Année 2020-2021

Polytech'Paris-Sud – 4ème année

Département Informatique

« Compléments Objets »

Frédéric Voisin

Département Informatique

Polytech Paris-Saclay

Contenu du module

Pré-requis:

Bases de Java (ou C++) : classes, interfaces, exceptions, packages…

COURS :

L'approche « objets » dans différents langages (Java, C++, Scala)

Aspects statiques / dynamiques; surcharge / redéfinition

Focus sur certaines opérations importantes (égalité, relations d'ordre, ...)

Bon usage de l'héritage ; « Design Patterns »

Généricité en Java et en Scala

Lambda-expressions et aspects fonctionnels (Java/Scala/OCaml)

TDs/TPs :

« conception abstraite » (diagrammes de classes ; exercices « théoriques »)

Quelques TPs

Dans ce module on s'intéresse aux principes, pas aux détails syntaxiques.

Contenu du module (suite)

Contrôle des connaissances (sous réserve)

- un examen
- « mini-projet » : TPs à rendre. Interros ou petits DM.

Cours associés:

« Outils de Programmation et C++ » (ET4)

Programmation C++ avancée (ET5)
C# (ET55)

Java/J2EE *vs* C#/.Net, WebServices et SOA (ET4/5)

Programmation Fonctionnelle (ET4)

Bibliographie

- C. S. Horstmann et G. Cornel : *Au cœur de Java 2*, Vol. 1 « Notions Fondamentales », Campus Press
- A. Hejlsberg, M. Torgensen, S. Wiltamuth et P. Golde : *The C# Programming Language*, Addison-Wesley
- M. Odersky & *alli* : *Programming in Scala*, Artima, 3rd edition, 2016
- P. Chianuso & R. Bjarnason: *Functional Programming in Scala*, Manning Pub., 2014
- C. S. Horstmann : *Scala for the Impatient, Addison-Wesley 2016*
- B. Stroustrup : *the C++ programming Language*, Addison Wesley, 2013 *(avec une partie sur la norme C++11, sachant que C++17 vient de sortir)*
- *S. Meyers : Effective Modern C++*, O'Reilly, 2014
- E. Gamma et *alli* : *Design Patterns : elements of reusable Object-Oriented Software*, Addison Wesley

Révisions conseillées

- UML (pour décrire les hiérarchies de classes en TD....)
- Les exceptions (voir poly sur site. *Travail personnel*)
- Interfaces

  `Comparable`, `Comparator`, `Collection`, `Stream` (Java8) ...

  Les interfaces « riches » en Java 8

  Les interfaces « fonctionnelles » : `Predicate`, `Function`, ...
- Les méthodes particulières

  `clone`, `hashCode`, `equals`, `compareTo`, `compare`
- Les paquetages

Pour accompagner vos révisions...

Devoir Maison, **pour lundi prochain,** à faire individuellement

- Dessiner les liens qui relient les classes et interfaces ci-dessous, en distinguant les relations d'héritage et celles d'implémentation d'interfaces. Quand l'unité est générique, indiquez son paramètre.

  `Collection, List, Set, SortedSet, HashSet, Iterable, TreeSet, AbstractSet, Vector, Queue, Stack, Deque, AbstractList, AbstractSequentialList, LinkedList  AbstractCollection`

- Un `TreeSet` peut nécessiter une méthode de comparaison pour ajouter un élément à un ensemble. Pourquoi n'y a-t-il pas de relation avec `Comparable` ? Qu'en déduire ?

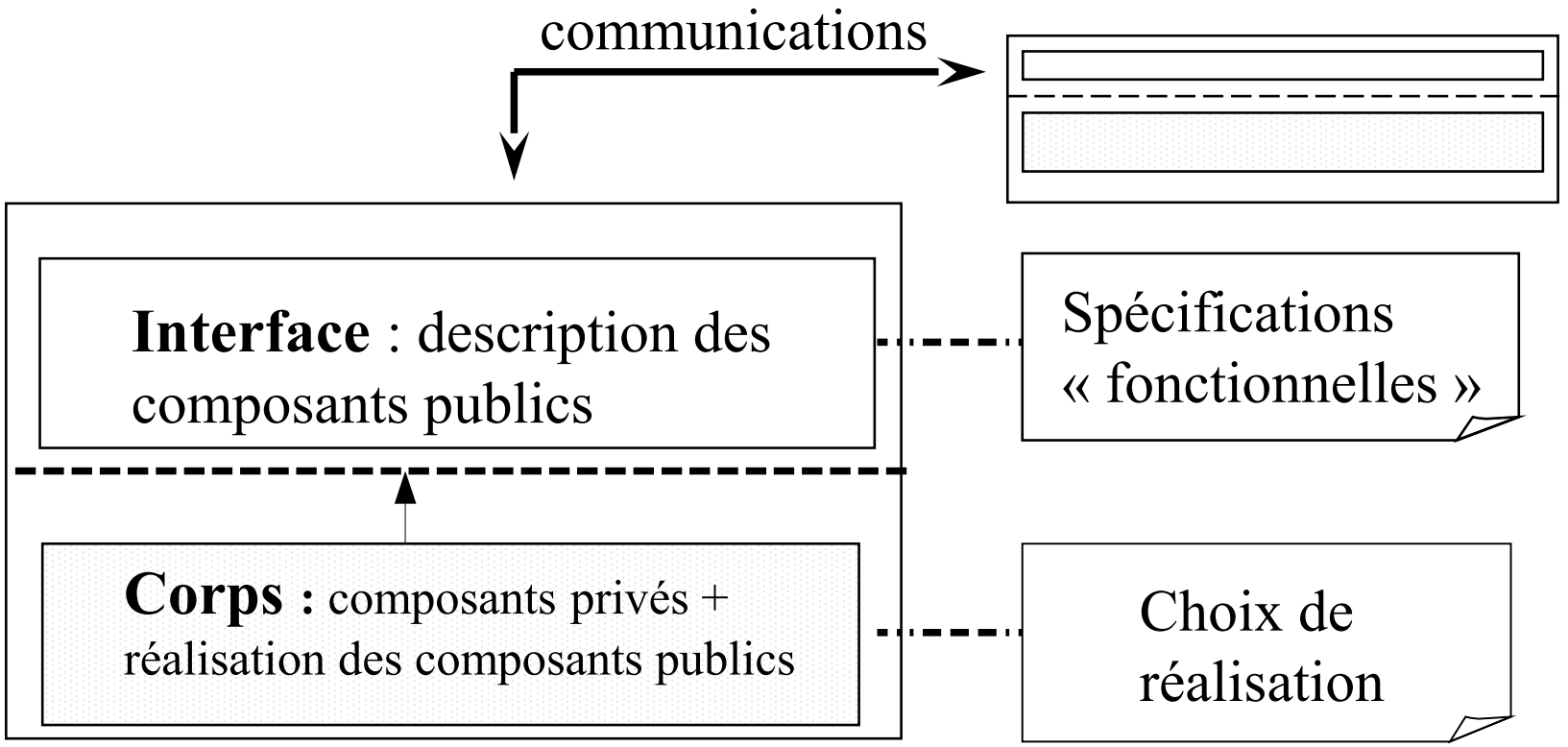
- Commentez les en-têtes des méthodes suivantes de `Collection` : `add, remove, contains, addAll, removeIf, toArray` (deux versions)

Un module ?

Une unité de programme qui peut

- **être développée de façon séparée :** *on y arrive souvent …*
- **être modifiée sans troubler le reste du système :** *plus dur !*
  - *mise au point/changement de représentation* :
    Laisse-t-on filtrer des informations internes ?
    Comment «confiner » des modifications?
  - *extension* : *faut-il modifier l'existant pour **ajouter** une nouvelle facette ?*
- une **construction syntaxique** : fixe la « frontière » du module
  - distinction syntaxique entre ce qui est dans/hors du module
  - distinction syntaxique entre ce qui est public/privé
- Fort couplage interne, faible couplage externe entre modules :
  Liaisons **explicites** entre modules via leur **interface**.

Un module (graphiquement)



Java/C++: *public/privé dans le même composant, notions de paquetages / « amis »*

Un Module = un Type ou une Classe ?

Non, pas forcément !

On peut être amené à définir plusieurs types ou classes dans un même module (types ou classes inter-dépendants)

le module doit être une construction syntaxique spécifique !

- Java = notion de « paquetage »
  - Rôles d'un paquetage ? Rôle de **`import`**?
  - Notion de « classes internes »
- C++ = pas de construction syntaxique directe mais
  - « Amitié » entre classes pour relâcher la protection,
  - « Espaces de nommage » pour résoudre les conflits de noms

Interface d'un module ?

- Description des **composants exportés** (publics)
  - types, constantes, objets
  - sous-programmes, paquetages
  - *Exceptions* : comportements exceptionnels ou incorrects

  Tout ce qui n'est pas explicitement public devrait être caché !

- Modules **importés**

- Les **paramètres** d'un module générique

  ```
  public class Paire<C1, C2> { … }
  ```

en Java ? en C++ ?
en C# ? en Scala ?

