CENTRALE MARSEILLE

Modélisation et Conception Objet

Anne Gelly

Nom de l'enseignement : Modélisation et Conception Objet

Code enseignement : MCO-2

1ère année
2014/2015

Notion de classe – Notion d'objet

Une classe regroupe dans une même entité des données (appelées attributs) et des fonctions (appelées méthodes) qui manipulent ces données. Lorsqu'on programme, on ne manipule pas directement les classes, on manipule des objets construits d'après des classes. Une classe est donc un modèle pour fabriquer des objets. Lorsqu'on crée un objet, on dit aussi que l'on instancie une classe. Un objet est par conséquent aussi appelé, une instance de classe.

- Une classe contient à la fois des données (appelées attributs) et des fonctions (appelées méthodes).

attributs: données membres de la classe

méthodes: fonctions membres de la classe

Exemple de classe

```
// fichier Personne.java contenant la classe Personne
public class Personne
 {
 private String Nom;
 private int Age;
 public Personne(String n, int a)
 {
  Nom=n;
  Age=a;
  }
 public void modifNom(String nouvNom)
   {
    Nom=nouvNom;
    }
public String getNom()
 {
 return Nom;
 }
}
```

Constructeur

Dans la classe, il y a une méthode spéciale qui porte le nom de la classe: le constructeur.

Le constructeur n ’est pas obligatoire. L ’appel au constructeur est généré à la création de chaque objet de la classe. Il peut avoir des paramètres. Il n ’a pas de valeur de retour, même pas void.

Utilisation de la classe

Pour créer un objet de la classe, on invoque l'opérateur new et le constructeur, et on fait un passage de paramètres (si le constructeur a des paramètres)

Personne P=new Personne (« Calimero »,10);

P est du type Personne

P est un **objet** de la classe Personne. On dit aussi que P est une **instance** de la classe Personne.

this

Le mot clé ***this*** désigne l'objet courant.

```
// fichier Personne.java contenant la classe Personne
public class Personne
 {
 private String Nom;
 private int Age;
 public Personne(String n, int a)
 {
  this.Nom=n;  // équivalent ici à Nom=n;
  this.Age=a;  // équivalent ici à Age=a;
  }
 public void modifNom(String nouvNom)
   {
    this.Nom=nouvNom;  // équivalent ici à Nom=nouvNom;
    }
public String getNom()
 {
 return this.Nom;  // équivalent ici à return Nom;
 }
}
```

Un peu plus en détail …

Personne p;

Cette instruction déclare une variable p correspondant à un objet de la classe Personne. Mais en fait, aucun objet (aucune instance) de la classe Personne n'est créé. L'espace mémoire nécessaire à stocker une Personne, n'est même pas alloué. La variable p correspond en fait à une référence. En première approximation, une référence correspond à une adresse en mémoire.

Si on ajoute l'instruction:

P=new Personne (« Calimero »,10);

. Un espace mémoire est alloué permettant de stocker un objet de type Personne.

. Le constructeur est appelé pour initialiser l'objet.

3 étapes en une seule instruction

Personne P=new Personne(« Calimero »,10);

1. déclaration d'une référence: p
2. création d'un objet à l'aide de l'opérateur new
3. affectation : la référence (stockée dans l'espace mémoire reservé lors de la déclaration de p) prend la valeur de l'adresse mémoire de l'objet créé avec new. On dit que l'objet créé par new est référencé par la variable p.

Copie de références et copie d'objets

Si l'on écrit:

Personne P1=new Personne(« Calimero »,10);

Personne P2=P1;

Dans ce code, même si 2 variables correspondant à des personnes sont créées, une seule instance de Personne est créée. Une règle générale : une instance ne peut être créée que par new. Il n'y qu'un seul new appelé lors de l'exécution de ce code, donc un seul objet créé.

Suivons l'exécution pas à pas

1. *Personne p1 = new Personne(« Calimero »,10);*

Création d'une instance de Personne, référencée par la variable p1. Initialisation des attributs de l'objet (instance), grâce à l'appel au constructeur.

2. *Personne p2=p1;*

Déclaration d'une variable p2 : aucune Personne n'est créée, on réserve juste un espace mémoire pour une référence, et on stocke à l'intérieur la valeur de la référence p1.

p2 référence désormais aussi l'objet qui était référencé par p1.

A la suite de l'exécution de ces 2 lignes de code, un seul objet a été créé, et cet objet est référencé par les 2 variables p1 et p2. p2=p1 effectue une copie de références, et non pas une copie d'objets.

On continue notre exemple

Personne p1 = new Personne(« Calimero »,10);

Personne p2=p1;

p2.modifNom(« Puccino »);

System.out.println(p2.getNom()); *// cette ligne affichera « Puccino »*

System.out.println(p1.getNom()); *// cette ligne affichera aussi « Puccino »*

P1 et p2 référencent le même objet

Copie des objets (copie profonde)
Constructeur par copie

```
public class Personne {

private String Nom;
  private int Age;
  public Personne(String n, int a)
  {
   Nom=new String(n);  // utilisation du constructeur par copie de la classe String
   Age=a;
   }

  public Personne(Personne P)     // constructeur par copie de la classe Personne
  {
Nom=new String(P.Nom);
Age=P.Age;
  }
  public void modifNom(String nouvNom)
    {
     Nom=nouvNom;
     }
 public String getNom()
  {
  return Nom;
  }



public static void main(String[] args)
 {
Personne p1 = new Personne("Calimero",10);
Personne p2=new Personne(p1);  // appel du constructeur par copie de la classe Personne
p2.modifNom("Puccino");
System.out.println(p2.getNom());  // cette ligne affiche « Puccino »
System.out.println(p1.getNom());  // cette ligne affiche « Calimero »

}

}
```

Surcharge

Profitons-en pour dire un mot sur la surcharge des méthodes.

Dans l'exemple précédent, on a finalement deux versions du constructeur dans la même classe. Mais, on voit aussi que les signatures de ces deux constructeurs sont différentes.

public Personne(String n, int a)

public Personne(Personne P)

Les méthodes portent le même nom mais leurs signatures (entête) sont différentes -> il s'agit d'une surcharge

Nous verrons plus loin la notion de redéfinition, qui est différente de celle de surcharge.

Utilisation de la classe

On invoque les méthodes de la classe de la manière suivante:

P.modifNom(« Charles »);

String myName=P.getNom();

Accessibilité

Les attributs étant privés (en général), on est obligé d'utiliser des méthodes pour les manipuler.

```
// fichier Principal.java contenant le Main
public class Principal
  {
public static void main(String[] args)
  {
  Personne P1= new Personne("Toto",40);
  //P1.Nom="Charles"; la méthode qui ne fonctionne pas
  P1.modifNom("Jules"); // la méthode qui fonctionne
}
```

P1 est un objet (ou une instance) de la classe Personne.

Avantages de la programmation orientée objet

1. Dans une même entité sont réunies les données et les méthodes qui manipulent ces données.
2. Protection des données.

Les données étant privées (fortement conseillé!), seules les méthodes de la classe peuvent y accéder.
Rq:Les méthodes sont généralement publiques, elles peuvent par conséquent être appelées partout, y compris dans le Main. On peut également trouver des méthodes privées au sein des classes, elle ne peuvent être appelées que par des méthodes de la classe.

Avantages de la programmation orientée objet

3. Initialisation des données garanties. C'est le rôle du constructeur. A chaque création d'objet, il y a un appel au constructeur avec passage de paramètres. Le rôle principal du constructeur est d'initialiser les attributs de l'objet, soit grâce aux paramètres effectifs du constructeur, c'est-à-dire aux paramètres lors de l'appel au constructeur, soit grâce à des valeurs choisies ou récupérées ailleurs.
Rq: Le constructeur peut avoir d'autres rôles, par exemple celui de réserver de la place mémoire.

