Concepts et langages orientés objets

Notes de cours – Mars 2004

Par Gregory Loichot

Concepts et langages orientés objets

Plan de cours

I. Notion d'objet
II. Modélisation
III. Apprendre UML
IV. Apprendre les bases de Java au travers des TPs.

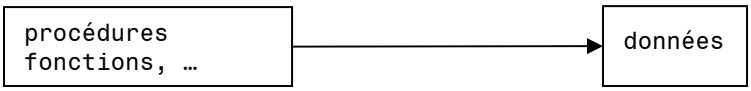
Définition *Modélisation*
La modélisation est une simplification d'un problème (de la réalité) qui permet de résoudre un problème.

Définition *Modèle*
Un modèle est une représentation simplifiée pertinente de la réalité pour résoudre un problème.

Pourquoi n'est-il pas simple de trouver un problème dans un logiciel ?
Parce qu'il n'y a pas de limite physique, tout est virtuel et arbitrairement complexe. Il est donc très difficile de se le représenter. C'est pour cela qu'il nous faut un modèle.

L'idée d'objet

Traditionnellement nous avons :



Les données ne sont pas situées au même endroit que les procédures (bdd par exemple). On aimerait pouvoir regrouper tout cela.

Pour ce faire on défini la notion d'objet.

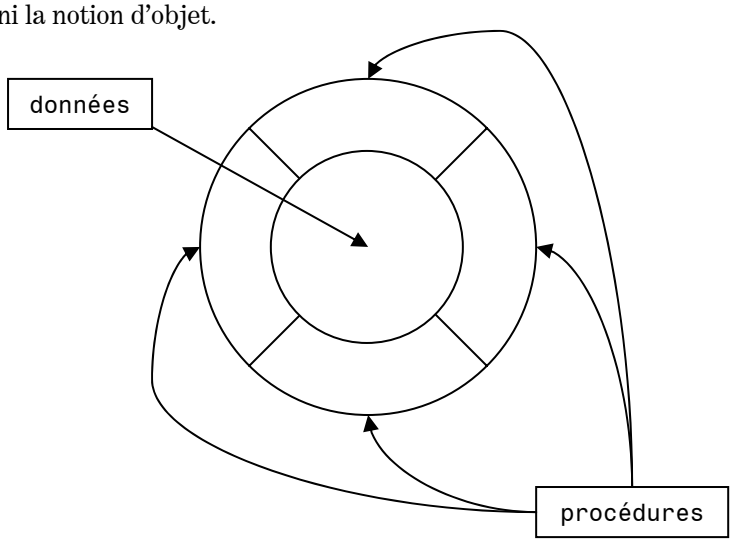


Fig. I, Un objet

Les procédures sont en fait les traitements sur les données de l'objet.

Définition *L'objet*

Un objet est une entité informatique regroupant les données et les traitements d'une entité des problèmes.

L'avantage d'une approche « objet » est que les objets peuvent représenter les aspects statiques et dynamiques des objets du domaine du problème.

Exemples

- Clients, comptes, … (domaine bancaire)
- Fenêtres, fichiers, boutons, … (domaine informatique)
- Bâtiment, portes, escalier, pilier, … (domaine architectural)

Représentation des modèles

Dans ce cours nous allons utiliser le langage UML (Unified Modeling Langage) standardisé par l'OMG. C'est un langage graphique de modélisation centré sur la modélisation par objet.

La structure des objets

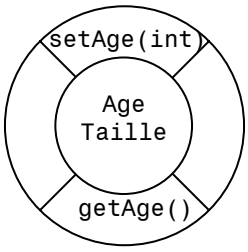
Définition *L'encapsulation*

Les données de l'objet sont encapsulées dans une « couche de procédure ». Nous sommes donc obligés de passer par les procédures de l'objet pour accéder aux données de celui-ci.

Note : certains langages de programmation permettent de casser l'encapsulation, c'est-à-dire d'accéder directement aux données.

L'intérêt principal de l'encapsulation est que l'état de l'objet est toujours sous contrôle de l'objet (des procédures de l'objet).
L'utilisation de l'objet passe par l'invocation des ses procédures. On ne touche jamais aux variables à distance (encapsulation stricte).

Exemple



Soit un objet représentant une personne.
Pour manipuler l'âge de la personne on utilise ces procédures

Traditionnellement les procédures des objets sont appelées « méthodes ».

Pour renforcer l'idée d'entité informatique avec lesquelles on communique, on appelle « invocation d'une méthode » « envoi de messages ».
Un logiciel entièrement basé sur les objets est formé d'objets qui communiquent par l'envoi de messages.

A l'origine l'objet a été proposé pour faire des simulations (Simula 68).

L'abstraction

Un objet a deux vues :

- Une vue interne : la manière dont il est programmé.
- Une vue externe : ce qu'il est capable de faire.

Utiliser un objet revient à invoquer une méthode de son interface.

Définition *L'interface*
L'interface est l'ensemble de méthodes que l'on peut invoquer par l'envoi de messages.

Définition *Le protocole*
Le protocole est l'ensemble des signatures de méthodes de l'objet avec leur sémantique en français.

Définition *La signature*
Une signature est composée d'un nom, d'un type ainsi que de la position de ses arguments.
Par exemple : `print(int)`, `print(int, double)`, `print(double, int)`.

Exemple
Si on reprend l'exemple de l'objet représentant une personne, on définit son protocole par :

- `getAge()` : retourne l'âge de la personne (int).
- `setAge(int)` : assigne un âge à la personne.

L'idée de l'abstraction : grâce à l'interface et au protocole on peut utiliser l'objet sans se préoccuper de l'implémentation des méthodes.

Il faut toutes fois garder à l'esprit que le programmeur a accès à toute l'implémentation du programme.

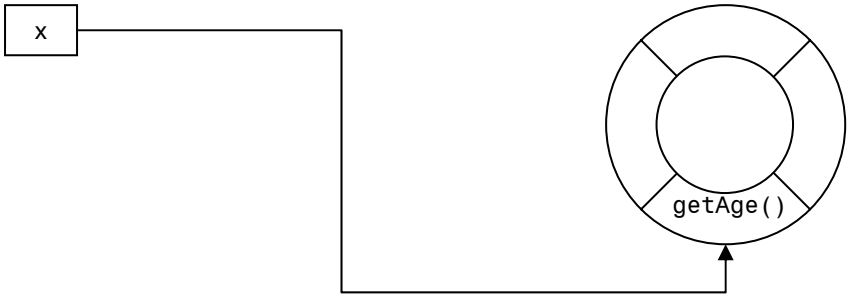
Un programmeur objet a pour tâche de
- programmer ses objets spécifiques,
- utiliser des objets de la librairie à travers leurs protocoles.

La connaissance de « l'existant » (l'ensemble de toutes les classes prédéfinies en Java) est le plus gros de l'apprentissage de ce langage.

L'envoi de messages

Rappelons qu'invoquer une méthode revient à envoyer un message.

Soit la situation suivante avec « x » une variable qui référencie un objet :



L'instruction

```
x.getAge() ;
```

aura comme effet d'envoyer le message `getAge()` à l'objet référencé par `x`.

Si nous avons plusieurs objets qui sont semblables au niveau de leurs structures et de leurs comportements, comment faut-il les déclarer ?
Cela se fait à l'aide de classes.

Définition *La classe*
Déclaration de structures de données et de méthodes d'objets identiques.

Représentation d'une classe selon UML :

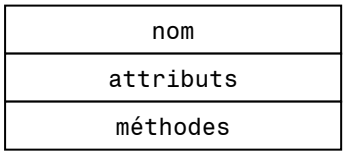


Fig. II, Représentation d'une classe

Les champs de la classe sont :

- `nom` : son nom.
- `attributs` : déclaration des structures de données.
- `méthodes` : déclaration des comportements des objets.

On peut imaginer, par exemple un écran sur lequel sont affichées plusieurs droites. Toutes ces droites sont identiques dans leur comportement et dans leur structure de données. On pourrait alors représenter la classe « droite » comme suit :

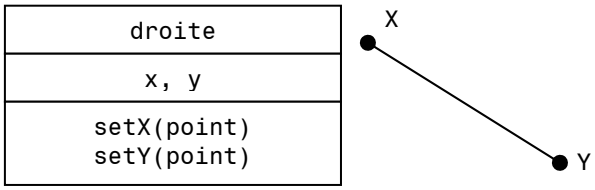


Fig. III, La classe « droite ».

A partir de cette classe on pourra générer plusieurs objets ($droite_1$, $droite_2$, …) de type « droite ». Ces objets ainsi générés sont appelés instances de la classe « droite ».
Virtuellement les objets restent encapsulés par leurs méthodes, mais le code est centralisé.

Définition *L'instanciation*

La création d'un objet à partir d'une classe s'appelle l'instanciation.

La seule différence entre deux instances de la classe est le contenu de leurs variables (attributs).

Définition *Information Hidding*

Cela caractérise le fait qu'un utilisateur d'objet connaisse son protocole mais pas son implémentation qui lui est cachée.

En résumé, programmer avec des objets, c'est :

- Créer des classes,
- Instancier ces classes,
- Manipuler les instances par des envois de messages.

L'instanciation d'une classe

Définition *Instanciation d'une classe*

C'est l'opération qui consiste à créer un nouvel objet à partir (qui satisfait les déclarations de la classe) des déclarations de la classe.

Cette opération est réalisée par un opérateur prédéfini du langage Java.

Exemple

En Java on utilise l'opérateur qui s'appelle « `new` ».
Soit la classe suivante et regardons comment l'instancier :

`personne`

L'expression

```
new Personne() ;
```

retourne une instance de la classe Personne.

Les variables

Dans les langages à objets « mixtes » (c'est-à-dire qui contiennent d'autres éléments que des objets), on a deux sortes de valeurs :

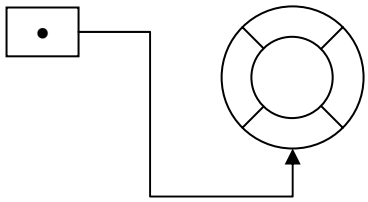
- Les valeurs primitives.
- Les objets.

Exemple

C#, C++, Java sont des langages de ce types (mixtes). On manipule aussi bien des valeurs de types primitifs (int, float, boolean, …) que des objets.

Par contre certains langages ne travaillent qu'avec des objets (Smalltalk, …). Dans ce cas même les entiers sont des objets.

Assigner un objet à une variable



Cela veut dire que l'on assigne une référence à l'objet à la variable.
Quand on crée un objet, le système alloue une zone mémoire qui correspond aux déclarations de variables (+ structure de contrôle) et retourne un pointeur sur cette zone.

Dans le schéma, le ● représente l'adresse de l'objet.

Note : Dans la structure de contrôle on a également un pointeur sur la classe de l'objet et ceci pour pouvoir utiliser les méthodes.

Déclarations de variables

Une déclaration d'une variable en Java est de la forme :

```
<type> <nom> ;
```

Avec `<type>` défini comme :

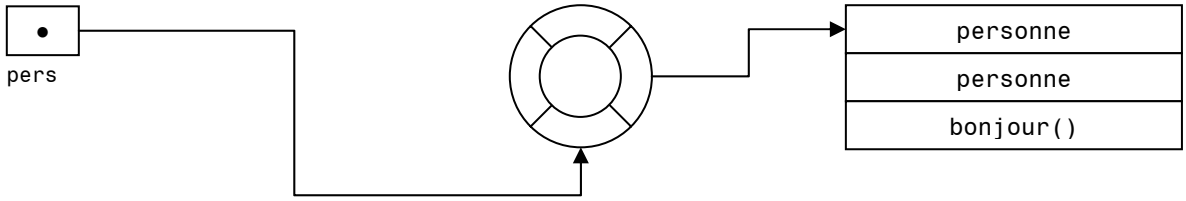
```
<type> = <type primitif> | <nom classe>
```

Dans une variable de type objet il y a une adresse (de l'objet) sur 32 bits par exemple.

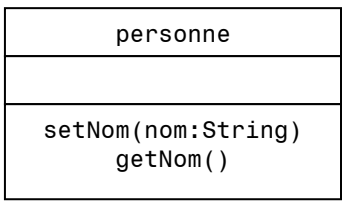
Exemple
Soit le fragment de code suivant :

```
Personne pers ; //Crée une variable de type Personne
pers = new Personne() ; //Assigne un objet à la variable
```

Cela peut se représenter de la façon suivante :



La flèche reliant `pers` à l'objet est créée par le signe « = », l'objet lui-même par « `new Personne()` ».
On peut compliquer un peu l'exemple en ajoutant deux méthodes à notre classe Personne :

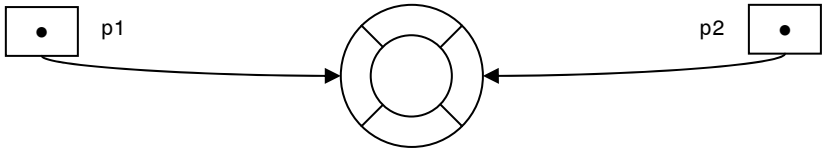



```
Personne p1, p2 ;
p1 = new Personne() ;
p1.setNom(« Jules ») ; //Assigne Jules comme nom de la personne
représentée par l'instance pointée par p1.
p1.getNom() ; // Retourne « Jules »

p2 = p1 ; //Recopie l'adresse de p1 dans p2.
```

```
p2.getNom() ; //Retourne « Jules »
p2.setNom(« Charles ») ; //
p1.getNom() ; // Retourne « Charles ».
```

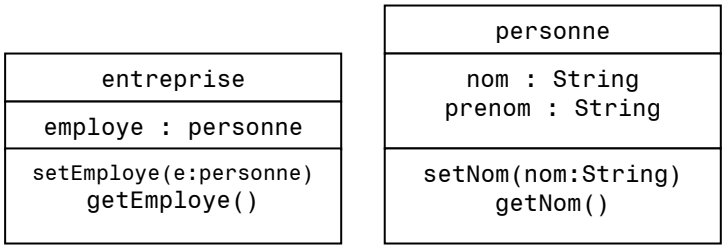
`setNom(« Jules »)` envoie le message `setNom` à l'objet. Cela implique que l'instance possède « Jules » dans sa structure de données.
`getNom()` retourne le nom de la personne représentée par l'instance.

Pour mieux comprendre ce qui se passe, faisons un petit croquis :



Le danger des pointeurs vis-à-vis de l'encapsulation

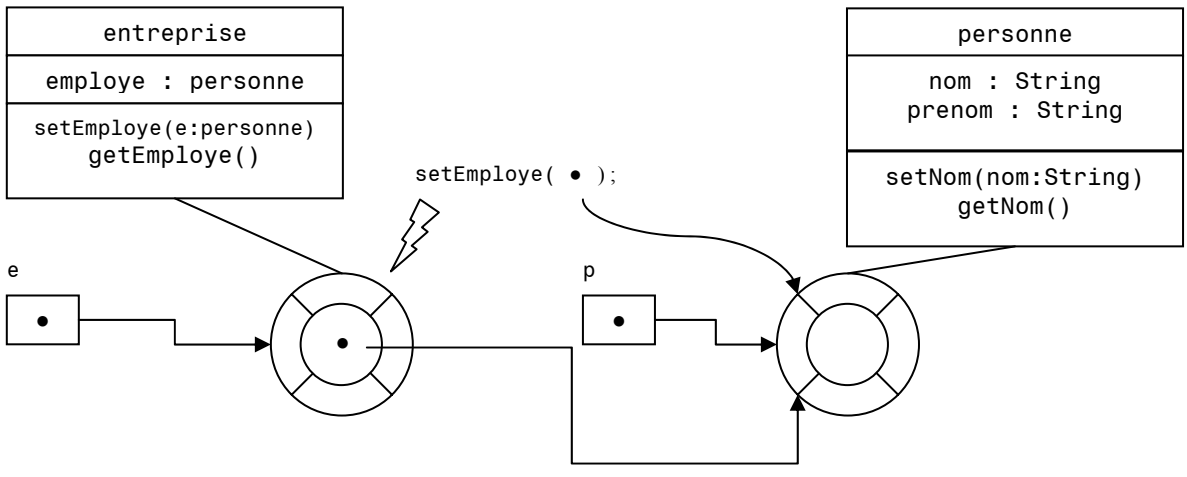
Soient les deux classes suivantes :



Soit le code Java :

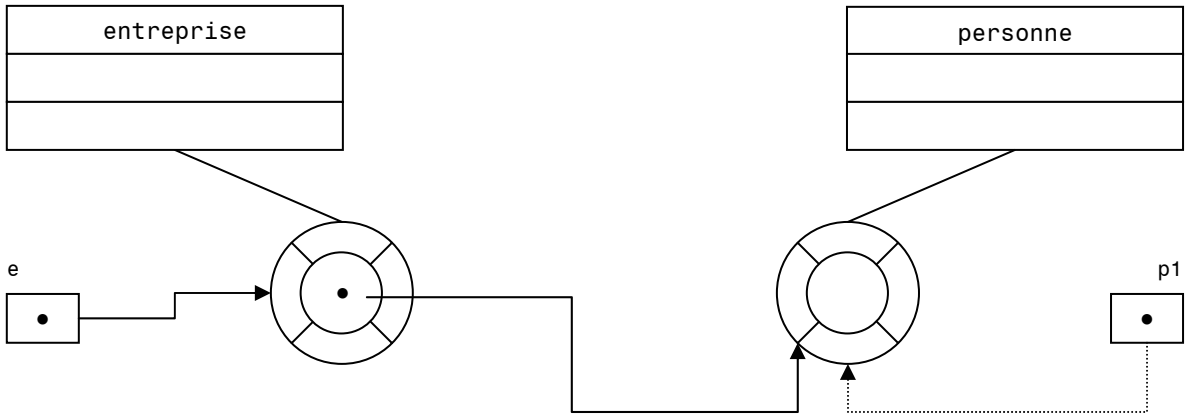
```
Personne p ;
p = new Personne ;
Entreprise e ;
e = new Entreprise ;
e.setEmploye(p) ;
```

Ce code nous produit la situation suivante :



Exemple de contournement de l'encapsulation

Dans cet exemple on va accéder aux variables de l'objet « à distance ».

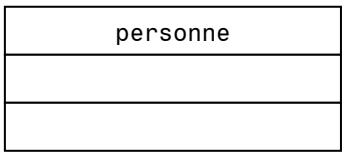


On va, ici, modifier Personne sans passer par Entreprise. C'est une chose à éviter absolument car on « viole » le principe d'encapsulation stricte.

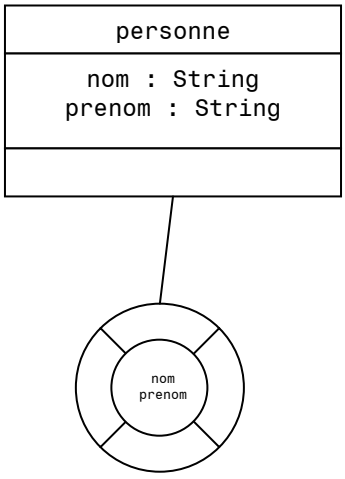
```
Personne p1 ;
p1 = e.getEmploye() ; //Récupère une référence sur l'objet contenu
dans la variable.
p1.setNom(« Voila ») ; //On change le nom en cassant l'encapsulation.
```

On peut se demander alors comment lutter contre cela ? Il faut que la méthode `getEmploye()` retourne un clone de l'objet : `return employe.clone() ;`.