- Les langages de programmation permettent de décrire les aspects syntaxiques… pas les aspects sémantiques :

  Java : quelles propriétés doivent satisfaire `equals()` *et* `compareTo()` *?*
  quelle(s) fonction(s) doit-on (re)définir si on (re)définit *equals ?*

Un exemple classique: les nombres complexes

- Deux représentations classiques:
  - cartésienne: `a + ib`
  - géométrique: (ρ, θ) $\rho\ e^{i\theta}$
- Chaque représentation a ses propres caractéristiques :
  - domaine de définition (« invariant »)
  - règles de calcul pour égalité, addition, multiplication,…
  - représentation de la constante *i*

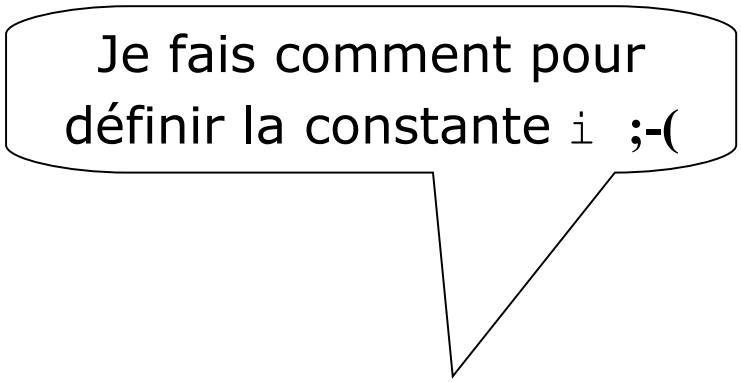
L'abstraction existe indépendamment des représentations:

Les opérations définies et les théorèmes restent les mêmes !

Une représentation peut être plus ou moins adaptée à une application...

Interface à la Java

```
public interface Complexe {   -- version Java 7. En Java 8 ?
   double getRel();
   double getIm();
   double getModule();
   double getArgument();
   Complex add(Complex arg);
   boolean equals (Object O);
   … // et toutes les opérations classiques (ou presque…)
}
public class MonApplication { -- module utilisateur
   public void traitement(Complexe c1, Complexe c2) {
   // On manipule c1 et c2 par les méthodes de l'interface !
   // On est indépendant de la représentation…
   }
}
```



Interface à la Java … (suite)

```
public class CartesianComplex implements Complexe {
  private double relPart, imPart;
  public double getRel() { return relPart; }
  public double getModule() { return … ; }
  public boolean equals (Object O){ … }
  Complex add(Complex arg);
  // et toutes les autres méthodes promises par l'interface
}
```

Programmation de *equals* ?

```
public class GeometricComplex implements Complexe {
  private double module, argument;
  public double getRel() { return … ; }
  public double getModule() { return module ; }
  public boolean equals (Object O){ … }
  Complex add(Complex arg);
  // et toutes les autres méthodes promises par l'interface
}
```

Programmation de *equals* ?

Interface à la Java … (fin)

On peut utiliser les complexes sans dépendre d'une représentation … **mais** attention si on mélange les instanciations !

```
public static void test(Complexe c1, Complexe c2) {
 … c1.equals(c2); …                                  // Compile ?
}
test(new CartesianComplex(1.0, 0.0),
     new CartesianComplex(1.0, 0.0));                // Quelle equals ?

test(new CartesianComplex(1.0, 0.0),
     new GeometricComplexe(1.0, 2 * Math.PI));       // Idem
```

Soit programmer **TRES** soigneusement `equals` (Où ? Comment ?)
Soit garantir qu'on ne mélange pas les implémentations (Comment ?)

Autre exemple Java:

Quand deux « instances » de `Collection` sont-elles égales ?

Quand deux instances de `Set` ou de `List` sont-elles égales ?

Polymorphisme et généricité

Polymorphisme : capacité d'une procédure à admettre des paramètres de plusieurs types (éventuellement une infinité !)

- Ce n'est pas du vrai polymorphisme
  - Conversion implicite de types
  - Surcharge : une facilité syntaxique

- polymorphisme "universel"
  - par **inclusion** (langages à objets) : construction de hiérarchies de types par **héritage** (sous-classes)
  - par **passage de paramètres** (généricité)

Ce n'est pas du vrai polymorphisme

- **Conversions implicites**

  C: `int -> float` ? *Masque un changement de représentation !*

  `char -> int` ? `'c' + 2` Ok*, mais quel est le sens de* `'c' * 2` ?

  `int <-> bool` ?

  ```
  if (3 < 5 < 2) …
  if (a == b == c) ...
  ```

  C++ : via les constructeurs et certains opérateurs… ;-(

le compilateur ne peut pas autoriser uniquement « celles qui ont un sens »

- **Surcharge**: facilité syntaxique. Les opérateurs surchargés n'ont pas forcément de lien sémantique, leurs implémentations diffèrent !
  - Exemple : les constructeurs en Java, …
  - Surcharge limitée en Java : les méthodes mais pas les opérateurs
  - C++, Scala: surcharge des méthodes et des opérateurs : +, ==, =, [ ], () ...

Polymorphisme et Langages à Objets

Polymorphisme par inclusion: hiérarchies de “types” / “sous-types”, avec héritage de la structure et des fonctionnalités…

Principe de substitution (B. Liskov) :
si `ST` est un sous-type de `T`, on peut utiliser un objet de type `ST` là où on attend un objet de type `T`

« sous-type » ? extension de structure, ajout/redéfinition de fonctionnalités.

Toute classe est potentiellement racine d'une sous-hiérarchie :
elle définit implicitement « une famille (infinie) de types »

Un exemple de **polymorphisme par inclusion**

Définir une hiérarchie pour un « éditeur de figures géométriques »

- des **figures simples**: cercles, triangles, rectangles, etc.
- des **diagrammes** composés de figures simples et de sous-diagrammes, avec un niveau d'imbrication quelconque

Pour les figures et les diagrammes, on veut pouvoir:

- tester l'appartenance d'un point à une figure (d'un diagramme)
- déterminer les dimensions d'un rectangle englobant
- déplacer, effectuer une rotation, etc.

En plus :

pour les diagrammes: ajouter/retirer des éléments, dessiner son contenu
pour une figure simple: la dessiner, connaître son périmètre

Les Figures Géométriques

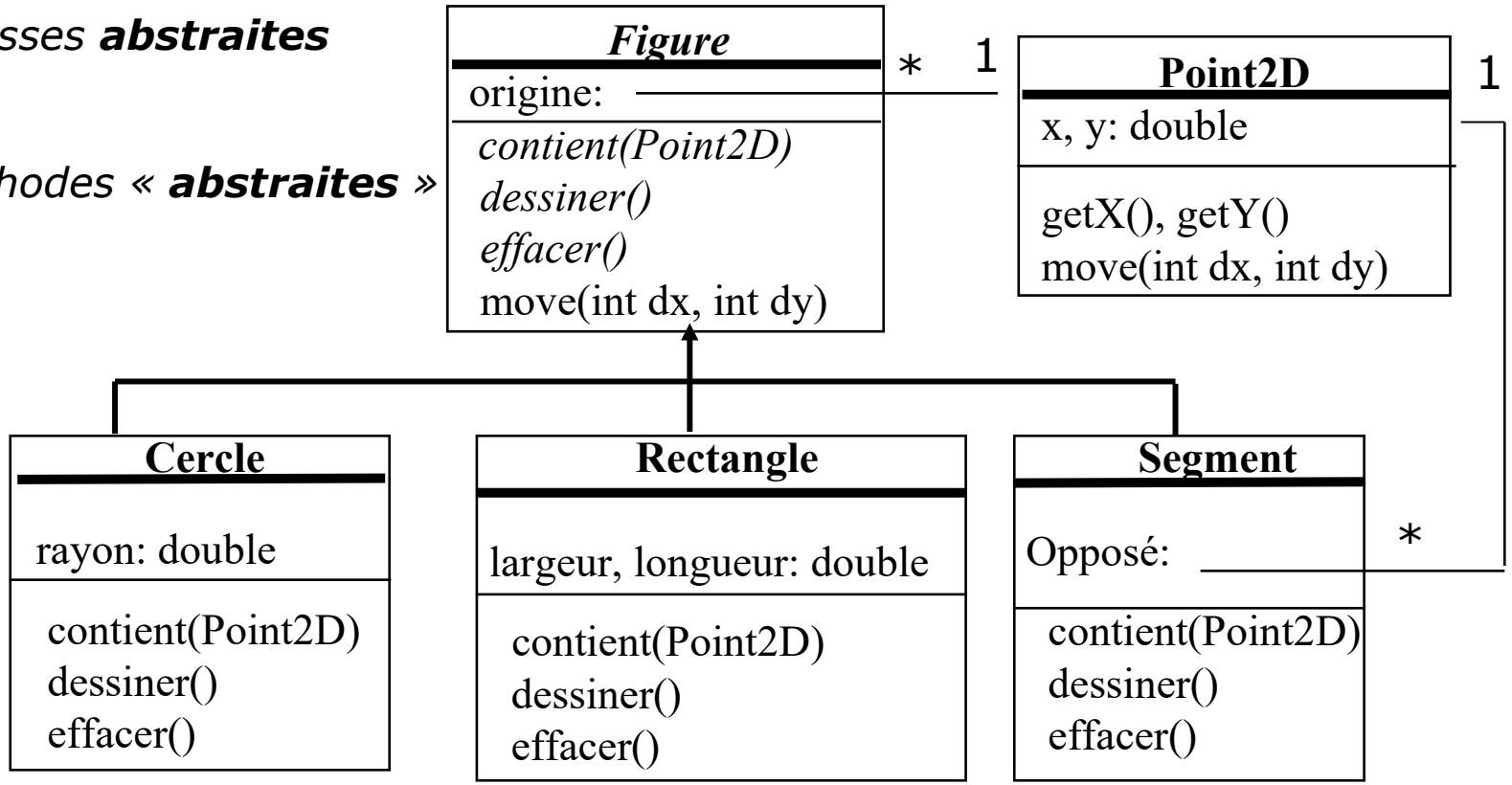
Ce qu'on veut pouvoir écrire directement... (syntaxe imaginaire)

```
P: Point(0.0, 1.0); C: Cercle(P, 10); R: Rectangle(P, 5, 15);
L2 : array[1..10] of Cercle ;
L :   array[1..10] of Figure ;                    -- OK ?

L[1] := C;   L[1] := R;  C:= L[1] ;               -- OK en Java ? En C++ ?
C.move(0, 1.2);
L[1].move(1, 3);                                  -- move ? En Java ? En C++ ?
if  (C.contient(P)) ...
for (Figure F : L) if (F.contient(P)) …           -- mêmes questions

L   := L2 ;                                       -- OK en Java ?
L2 := L ;                                         -- OK en Java ?
```