Exemple de constructeur

```
Personne(String n)
 {
  Nom=n; // ou Nom=new String(n);  -> initialisation grâce au paramètre
   Age=12; -> initialisation grâce à une valeur
 }
```

Maintenant que nous avons à peu près compris les grands concepts de la programmation orientée objet, et avant d'aller plus loin dans ces concepts, nous allons nous pencher sur tout ce qui est issu du langage historique, le langage C, à savoir: les types de données élémentaires, les structures de contrôle etc…

Les types de base

- **boolean** : un booléen ne pourra prendre que les valeurs **true** ou **false**.
- **byte** : un entier relatif très court (entre -128 et 127)
- **short** : un entier relatif court (entre -32 768 et 32 767)
- **int** : un entier relatif entre -2 147 483 648 et 2 147 483 647)
- **long** : un entier relatif long (entre -9 223 372 036 854 775 808 et 9 223 372 036 854 775 807)
- **float** : un nombre décimal
- **double** : un nombre décimal à double précision.
- **char** : un caractère

Déclaration et initialisation de variables de type simple

```
int monEntier=0;          // ou bien
char monChar='A';
double monDouble=12.0;
boolean monBooleen=true;
```

```
int monEntier;
monEntier=0;
```

Les tableaux

Un tableau est un ensemble indexé de données d'un même type, qu'il s'agisse d'un type simple ou d'un type classe.

Un tableau se déclare et s'instancie comme une classe.

int *monTableau**[ ]*** = ***new int[****10****];***

Maintenant, on peut remplir le tableau.

*monTableau**[5]*** *= 23;*

L'indexation démarre à partir de 0, ce qui veut dire que, pour un tableau de N éléments, la numérotation va de 0 à N-1.

Dans l'exemple ci-dessus, la 6ème case contient donc la valeur 23.

Nous pouvons également créer un tableau en énumérant son contenu :

int [] *monTableau =* ***{****5,8,6,0,7****};*** *// Ce tableau contient 5 éléments*

Lorsque la variable est déjà déclarée, nous pouvons lui assigner d'autres valeurs à la place, en utilisant l'opérateur new :

monTableau = ***new int[]{****11,13,17,19,23,29****};***

Le type String

Ce qui était un tableau de caractères avec des propriétés particulières dans le langage historique, est une classe dans le langage Java: la classe String.

La classe *String* est une classe spéciale :

- les chaînes de caractères peuvent se concaténer à l'aide de l'opérateur +.
- les instances peuvent ne pas être créées explicitement:

String s = "abc" ; au lieu de *String s = **new** String("abc") ;*

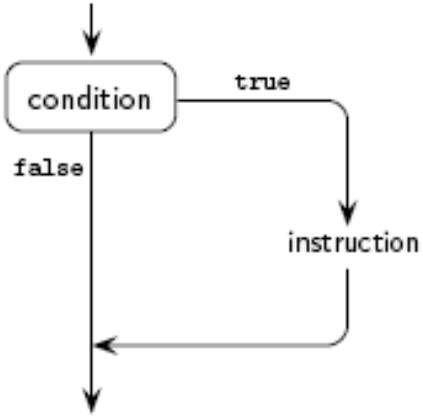
Les instructions conditionnelles

Les instructions conditionnelles, ou instructions de test, permettent de faire des choix dans un programme. Elles permettent d'altérer le déroulement du programme en fonction de conditions. Il y a trois instructions conditionnelles différentes en Java : l'instruction if, l'instruction if-else et l'instruction switch.

L’instruction if

L'*instruction if* permet d'exécuter une instruction si une *condition* est vérifiée. La condition est une expression booléenne et elle est dite vérifiée si sa valeur est true.

if (valeur<0) System.out.println(« Négatif »);

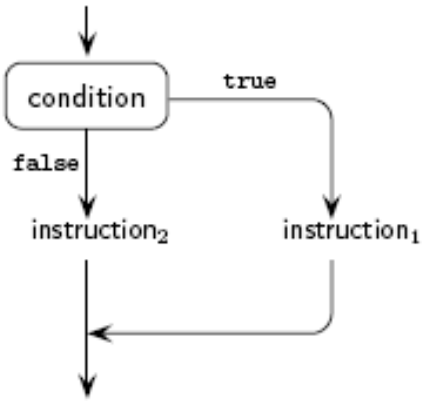


L’instruction if … else

Lorsqu’on veut exécuter autre chose si la condition est fausse:

if (valeur<0) System.out.println(« Négatif »);

else System.out.println(« Positif ou nul »);

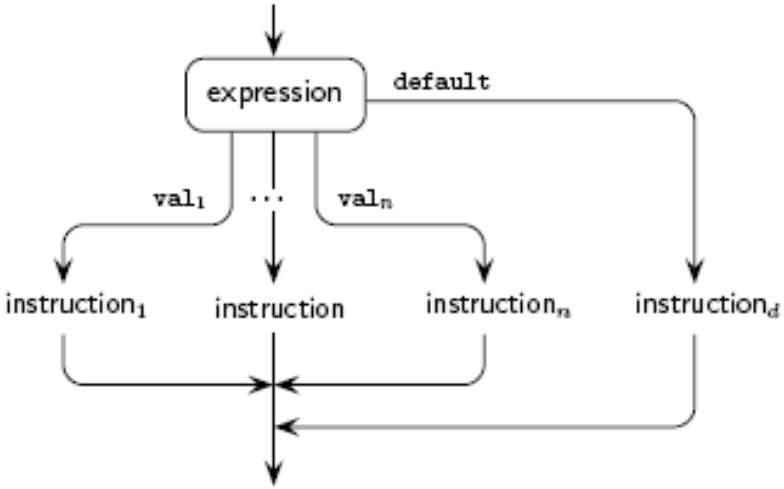


Le switch

. L'expression du switch doit être de type (ou doit pouvoir être converti en) int ou char. De plus, les valeurs utilisées dans les case doivent être des expressions constantes : c'est-à-dire soit des littéraux, soit des constantes (définies avec final), soit des expressions dont tous les opérandes sont des constantes.

. Le *break* est obligatoire, sinon la suite s'exécute.

. *Default* est facultatif.



Exemple avec monChoix de type entier (int)

```
switch(monChoix)
 {
 case 1: System.out.println(« Vous avez choisi 1 »);
          break;
 case 2: System.out.println(« Vous avez choisi 2 »);
         break;
  default: System.out.println(« Vous n'avez choisi ni 1 ni 2 »);
}
```

Exemple avec monChoix de type caractère (char)

```
switch(monChoix)
 {
 case '1': System.out.println(« Vous avez choisi 1 »);
          break;
 case '2': System.out.println(« Vous avez choisi 2 »);
         break;
  default: System.out.println(« Vous n'avez choisi ni 1 ni 2 »);
}
```

Les instructions répétitives

Il y a quatre types de boucle en Java :

while

do … while

for

for each

L’instruction while

```
int i=0;
while(i<10)
  {
  …
  System.out.println(i);
  }
```

Tant que l’expression (toujours entre parenthèses) est vraie, les instructions dans les accolades sont exécutées.

1. L’expression est évaluée. Si elle est fausse, aller en 4 sinon aller en2.
2. Les instructions qui constituent le corps de la boucle sont exécutées.
3. Aller en 1.

4 Suite du programme.

Le test s’effectue en début de boucle. Si la condition est fausse dès le départ, les instructions de la boucle ne seront jamais exécutées. Dans notre exemple, si l’on avait initialisé i à10 par exemple, le programme ne serait jamais entré dans le corps de la boucle.