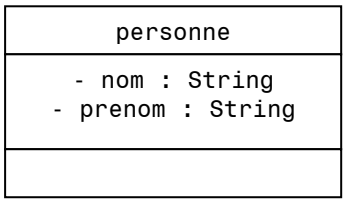
Eléments de syntaxe




```
class Personne {
      Déclaration de variables et méthodes de la
classe
}
```



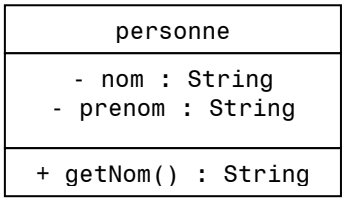

```
class Personne {

      String nom ;
      String prenom ;

}
```



L'encapsulation représente le fait que les variables soient *privées*.

```
class Personne {
      private String nom ;
      private String prenom ;
}
```

Au niveau de la notation UML, le « -« signifie `private`.




```
class Personne {
      private String nom, prenom ;
      public String getNom() {
            return nom ;
      }
}
```

Au niveau de la notation UML, le « +« signifie `public`. Ici la méthode `getNom()` renvoie, par l'expression `return nom;` le contenu de la variable privée `nom`.

Déclarations de méthodes

La déclaration d'une méthode se fait de la manière suivante en Java :

```
<visibilité> <type> <nom> (<arguments>) { <suite d'expressions>}
```

La suite d'expressions est un ensemble d'expressions élémentaires séparées par un « ; ». La liste d'arguments peut être vide ou du type `<argument> [',' <argument>]*`.

De plus un argument a la forme

```
<argument> = <type> <nom>
```

Exemple
Méthode d'assignation du nom de la personne :

```
public void setNom(String  s) {
      nom = s ;
}
```

`void` est un élément spécifique représentant l'absence d'un type.

Les constructeurs

Définition *Constructeur*
Un constructeur est un élément du langage Java qui permet de créer des instances et de les initialiser.

La syntaxe est de la forme : `new XYZ();`, où `XYZ()` est un constructeur de la classe `XYZ`.

Les constructeurs d'une classe ont le même nom que la classe (avec une majuscule comme première lettre). Le nombre de paramètres est quelconque et est défini par le programmeur.
Un constructeur ne possède pas de type retourné, en revanche il possède une visibilité (en général `public`).

Pour se faire une meilleure idée, regardons tout de suite un exemple.

Exemple
Essayons de voir comment faire un constructeur pour notre classe `Personne` :

```
class Personne {
      public Personne() { …code d'initialisation de l'instance… }
      public Personne(String nom, String prenom) {
            … code qui assigne le nom et prénom à l'instance…
      }
}
```

Avec ce code, la classe `Personne` possède deux constructeurs (qui sont différenciés par leur signature).

Nous pouvons alors essayer d'initialiser une instance de cette classe :

```
Personne p = New Personne(« Jules », « Bonnot »);
```

Par défaut, toute classe possède le constructeur sans argument : `nomClasse();`.
Un constructeur est exécuté dans l'environnement de la nouvelle instance, ce qui implique que l'on ait accès à toutes les variables.

Lorsque l'on fait `new Personne();`, que se passe-t-il ?

1. Création de l'instance,
2. exécution du constructeur « dans » l'instance (initialisation des variables).

Virtual Machine

La JVM (Java Virtual Machine) fait le lien entre des fonctions systèmes standard et les fonctions systèmes de la plate-forme choisie (O.S.)

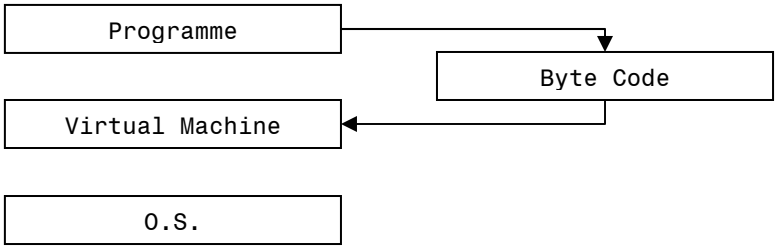


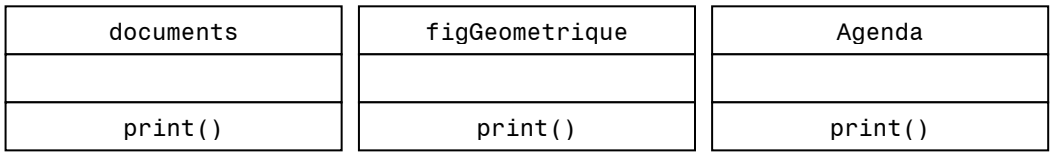
Fig. IV, La JVM

Ainsi un programme compilé dans un environnement Windows sera exécutable sous Mac OS ou UNIX pour autant qu'il existe une JVM pour chacune de ces plates-formes.

Le polymorphisme

Définition *Le polymorphisme*
Capacité d'un langage (objet) à permettre de multiples définitions d'une méthode de même signature dans des objets différents (une seule définition par objet).

Exemple
Soient les classes suivantes :

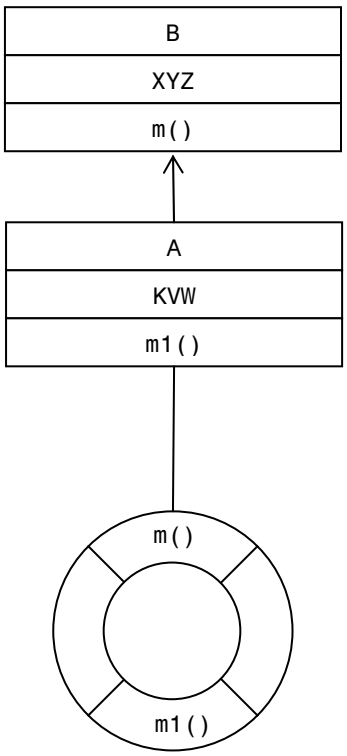


A une sémantique correspond une signature et réciproquement.
`print()` dans chaque classe possède une implémentation différente mais la sémantique doit être la même.

Le polymorphisme nous dit qu'à une signature on a n « formes » qui sont les implémentations dans les n différents objets.

Héritage, Spécialisation & Généralisation

Une situation d'héritage :



Que remarque-t-on sur ce schéma ?

- B est une classe représentant un concept plus général.
- A est une classe représentant un concept plus spécifique.
- B est la *super*-classe de A.
- A est la *sous*-classe de B.
- La flèche entre les deux classes signifie, en UML : généralisation.

Dans cette configuration on peut envoyer les messages `m()` et `m1()` à l'objet créé.

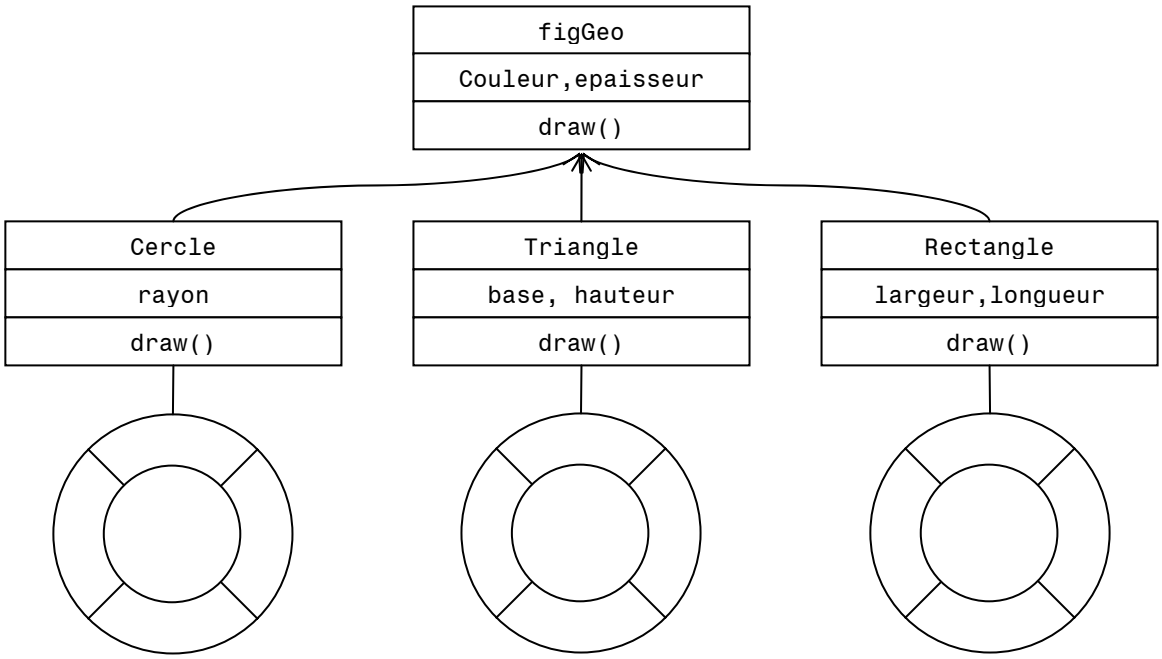
L'héritage est une technique qui permet à une sous-classe de récupérer les déclarations faites dans sa super-classe. L'instance de la sous-classe sera construite à partir des déclarations des deux classes. L'héritage est la « face » technique d'un élément de modélisation qu'on appelle la généralisation.

1. On utilise la généralisation pour modéliser des concepts généraux v.s. spécifiques.
2. Cette modélisation correspond du point de vue technique à l'héritage.

Exemple
On veut modéliser un système d'affichage de figures géométriques.
Les propriétés de ces figures sont :

- Largeur, longueur pour les rectangles.
- Hauteur, base pour les triangles.
- Rayon pour les cercles.

En plus, toutes les figures ont en commun : la couleur et l'épaisseur du trait.



Les méthodes `draw()` des classes `Cercle`, `Triangle` et `Rectangle` sont des spécialisations de la méthode `draw()` de la classe `figGeo`. Il se peut que la manière d'afficher un cercle soit différente de celle pour un triangle. Il n'est pas toujours nécessaire de redéfinir les méthodes.

▲ Ce schéma doit représenter une vraie généralisation conceptuelle !

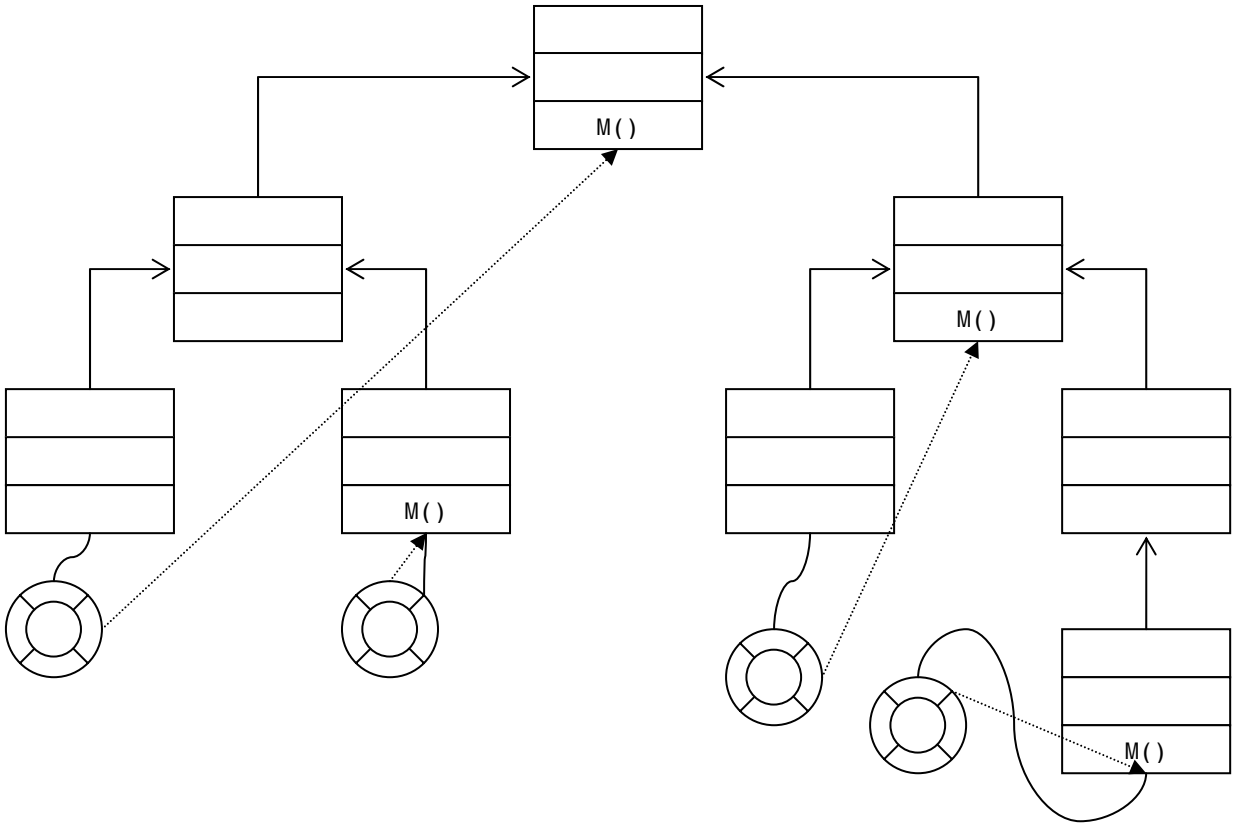
Exécution – Recherche de méthodes dans un arbre d'héritage

Lorsqu'une instance reçoit un message, on recherche dans l'arbre d'héritage la première déclaration de méthode de même signature et c'est celle que l'on exécute.

Exemple
La recherche de la méthode à exécuter suit toujours le chemin ascendant dans l'arbre.

Voyons un exemple d'arbre :

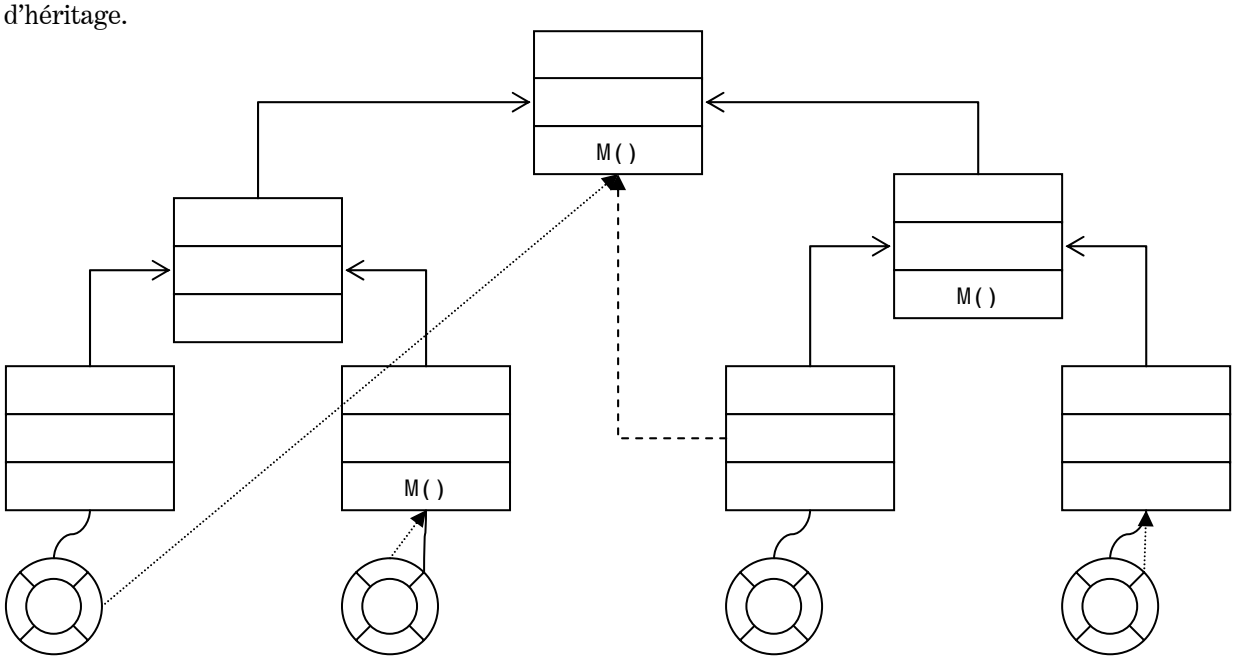
Ici `M()` a partout la même signature.



Ici les flèches en … font référence à la méthode `M()` que l'objet utilisera lorsqu'il reçoit le message `M()`.
Une déclaration de méthode dans une classe « masque » les méthodes héritées des super-classes. On parle alors de « method overriding ».

Héritage simple et multiple

Lorsque la structure de généralisation forme un arbre on a un héritage simple. La recherche des méthodes est alors univoque.
Si cette même structure forme un graphe on a alors un héritage multiple. Il peut alors y avoir ambiguïté d'héritage.

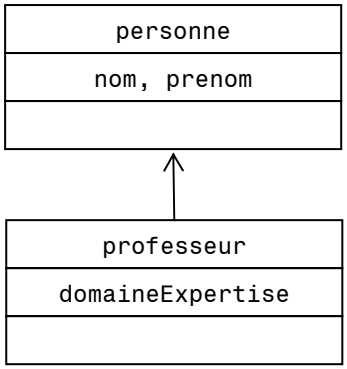


Dans le schéma ci-dessus la situation ambiguë est créée par la flèche en ---. Il y a alors deux choix possibles pour le 3ème objet. Lequel choisir ?

Cette situation n'est pas possible en Java car il ne supporte pas l'héritage multiple.

Syntaxe de déclaration de l'héritage

Soit le cas suivant :




```
class Personne {
      String nom, prenom;
}

class Professeur extends Personne {
      String domaineExpertise
}
```

Ici la classe `Personne` est la super-classe de la classe `Professeur`.
On utilise le mot clé `extends` pour créer la notion d'héritage.

Quand parcourt-on un arbre pour rechercher les méthodes correspondantes à un message ? Il y a deux techniques :

1. « Static binding » : le lien est fait à la compilation.
2. « Dynamic binding » : le lien est fait au run-time à l'exécution.

Définition *Le « binding »*
Lien entre une référence symbolique et le code exécutable.

Le langage Java utilise le « dynamic binding ». Le lien est fait au moment où l'objet reçoit un message.

Avantage de la version statique :
On évite la phase de recherche à chaque exécution (gain en performances).

Avantage de la version dynamique :
La souplesse d'utilisation + définition, utilisation de nouvelles classes au run-time (évite la recompilation totale au lors de la modification d'une classe).

Exemple

```
if(flag) {
      myvar = new C1() ;
}
else {
      myvar = new C2() ;
}
```

Cette situation ne peut pas être résolue avec un « static binding » car tout dépend de l'état de `flag` qui n'est pas connu avant l'exécution.