Une hiérarchie des figures simples (à la UML)

*Classes **abstraites***

*Méthodes « **abstraites** »*



Ajouter `CerclePoilsCourts`, `CerclePoilsLongs`, `Carre`, `Triangle`

Polymorphisme **paramétrique** à la Ada

```
generic
type elem is private;
with function compareTo(E1,E2:elem) return boolean;
type table is array(integer range <>) of elem;
procedure sort(T: in out table) is begin
   -- ici votre algorithme de tri préféré exprimé en
   -- fonction des paramètres elem, table, compareTo
end Tri;
```

paramètres

Pour utiliser `sort`, il faut en dériver des « procédures concrètes » :

```
procedure sort1 is new sort(elem => integer, compareTo => "<=", ...);
procedure sort3 is new sort(elem => integer, compareTo => ">=", ...)
procedure sort2 is new sort(elem => monType, compareTo => cmp,...);
```

Autant de versions distinctes que d'instanciations…
… mais une seule version du source à maintenir

Polymorphisme **paramétrique** à la C++

```
template < class RandomAccessIterator,
           class StrictWeakOrdering >    -- Optionnel
void sort( RandomAccessIterator first,
           RandomAccessIterator last,
           StrictWeakOrdering comp);
```

`RandomAccessIterator` : itérateur de parcours

`StrictWeakOrdering` : notion de « relation d'ordre, stricte, faible »

```
int A[]= { 1, 4, 2, 8, 5 }; double B[50];
```

sort(A, A+5); -- avec "<=" et conversion pointeurs -> itérateurs

sort(B+3, B+10); -- concrétisations implicites de ***sort***

Les instanciations sont **déduites** par le compilateur (correspondance **par nom** pour les méthodes et/ou conversions implicites)

Le code final contient autant de procédures différentes que de concrétisations dans le code.

Polymorphisme paramétrique à la Java ?

- Jusqu'à Java 1.4 : pas de classe/méthode paramétrée !
  - Que du « polymorphisme par inclusion » (héritage)
  - `ArrayList`, `TreeMap`,… sont « génériques » par rapport à `Object`.
  - On « transtype » et/ou on gère `ClassCastException`
  - On perd le type statique des objets par passage dans une collection
  - Pour passer une fonction en paramètre il faut en faire un objet !
- Depuis Java 5 : classes et interfaces génériques comme `Collection<T>`, `Comparable<String>` … mais c'est « compliqué »
- Amélioration Java 8 (« lambda » : fonctions anonymes, passage de fonctions en paramètres, interfaces « riches »)

« Généricité » en Java 1.4

Version 1 : Utilise « l'ordre naturel » sur les éléments.
Les éléments de la collection doivent pouvoir se comparer, i.e. être d'une classe qui implémente `Comparable`

```
Integer[] tab = { 1, 3, 2, 7, 5};
TreeSet ts = new TreeSet();
for(int i = 0; i < tab.length; i++) ts.add(tab[i]);
Integer x = (Integer) ts.first(); ...
```

`ts` va utiliser une relation d'ordre implicite (*`compareTo`*)

Besoin de « transtyper » (d'où risque d'échec à l'exécution)

On peut mettre une instance de n'importe quelle classe dans `ts`.

Une méthode *`compareTo`* par classe donc une seule « comparaison naturelle » entre éléments !

Java 1.4 : « Réification » de méthodes …

Q: Passer des méthodes en paramètre quand on ne peut passer que des objets ?

R: « Faire un objet de cette méthode », en créant une classe dont la seule raison d'exister est de permettre de définir la comparaison voulue

Définition de « comparateurs », chacun avec sa comparaison

```
public class monComp1 implements Comparator { … }
public class monComp2 implements Comparator { … }
```

On passe au constructeur de `TreeSet` le comparateur à utiliser :

```
TreeSet T1 = new TreeSet(new monComp1());
TreeSet T2 = new TreeSet(new monComp2());
TreeSet T3 = new TreeSet();                    -- utilise compareTo
```

La classe d'intérêt n'est plus obligée d'implémenter `Comparable`.

Q : Différence entre `Comparator` et `Comparable` ?

Q : Quelle différence entre deux instance de `monComp1` ?

Java 5+ : idem sauf que `Comparable` et `Comparator` sont paramétrées

Polymorphisme paramétrique ou d'inclusion ?

Java 1.5+ :

- les deux, mais cohabitation « délicate ».
  Ex: `TreeSet<Cercle>` sous-classe de `TreeSet<Figure>` ?
- Problème accru par le besoin de compatibilité avec les programmes écrits dans les versions antérieures de Java !

```
ArrayList<Integer> a1 = new ArrayList<Integer>();
ArrayList          a2 = new ArrayList();
a1 = a2;                                  // OK ?
a1 = (ArrayList<Integer>) a2;             // OK ?
a2 = a1;                                  // OK ?
```

On n'écrit jamais cela directement mais des problèmes similaires apparaissent via des appels à des fonctions écrites en Java1.4

C# : les deux, et encore différent.

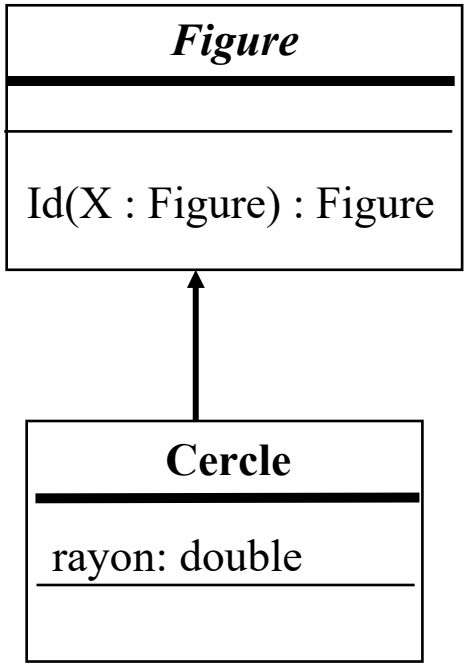
Scala : encore différent (plus large, plus précis, des contraintes)

Héritage et sous-typage

Cercle *sous-classe de* Figure

```
function Figure::Id (X: Figure) return Figure { return X; }
```
// Ici on suppose une affectation entre références, à la Java

```
F: Figure; C:Cercle; R: Rectangle;

F := C;                 -- OK ?
C := F;                 -- OK ?
R := C;                 -- OK ?

F := F.Id(F);           -- OK ?
F := F.Id(C);           -- OK ?
F := C.Id(C);           -- OK ?
C := C.Id(C);           -- OK ?
C.Id(C).Rayon           -- OK ?
```

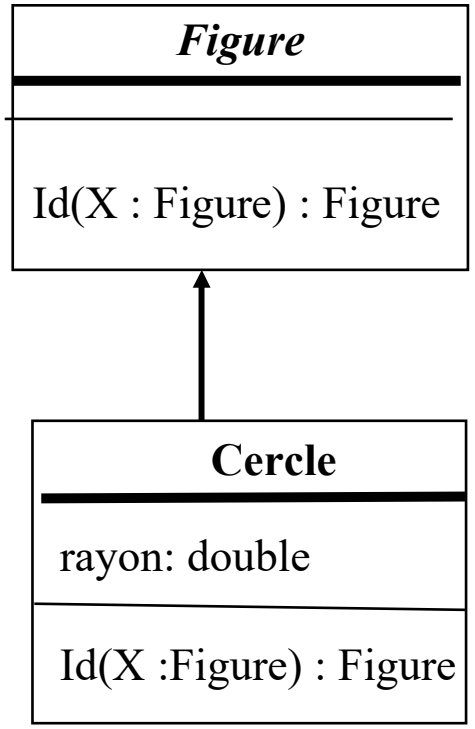


Liaison statique *vs* liaison dynamique de fonction

On **redéfinit** `Id` dans `Cercle`, **en gardant son en-tête**:

```
function Cercle:: Id (X: Figure) return Figure { … }
```

```
F.Id(F);              -- Quelle Id ?
C.Id(F);              -- Quelle Id ?

F := C;               -- OK ?

F.Id(F);              -- Quelle Id ?
F.Id(F).rayon = 1.0;  -- OK ?
```



Covariance et Contravariance

Lors d'une **redéfinition** pourrait-on modifier (élargir / spécialiser) le type des paramètres ou du résultat ?

```
function Cercle::Id(X: Cercle) return Figure …          -- OK ??

function Cercle::Id(X: Figure) return Cercle...         -- OK ??

function Cercle::Id(X: FigureAbstraite) return Figure   -- OK ??

function Cercle::Id(X: Figure) return FigureAbstraite   -- OK ??
```

Possibilités différentes selon les langages (ou versions !)