Les variables servant dans le test doivent être initialisées.

L’instruction do … while

```
int i=0;
do
{
  …
  System.out.println(i);
  } while (i<10)
```

Le corps de la boucle est exécuté tant que l’expression est vraie.

1. Les instructions constituant le corps de la boucle sont exécutées.
2. L’expression est évaluée. Si elle est vraie, aller en 1.
3. Suite du programme.

Les instructions constituant le corps de la boucle sont exécutées au moins une fois, même si la condition est fausse au départ, car elle est testée à la fin de la boucle. Donc, même si i est initialisée à une valeur supérieure ou égale à 10, le corps de la boucle sera quand même exécuté une fois.

L’instruction for

for (<expression1>**;** <expression2>**;** <expression3>**)**
    <instruction>

Cette condition est équivalente à
<expression1>;
while (<expression2>)
  {
   <instruction>;
   <expression3>;
  }
Exemple:
for *(int i=0; i<10; i++)*
  System.out.println(i);

L’instruction for

Attention aux erreurs d’inattention. Elles n’engendrent pas forcément d’erreur à la compilation.

```
for (int i=0; i<10; i++);
  System.out.println(i);
```

Le programmeur s’attend à ce que son code affiche les 10 premiers entiers. En fait, il n’y a que le chiffre 10 qui est affiché, en raison du point virgule après le **for**. En effet, ce morceau de programme est équivalent à:

```
int i=0;
while (i<10)
  {
    ;
    i++;
  }
System.out.println(i);
```

La boucle for each

La boucle *for each* n'est pas une boucle comparable aux trois précédentes, car elle s'utilise dans un cadre très particulier, lorsqu'il faut itérer les différents éléments d'un ensemble (un tableau, une liste etc...).

La syntaxe est la suivante :

```
for (type variable : ensemble)
{
  instructions
}
```

Par exemple pour afficher tous les éléments d'un tableau d'entiers, ces deux codes sont équivalents :

```
int n=tableau.length;
for (int i=0; i<n; i++)
System.out.println(tableau[i]);
```

```
for (int e : tableau)
System.out.println(e);
```

Dans le premier code, on itérait sur les indices du tableau, alors que dans le deuxième, on itère directement sur les éléments du tableau.

Remarques :Dans la majorité des cas où elle peut s'utiliser, la boucle for each est plus rapide qu'une boucle for.

On a ici un contrôle moindre sur l'ordre des éléments, mais Java assure que tous les éléments seront utilisés (on n’a pas à compter soi-même le nombre d'éléments) et qu'ils seront pris "dans l'ordre" (lorsque cet ordre existe).

Le &&

Il s'agit du ET entre expressions

if (x>3 && x<5) …

Ce qui signifie: si x est supérieur à 3 et que x est inférieur à 5

Ce qui est évalué: si (x>3) est vraie et que (x<5) est vraie …

Le ||

Il s’agit du OU entre expressions

if (x<0 || x>BORNEMAX) …

Ce qui signifie: si x est inférieur à 0 ou que x est supérieur à BORNEMAX

Ce qui est évalué: si (x<0) est vraie et que (x>BORNEMAX) est vraie …

Exemple d'utilisation

```
int i=0;
boolan trouve=false;
while( trouve==false &&  i<tableau.length)
  {
   if (tableau[i]==valeur)
          trouve=true;
   else i++;
}
```

Les paramètres des méthodes

```
class Animal
  {
  private int nbrePattes;
 …
public void modifNbrePattes(int nbre) { nbrePattes=nbre;}
public static void main(String[] args)
  {
  Animal Bill=new Animal(4);
  Bill.modifNbrePattes(6);
  }
}
```

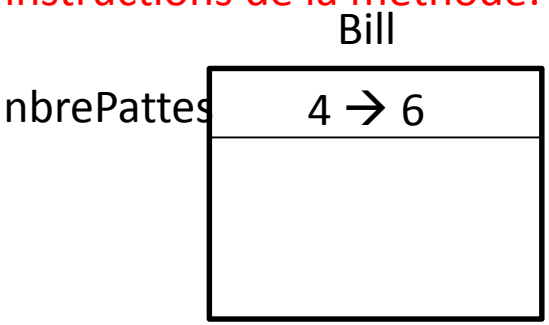
Ici, le paramètre est d'un type **primitif**. Il est donc passé par **valeur.**

Mécanisme d'appel

Lors de l'appel d'une méthode, il y a création des variables locales et des paramètres formels de la méthode, et initialisation des paramètres formels à la valeur des paramètres effectifs.

Etape 1: nbre s'initialise à 6

Etape 2 Exécution des instructions de la méthode: *this.nbrePattes=nbre;*



Passage d'un paramètre de type objet

```
// un exemple stupide juste pour expliquer le passage par référence des objets

class Tartampion
{
….
public void modifAge(Personne P)
  {
  P.age+=10;
  }
public static void main(String[] args)
  {
Personne P1=new personne(….);
Tartampion T1= new Tartampion( …);
T1.modifAge(P1);
   }
}
```

On recommence: Lors de l'appel d'une méthode, il y a création des variables locales et des paramètres formels de la méthode, et initialisation des paramètres formels à la valeur des paramètres effectifs. Donc Personne p=p1 (on fait une copie des références (cf page 10)).

Pendant toute la durée de l'exécution de la méthode modifAge, P et P1 référencent le même objet. Donc, modifier l'objet référencé par P revient à modifier l'objet référencé par P1, puisqu'il s'agit d'un seul et même objet.

Après l'instruction T1.modifAge(P1), l'âge de l'objet référencé par P1 aura été incrémenté de 10. La modification de l'objet référencé par P a donc entraîné la modification de l'objet référencé par P1, puisque P et P1 référencent le même objet durant tout le temps de l'exécution de la méthode modifAge.

Afficher des objets

Nous avons vu que pour afficher à l'écran, il fallait utiliser la méthode System.out.println.

Ce que nous n'avons pas précisé, c'est que l'argument de cette méthode doit être de type String. Lorsqu'on affiche des variables de type simple, pas de problème elles sont traduites en String;

```
int i=10;
System.out.println(i); //  affiche bien 10
```

Afficher des objets

Reprenons la classe Personne

Personne P1=new Personne(« Toto »,15);

System.out.println(P1); //Erreur

Le compilateur détecte une erreur car il ne sait pas traduire un objet Personne en objet String.

Il est temps de présenter la méthode ***toString***.

La méthode toString définie dans la classe mère de toutes les classes (la classe Object) retourne une représentation d'un Objet sous forme de String.

=> Lorsque vous voulez utiliser System.out.println pour afficher un objet de votre classe, vous devez tout d'abord surcharger la méthode toString de votre classe.

Surcharge de toString (ou plutôt redéfinition)

```
public class Personne
  {
  …
  public String toString() // retourne une représentation sous forme de String de l'objet courant
    {
    return  ''son age: '' + Age + ''son nom: ''  + Nom;
    // ou bien return new String( ''son age: '' + Age + ''son nom: ''  + Nom);
    }
}

// fichier Principal.java contenant le Main
public class Principal
  {
public static void main(String[] args)
  {
  Personne P1= new Personne("Toto",40);
System.out.println(P1);                    // l'appel à la méthode to String() est implicite
System.out.println(P1.toString()) ;  // c'est également juste, mais inutile
}
```

Attributs de classe

Dans la définition de la classe, on peut déclarer *static* un attribut. On dit alors que c'est un attribut de classe.

--> cet attribut est alors commun à toutes les instances de la classe. Il n 'y en a donc qu 'une copie pour toutes les instances.