Le mot clé super

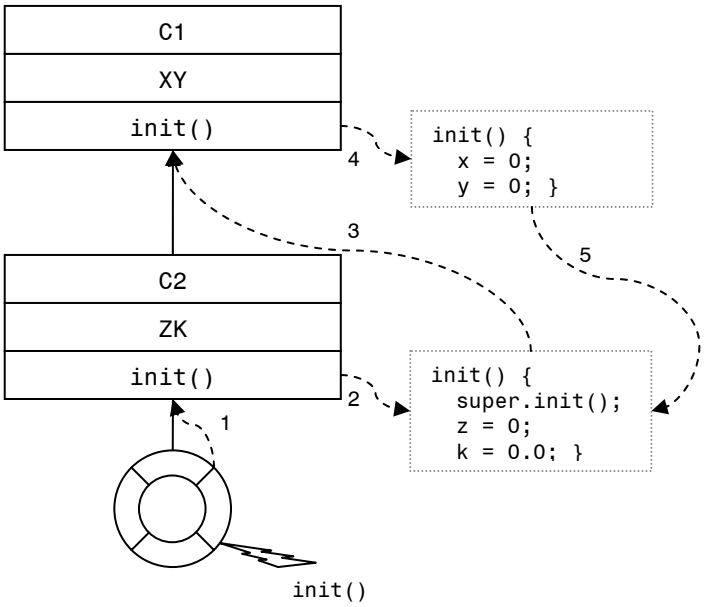
Ce mot permet à une méthode d'appeler une méthode de même signature située dans une super-classe de la classe actuelle (celle qui contient la méthode).

On utilise le mot clé super de la manière suivante :

```
super.methode(…);
```

où « `super` » désigne le receveur et « `methode(…)` » désigne la méthode qui l'on appelle dans la super-classe.

Exemple
Soit la situation suivante :



Que nous apprend cette figure ?

Quand on fait appel à une méthode avec le mot clé `super`, le système recherche la méthode de la super-classe de la classe qui contient le mot clé `super`. Mais la méthode n'est pas forcément trouvée dans la super-classe immédiate.

En imaginant une classe C3 entre C1 et C2 dans cette même hiérarchie, si cette classe C3 ne contient pas de méthode `init()`, l'objet serait allé chercher la méthode `init()` de la classe C1.

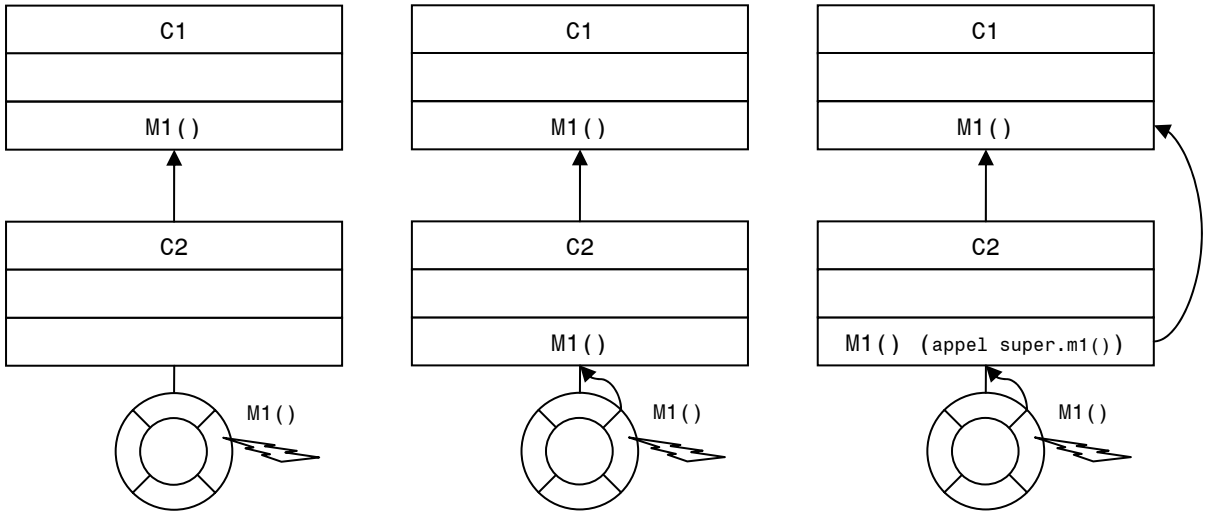
L'ordre par lequel passe la recherche est noté sur les flèches.

L'utilisation d'un appel avec super

Spécialise une méthode déjà déclarée dans les super-classes en réutilisant ces méthodes.
Comme on favorise le polymorphisme, si la méthode a la même sémantique, elle a forcément la même signature.

Trois situations d'héritage

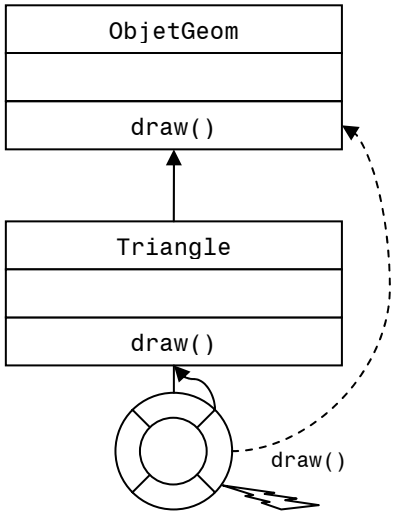
On distingue principalement ces 3 cas :



Le principe de substitution

C'est un corollaire de la généralisation.
Dans toutes substitutions où on attend une instance d'une classe, on peut lui substituer une instance d'une sous-classe. Cela implique que toute instance d'une sous-classe doit, conceptuellement, pouvoir être considérée comme une instance de la super-classe.

Exemple
La hiérarchie d'héritage doit représenter une vraie généralisation / spécification.

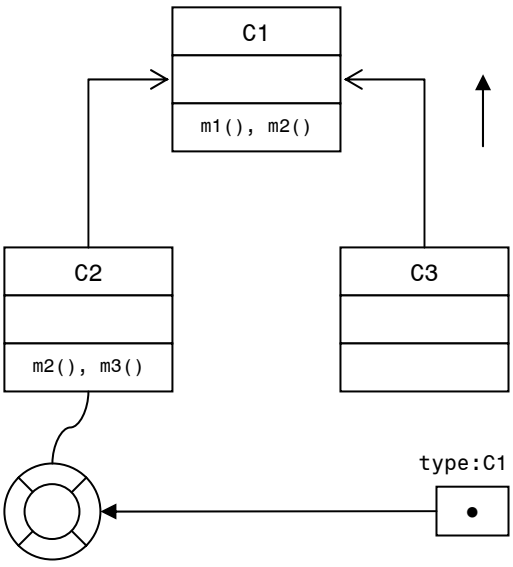


Dans la situation de cet exemple, c'est comme si l'objet avait été créé à partir de la classe `ObjetGeom`.

Typage par super-classe

Soient la situation et le code Java suivant :

```
C1 x;
x = new C2() ;
```



Dans ce cas, il est possible d'envoyer à `x` tous les messages définis dans C1 même si, *in fine*, les méthodes sont implémentées dans C2.

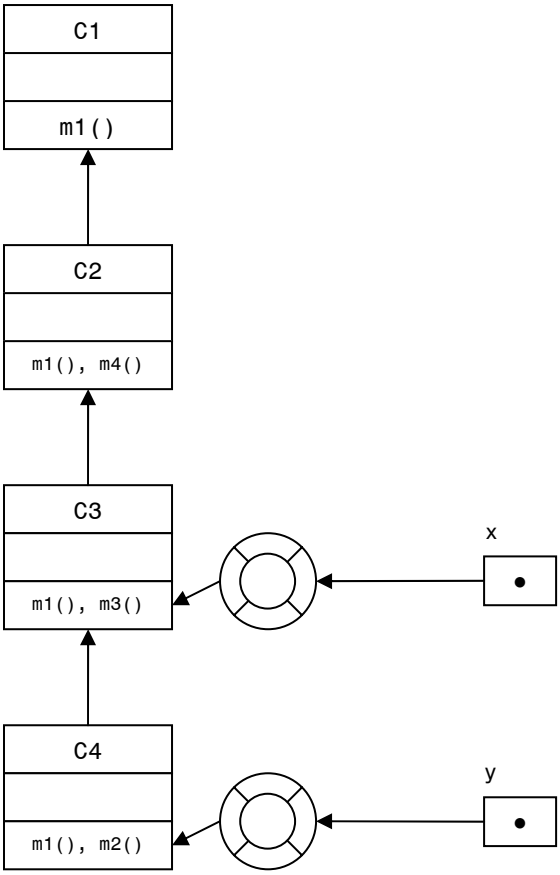
On peut alors envoyer à `x` :

- `m1()`
- `m2()`

En revanche il n'est pas possible d'envoyer `m3()` car C1 ne supporte pas `m3()`.

Ceci démontre que le typage par classe est l'équivalant d'un type abstrait, c'est-à-dire un type défini par les opérations qu'on peut faire sur lui.

Exemple
Etudions un autre exemple :



En regard de ce schéma on peut écrire :

```
C1 x;
C2 y;

x = new C3();
y = new C4();

x.m1();
y.m1();
```

Cependant les instructions suivantes sont fausses:

```
x.m3() ;
y.m2() ;
x.m4() ;
```

Classe abstraite

Définition *La classe abstraite*
Elle représente un concept abstrait du domaine. Une classe abstraite ne peut pas être instanciée car une partie de son protocole doit être spécialisé dans une classe concrète pour que la description soit complète.

Déclaration

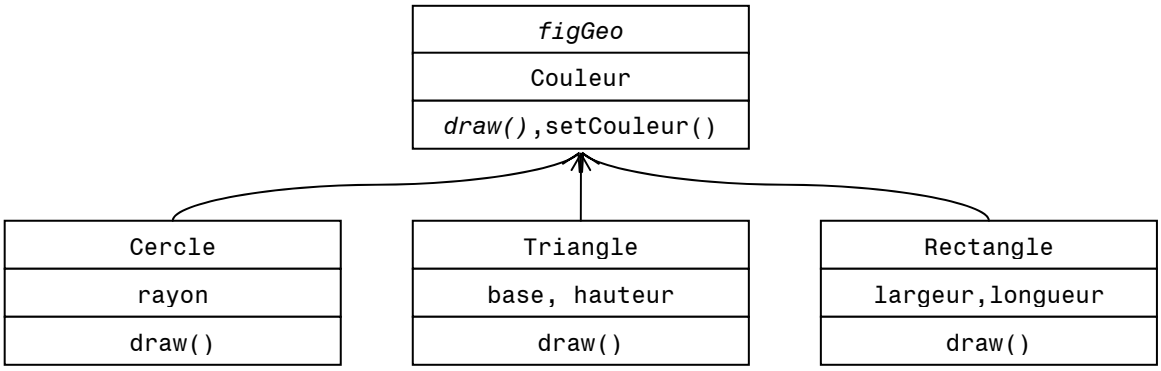
```
abstract class FigureGeo {…}
```

Les méthodes d'une classe abstraite peuvent être soit abstraites soit concrètes. Une méthode abstraite est une déclaration limitée à la déclaration de la méthode, sans implémentation. Une méthode abstraite doit être implémentée dans les sous-classes concrètes. Si ce n'est pas le cas, la sous-classe doit être abstraite.

Exemple
Dans le schéma ci-dessous :

- La classe `FigGeo` est abstraite.
- La méthode `draw()` de cette même classe est abstraite et a la forme : `void draw()` ce qui oblige les sous-classes concrètes à implémenter cette méthode.

- Les classes `Cercle`, `Triangle` et `Rectangle` sont concrètes ce qui oblige qu'elles possèdent des méthodes `draw()` implémentées.



On remarque que certains mots sont en italique. Cela signifie, en UML, que la classe ou la méthode est *abstraite*.

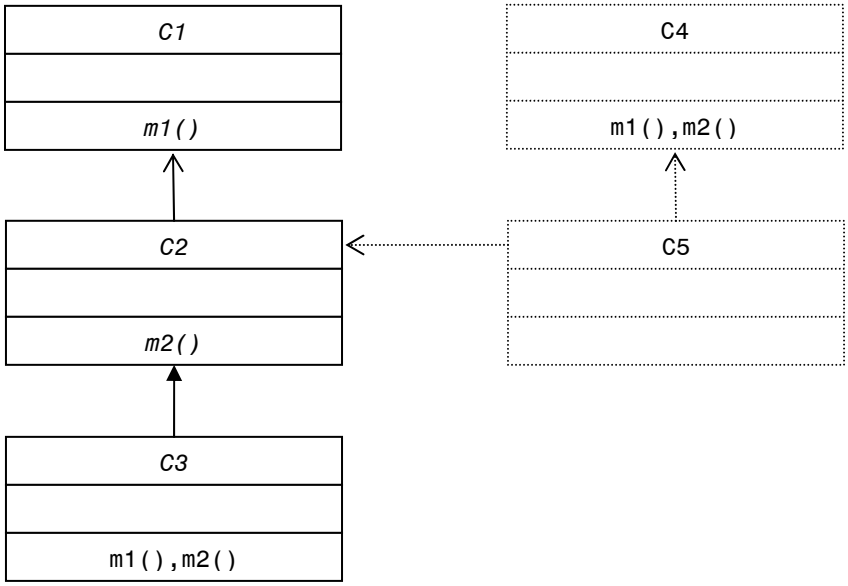
Remarque : Une variable peut être typée avec une classe abstraite. Cela oblige toute instruction à passer par une sous-classe.

Toute sous-classe d'une classe abstraite qui n'implémente pas l'ensemble des méthodes abstraites de sa super-classe doit être déclarée abstraite.

Exemple

```
abstract class FigGeom {
      void draw() {…}
      abstract int perimetre() {…}
}
class Cercle extends FigGeom {
      void draw() {…} //Spécification
      int perimetre() {…} //Implémentation de la méthode abstraite
}
```

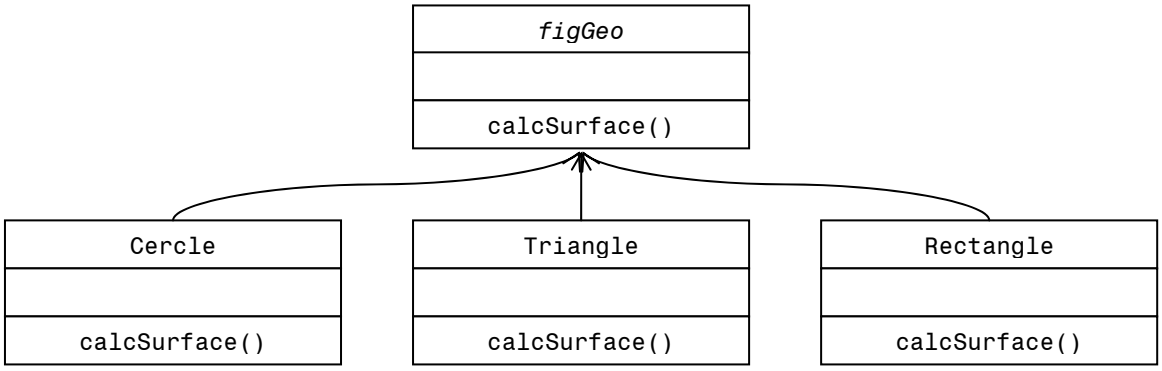
Une classe peut être instanciée (c'est-à-dire concrète) que si l'ensemble de ses méthodes abstraites sont implémentées.



La classe C3 serait concrète si et seulement si `m1()` et `m2()` étaient implémentées.

Sur le schéma précédent, les classes et liens d'héritage en pointillés représentent une situation d'héritage multiple (C++). La classe C5 hérite de l'implémentation de `m1()` et `m2()` depuis C4 donc elle peut être concrète.

Pour en terminer avec ce chapitre, revenons encore quelques instants sur notre exemple des figures géométriques :



En créant des objets Cercles, Triangle et Rectangle, on s'apercoit qu'ils ne sont pas issus d'une même classe.
Si on veut envoyer le même message à tous les objets, comment typer la variable qui va référencer ces objets ?

```
x.calcSurface() ;
```

Le type de x sera la super-classe commune. Dans cette super-classe on déclare l'ensemble des signatures des messages que l'on veut envoyer aux instances des sous-classes.

Cela nous donne, en pseudo code :

```
FigGeom x;
Boucle :
      prochain objet → x ;
      x.calcSurface() ;
```

Super() ≠ Super

`super()` n'a rien à voir avec la pseudo variable `super`. C'est une déclaration liée au constructeur.

Si la classe C2 hérite de C1, et que l'on a

```
new C1();
new C2();
```

alors cette seconde ligne de code exécute le constructeur de C2. Cependant on aimerait réutiliser le constructeur de C1. Comment faire ?

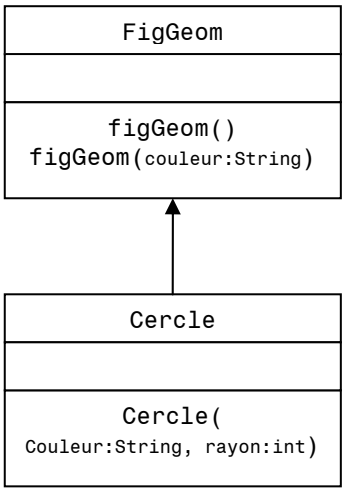
On va utiliser :

```
super(...);
```

En lieu et place des « ... », on met les paramètres qui correspondent à signature du constructeur de la super-classe que l'on veut exécuter.

Voyons de suite un exemple :

Exemple



Supposons que dans la classe `Cercle` le constructeur soit défini par :

```
public Cercle(String couleur, int rayon) {
      super(couleur) ;
      ...
}
```

Alors on va invoquer le constructeur de la super-classe avec un paramètre de type String.

Si au lieu de `super(...)` on avait mis `this(...)`, alors on aurait invoqué l'exécution du constructeur de la même classe qui possède les paramètres compatibles (à savoir un constructeur de la classe `Cercle`).

Notation de la pseudo variable this

Soit le code suivant :

```
m1(String toto) {
      this.toto = toto ; }
```

Comme le paramètre a le même nom qu'une variable de l'objet, on distingue les deux en la pseudo variable `this` point le nom de la variable.

Comment représenter le this ?

Graphiquement on peut voir le `this` comme :

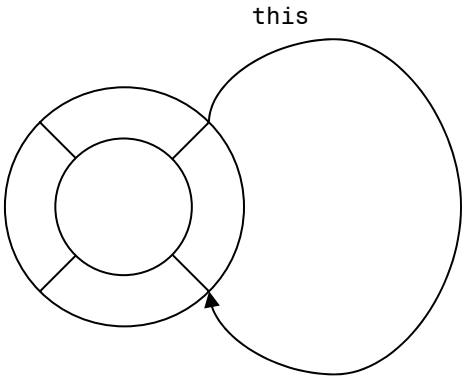
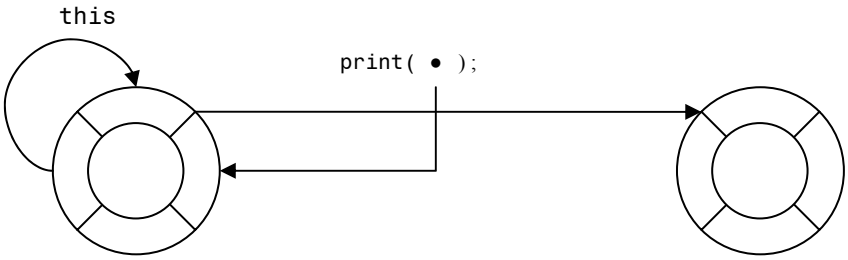


Fig. V, Représentation du this

On va voir maintenant comme on peut utiliser efficacement cette pseudo variable lors de passages de paramètres entre objet.