Des langages autorisent certains de ces cas problématiques mais en les voyant comme des **surcharges** et non pas comme des redéfinitions.

Retour sur les langages “à objets”

“à la mode” depuis le début des années 80, même s’ils existent depuis plus longtemps (Simula = 1967, Smalltalk = 1974)

Choix techniques différents :

- Typé statiquement (C++, Eiffel, Java, C#) ou non (Smalltalk, Self) ?
- héritage simple (Java, C#, Scala) ou multiple (C++) ?
- liaison statique (C++ et C# par défaut) ou dynamique (Java) ? Ou une combinaison des deux (C++, C#, Ada95) ?
- Classes : notion statique (C++) ou dynamique (Java, C#) ?
- « pur Objet » (Java, Eiffel) ou “orienté objet” (C++, C#)
- Généricité ou non (Java ? C++ ? C# ? Scala ?)

Principes de la “programmation par les objets”

- Programmation « procédurale » à la C, ...
  Structuré autour des traitements : types + procédures
  - Résiste mal à un changement dans les types de données :
  - **Difficulté à confiner les modifications !**

Exercice : Définition de types C pour les « figures géométriques »

- Programmation « objet » :
  - **encapsuler données et comportements**
  - **définir des hiérarchies de description**
  - **coopération de traitements** répartis dans les classes concernées
    Difficulté : identifier les méthodes nécessaires de chaque classe.
  - **liaison dynamique** des appels de méthode comme mécanisme de base

On ne devrait pas avoir besoin de savoir comment est implémentée la hiérarchie au-dessus d'une classe.

Les classes `Figure` et `Cercle` en Java

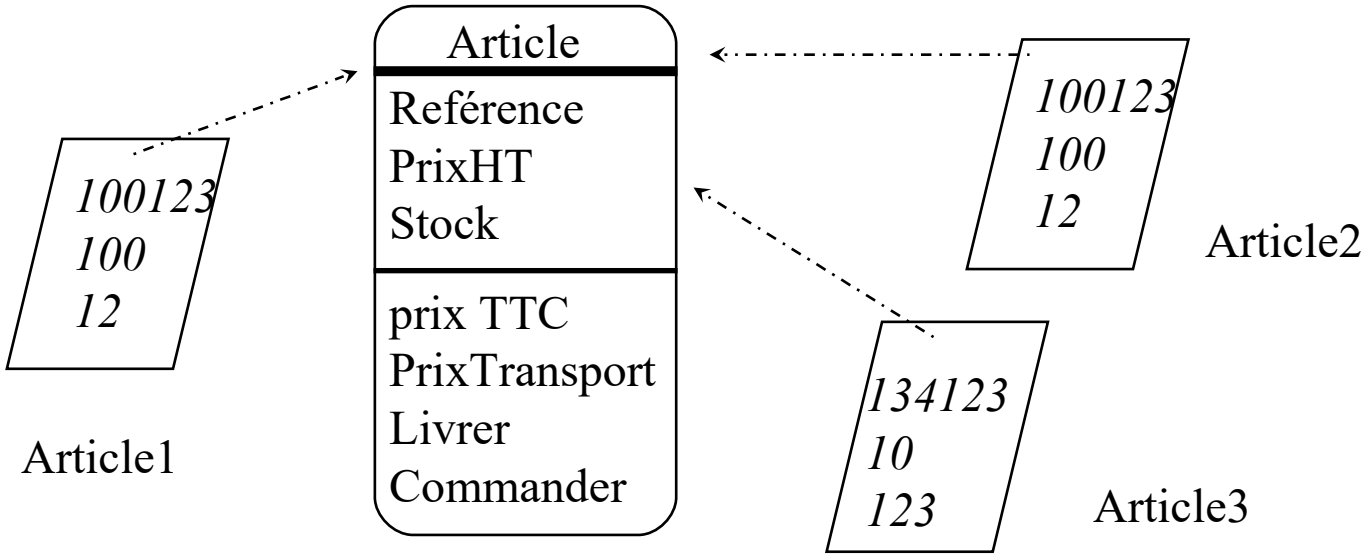
```
abstract public class Figure {
  protected Point2D origine; // ou private ?
  public abstract float aire();
  public abstract boolean contient (Point2D P);
  public void move(int x, int y) {
    this.effacer(); origine.move(dx, dy); this.dessiner();
  }
}// nécessiterait des accesseurs en plus
```

```
public class Cercle extends Figure {
  private float rayon;
  public void dessiner() { ... }
  public boolean contient (Point2D P) { ... }
  public float aire() { … }
...
}// nécessiterait des accesseurs en plus
```

1. Un objet ?

Un objet = Une **identité** + un **état** + des **comportements possibles**

- chaque objet a son **identité** propre et appartient à une classe
- l'**état** d'un objet est défini par la valeur courante de ses champs.
  Il y a plusieurs notions d'**égalité** (entre identités ou états ?)
- les **comportements** sont définis par la hiérarchie de classes à laquelle appartient l'objet et **communs** à toutes les instances d'une classe.



2. Classes et contrôle de types

On ne décrit pas un objet isolément (sauf via `object` en Scala), mais une classe :

- Description d'une famille d'objets (instances) ayant
  même structure (variables d'instances)
  mêmes comportements (méthodes applicables)

- Une description factorisée de la famille = un moule à partir duquel sont obtenues les instances…
  Chaque instance a son propre jeu de variables locales (« état »)

- **Le lien dynamique entre instance et classe guide la recherche du comportement à exécuter dynamiquement**

Classes (suite)

- "**variables de classes**" : attribut associé à la classe plutôt qu'à une de ses instances (*ex: gérer un compteur d'instances*)
- "**méthodes de classes**"
(*ex : retourner le nombre d'instances créées*)

Dans certains langages une classe est représentée à l'exécution et est elle-même une instance (de quelle classe ?).

Notion **d'introspection** (accès dynamique au graphe d'héritage).

D'autres langages (C++) ont une vision **statique** des classes

3. Héritage

Structuration entre classes pour factoriser la description

- favoriser la réutilisabilité (« partage de code »)
- favoriser l'évolutivité : extension, redéfinition de méthodes.

Obtenir ce qui résulterait d'une inclusion textuelle, tout en n'ayant qu'une seule version à maintenir

Une classe peut avoir un lien avec

- 1 "super-classe" (héritage **simple** à la Smalltalk, Java, Scala)
- 0, 1 ou plusieurs "super-classe" (héritage **multiple** à la C++)

dont elle va partager (une partie de) la description

Héritage (suite)

Une sous-classe pourra

- ajouter de nouveaux champs (la **redéfinition** ou **l'oubli** de champs est interdit, le **masquage** est possible)
- ajouter de nouvelles méthodes
- redéfinir (spécialiser) des méthodes de sa super-classe

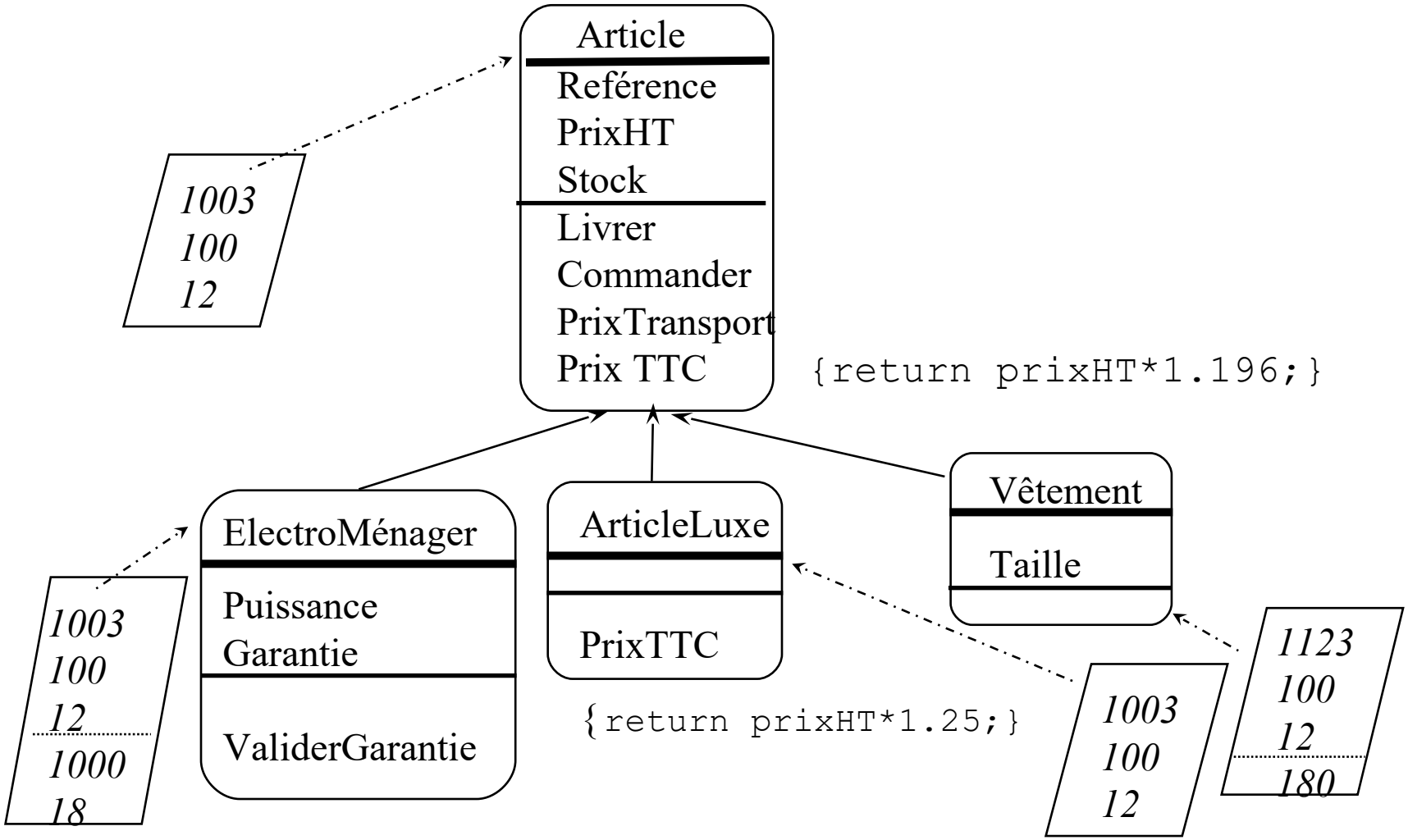
- Selon les langages, on peut interdire
  - que certains "membres" de la super-classe soient visibles des sous-classes,
  - que certaines méthodes soient redéfinies,
  - que certaines classes soient dérivées…