Ex: une classe Personne, avec le nombre de Personnes en donnée membre statique.

```
class Personne
 {
  public Personne(String nom, int age)
  {
  nomIndividu = nom;  // ou copie profonde nomIndividu=new String(nom);
 ageIndividu=age;
nombreIndividus++;  //-> chaque fois qu'une personne est créée, donc à chaque appel au constructeur,
                          //le nombre d'individus est incrémenté.
 }
private String nomIndividu;
private int ageIndividu;
private static int nombreIndividus=0;
}                                                                                    }
```

Méthodes de classe

Dans une classe, on peut déclarer static une méthode qui ne s ’applique pas à un objet particulier. On dit alors que c’est une méthode de classe.

Les méthodes de classe sont souvent associées à la manipulation d’attributs de classe (mais pas nécessairement).

```
class Personne
  {

    …
    public static int getNombreIndividus()
  {
  return nombreIndividus;
  }
}
```

Utilisation

public static void main(String[] args)

{

Personne P1=new Personne(« Titi »,12);

Personne P2=new Personne(« Tutu »,15);

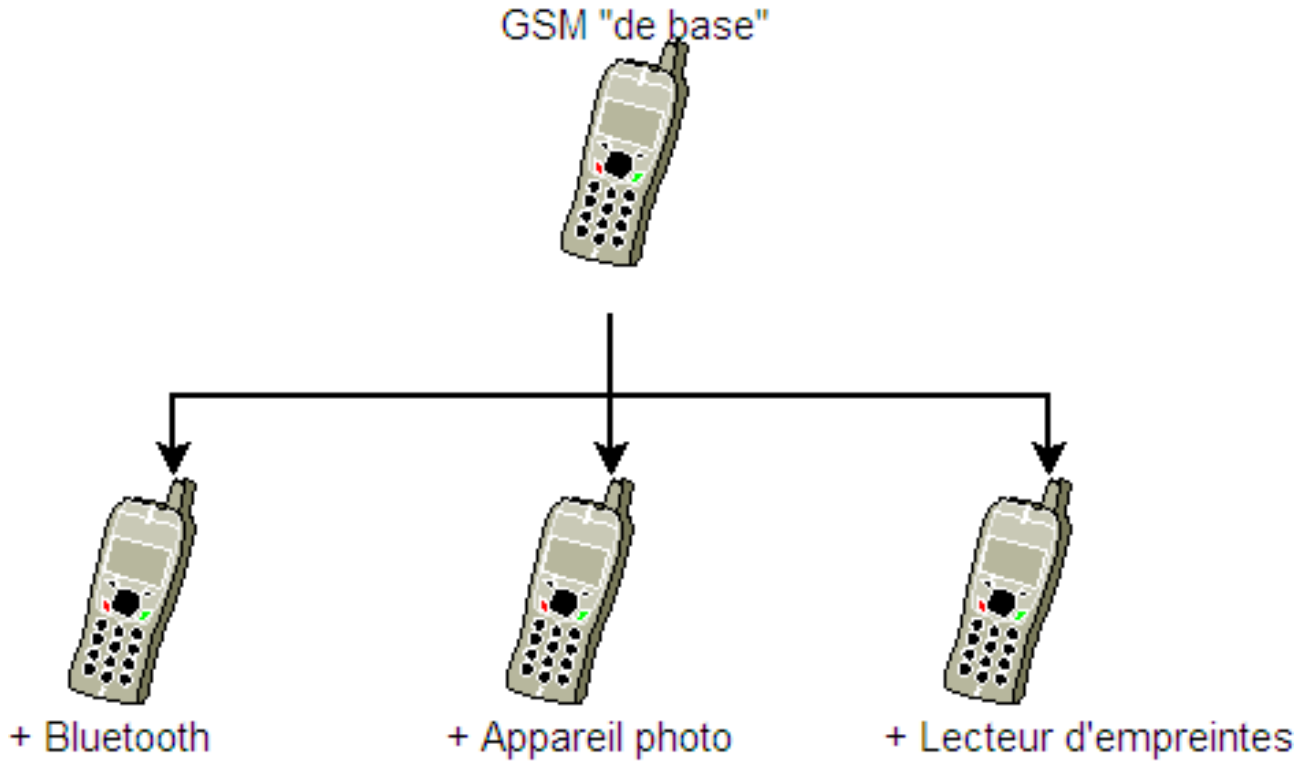
System.out.println(Personne. *getNombreIndividus()); // affiche 2*

}

Comme la méthode ne porte pas sur un objet en particulier, l'appel est précédé du nom de la classe.

L’Héritage

Il arrive assez souvent, que l'on désire écrire une nouvelle classe alors que certains attributs et méthodes sont déjà définis dans une autre classe. Heureusement, il ne faudra pas tout réécrire, il existe un mécanisme très utilisé en programmation orientée objet qui permet de récupérer le travail déjà fourni : *l'héritage* .



Source UKO

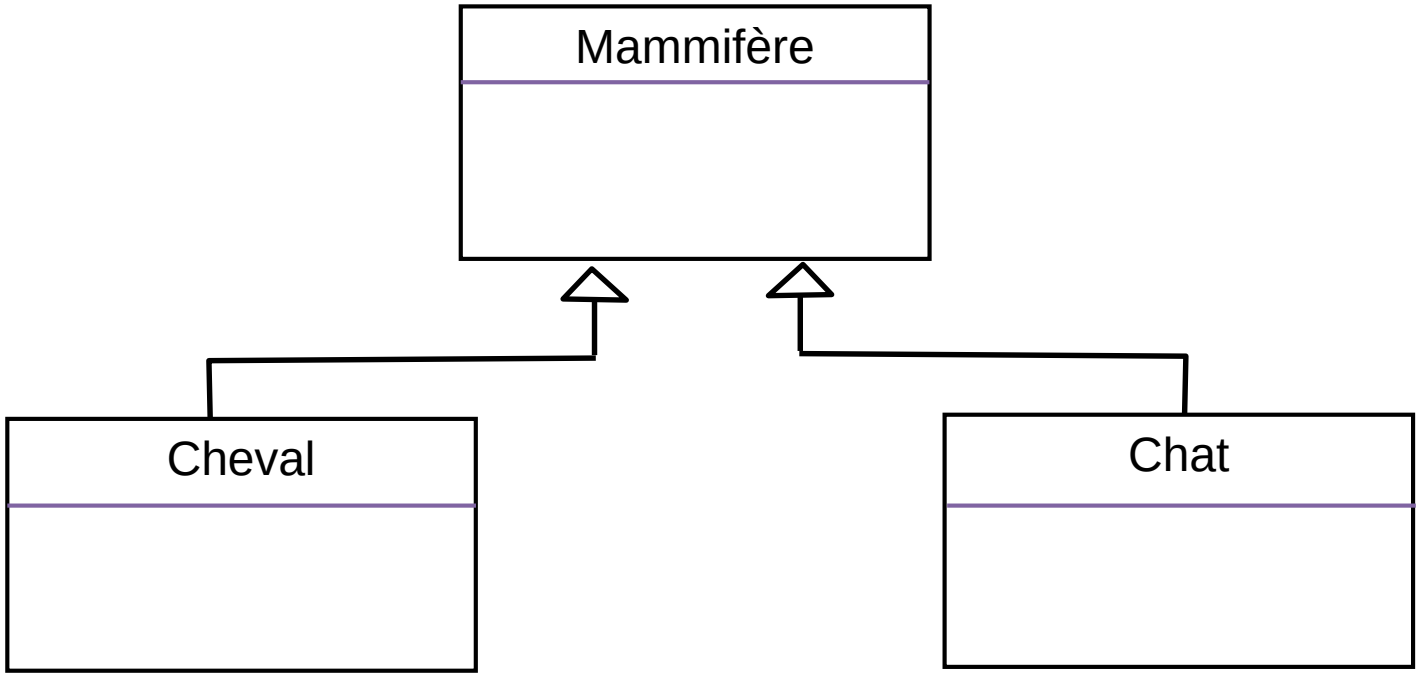
On possède donc une classe, et à partir de celle-ci, on va *dériver* une nouvelle classe (ou plusieurs). Cette nouvelle classe est appelée *classe dérivée, sous-classe* ou *classe fille* tandis que l'autre classe est appelée *classe mère ou superclasse.*

Cette manière d'écrire des classes est plus économique, en effet, les sous-classes vont hériter des attributs et méthodes définis dans la classe mère.

