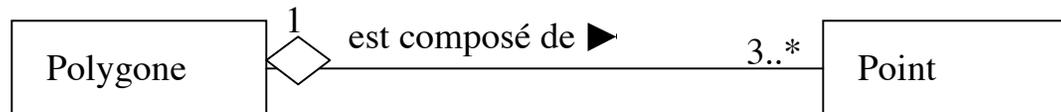
- Association
  - Relations structurelles entre objets (durables) dans le temps
  - Chaque objet joue un rôle dans l'association, qui peut-être noté lors de la définition de l'association
  - Association peut-être nommée
  - Cardinalité peut être indiquée



# Concepts de base

## Relations Conceptuelles entre Objets

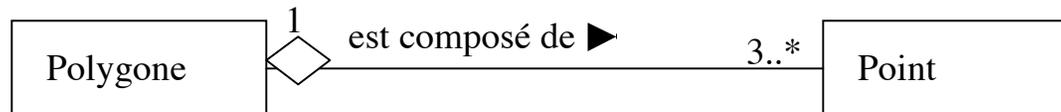
- Association particulière (Agrégation ou composition)
  - Un objet fait partie (ou compose physiquement) un autre objet
- Exemple
  - La tête fait partie du corps
  - Le dossier fait partie (compose) la chaise
- Notation

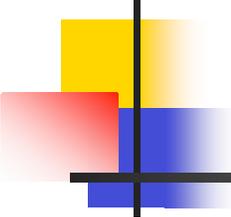


# Concepts de base

## Implémentation des associations

- L'association d'une classe A à une classe B est implantée par la définition d'un attribut de type A dans la classe B.
- Le nom de l'attribut est le rôle (s'il est défini) dans l'association
- La tête d'une association de cardinalité multiple définit un attribut de type collection
  - Exemple
    - Attribut: Liste de points dans une classe Polygone



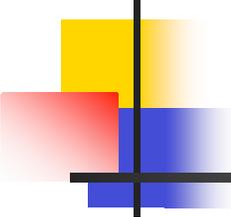


# Concepts de base

---

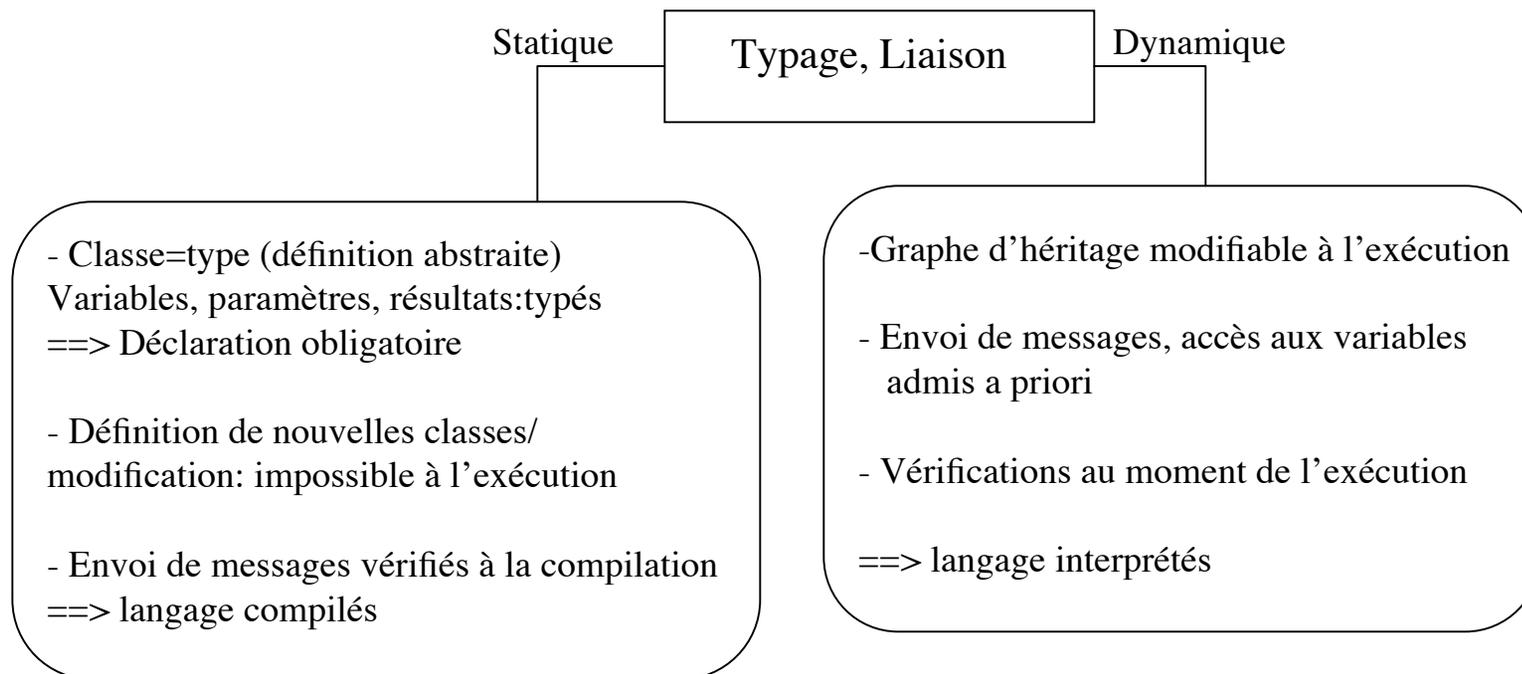
## Héritage, Typage et Liason

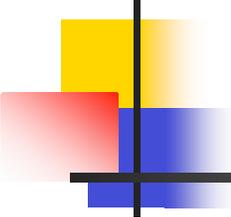
- **La liaison** définit le rattachement d'une méthode à un objet. Elle peut être:
  - statique : la liaison entre le message et la méthode se fait sur la base du type *déclaré* de la variable
    - Ex: *Les librairies en C où le code est recherché à la compilation*
  - dynamique: la liaison message-méthode s'effectue sur la base du type de la *valeur* que prend la variable à l'exécution
    - *Dans o. **surface()** La variable peut référencer un objet de la classe Polygone, de la classe Carré, ou de la classe Triangle au cours de son existence*



# Concepts de base

## Héritage, Typage et Liaison





# Concepts de base

## Héritage, Typage et Liaison