- Niveaux de visibilité: qui voit quoi ? Une sous-classe peut-elle modifier la visibilité de ses champs/méthodes ?

Que veut dire « `protected` » en Java ?

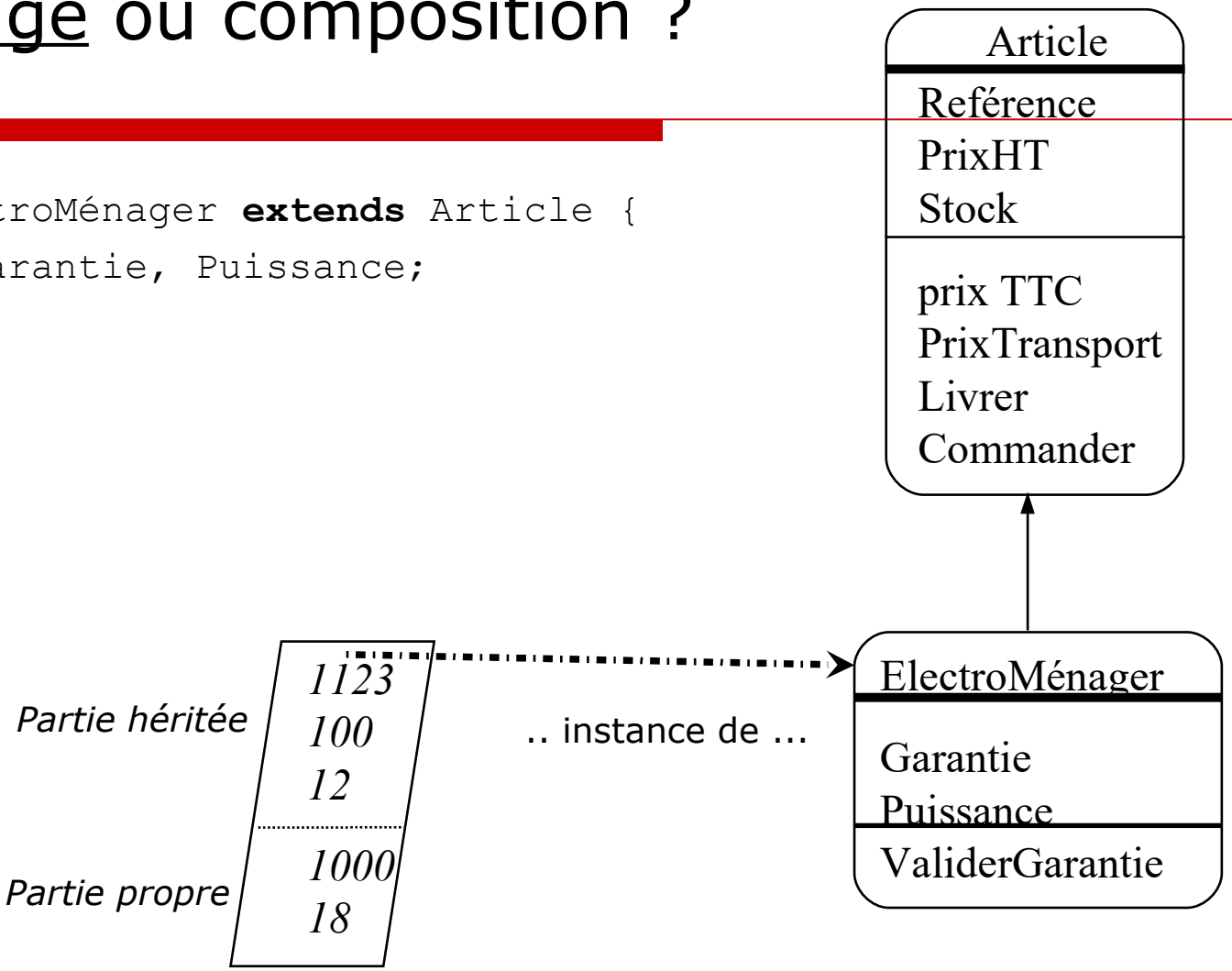
Quelle relation entre « `protected` » et « paquetage » ?

Un exemple d'héritage



Héritage ou composition ?

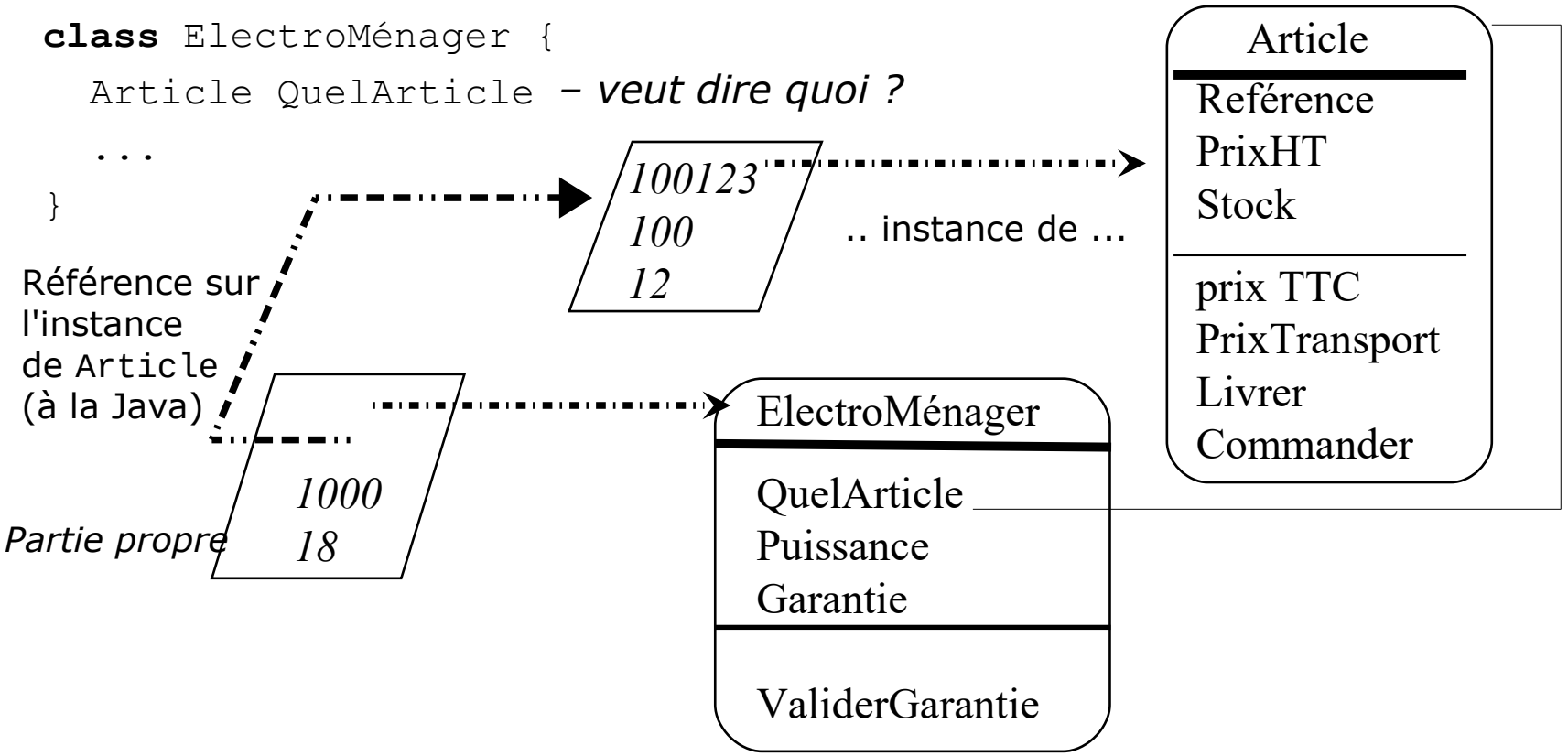
```
class ElectroMénager extends Article {
 integer Garantie, Puissance;
 ...
}
```



Quels comportements et quelle structure pour `ArticleElectro` ?