Exemple

La sous-classe est toujours plus spécifique que la classe mère.

Prenons par exemple une classe qui représente les mammifères, ce sera la classe mère. Un mammifère possède toute une série de caractéristiques. On peut ensuite créer deux classes filles qui représentent les chevaux et les chats. Ces classes auront des caractéristiques supplémentaires spécifiques aux chevaux pour la première, et aux chats pour la seconde.


Mammifère
Cheval
Chat

En Java, on utilise le mot réservé extends pour signaler qu'une classe hérite d'une autre.

```
public class Employe
 {
...
  private float salaire;
   public Employe(float sal){salaire=sal;}
  public float calculSalaire {return salaire;}
  };

class Cadre extends Employe
{
private float prime;
...
}
```

Héritage: constructeur

- Si le programmeur n'a rien prévu, c'est le constructeur sans arguments de la super-classe qui est appelé, à condition qu'il existe.
- Sinon, tout constructeur de la sous-classe commence nécessairement par l'appel d'un constructeur de la super-classe avec la syntaxe:

super (*arguments*);

Héritage: constructeur

```
public class Cadre extends Employe
{
private float prime;
public Cadre(int sal, int pri)
  {
  super(sal);
  prime=pri;
 }
...
}
```

Héritage:invocation d'une méthode de la super-classe

```
public class Cadre extends Employe
{
private float prime;
public Cadre(int sal, int pri)
  {
  super(sal);
  prime=pri;
 }
public float calculSalaire()
  {
  return (super.calculSalaire() + prime);
  }
…
}
```

Redéfinition de méthodes

Profitons-en pour dire un mot sur la redéfinition. Dans l'exemple précédent, on a redéfini dans la classe Cadre, la méthode calculSalaire que l'on avait définie dans la classe mère Employe. En effet, les deux méthodes ont bien la même signature. Cependant leur corps est différent. Il s'agit donc bien d'une redéfinition et non d'une surcharge.

Les interfaces

Interfaces

La notion d'interface est liée à la notion de services. Une interface regroupe des « services », c'est-à-dire généralement un certain nombre de méthodes (dont le corps ne sera pas défini dans l'interface). Imaginons par exemple une interface ***télécommande de télévision***. Cette interface définit diverses méthodes publiques qui sont par exemple *augmenter ou diminuer le son, monter ou descendre de chaine.*

On dit qu'une classe *implémente* une interface lorsqu'elle offre toutes les méthodes publiques définies dans l'interface, c'est-à-dire lorsqu'elle implémente toutes les méthodes de l'interface. Lorsqu'une classe implémente une interface, elle est alors obligée de définir le corps de chacune des méthodes dont l'entête figure dans l'interface.

On déclare une interface avec le mot réservé ***interface***.

Une interface contient les signatures des méthodes et éventuellement des déclarations de constantes . La signature d'une méthode est une méthode sans corps (sans implémentation), il y a donc juste l'entête de la méthode se terminant par un point-virgule.

Voici l’exemple d'une interface représentant un animal et offrant deux méthodes publiques : la première permet d'obtenir le cri de l'animal et la seconde la famille.

```
public interface AnimalInterface
{
    public void getNoise();
    public void getFamily();
}
```

On dit qu’une classe *implémente* une interface lorsque la classe fournit une implémentation pour toutes les méthodes de l'interface. Pour signaler qu'une classe implémente une certaine interface, on utilise le mot réservé *implements*.

Exemple

```
public class Dog implements AnimalInterface
{
   private String race;

   public Dog (String rac)
   {
      this.race = rac;
   }

   public void getNoise()
   {
      System.out.println ("Wouf");
   }

   public void getFamily()
   {
      System.out.println (« Je suis un mammifère");
   }

   public void getBreed()
   {
      System.out.println (« Je suis un" + race);
   }
}
```

La classe Dog implémente donc bien l'interface AnimalInterface étant donné qu'on y retrouve les méthodes getNoise et getFamily déclarées dans l'interface AnimalInterface. La classe Dog contient une méthode supplémentaire getBreed pour obtenir la race du chien, et ce n'est pas du tout interdit. Tant qu'elle contient toutes les méthodes déclarées dans l'interface, c'est bon.

La classe suivante implémente toutes les méthodes de l'interface, et rien d'autre.

```
public class Fly implements AnimalInterface
{
    public void getNoise()
    {
        System.out.println ("Bzzz");
    }
    public void getFamily()
    {
        System.out.println (« Je suis un insecte");
    }
}
```

Implémentation multiple

Une classe peut implémenter plusieurs interfaces. Dans ce cas, on les énumère toutes, après le mot réservé ***implements***, séparées par des virgules. La classe devra donc implémenter toutes les méthodes définies dans toutes les interfaces qu'elle implémente.

Dans un tel cas, il faut faire attention. Si les interfaces ont des méthodes en commun, c'est-à-dire avec la même signature (entête), mais avec une sémantique différente, la classe qui implémente toutes les interfaces ne définira qu'une seule fois la méthode. Si ces méthodes ont une sémantique différente, une des deux sera masquée par l'autre, il faut donc être vigilant.

Objet de type Interface

On peut se servir d'une interface comme type pour une variable.

On restreint ainsi l'accès aux méthodes de l'objet.

AnimalInterface AI=new Dog(« Caniche »);

Seules les méthodes de AI déclarées dans AnimalInterface sont accessibles.

Donc, les instructions *AI.getNoise();* et *AI.getFamily();* sont correctes.

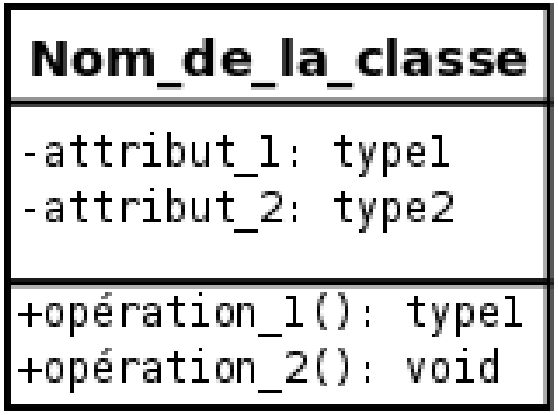
Par contre, l'instruction *AI. getBreed();* est incorrecte.