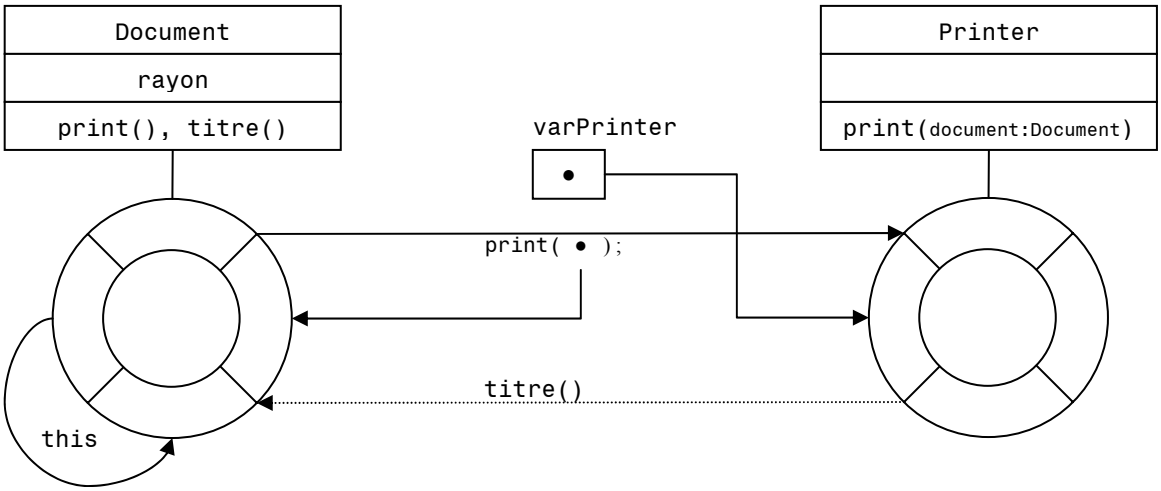
Soit la situation suivante :



• représente une référence à soit-même (à l'objet de gauche).

L'objet de droite est une imprimante, et l'objet de gauche souhaite dire à cette imprimante : « Imprime moi ! »
Il va envoyer à l'objet imprimante le message `print(moi)` où `moi` est l'adresse de l'objet de gauche.

Formellement :



Dans la classe `Document`, supposons avoir :

```
print() {
      varPrinter.print(this) ;
}
```

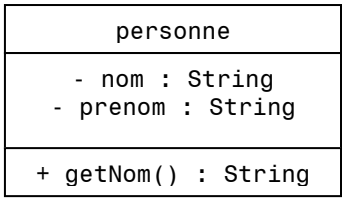
Dans la classe `Printer`, supposons avoir :

```
print(Document document) {
      document.titre() ;
}
```

Ainsi l'objet `printer` va renvoyer un message `titre();` à l'objet `document`. On assiste ici à un échange de messages entre les deux objets.

Modificateurs de visibilité

A tout élément de modélisation des classes UML, on peut associer une « visibilité ».
Nous avions déjà vu dans ce cours les notations UML, mais rappelons-les :

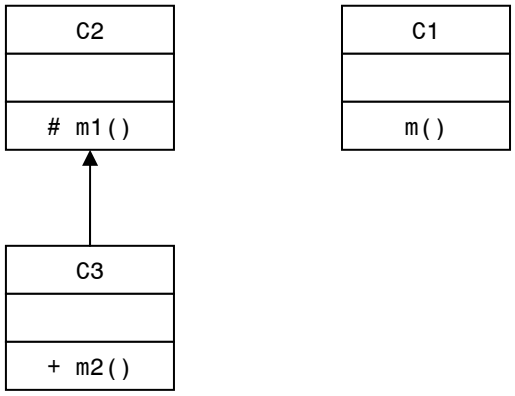


- `public (+)`
  L'élément de la classe (attribut ou méthode) est visible par tous les objets qui voient la classe.

- `private (-)`
  L'élément de la classe (attribut ou méthode) est invisible en dehors de la classe.

- `protected (#)`
  L'élément de la classe (attribut ou méthode) est visible dans la classe et ses sous-classes.

Exemple
Soit la hiérarchie d'héritage suivante :



Ici on peut invoquer `m1()` car C3 est une sous-classe de C2.

Cependant `m()` ne peut pas invoquer `m1()` car ce n'est pas une sous-classe de C2.

Attention, cet exemple est valable pour l'UML.

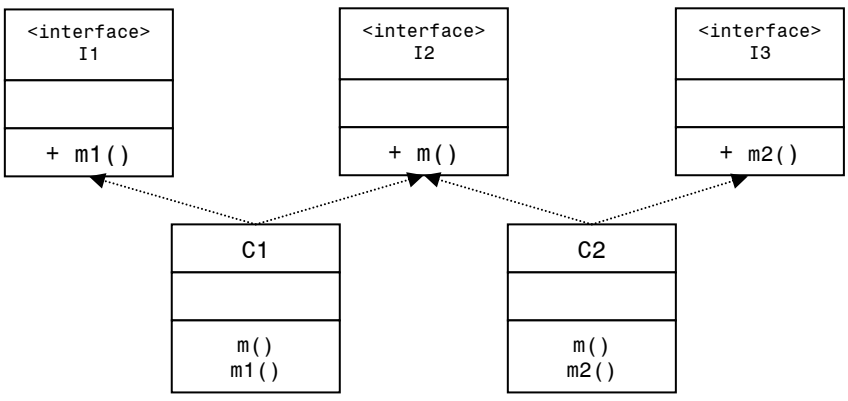
Recommandation

En UML comme en Java, les attributs doivent être privés (–), ceci pour respecter une stricte encapsulation.
Si on doit accéder à un attribut privé, on le fait en déclarant des méthodes d'accès (`set` / `get`).

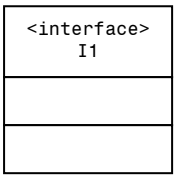
Interface et héritage

Un ensemble de classes peut implémenter plusieurs interfaces.

Ainsi la situation suivante est tout à fait possible :



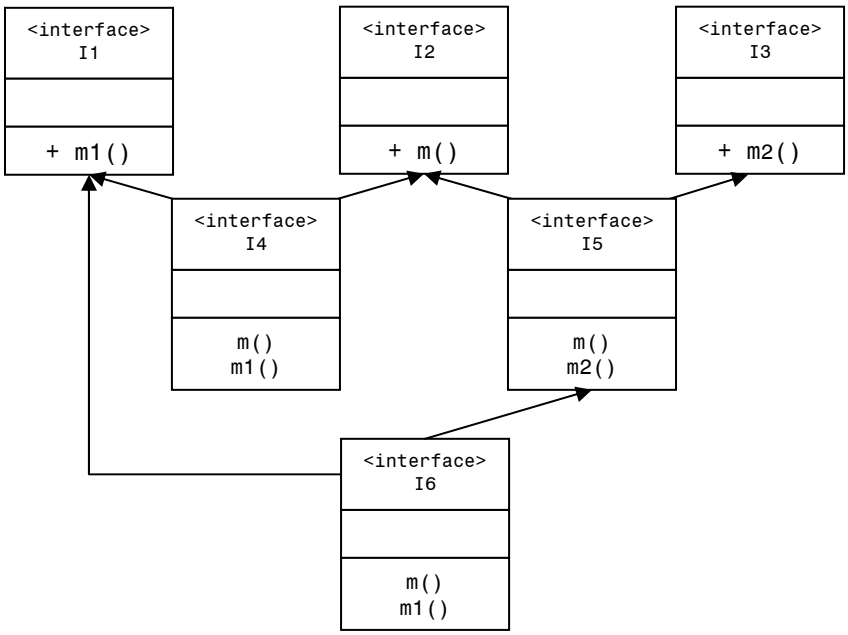
La déclaration de l'interface en UML est le symbole de classe avec le label `<interface>`.
`<interface>` est un stéréotype, c'est-à-dire un symbole UML avec une sémantique spécifique déclarée par l'étiquette du stéréotype.



Dans le cas d'une interface, on a en lieu et place de la déclaration des méthodes pour une classe, la déclaration de l'ensemble des signatures déclarées dans l'interface.

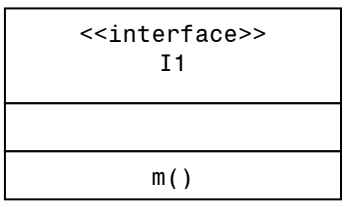
La visibilité d'une signature dans une interface est `public`.

Avec les interfaces il n'y a pas de problème avec un héritage multiple. On peut ainsi avoir la situation suivante :



Il n'existe pas d'implémentation de méthode dans une interface, on n'a donc pas d'ambiguïté d'héritage.

Une interface peut être utilisée pour typer une variable et pour typer l'objet retourné par une méthode, c'est-à-dire :

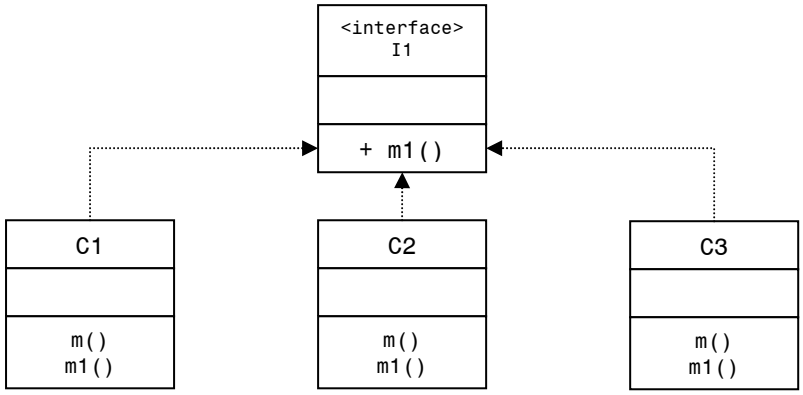


On peut alors écrire :

```
I1 x ;      //On déclare que tout objet associé à x support le protocole
            //(méthode) m().
public I1 methode1() ;
```

En fait, on va typer les objets en fonction des opérations que l'on peut faire sur l'objet.

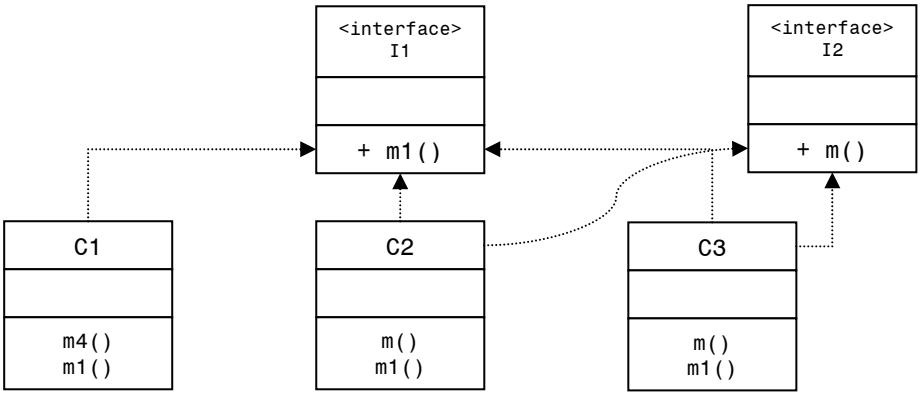
Faisons évoluer un peu la situation :



Dans ce cas, on peut alors faire :

```
I1 x ;
x = new C1() ; ou x = new C2() ; ou x = new C3() ;
```

Rajoutons encore une interface :



On peut alors écrire, en plus des deux lignes de codes précédentes :

```
I2 x ;
x = new C2() ; ou x = new C3() ;
```

Exemple

Reprenons nos fameuses figures géométriques.
Soit le code Java suivant :

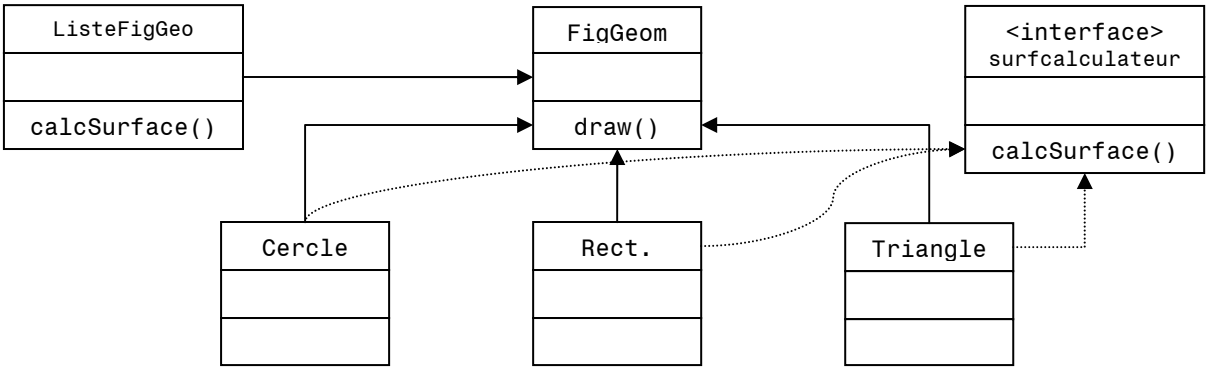
```
1     class ListeFigGeo {
2           private Vector figures ;
3           public void addFigureGeom (Object o) {
4                 figures.addElement(o) ;
5           }
6           public int calcSurface() {
7                 int surface ;
8                 Surfcalculateur sc ;
9                 Iterator iter = figures.iterator() ;
10                while(iter.hasnext()) {
```

```
11                      sc = (Surfcalculateur)iter.next() ;
12                      surface = surface + sc.calcSurface() ;
13                  }
14                  return surface ;
15          }
16      }
```

Remarques sur le code

Ligne 8 : Les objets supportent `Surfcalculateur`.
Ligne 9 : L'itération sur la collection retourne un objet de type `Iterator`.
Ligne 10 : On cherche à savoir s'il y a un élément suivant. `next()` retourne un objet de type `Object`, il faut donc faire une conversion explicite.
Ligne 12 : On peut utiliser la méthode `calcSurface()` car `sc` est du type `Surfcalculateur`.

Représentation de la situation qui a permis d'écrire ce code :



Les packages

En UML, un package se représente de la manière suivante :

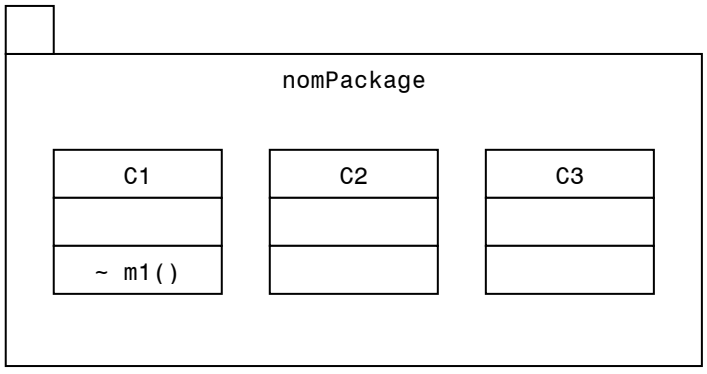


Fig. VI, Représentation UML d'un package

Un package représente un groupement d'éléments de modélisation.
Le package permet d'éviter des collisions entre les noms des éléments de modélisation. Les éléments à l'intérieur d'un package sont protégés vis-à-vis de l'extérieur à moins de les rendre explicitement visibles.

Le modificateur de visibilité de package est : ~

Dans la figure ci-dessus, `m1()` n'est visible que par les éléments du package.

Les classes dans un package

Il existe deux modificateurs possibles :

- ~ : (par défaut) package.
- + : public

Une classe de visibilité `public` est visible à l'intérieur du package.
Une classe de visibilité `package` n'est visible qu'à l'intérieur même du package où elle est déclarée.

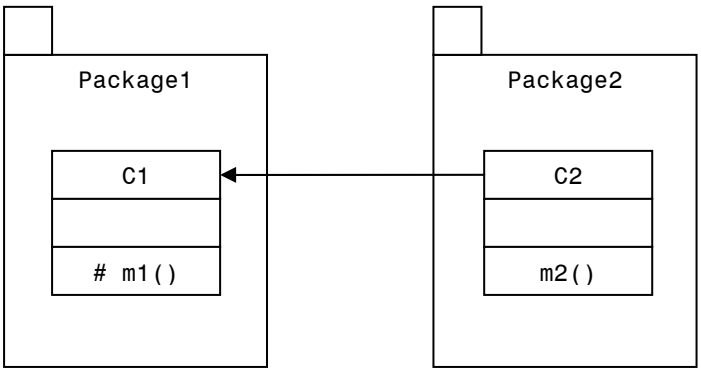
Si on veut pouvoir exécuter une méthode d'une classe depuis l'extérieur du package dans lequel elle est, il faut :

1. Que la classe soit `public`.
2. Que la méthode soit `public`.

Package & protected

Rappelons que `protected` implique une visibilité dans les sous-classes.

Si on a :



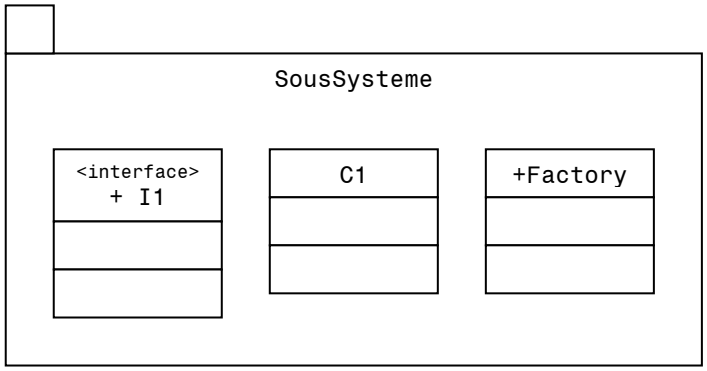
`m2()` peut référencer `m1()` car C2 est une sous-classe de C1.

```
m2() {
       m1() ;
       …
}
```

La visibilité `protected` traverse les frontières package.