Héritage ou composition (suite)

```
class ElectroMénager {
   Article QuelArticle – veut dire quoi ?
   ...
}
```



Quels comportements et quelle structure pour `ArticleElectro` ?

Héritage ou Composition ?

```
class PointColore {
   Point2D lePoint;
   Couleur laCouleur;
…
}
```

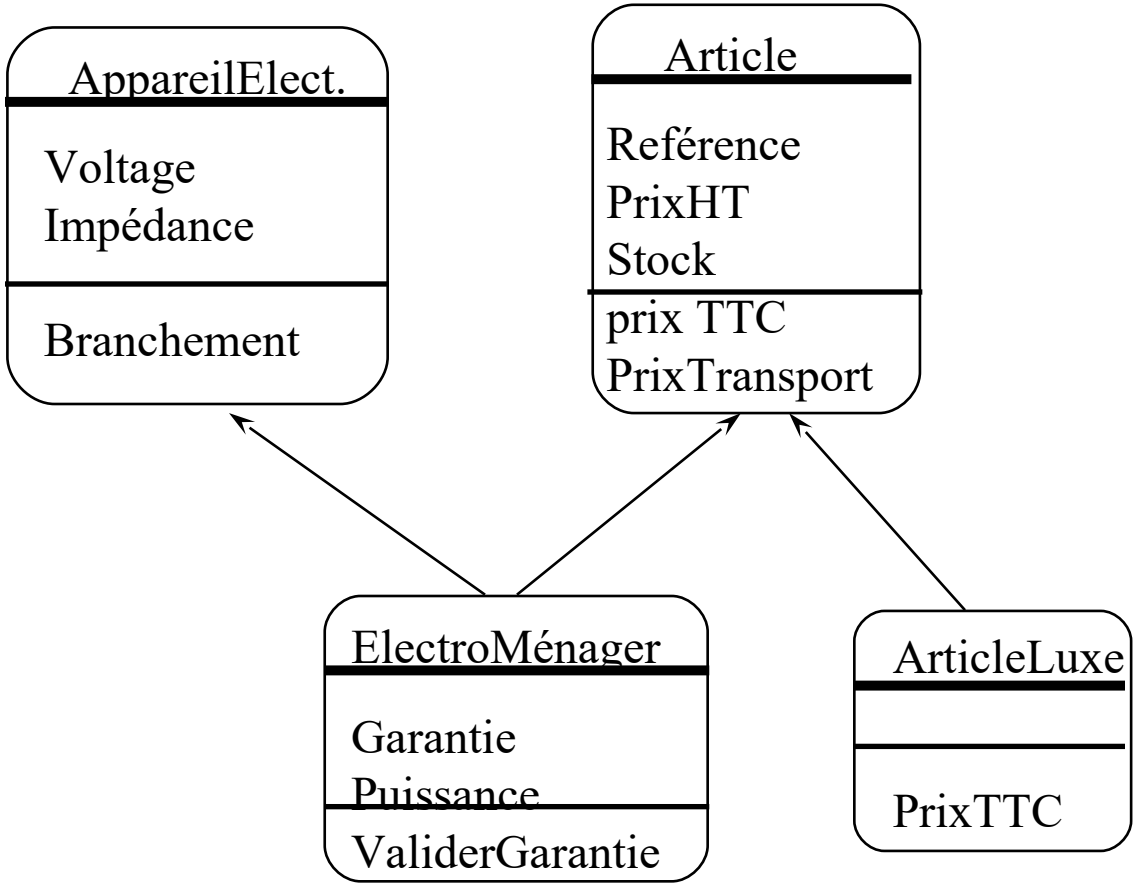
ou

```
class PointColore
   extends Point2D {
Couleur laCouleur;
…
}
```

Question : héritage ou composition entre les classes suivantes

- `PointColoré` et `Point2D` ?
- `Point3D` et `Point2D` ?
- `Point2D` et `Segment2D` ?

Héritage multiple



Plusieurs superclasses pour la classe `ElectroMenager` ...

Héritage multiple et conflits d'héritage

Possibilité de “conflit d’héritage”:

- Deux superclasses définissent un champ avec le même nom ?
- Deux superclasses définissent une méthode de même signature ?
- La résolution des conflits dépent-elle de l’ordre de parcours du graphe d’héritage ? Comment « confiner » le problème ?

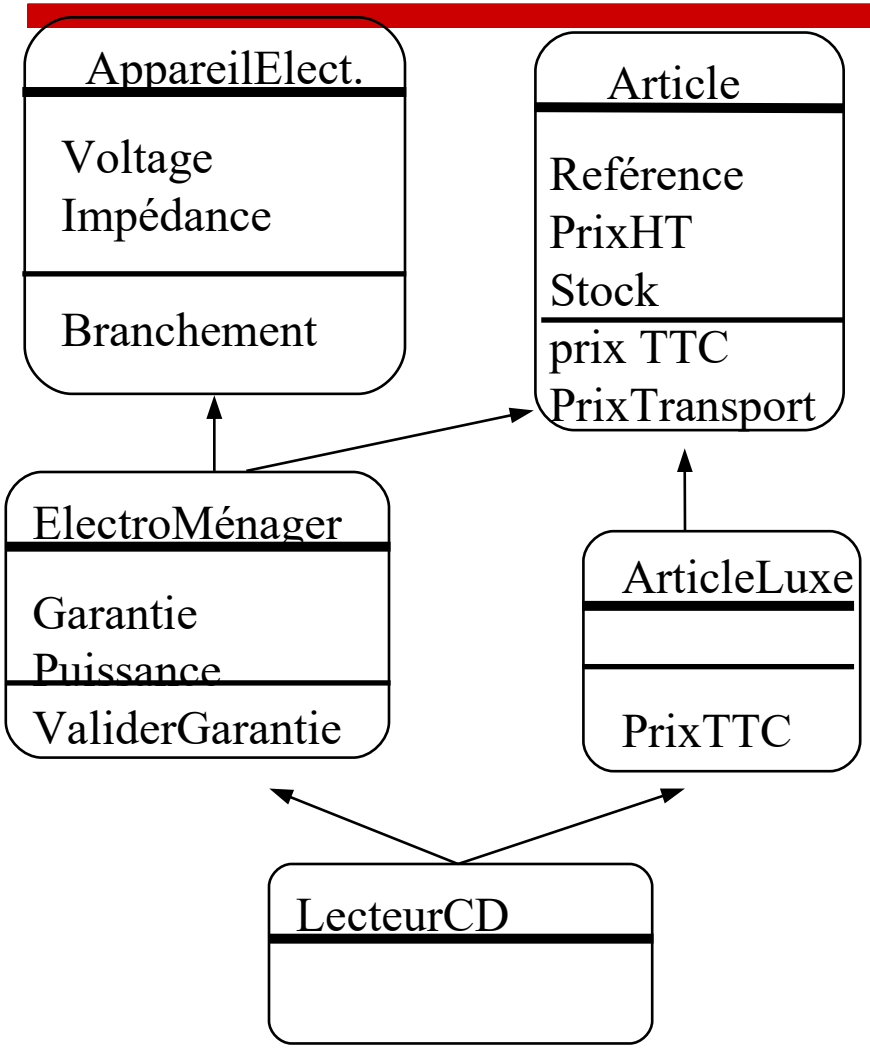
Et si une classe est héritée via plusieurs chemins par une sous-classe ?

- Quelle structure pour ses instances ?
- Comment spécifier les partages voulus ?
- Comment gérer les appels entre constructeurs ?

Java : problème « similaire » avec les interfaces

C# : solution plus élégante

Héritage multiple (suite)



La multiplicité des chemins d'héritage vers `Article` n'apparaît que quand on écrit `LecteurCD`, pas quand on a écrit `ElectroMenager` ou `ArticleLuxe`.

Il faut pourtant l'anticiper puisque ça influe sur la manière d'implémenter le champ `Article` des instance de `ArticleLuxe` et `ElectroMenager`

La création d'instances

- statique
  C++ : `Article a;` -- *Pas forcément ce qu'on croit !*
  Java : `Article a;` -- *ne crée rien (référence) !*

C++ : différences entre `Article a, *pa, &refa;`

- dynamique
  C++ : `Article *ptrA = new Article(...);`
  Java : `Article a = new Article(...);`

- via des méthodes d'instance
  `Article b = a.clone();` -- *retourne une instance de quelle classe ?*

- via des « méthodes de classes » :
  Java: `Article a = Article.makeInstance(...);`

- via des « fabriques » (ou « Factory »)
  `Article a = maFabrique.makeInstance(...);`

`makeInstance` sert de **générateur d'instances** pour contrôler la création d'instances. Les constructeurs de `Article` sont alors « privés »

4. Envois de messages - Liaison de fonctions

Envoi de message: exécution d'une action par un objet
Quelle différence de nature entre `f(x,y)` et `x.f(y)` ?