Notions d'UML

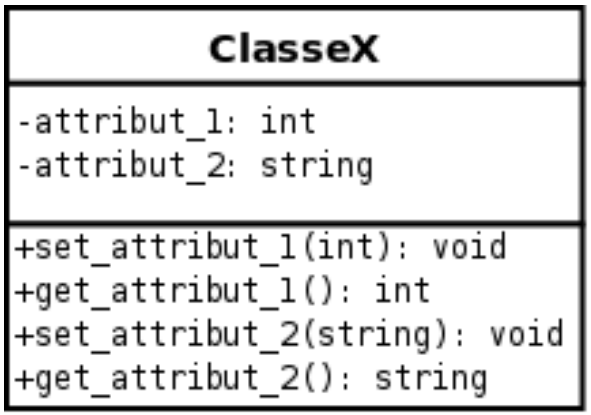
(d'après le cours de Laurent Audibert sur developpez.com

http://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-classes#L3-2-1)

Représentation d'une classe en UML

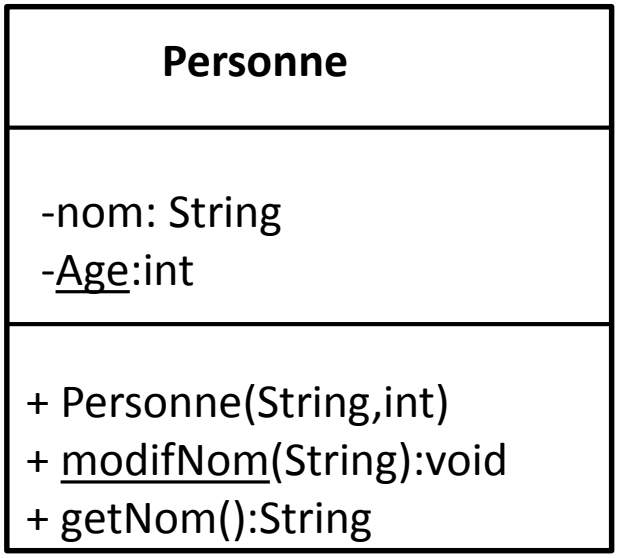


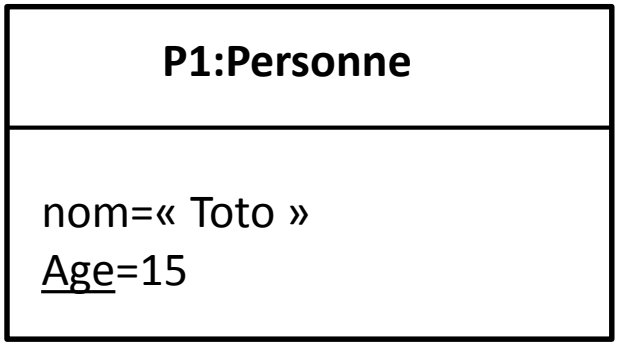
Exemple de classe



public ou +
protected ou #
private ou -

Représentation d'une classe et d'une instance de cette classe





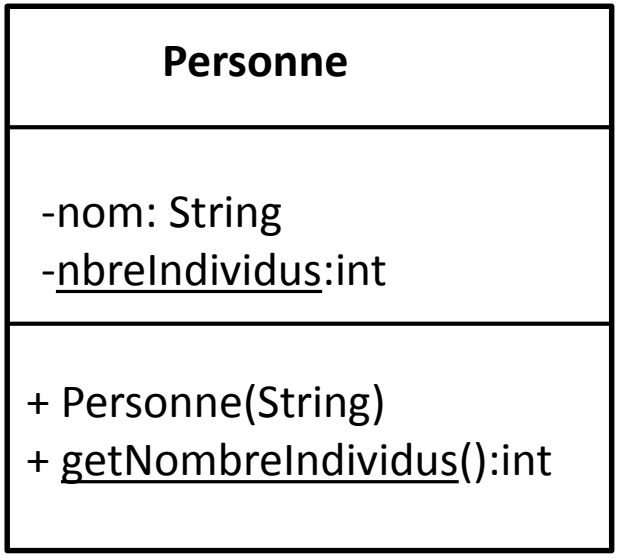
Attribut de classe

Un attribut commun à la classe et non spécifique de chaque instance de classe, est un attribut de classe. En java ou en C++, c'est un attribut déclaré static. En UML, il est souligné pour mettre en évidence sa particularité.

Méthode de classe

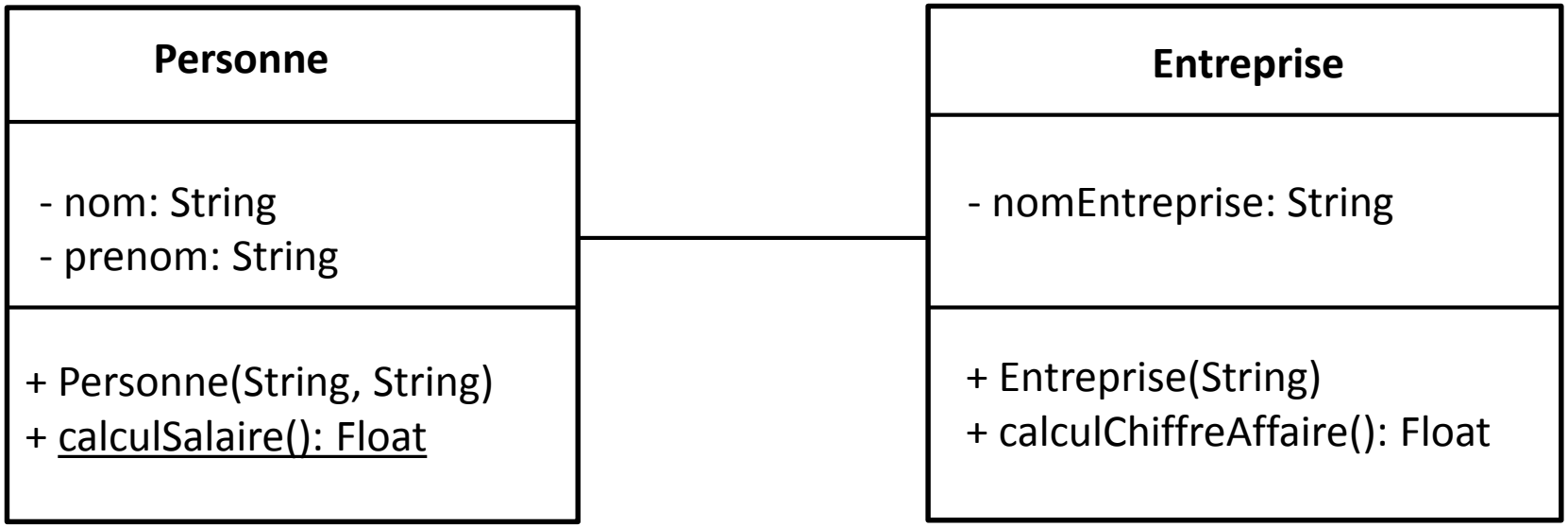
Comme pour les attributs de classe, il est possible de déclarer des méthodes de classe. Une méthode de classe ne peut manipuler que des attributs de classe et ses propres paramètres. Cette méthode n'a pas accès aux attributs de la classe (i.e. des instances de la classe). L'accès à une méthode de classe ne nécessite pas l'existence d'une instance de cette classe.

Graphiquement, une méthode de classe est soulignée.


Personne
-nom: String
-nbreIndividus:int
+ Personne(String)
+ getNombreIndividus():int

Notion d’association

Une association est une relation entre deux classes (association binaire) ou plus (association n-aire), qui décrit les connexions structurelles entre leurs instances (objets). Une association indique donc qu'il peut y avoir des liens entre des instances des classes associées.



Par exemple, *Toto* travaille pour *EDF.*
Il y a donc un lien entre Toto (instance de la classe Personne)
et EDF (instance de la classe Entreprise).

Terminaison d’association

Une terminaison d'associations est une extrémité de l'association. Une association binaire en possède deux, une association n-aire en possède n.

Agrégation

Une association simple entre deux classes représente une relation structurelle entre pairs, c'est-à-dire entre deux classes de même niveau conceptuel : aucune des deux n'est plus importante que l'autre.