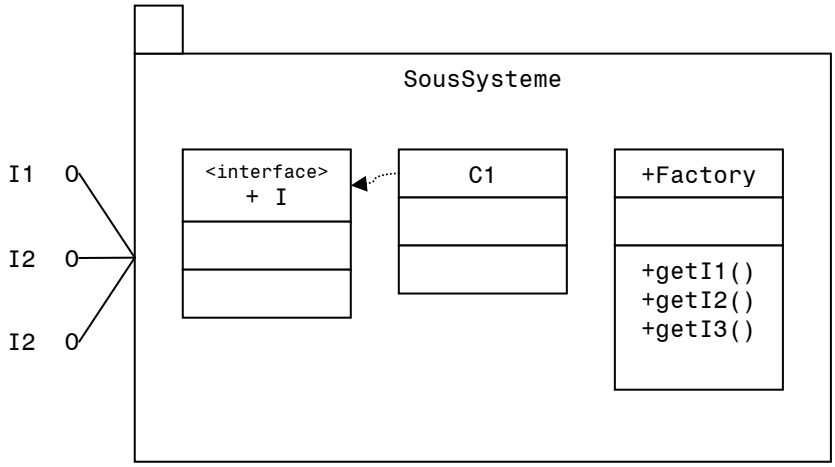
La classe « Factory »

Soit :



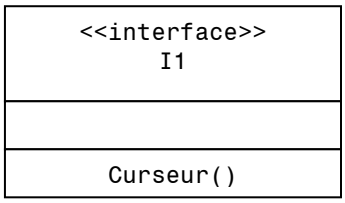
Dans ce cas, nous avons une seule classe `public` qui représente l'unique point d'entrée dans le `SousSysteme`. Les autres classes sont invisibles.
On appelle, en général, cette classe la classe « `Factory` ».

Voyons cette situation plus en détail :



`Factory` retourne un objet qui correspond à l'interface choisie (`I1, I2, I3`).

On pourrait alors avoir l'implémentation suivante :



La méthode `getI1()` s'implémente de la façon suivante :

```
getI1() {
      return new C1() ;
}
```

Après avoir fait un `import SousSyseme`, on peut écrire :

```
I1 x = (new Factory()).getI1() ; // Retourne l'objet du package qui
implémente I1.
x.Curseur() ;
```

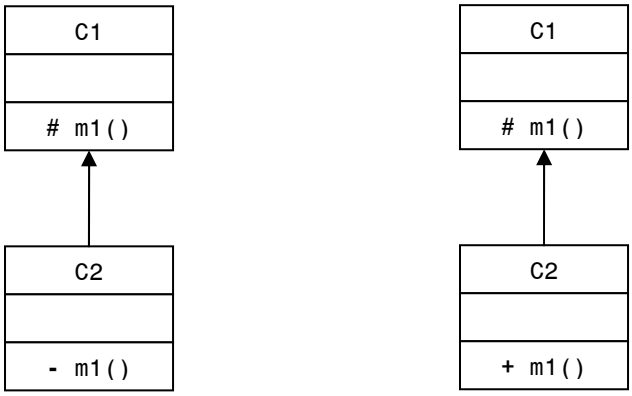
La visibilité & Java

Les concepts de visibilité sont semblables à ceux définis en UML. La grande différence est que la visibilité `protected` de Java est équivalente à `package` + `protected` de UML. C'est-à-dire que les éléments déclarés `protected` en Java sont visibles dans tout le package dans lequel ils se trouvent.

Attention : On ne peut jamais diminuer la visibilité d'un élément dans une sous-classe. On ne peut que l'augmenter.

Pour illustrer cette dernière remarque, voyons un exemple.

Exemple



L'attribut « -« dans la version de gauche n'est pas légale.
En revanche la version de droite est tout à fait correcte.

Cela nous amène à parler des hiérarchies de visibilité.

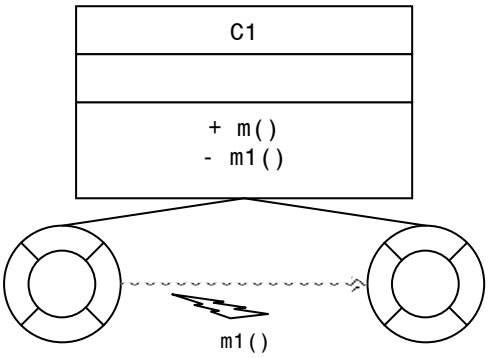
Hiérarchie de visibilité

1. private (-)
2. protected (#)
3. package (~)
4. public (+)

On ne peut que descendre dans cette hiérarchie (c.f. exemple ci-dessus).

Une question vient alors à l'esprit : « Pour quelle raison a-t-on une telle restriction ? »
Car sinon le principe de substitution serait violé.

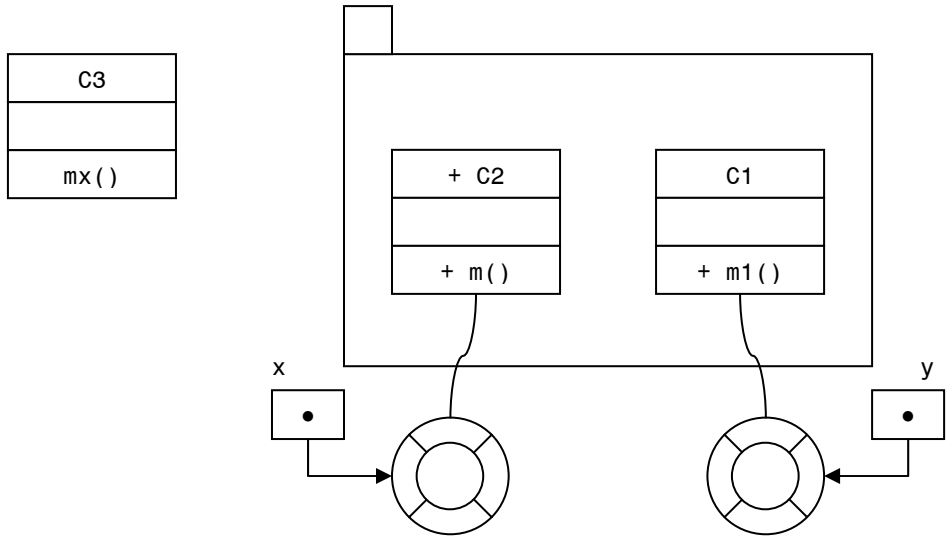
La déclaration de la visibilité d'une méthode s'interprète par rapport à la déclaration du code, c'est-à-dire :



La visibilité d'une classe indique si on voit la classe à l'extérieur. Mais il est possible de manipuler des instances d'une classe non-visible depuis l'extérieur.

Voyons comment cela est possible en regardant cet exemple :

Exemple

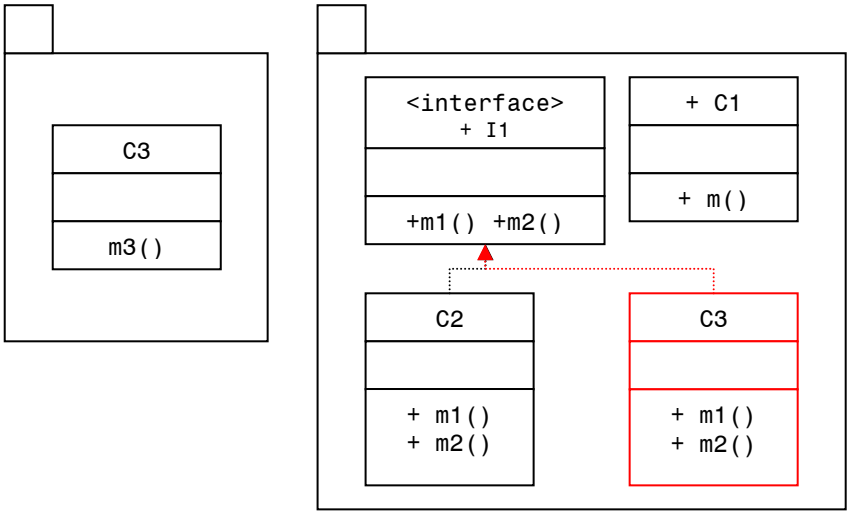


Dans cet exemple, `C1` est de visibilité `package`.

```
mx(){
      C2 x = new C2() ;
      Object y = x.m() ;
}

C2 m() {
      return new C1() ;
}
```

Ainsi, dans l'objet `y` on a une instance de `C1` qui est une classe invisible.

Modifions un peu cette situation :



Ici, `C2` est de visibilité `package`.

Sans se préoccuper des modifications de couleur rouge, on peut écrire que :

```
public I1 m() {
      return new C2() ; }
      m3() {
```

```
            C1 x = new C1() ;
            I1 y = x.m() ;
            y.m1() ;
      }
```

On a donc que `y` contient une instance de la classe `C2` « invisible » pour `C3`.

Si, maintenant, on tient compte des modifications de couleur rouge sur le schéma précédent, on peut réécrire les deux premières lignes comme :

```
public I1 m() {
      return new C3() ; }
```

Tout le reste est inchangé ! Cela représente le concept d'encapsulation au niveau package.

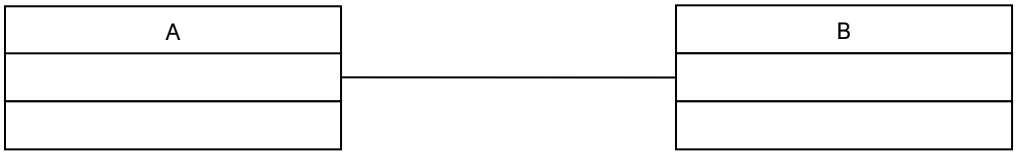
Polymorphisme & élimination de test de cas

Après tous les exemples donnés par M.Dugerdil sur transparents, il en ressort que, si dans une méthode on trouve des tests de type de l'objet, il y a une erreur de composition quelque part.

Associations

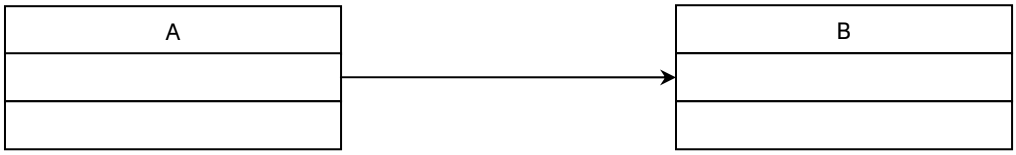
Définition *Une association*
Déclaration que les instances des classes associées peuvent communiquer.

Association bidirectionnelle



L'association bidirectionnelle est symbolisée par un trait reliant les deux classes.
Dans ce cas-ci, les instances de `A` et de `B` peuvent communiquer ensemble.

Association directionnelle



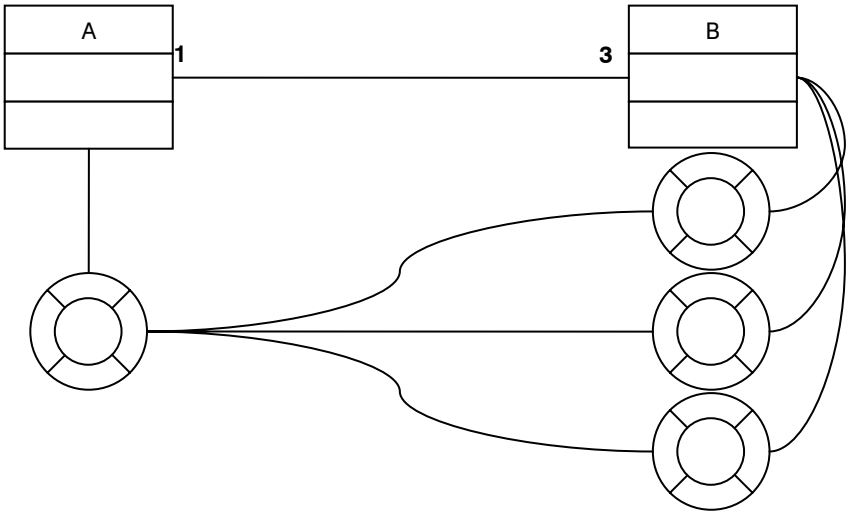
L'association directionnelle est symbolisée par une flèche reliant les deux classes.
Dans ce cas-ci, les instances de `A` peuvent communiquer avec les instances de `B`, mais l'inverse n'est pas possible.

En Java, il y aura une variable (attribut) dont le type est égal à `B`.

Remarque : On appelle `link` le lien qui unit deux objets « associés ». Dans les versions UML > 1.5. le « `link` » devient une « `instance d'association` ».

Cardinalité d'une association

Soit



Ici une instance de `A` doit être associée à trois instances de `B`.
Le « 3 » veut dire : exactement 3.
Le « 1 » veut dire : exactement 1.

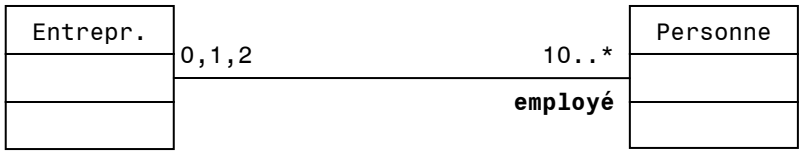
Syntaxe pour la cardinalité

- 1 = exactement 1.
- 1..3 = 1 ou 2 ou 3 (notion d'intervalle).
- 1,3,5 = exactement 1 ou exactement 3 ou exactement 5.
- 0..1 = 0 ou 1 (cardinalité optionnelle).
- * = 0..n.

Exemple
Toute instance de `A` peut être associée à 2, 3 ou 8 instances de `B` et toute instances de B à 5 ou N instances de A, avec N>5.

▲ L'association est une représentation de la communication entre instances. Ce n'est pas une « relation » au sens base de données.
Deux instances vont collaborer pour résoudre un problème (communication).

Exemple
Une entreprise communique avec ses employés qui sont au moins dix. Un employé doit communiquer avec son entreprise mais, s'il travaille à temps partiel, il peut être associé à deux entreprises. Par contre il peut aussi être sans travail (associé à zéro entreprise).

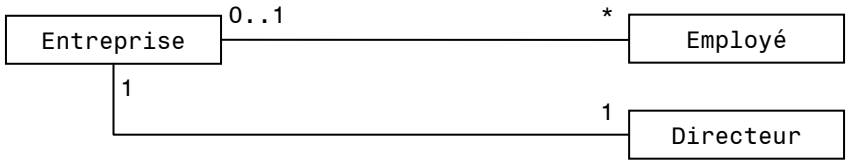


Avec cet exemple, on introduit la notion de « rôle ». En UML, un rôle est indiqué en dessous de la cardinalité. (ici `employé`).

Définition *Un rôle*

Façon dont un objet est « vu » par un autre.
On le note du côté de l'association où le rôle est « joué ».

Augmentons maintenant le schéma de classes pour inclure le directeur :



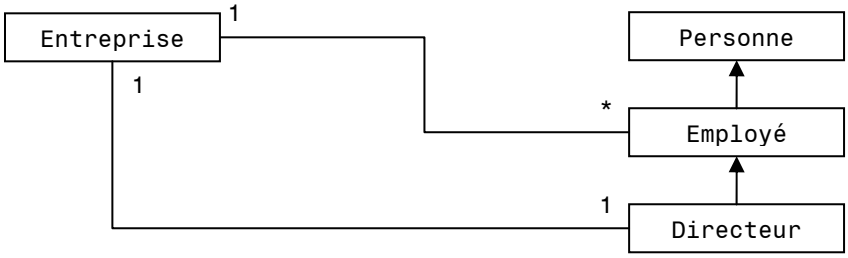
Ici nous n'utilisons absolument pas la notion de rôle. Ainsi, si le directeur est aussi un employé de l'entreprise, il faudra deux objets pour le représenter. Cela n'est donc pas une bonne modélisation.

Nous allons mettre de côté cet exemple et voir quelques exemples de « rôle » puis nous reviendrons à cet exemple pour le modéliser avec la notion de « rôle ».

Exemple
Représentation de rôles sous forme de sous-classes :

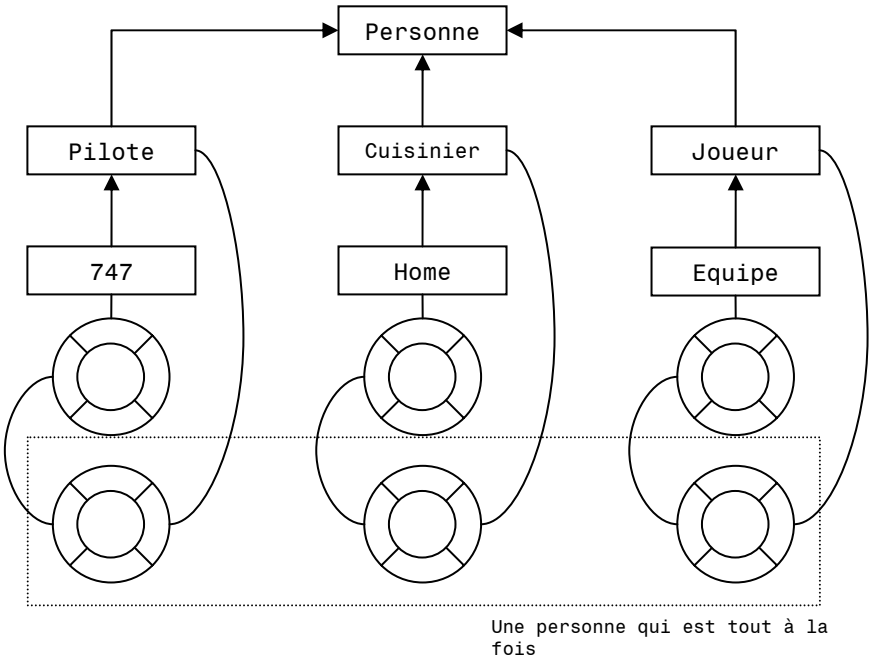
Cette modélisation est juste si une personne donnée ne joue d'un seul rôle dans l'entreprise et que son rôle est immuable.

Maintenant, si on veut qu'une entreprise ait un seul directeur, il faut :

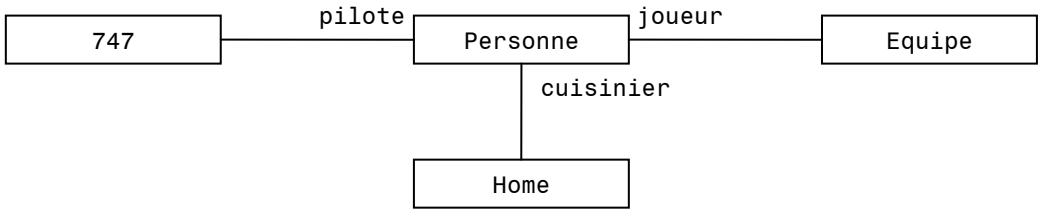


Exemple
Représentation d'une personne et de ses activités.
Disons qu'une personne peut être : pilote (un 747), cuisinier (at Home), joueur de foot (dans une équipe).

Alors on a le modèle suivant (sans notion de rôle).