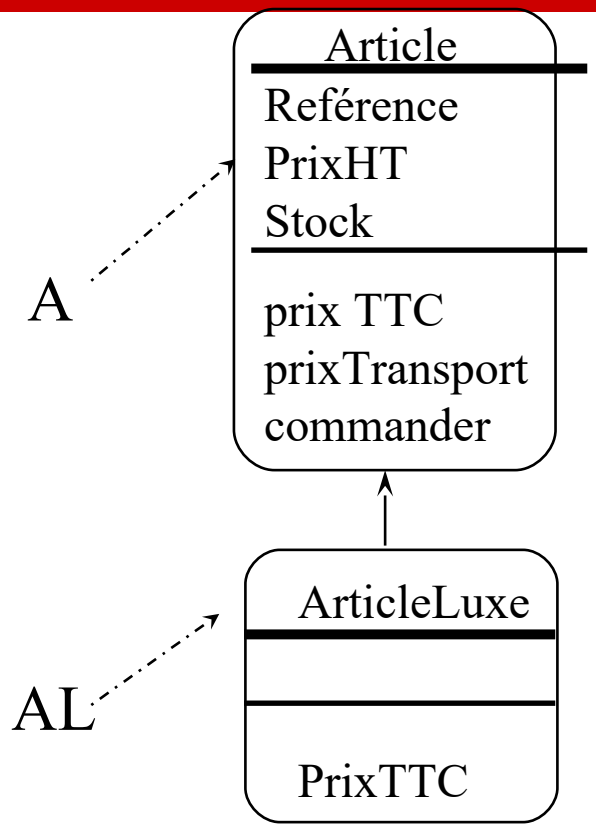
- le **message** est constitué du **sélecteur** `f` et des **arguments** (`y`)
  l'envoi associe le **message** à son **destinataire** (`x`)
- l'action dépend de la **classe du destinataire et du graphe d'héritage** (recherche du corps de méthode à exécuter)
- Cette association peut être faite statiquement ou dynamiquement (**type apparent ≠ type réel**)

```
public static void f(Article a) {
   a.prixTTC();                    -- Quel prixTTC() ?
}
```

Que peut-on vérifier statiquement ?

- Importance du lien d'instanciation, du graphe d'héritage et du type de liaison de fonction (dynamique / statique)

Envois de messages (suite)




```
{ return prixHT * 1.2; }
{ … }
{ return this.prixTTC() +
         this.prixTransport();}
```

```
{ return prixHT * 1.33; }
```

Quelle est la classe de `this` dans `commander()` ?

```
A.commander();
AL.commander();
```

Envois de message (fin)

«Le prix du transport d'un article de luxe comprend une assurance forfaitaire fixée à 50% du prix de transport de la version de base…»

Q : Comment définir la méthode `PrixTransport` dans `ArticleLuxe` ?

- On copie, en l'adaptant, le corps de la méthode définie dans `Article` ?

```
PrixTransport () = (…) * 1,5;
```

- On nomme *explicitement* la méthode de `Article`

```
ArticleLuxe::PrixTransport() =
   this.Article::PrixTransport() * 1.5;
```

- On référence *implicitement* la méthode

```
super.PrixTransport() *1.5
```

Quelle veut dire `super` ci-dessus ?

Auto-référence

- `this` (C++, Java) : un mécanisme pour désigner, à l'intérieur du corps d'une méthode d'instance, le destinataire du message :
  - pouvoir lui envoyer d'autres messages : `this.f()`
  - pouvoir le passer en paramètre de messages envoyés à d'autres destinataires : `x.f(this)`

- `super` (Java): imposer statiquement dans le corps d'une méthode, l'usage d'une méthode d'une super-classe qui risque d'être redéfinie
  - équivalent à `this`, sauf en tant que receveur d'un message : dans ce cas la recherche du corps associé au sélecteur commence dans la super-classe de la méthode et la liaison est statique.

  - C++ : pas de mécanisme, il faut nommer explicitement la classe (plusieurs superclasses en cas d'héritage multiple)

Surcharge *vs* Redéfinition ...

- **Surcharge** ? Il existe une méthode de **même nom**, dans la même classe ou dans une autre, avec une signature éventuellement différente.
- **Redéfinition** ? Il existe une méthode avec le **même nom** et la **même signature** dans la sous-classe et une des super-classes…
- L'utilisation de `@override` (Java) permet de vérifier qu'on fait bien une redéfinition et non pas une surcharge accidentelle !
- Surcharge implique masquage. Mais que masque-t-on ?

  Supposons que

  `Point2D` définit `move(double dx, double dy)`

  `Point3D` hérite de `Point2D`

  `Point3D` définit `move(double dx, double dy, double dz)`

  `(new Point3D()).move(1.0, 2.0)` -- *OK ou pas ?*

  La méthode de `Point2D` est-elle visible dans `Point3D` ?

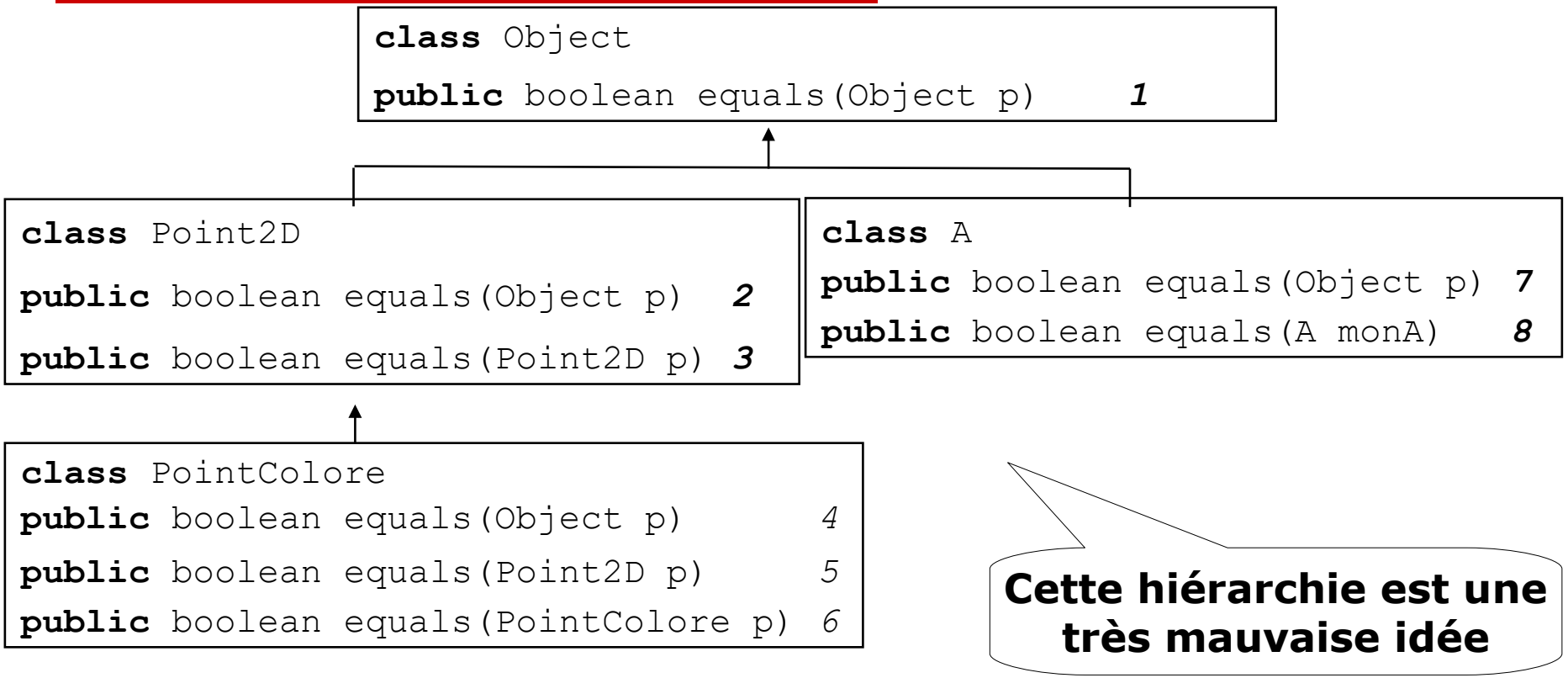
Surcharge *vs* redéfinition (suite)

A la compilation, on détermine la « famille » (ensemble des redéfinitions dans une hiérarchie) de méthodes appelables, compte-tenu de la surcharge et du profil.

Le type du résultat ne compte pas (heureusement pour la covariance) dans le profil.

- **A l'exécution**, la liaison dynamique appelle la « bonne » méthode (selon la classe du receveur) **au sein de la famille déterminée statiquement**,
- La liaison dynamique se fait donc **« à profil constant »** : le profil a été déterminé **statiquement**

Héritage, surcharge et redéfinition (Java)



Déterminez les « familles » de méthodes

Des exemples pour vérifier si on a compris...