Lorsque l'on souhaite modéliser une relation tout/partie où une classe constitue un élément plus grand (tout) composé d'éléments plus petits (partie), il faut utiliser une agrégation.

Agrégation

Une agrégation est une association qui représente une relation d'inclusion structurelle ou comportementale d'un élément dans un ensemble. Graphiquement, on ajoute un losange vide (◊) du côté de l'agrégat. Une agrégation n'entraîne pas de contrainte sur la durée de vie des parties par rapport au tout.

Composition

La composition, également appelée agrégation composite, décrit une contenance structurelle entre instances. (ex: Un camion a un moteur) Ainsi, la destruction de l'objet composite implique la destruction de ses composants. Une instance de la partie appartient toujours à au plus une instance de l'élément composite : la multiplicité du côté composite ne doit pas être supérieure à 1 (i.e. 1 ou 0..1). Graphiquement, on ajoute un losange plein (♦) du côté de l'agrégat

Exemple

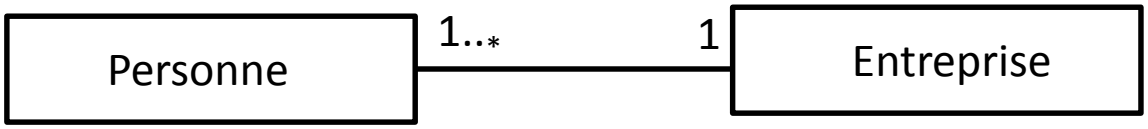
. Ici, on comprend bien qu'un objet camion et son moteur vivent et meurent en même temps. Si un objet camion meurt, son moteur aussi. De plus, un objet moteur n'appartient qu'à un objet camion. Il s'agit donc bien d'une **composition**.

. Une entreprise a des camions. Elle peut « contenir » des camions ou bien être liée à eux par une relation (L'entreprise A « possède » 3 camions C1, C2, C3). Les durées de vie de l'entreprise et de ses camions ne sont pas liées. Il s'agit donc d'une **agrégation**.

Multiplicité ou cardinalité

La multiplicité associée à une terminaison d'association, d'agrégation ou de composition déclare le nombre d'objets susceptibles d'occuper la position définie par la terminaison d'association. Voici quelques exemples de multiplicité :

- exactement un : 1 ou 1..1
- plusieurs : * ou 0..*
- au moins un : 1..*
- de un à six : 1..6



Le schéma ci-dessus traduit le fait
qu'une Entreprise emploie au moins une personne
et qu'une personne n'est employée que par une entreprise.

Le schéma ci-dessus traduit le fait
qu'une Entreprise a plusieurs camions (elle peut en avoir 0)
et qu'un camion n'a qu'un seul moteur.

Navigabilité

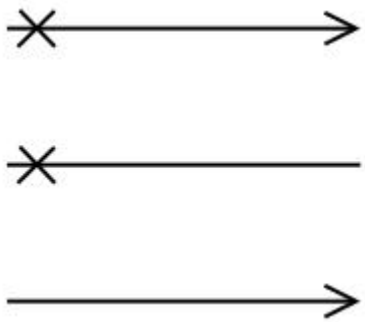
On représente graphiquement la navigabilité par une flèche du côté de la terminaison navigable et on empêche la navigabilité par une croix du côté de la terminaison non navigable.

Le schéma illustre le fait que:

Un objet **Commande** contient une liste de **Produit**s.

Un objet **Produit** ne contient pas une liste de **Commande**s.

Navigabilité

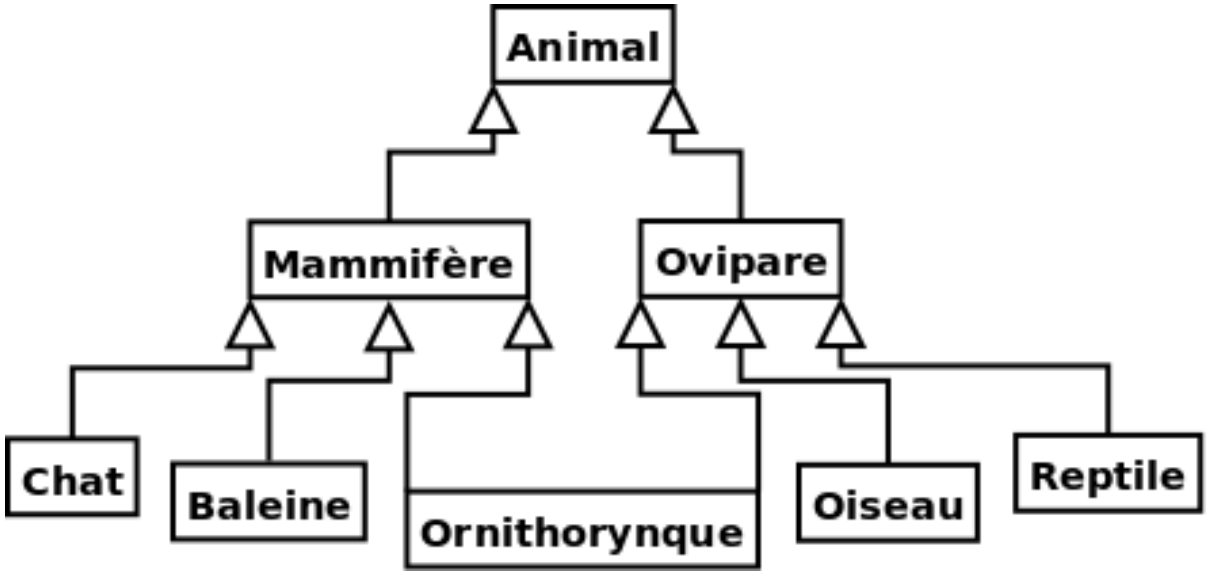


Ces 3 notations ont finalement le même sens.

Héritage

- La généralisation décrit une relation entre une classe générale (classe de base ou classe parent) et une classe spécialisée (sous-classe).
- Un objet de la classe spécialisée peut être utilisé partout où un objet de la classe de base est autorisé.
- Dans le langage UML, ainsi que dans la plupart des langages objet, cette relation de généralisation se traduit par le concept d'**héritage**. On parle également de relation d'héritage.
- Le symbole utilisé pour la relation d'héritage ou de généralisation est une flèche avec un trait plein dont la pointe est un triangle fermé désignant le cas le plus général