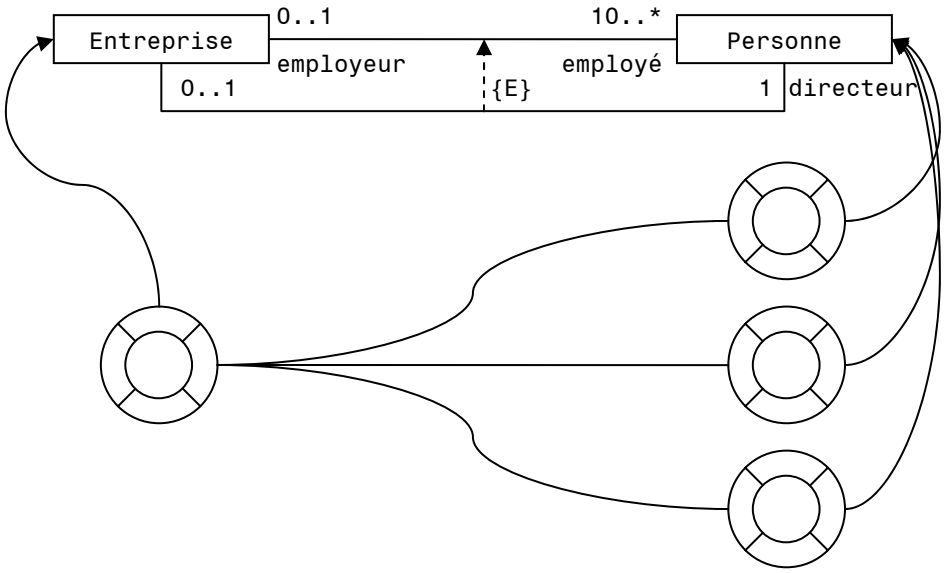


Voyons maintenant ce même exemple, mais en y ajoutant la notion de rôle :



Maintenant que nous avons bien compris cette notion de « rôle », nous pouvons revenir à l'exemple que nous avons laissé en suspend un peu plus haut.

Ainsi le cas du directeur et des employés devient, avec la notion de rôle :



On remarque ici l'introduction d'une contrainte d'association dont la notation UML est une flèche en trait tillés et entre {} une contrainte exprimée en langage naturel ou formel.

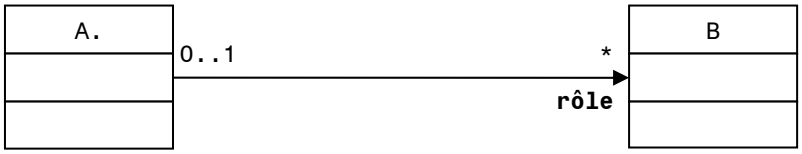
Implantation d'associations en Java

Soit


```
class A {
      private B role ;
…
}
```

Si le rôle n'existe pas, on choisi un nom de variable représentant la classe, comme : `myB, bObject`, …

Soit une autre situation




```
class A {
      private Vecteur role ;
      private void addB(B objet) {
            role.addElement(objet) ;
      }
}
```

La méthode `addB()` doit être la seule qui manipule le vecteur. Cette méthode permet de contrôler que le vecteur `role` contienne que des objets de type `B`.
C'est aussi cette même méthode qui va valider la cardinalité (elle compte le nombre d'objets du vecteur et s'il est plus grand que la cardinalité, elle retourne une erreur).

En conclusion, la cardinalité d'une association ne peut être validée que par une méthode qui manipule la collection.

Demande d'analyse d'un cahier des charges

Cela consiste à :

- Trouver les objets.
- Déterminer les associations.
- Etablir les comportements des objets (trouver les méthodes).

Nous allons utiliser la méthode d'analyse RDD (Responsability Driver Design). Elle consiste à analyser les objets en passant par leur responsabilité.

Définition *La responsabilité*

- Effectuer un calcul.
- Stocker / Retourner une valeur.
- Connaître un autre objet.

Démarche

1. Identifier un premier ensemble d'objets. C'est les noms dans le cahier des charges. Cela nous donne un ensemble de classes potentielles.
   Attention à lister tous les noms du cahier des charges et filtrer ces noms pour ne retenir que les objets à implanter.

2. Pour chaque objet, identifier ses responsabilités (quel est son rôle dans le système).
   Les responsabilités sont issues des verbes dans le cahier des charges. Pour chaque verbe il faut déterminer « qui est responsable de cette action ? » : un objet existant ou un nouvel objet ?

3. Pour chaque responsabilité assignée à un objet, on se demande : « Peut-il l'assumer seul ou doit-il être aidé par un spécialiste ? »
   On fait une analyse de l'information nécessaire à l'accomplissement de la tache.

   Si l'information n'est maîtrisée que partiellement par l'objet, il devra collaborer avec un spécialiste.

Exemple
Impression d'un document.

Ici le « `document` » est le responsable de l'impression du document.
« `Printer` » est le collaborateur. Logiquement l'association représente la collaboration.

Filtrage des noms

Rappel : Enumérer tous les noms du cahier des charges.

Le filtrage consiste à :

- Eliminer les noms qui représentent des attributs. De tels noms ne sont pas manipulés isolement : on a, par exemple, le nom ou le prénom *de* la personne.
- Eliminer les noms qui représentent des entités externes au projet.
- Choisir l'objet « fondamentalement » et non pas ses rôles. Les noms correspondants aux rôles sont écartés (mais enregistrés pour l'analyse future).
- Eliminer les noms représentants les responsabilités.

Il faut ensuite analyser les responsabilités représentées par les verbes du cahier des charges et les associer aux objets.

Quand toutes les responsabilités ont été associées aux objets, on parcourt les objets et ceux qui n'ont pas de responsabilité sont éliminés.

Notation pour RDD

Une carte CRC :

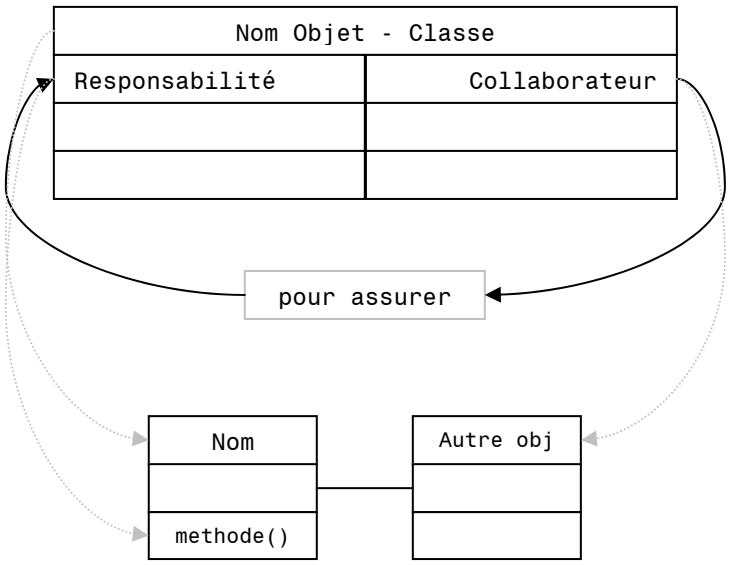
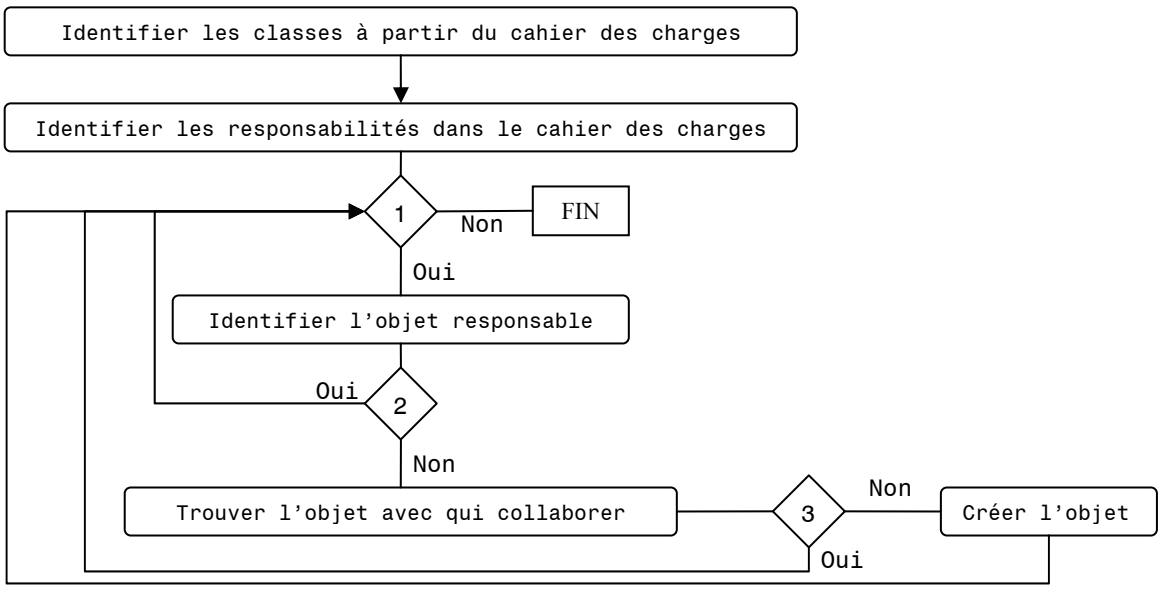


Fig. VII, une carte CRC

Heuristique pour assigner les responsabilités

1. Assigner les responsabilités de manière aussi générale que possible.
   C'est-à-dire, si possible, à la super-classe.
2. Gérer les informations à un seul endroit.
   C'est-à-dire que l'objet qui contient l'information assume toutes les responsabilités par rapport à cette information.
3. Pas d'objet omniscient.
   Pour ce faire il faut répartir les responsabilités de manière aussi granulaire que possible. Il faut donc déplacer les calculs ou les procédures dans les objets qui possèdent les informations pour effectuer le calcul.

Diagramme d'activité pour représenter le processus :



Exemple: bibliotheque

Réaliser un système de gestion de bibliothèque, gérant les prêts et les réservations de livres faites par les utilisateurs inscrits.

Chaque personne désireuse d'utiliser les services de la bibliothèque doit s'inscrire au moyen de ses nom et adresse.

Chaque client peut emprunter au maximum 5 livres à la fois. Il est toutefois possible de réserver les livres non disponibles. Les réservations pour un même livre sont traitées dans l'ordre d'arrivée.

Le responsable de la bibliothèque doit pouvoir interroger le système de façon à déterminer:

- Si un livre donné, identifié par ses titre et auteur, est emprunté ou réservé, par qui et jusqu'à quand pour le prêt ?
- Quels sont tous les livres empruntés et réservés par une personne donnée ?
- Quel est le nom et l'adresse de toute personne inscrite ?.

Légende des décisions :

« 1 » : Une responsabilité n'est pas assignée ?
« 2 » : L'objet peut-il assumer seul la responsabilité ?
« 3 » : Est-ce que l'objet existe ?

Analyse du cahier des charges de « La Bibliothèque »

Tout d'abord il nous faut chercher les noms sur le cahier des charges. Ils sont soulignés en rouge sur l'image qui précède cette page.

Parmi les mots retenus, il faut faire attention à :

Ligne 1 : « Système de gestion ». Il faut l'éliminer de la liste car il représente la totalité du système.

Ligne 2 – 3 : Il y a redondance entre « utilisateur » et « personne ». En effet, « utilisateur » est un rôle de la personne. On élimine donc « utilisateur ».

Ligne 3 : « Service de la bibliothèque ». A éliminer car ce terme est générique. Il signifie, ici, emprunter – réserver.

Ligne 4 : Nom – adresse sont des attributs, on élimine.
« maximum » est une contrainte.

Ligne 7 : « Ordre d'arrivée » est une contrainte de traitement (à cause de « traitées »).

Ligne 8 : « Responsable » est un rôle de la personne, à supprimer.

Ligne 9 : Titre – auteur sont des attributs, on supprime.

Ligne 12 : Nom – adresse sont des attributs, on les élimine.

Au final, il nous reste :

- `Bibliothèque`
- `Prêt`
- `Réservation`
- `Livre` (attributs = titre, auteur)
- `Personne` (attributs = nom, adresse).

`Bibliothèque` va gérer les prêts et les réservations. Elle devra donc collaborer avec `Prêt` et `Réservation`.

Il y aura des utilisateurs inscrits. Cela implique de conserver une liste des utilisateurs. C'est aussi `Bibliothèque` qui va gérer les utilisateurs inscrits, d'où une collaboration avec `Personne`.

Reprenons une analyse ligne par ligne du cahier de charges :

Ligne 3 : C'est `Personne` qui va gérer le nom et l'adresse.

Ligne 5 : « Chaque client peut emprunter (=prêt) 5 livres ».
Ici se pose la question de savoir qui gère les emprunts ? C'est `Bibliothèque`.

Ligne 6 : « Réserver des livres non disponibles ».
`Bibliothèque` gère les livres. Elle collabore donc avec `Livre` car seul le livre lui-même sait s'il est emprunté ou pas.

Ligne 8 : « … interroger le système… ».
Ici on est hors du contexte (utilisateur externe au système). On ne retient pas cette responsabilité.

Ligne 9 : Quel objet a la responsabilité de savoir qui a emprunté et réservé et jusqu'à quand ? C'est, respectivement : `Prêt`, `Réservation` et `Prêt`.

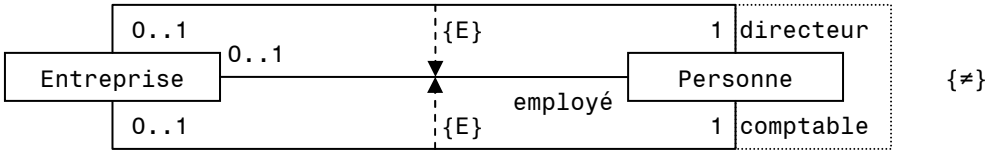
Donc `Prêt` gère : Qui a emprunté – Quand il a emprunté – Jusqu'à quand il a emprunté.

Ligne 11 : Quel objet sait quels livres sont empruntés (ou réservés) pour la personne ? C'est la personne elle-même.

Donc `Personne` collabore avec `Prêt` et `Réservation`.

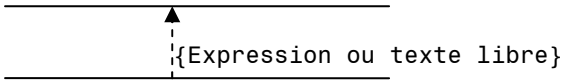
Pour le détail des cartes CRC engendrées par notre analyse, je vous renvoie aux documents de P.Dugerdil.

Exercice
Représenter une entreprise avec n employés dont fait partie le comptable et le directeur (qui lui n'est pas comptable).



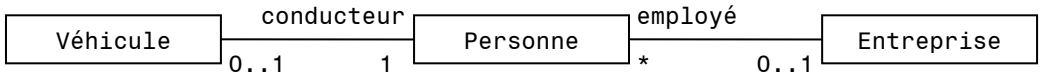
Contrainte entre associations

Notation :



où `expression` en texte libre ou en OCL (Object Constraint Language) ou bien encore une expression booléenne.

Problème
Où placer les propriétés associées aux rôles ?

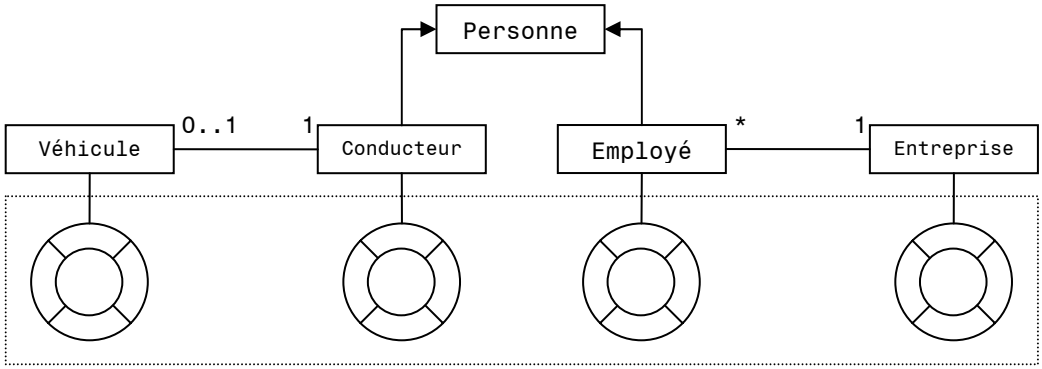


En Java le nom du rôle (ici : conducteur et employé) est utilisé pour nommer les variables qui implantent l'association :

```
class Vehicule {
      private Personne conducteur ;
      …
}
```

On va voir qu'il y a trois solutions à notre problème :

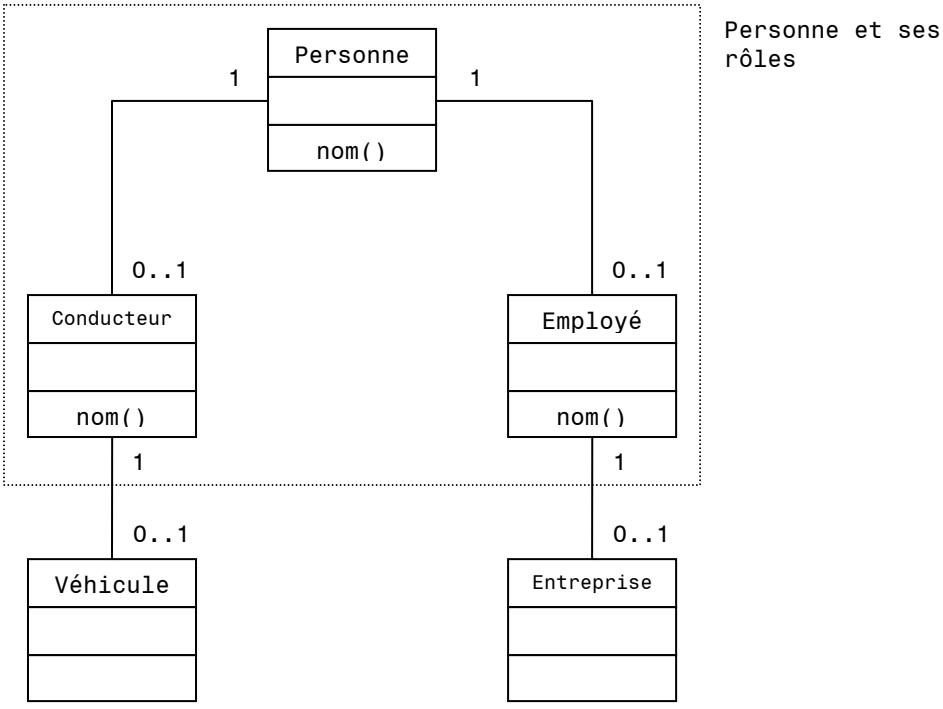
Solution 1



On remarque que si une personne joue deux rôles à la fois, il faudra deux instances. Cette modélisation est correcte si la personne ne joue qu'un seul rôle immuable. Dans le cas contraire il faudra prévoir une copie d'attributs entre instances, ce qui n'est pas une très bonne chose.

Solution 2

Représenter les rôles par des objets associés.