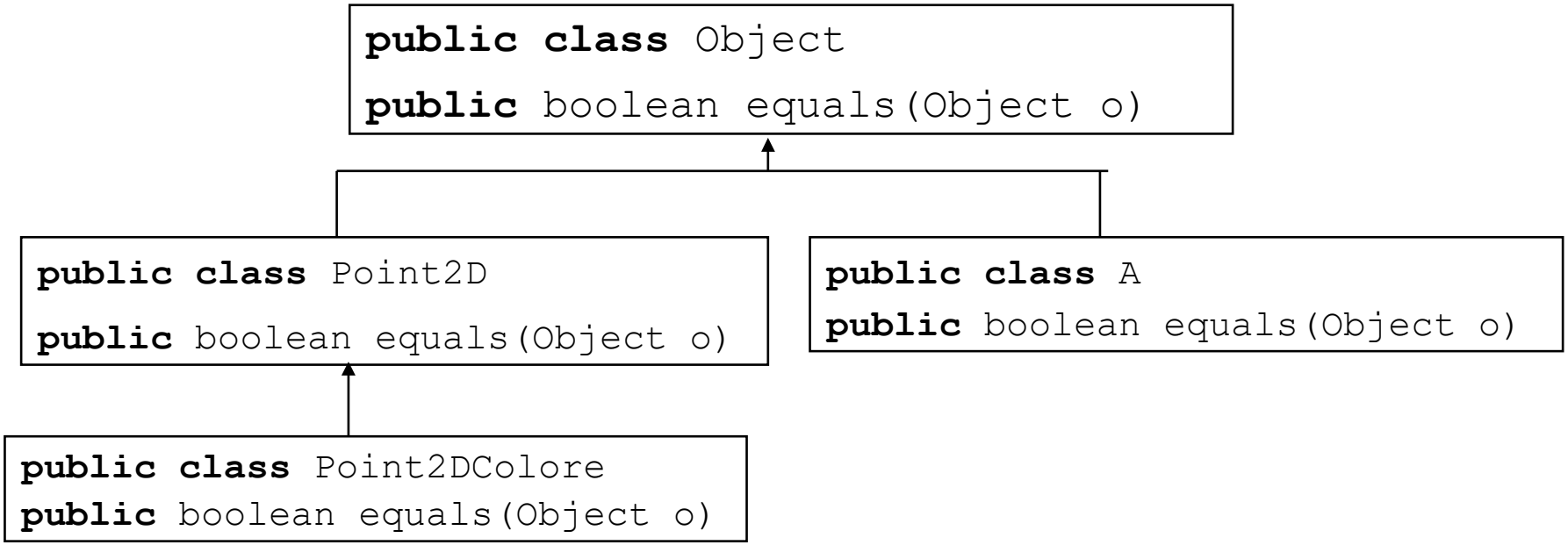
***t1* et *t2* ne diffèrent que par le type de leurs paramètres :**

```
static void t1(Object  o1,  Object o2)  {… o1.equals(o2) …  }
static void t2(Point2D p1, Point2D p2)  {… p1.equals(p2) …  }

Point2D p1 = new Point2D(1.0,1.0), p2 = new Point2D(1.0,1.0);
PointColoré pc1 = new PointColore(1.0, 1.0, Couleur.Blanc);
PointColoré pc2 = new PointColore(1.0, 1.0, Couleur.Vert);
Object o1 = pc1, o2 = pc2;

t1(p1, pc1); t1(pc1, p1);
t2(p1, pc1); t2(pc1, p1);

o1.equals(o2);
pc1.equals(p1);
p1.equals(pc1);

((Point2D) pc1).equals(pc1);
((Object) pc1).equals((Object) p1);
```

Déterminez la méthode `equals` appelée dynamiquement …

Héritage, surcharge et redéfinition (exercice)

Pour `equals` il faut impérativement une seule famille et ne pas confondre « surcharge » et « redéfinition »



Programmez chaque méthode de la famille...

Conclusion partielle sur `equals`

Vérifier la cohérence de `equals` dans votre hiérarchie de classes !

Quelques approches plus ou moins strictes :

- Les classes concrètes n'apparaissent qu'aux feuilles du graphe d'héritage
- Si les sous-classes définissent l'égalité, tester les classe via `==`, y compris vis-à-vis de la super-classe !
  Pas vérifié dans l'API standard (ex : `HashSet` vs `TreeSet`)
- Si l'égalité est toujours celle de la super-classe on peut utiliser `instanceof` (transtypage « descendant »)

- `equals` doit être « totale », réflexive, symétrique, transitive et ne lève jamais d'exception
- Vérifier la cohérence par rapport à des méthodes liées :
  `hashCode`, égalités indirectes (`compareTo`), etc.

MétaClasses et introspection

- **métaclasse** : la classe d'une classe lorsque celle-ci est vue comme une instance

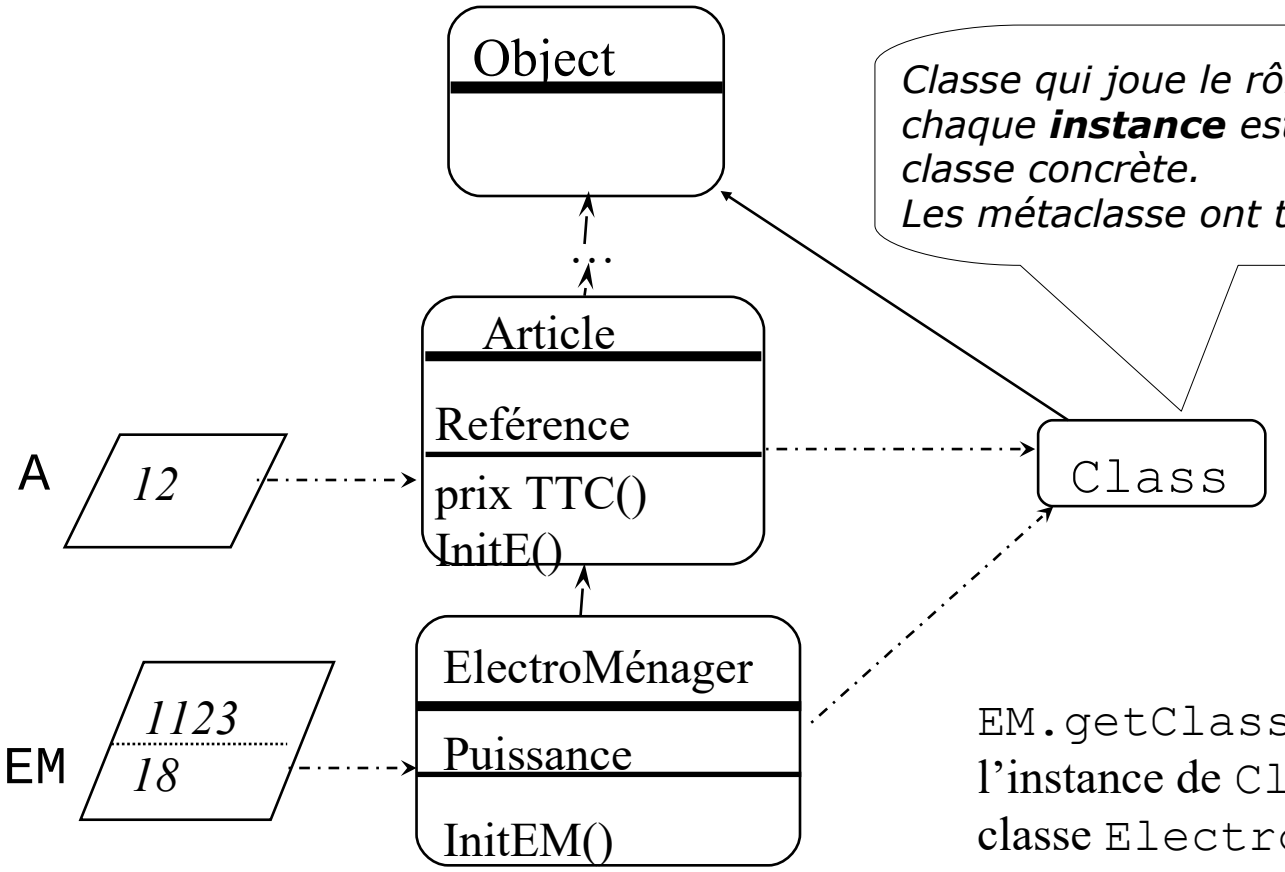
Ex : « variable de classe » = une variable d'instance de sa métaclasse :-)

- pouvoir envoyer des méthodes aux classes (**introspection**)
  - création d'instance si on ne connait pas statiquement la classe

```
Object f(Class c) { return c.newInstance(); }    -- Java 1.4
boolean g(Class c, Object o2) { return c.isInstance(o2); }
```

  - accès **dynamique** au graphe d'héritage
  - accès et/ou définition **dynamique** de méthodes, de classes…
- La hiérarchie entre métaclasses reflète (plus ou moins fidèlement) la hiérarchie entre les classes elles-mêmes.
- La notion n'existe pas dans tous les langages ! Les notions de variables ou méthodes de classe sont alors « artificielles ».

Classes et Métaclasses à la Java



`EM.getClass()` : retourne l'instance de `Class` qui représente la classe `Electroménager`

Introspection à la Java

Dans `Object` : **`public final Class<?> getClass()`**

Dans **`Class<T>`** : -- gère aussi les types primitifs et les tableaux

- Pas de constructeur utilisable : les instances sont construites par le « `classLoader` » de la JVM (`defineClass(byte [])`)
- Pour l'utilisateur : automatique quand on référence une classe ! Initialisation automatique des variables de classes, chargement des superclasses…

Quelques méthodes (publiques) :

- `static Class<?> forName(String)`
- `static Class<?> forName(String, boolean, ClassLoader)`
- `Class<? super T> getSuperClass()`
- `Method[] getMethods()` -- idem pour les champs…
- `boolean isInstance(Object O)`
- `T newInstance()`
- …