Héritage

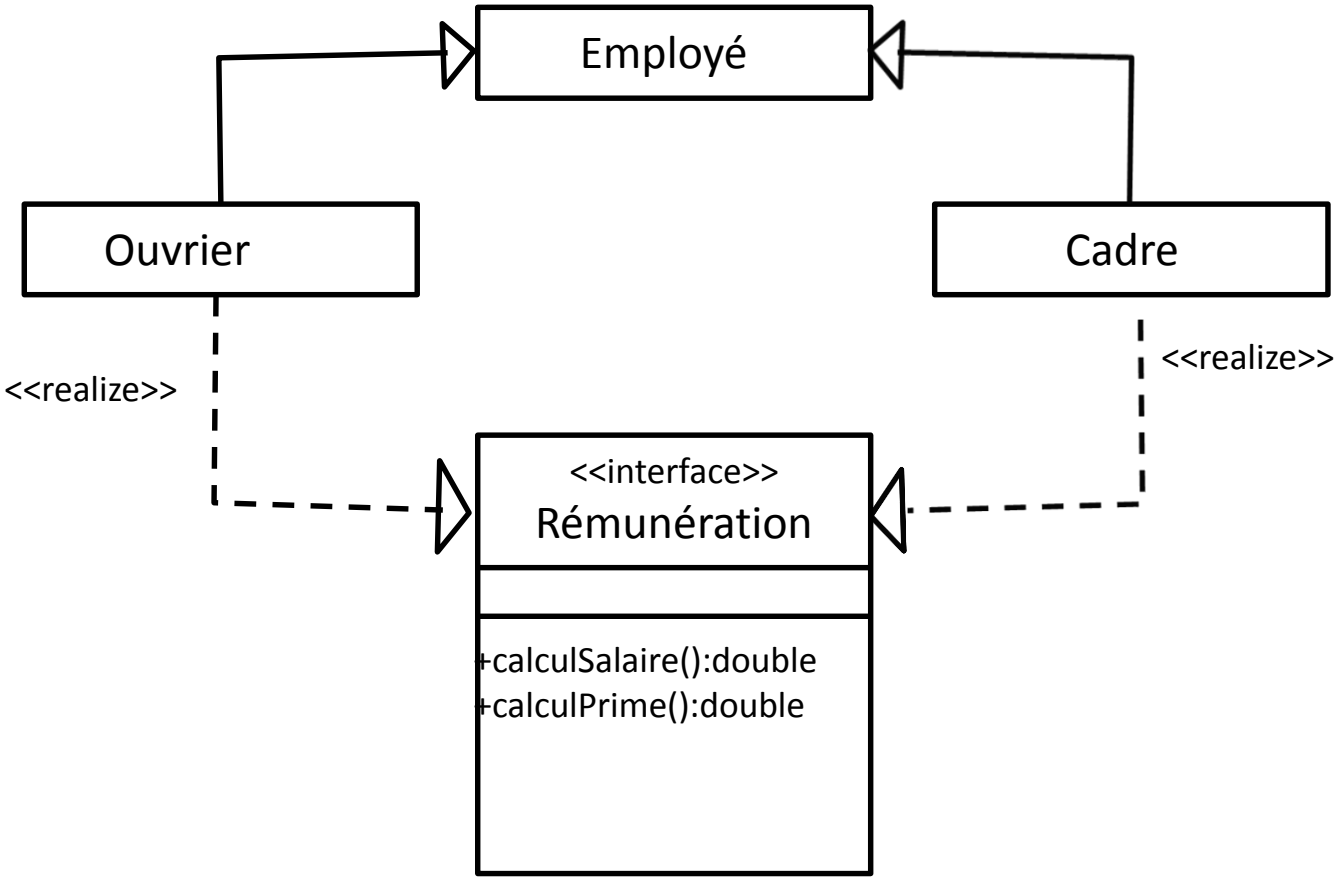


L'héritage multiple(La classe Ornithorynque hérite à la fois de la classe Mammifère et de la classe Ovipare) n'est pas implémenté en Java.

Interface

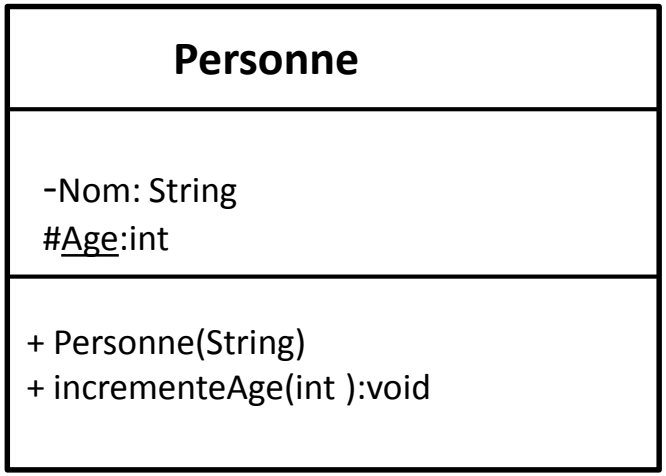
- Une interface est représentée comme une classe avec l'ajout du stéréotype << interface >> .
- Une interface doit être réalisée par au moins une classe et peut l'être par plusieurs.
  Graphiquement, cela est représenté par un trait discontinu terminé par une flèche triangulaire et le stéréotype « realize ». Une classe peut très bien réaliser plusieurs interfaces.

Exemple


Employé
Ouvrier
Cadre
<<realize>>
<<realize>>
<<interface>>
Rémunération
+calculSalaire():double
+calculPrime():double

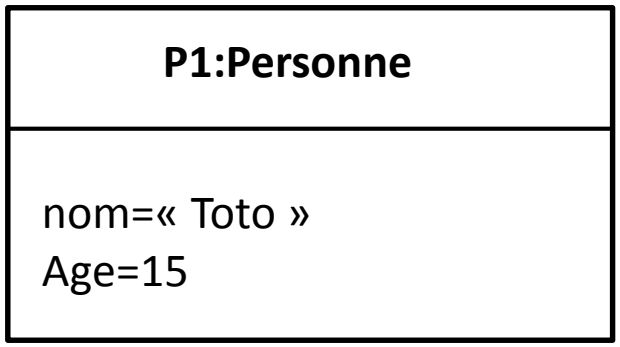
Mise en œuvre en java

Classe




```
public class Personne
  {
private String Nom;
protected int Age;
public Personne(String p)
  {
…
  }
public void incrementeAge()
  {
…
  }
}
```

Instance de classe

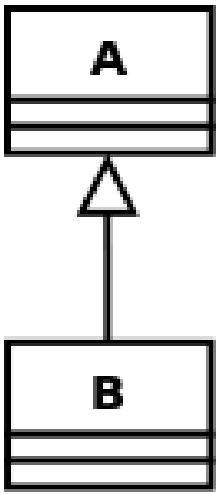


Personne P1=new Personne(« Toto »,15);

Interface


```
public interface A
 {
 …
 }
```

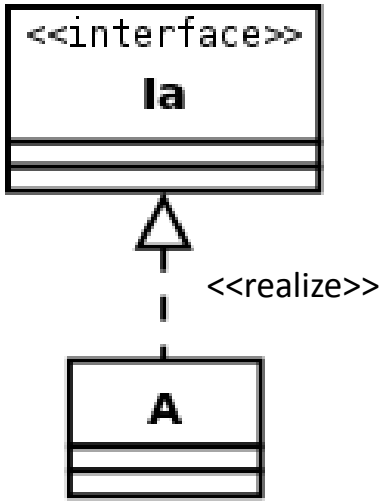
Héritage simple




```
public class A
 {
 …
 }
public class B extends A
 {
 …
 }
```

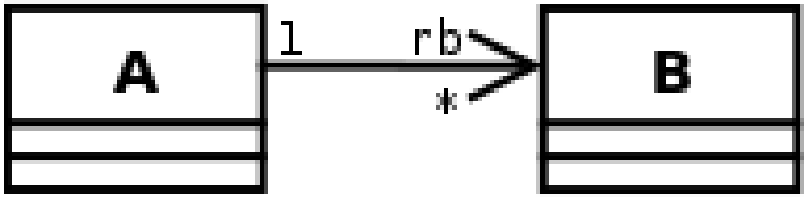
Réalisation d'une interface par une classe

```
public interface Ia
 {
 …
 }

public class A implements Ia
 {
 …
 }
```



Association unidirectionnelle 1 vers plusieurs

```
public class A
 {
 private ArrayList <B> rb;
 public A()
  {
  rb = new ArrayList<B>();
  }
 public void addB(B b)
   {
   if( !rb.contains( b ) )
       rb.add(b);
   }
 }

public class B
 {
  ...
  // B ne connaît pas l'existence de A
 }
```



Agrégation et Composition

. Les agrégations s'implémentent comme les associations.

. Une composition peut s'implémenter comme une association unidirectionnelle.