Dans `Conducteur` :

```
nom() {
      return personne.nom() ;
}
```

Dans `Employé` :

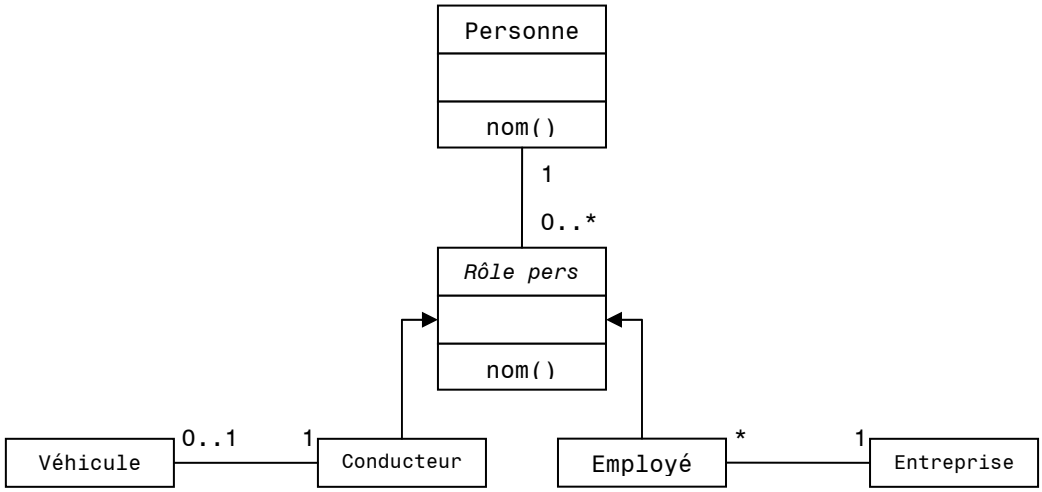
```
nom() {
      return personne.nom() ;
}
```

On a plus d'héritage mais on a des associations. Depuis `Employé` on ne fait qu'appeler `Personne`.

Le principal avantage de cette solution est qu'un objet peut jouer plusieurs rôles à un instant donné. De plus ces rôles peuvent être dynamiques.

Remarque : On n'a pas de duplication des informations comme dans le cas de la solution 1. Les propriétés de `Personne` sont localisées dans cette classe et nulle part ailleurs.

Solution 3
Introduire une super-classe abstraite de rôles qui regroupe toutes les méthodes qui délèguent l'exécution à `Personne`.

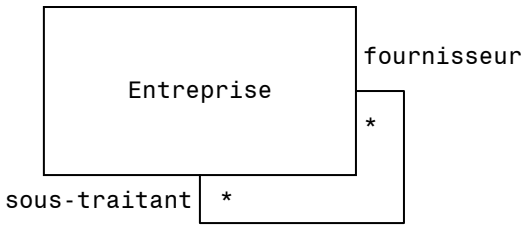


Cette modélisation possède les mêmes qualités que le schéma précédent, mais en plus les méthodes déléguées ne sont codées qu'une seule fois (grâce à l'héritage).

Revenons maintenant au chapitre des associations.

Association récursive

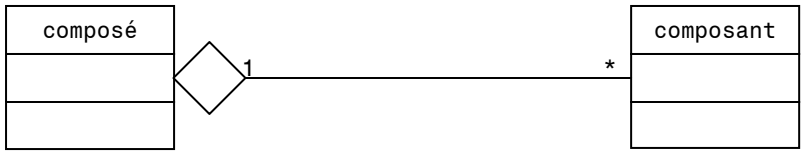
Représenter une entreprise et ses fournisseurs et sous-traitants lesquels sont dans l'entreprise :



C'est grâce à la notion de rôle que l'on peut utiliser les associations récursives.

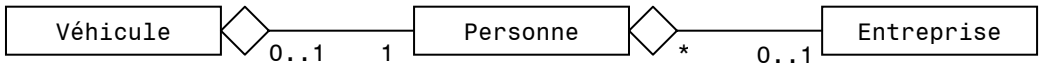
Agrégation

Définition *L'agrégation*
Elle représente l'association entre un objet « composé » et ses « composants ».



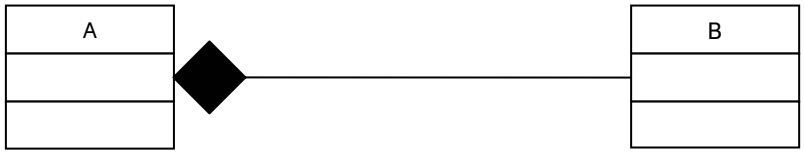
Le lien entre ces deux classes est une association d'agrégation qui est réflexive, anti-symétrique et transitive.

Exemple



Association de composition

Elle est semblable à l'agrégation, sauf que `B` n'existe pas si `A` n'existe pas.



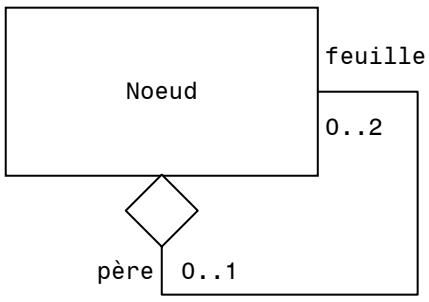
Des exemples : Un bâtiment avec ses chambres.
Un livre avec ses pages, …

N.B. : Quand on a une relation d'agrégation ou de composition, les responsabilités entre composé et composant sont souvent partagées. Comme exemple citons les figures géométriques composées :

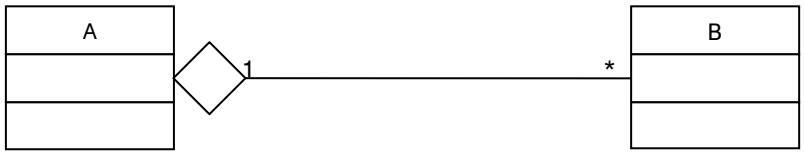
- `draw()`
- `delete()`
- `move()`

Ces trois méthodes sont implantées par délégation aux figures composées.

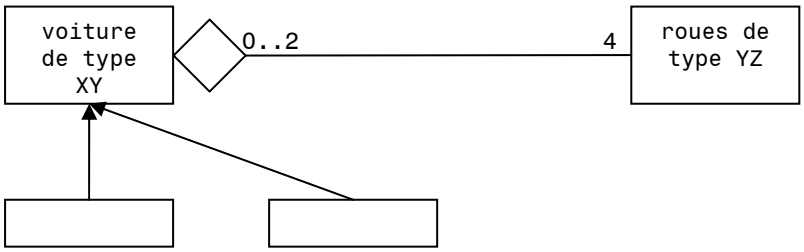
Exercice
Représenter, à l'aide de l'agrégation, la notion d'arbre binaire.



Question
On a souvent la situation suivante :



Ici nous avons une cardinalité de 1 du côté de A. Cela est souvent le cas avec des objets physiques car un composant ne fait partie, en général, que d'un seul composé.
Par contre la cardinalité peut être supérieure si on représente des types d'objets :

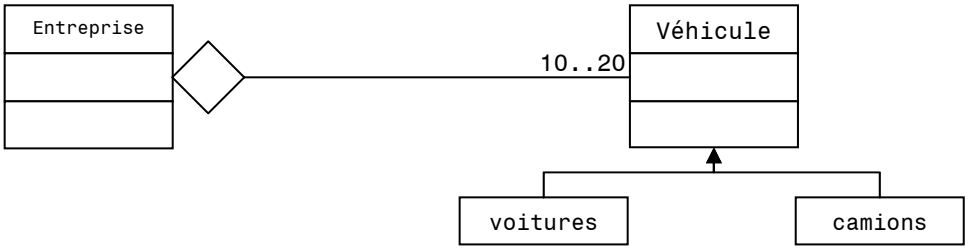


Les roues de type YZ peuvent aller sur deux modèles différents de voiture.
Lorsqu'on modélise, il faut prendre garde au niveau où on se place : soit objets physiques soit types d'objets.

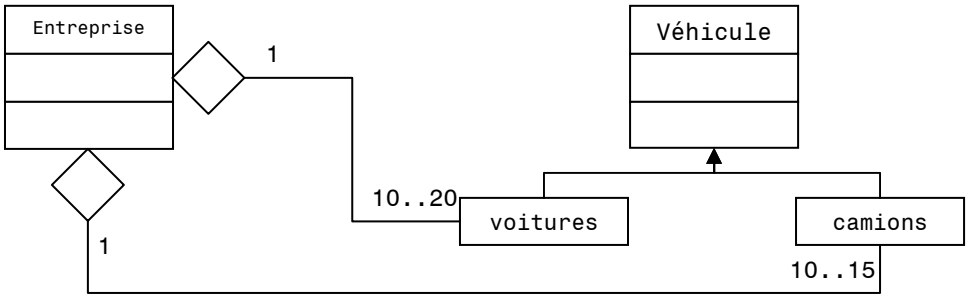
Héritage des associations

Problème
Comment représenter des cardinalités différentes en fonction des sous-classes ?

Exemple
Représenter la flotte de véhicules d'une entreprise constituée de 10 à 20 voitures et de 10 à 15 camions.



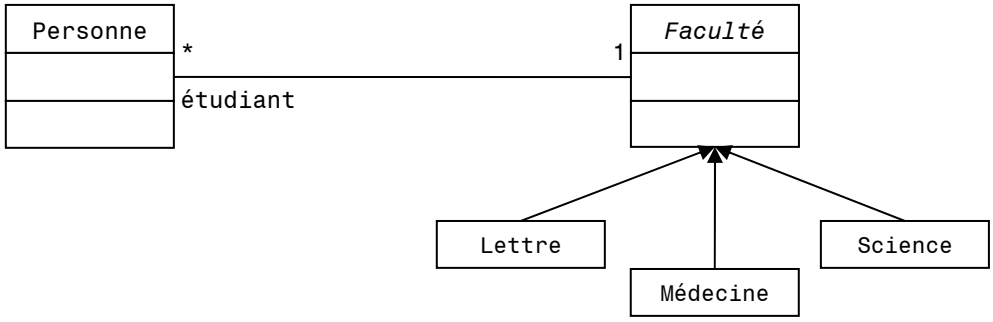
Ici il y a un gros problème : est-ce que l'on a 10 à 20 voitures ou 10 à 20 camions ? Impossible de le savoir.
Il faut trouver une autre solution.



Ainsi le problème est résolu.

Alternative & héritage

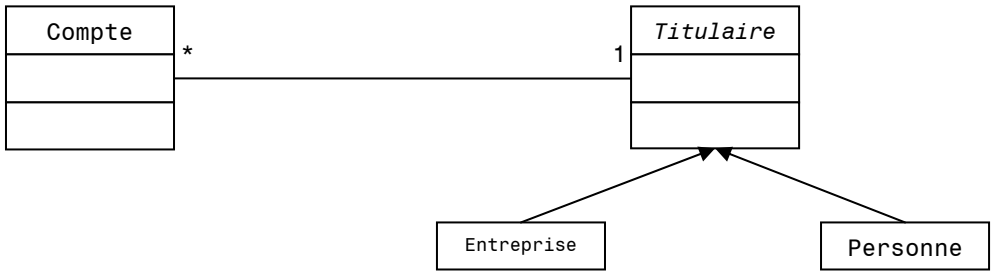
Exemple
Un étudiant est inscrit soit en lettres, soit en médecine, soit en science.



Contrainte d'association

Représenter le fait qu'un titulaire de compte bancaire soit une personne physique ou une entreprise. Un titulaire doit pouvoir fournir un nom et une adresse.

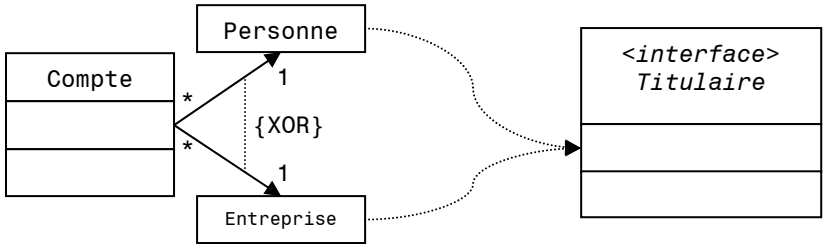
Attention Un titulaire ne peut pas être à la fois entreprise et individu.



Ici le rôle est représenté par la super-classe abstraite.

Cependant ce n'est pas une bonne solution car `Entreprise` et `Personne` ne sont pas des spécifications de titulaire.

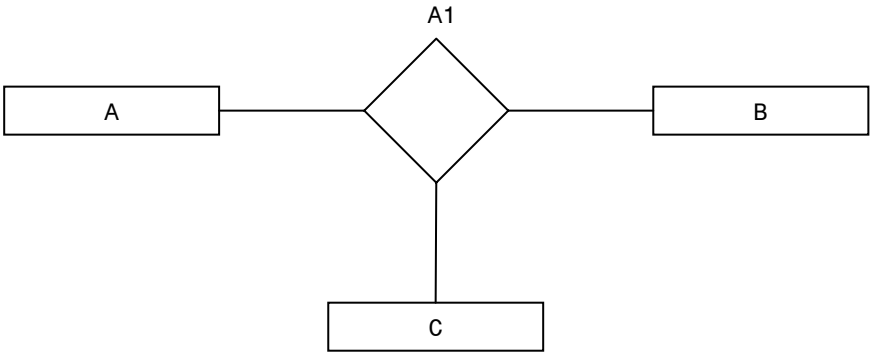
D'où :



Ce qui donne, en Java :

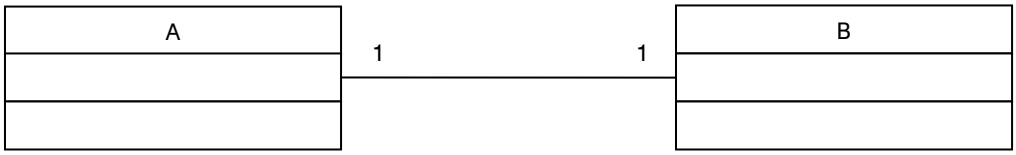
```
class Compte {
      private Titulaire titulaire ;
      …
}
```

Associations ternaires



Ce schéma signifie que les instances de `A`, `B` et `C` sont associées dans l'association `A1`.

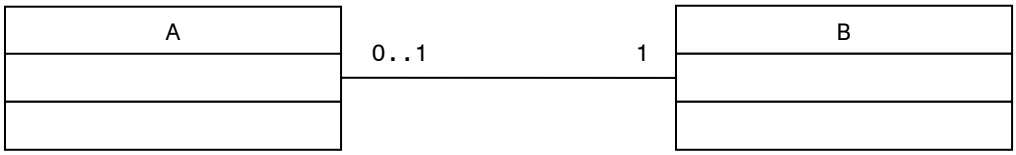
Cardinalité et contraintes d'intégrité



Quelque soit l'instance de `A`, on a exactement une instance de `B` associée et inversement. Cela implique que les instances ne peuvent pas préexister isolement : la création doit être simultanée.

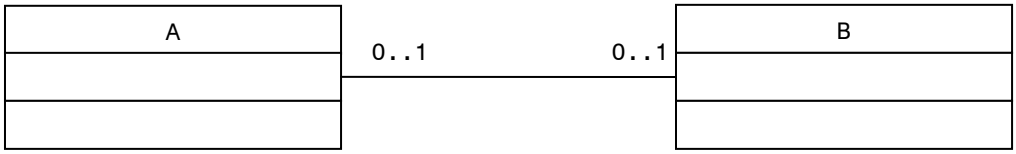
D'un point de vu programmation Java, le constructeur de `A` va provoquer l'instanciation de `B` pour satisfaire la cardinalité et inversement, si l'association est bidirectionnelle.

Il faut faire attention aux boucles récursives !



Des instances de `B` peuvent préexister. Cela implique que quand on instancie `A`, on peut soit forcer la création d'une instance de `B`, soit choisir parmi des instances disponibles.

Un schéma sans contrainte d'intégrité :

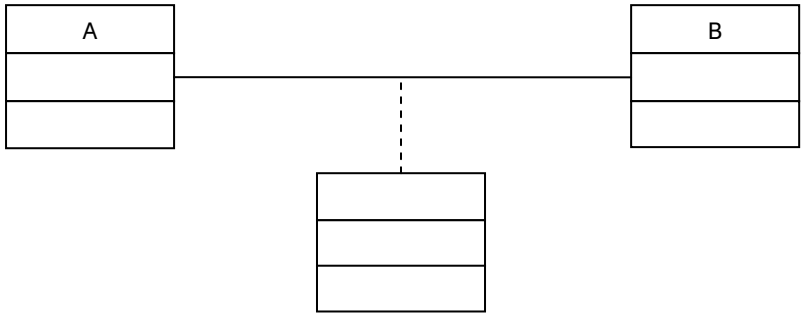


Classe d'association

Définition *Classe d'association*

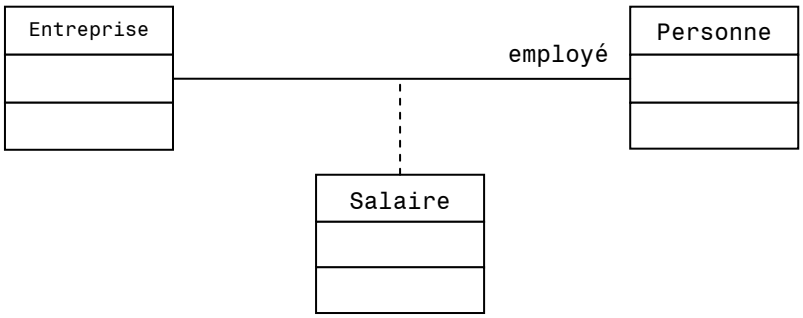
Elle permet de représenter une information qui émerge parce que deux classes sont associées.

On la représente de la manière suivante :



La classe d'association, reliée par des traits tillés, peut ne pas avoir de nom.

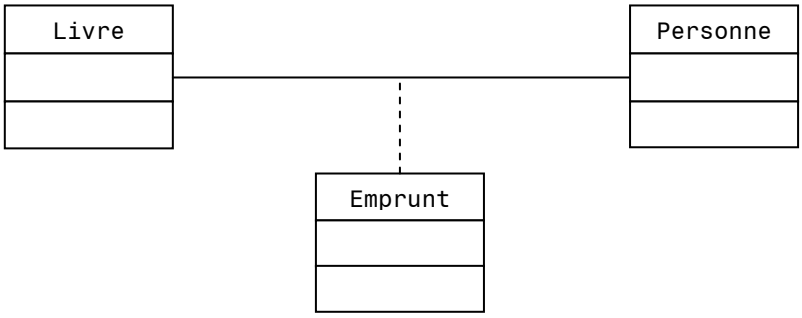
Exemple



Où mettre le salaire ? Il dépend du contexte (dépend de l'entreprise où la personne travaille, si elle travaille à temps partiel, …)

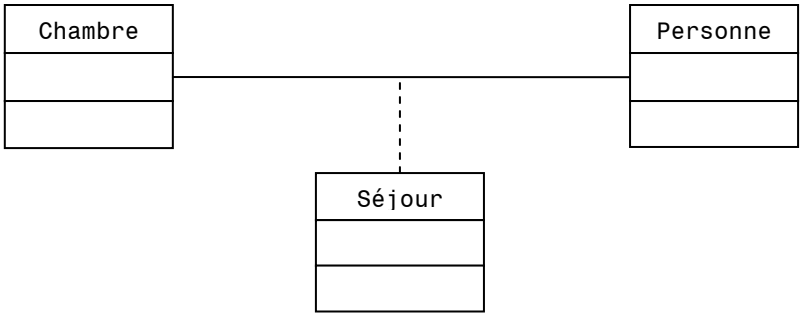
Finalement `salaire` émerge de l'association entre `Personne` et `Entreprise`.

Exemple



Ici, un emprunt n'a aucun sens s'il n'y a pas de lien entre une personne et un livre.

Exemple
Séjour dans une chambre d'hôtel :



Une question que l'on va souvent se poser : « Faut-il modéliser une classe d'association ? ».
Pour y répondre, il faut répondre à ces deux questions :

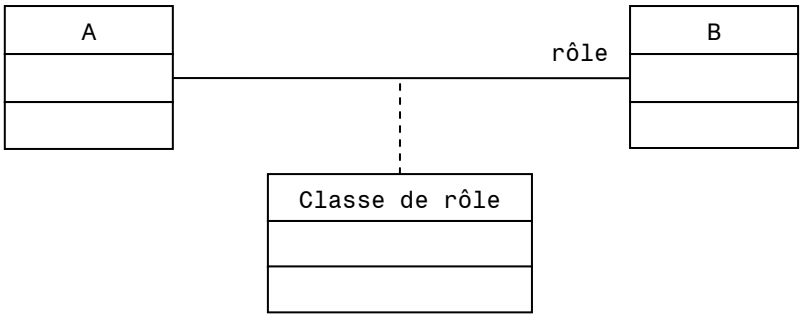
- Est-ce que la responsabilité est valide pour l'objet quelque soit le contexte ?
- La responsabilité est-elle unique quelque soit le contexte (possède d'autre interprétations suivant le contexte) ?

Si on répond *non* à une des deux questions au moins, alors il faut envisager la modélisation d'une classe d'association.

▲ Le contexte est l'ensemble des classes associées.

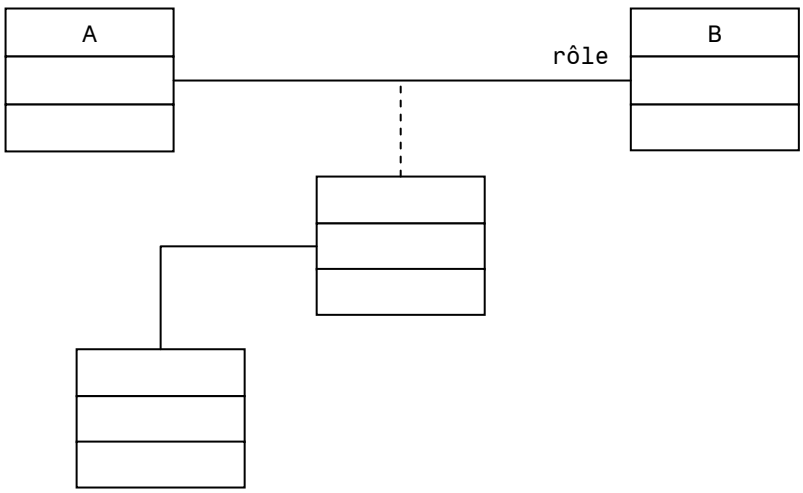
Rôle & classes d'associations

Est-ce que cette représentation est satisfaisante ?



La réponse est non, ce n'est pas une bonne façon. Cependant cette modélisation peut être valide si, dans la classe `Classe de rôle`, il y a les propriétés de gestion du rôle (ex : début de l'association du rôle, …)

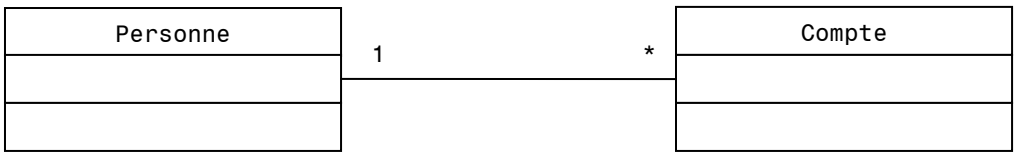
On peut faire des associations sur des associations



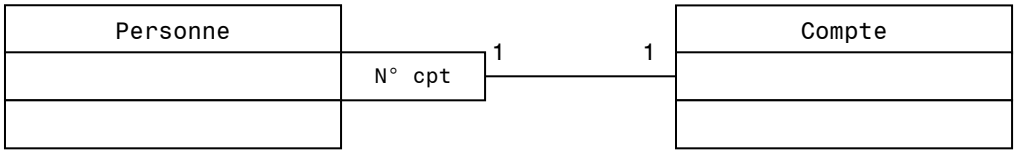
Ainsi la classe qui se trouve tout en bas du schéma n'existe que si l'association entre `A` et `B` est réalisée.
On a donc une contrainte d'intégrité implicite.

Qualificateur

Soit

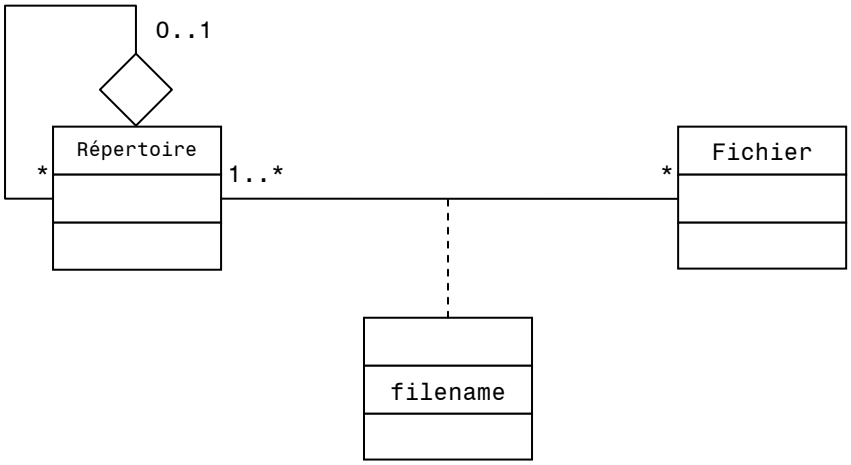


En introduisant un qualificateur, on obtient :

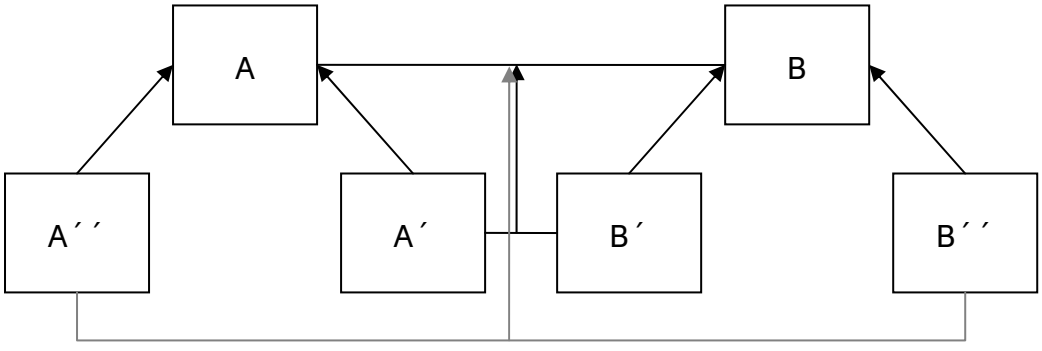


`N° cpt` est un attribut de `Compte` qui permet de discriminer les objets dans l'ensemble associé.
Cette méthode est relativement peu utilisée, sauf dans le cas des modélisations de base de données.

Exemple
Représenter un système de fichiers avec répertoires et sous répertoires. Un fichier possède un nom et des alias dans les divers répertoires. Dans un répertoire, un fichier est identifié par son nom (ou alias).

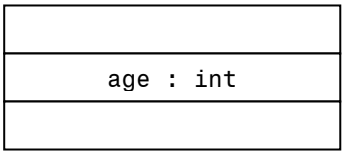


Héritage entre association

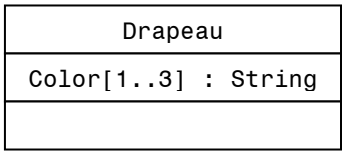


Cette façon de faire est très peu utilisée.

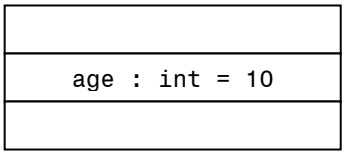
Eléments de syntaxe, suite et fin



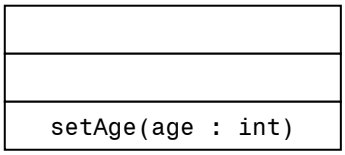
Type d'un attribut



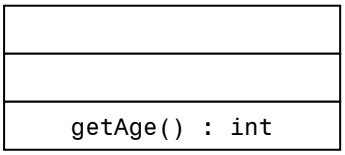
Cardinalité des attributs



Initialisation des attributs



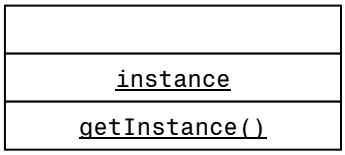
Type des arguments



Type de retour

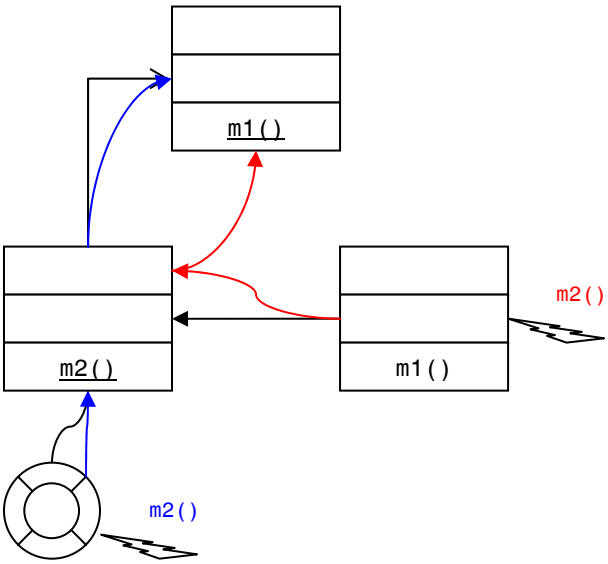
Représentation et attributs des méthodes static

En UML, `static` est symbolisé par un soulignage.



▲ En Java il n'y a pas de dynamic binding avec la méthode statique.

Dans ce schéma, on suppose que `m2()` fasse appel à `m1()` :

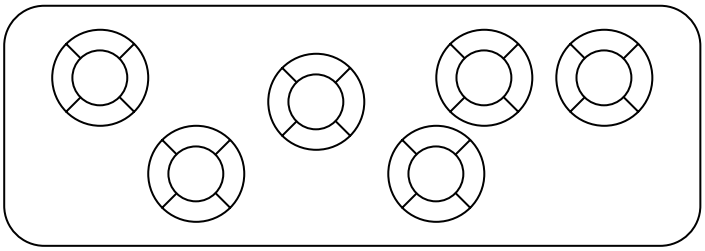


Attributs et associations dérivés

/ = attribut dérivé ou calculable à partir d'un autre. L'âge se calcule à partir de la date de naissance. Ce symbole défini aussi les associations dérivées (que l'on peut déduire à partir des autres associations).

Mémoire objet & Garbage Collector

Les objets créés prennent places dans un espace d'objet (image)



Si on a :

```
myVar = new A() ;
myVar = new A() ;
```

Que devient le premier objet créé par la première ligne de ce code ? Il est alors inatteignable.

Il faut gérer le problème de l'allocation mémoire (en C : `free`, Pascal : `dispose`) pour qu'elle ne sature pas.

Problème

- Trouver les objets vivants.
- Permettre la réallocation de l'espace inutilisé.
- Compactage : tasser la mémoire libre pour permettre l'allocation d'objets de taille quelconque.

Avant :

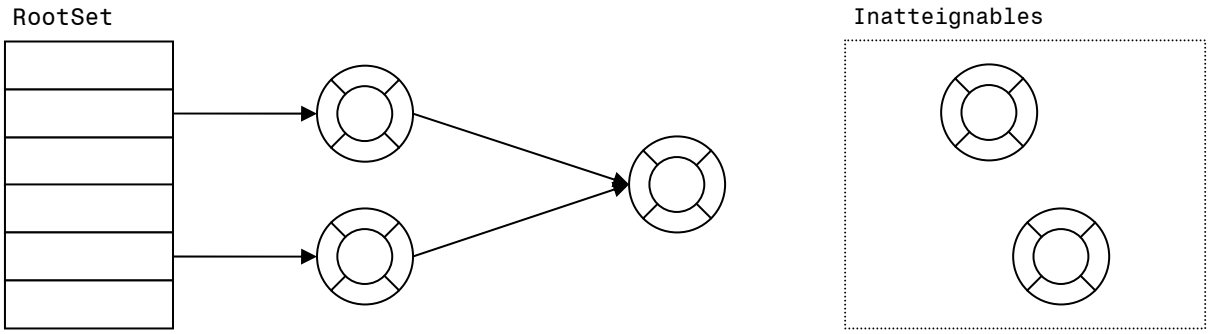
Après :

Définition *RootSet*

Ensemble des variables accessibles aux programmes à un instant donné : variables globales, variables locales des objets qui s'exécutent.

Définition *Objets vivants*

Se sont les objets qui sont atteignables transitivement depuis le RootSet.



Pour résoudre le problème posé, on distingue trois classes d'algorithmes :

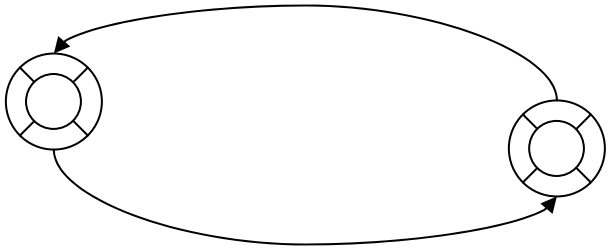
1. Comptage du nombre de références.
2. Marquage.
3. Copie.

Comptage du nombre de références

L'idée est d'incrémenter un compteur de référence quand un objet est référencé. Quand il est déréférencé, on décrémente le compteur. Conceptuellement le compteur fait partie de l'objet.

Quand le compteur est à zéro, l'objet peut être récupéré.

Le désavantage de cet algorithme est que l'on ne puisse pas récupérer des structures cycliques :



En effet dans ce cas, les deux objets ont un compteur qui indique 1, mais ils ne sont pas accessibles.

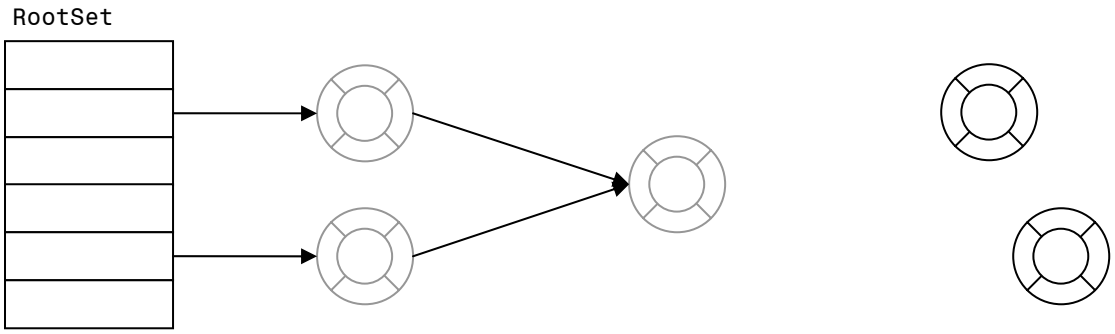
Un autre désavantage important de cette méthode est qu'elle est gourmande car elle travaille en parallèle au programme.
Il n'y a également pas de compactage, il faut le faire en plus.

Marquage

Cela se fait en deux étapes :

1. Parcourir tous les objets atteignables à partir du RootSet et les marquer (en gris sur le schéma).

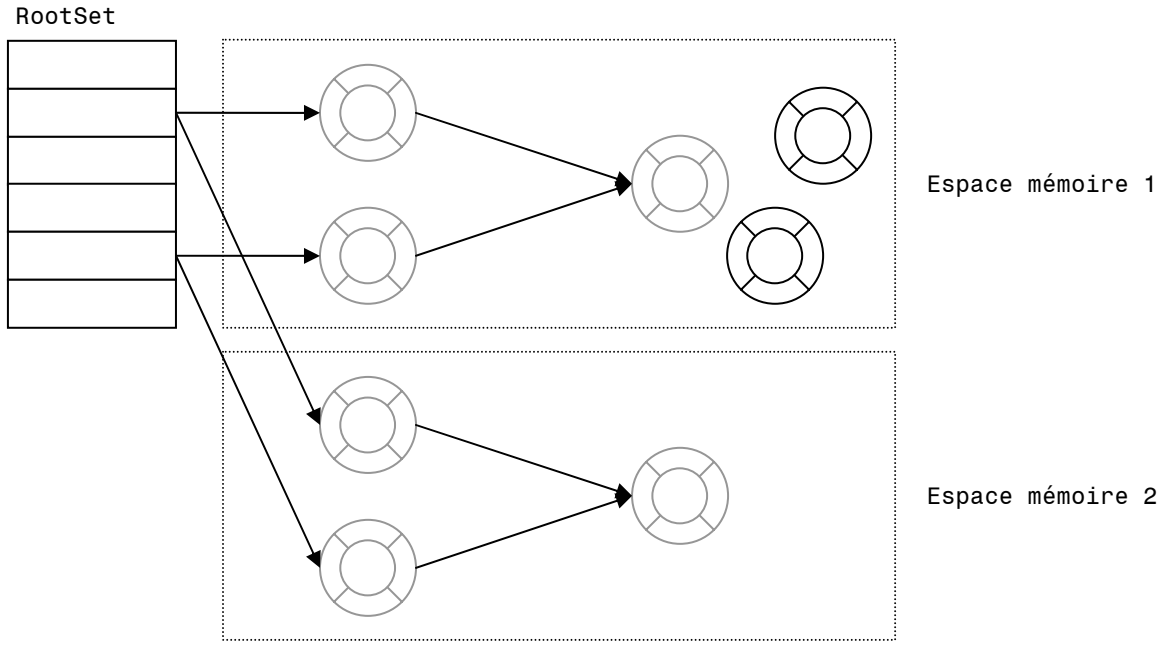
2. On balaye la mémoire et on repère les objets non marqués et on les supprime.



L'avantage est que l'on peut lancer la procédure de marquage quand on le veut.
Cependant on est obligé de parcourir toute la mémoire ce qui est très lourd et le compactage n'est pas effectué.

Algorithme par copie

L'idée est de parcourir les objets atteignables depuis le RootSet et chaque objet atteint est copié dans un autre espace mémoire.



Une fois que la recopie a eu lieu de l'espace mémoire 1 à l'espace mémoire 2 en éliminant les objets inatteignables, on efface le contenu de l'espace mémoire 1 et on recommence l'algorithme mais cette fois de l'espace mémoire 2 à l'espace mémoire 1.

L'avantage de cette méthode est que le compactage est automatique. Cependant il est encore possible de l'améliorer en évitant la recopie de certains objets (ceux qui ont survécu à un certain nombre de « swap ») que l'on place dans une autre zone mémoire. Sur cette nouvelle zone mémoire (que l'on double également), on applique aussi l'algorithme par copie.

Le seul désavantage de cette méthode est qu'il faut prévoir deux fois plus de mémoire.
