Bases de la conception orientée objet
Diagrammes de classe
ACDA 2.1

Luc Dartois

luc.dartois@u-pec.fr

Objectifs

Objectifs du cours d'aujourd'hui

- Comprendre les notions complémentaires de diagramme de classe (visibilité, attributs de classe, relation d'agrégation)
- Comprendre la notion de généralisation et sa relation avec l'héritage
- Maîtriser la notion de classe abstraite et interface.

La visibilité

La visibilité d'un attribut (ou une opération) désigne son accessibilité par les autre objets. Principalement, on a :

+	Public	:	accessible par tous
#	Protégé	:	accessibles aux sous-classes seulement
−	Privé	:	restreint à la classe seule

La visibilité

La visibilité d'un attribut (ou une opération) désigne son accessibilité par les autre objets. Principalement, on a :

+	Public	:	accessible par tous
#	Protégé	:	accessibles aux sous-classes seulement
−	Privé	:	restreint à la classe seule

Public

Pour : accès *simple* pour tous
Contre : Modifiable par tous. Aucune sécurité

La visibilité

La visibilité d'un attribut (ou une opération) désigne son accessibilité par les autre objets. Principalement, on a :

+	Public	:	accessible par tous
#	Protégé	:	accessibles aux sous-classes seulement
−	Privé	:	restreint à la classe seule

Protégé

Plus sécurisé que Public, ne peut être modifié que par les sous-classes, et donc par des objets que l'on connait et maîtrise (théoriquement).

La visibilité

La visibilité d'un attribut (ou une opération) désigne son accessibilité par les autre objets. Principalement, on a :

+	Public	:	accessible par tous
#	Protégé	:	accessibles aux sous-classes seulement
−	Privé	:	restreint à la classe seule

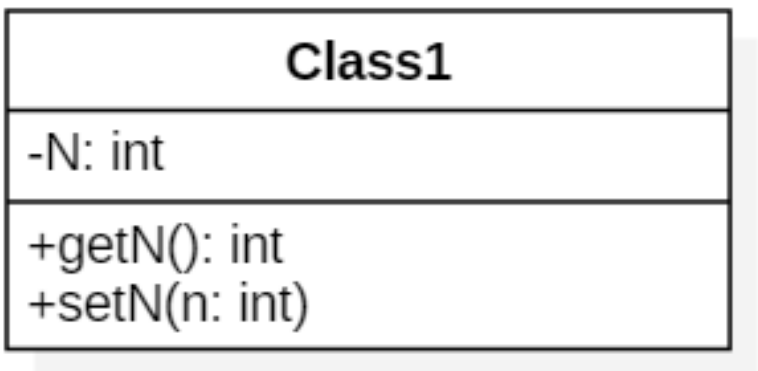
Privé

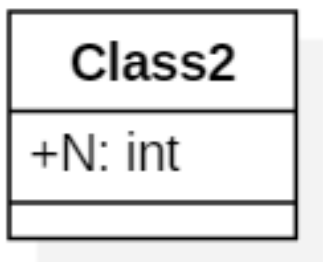
Complétement invisible aux objets extérieurs, l'objet garde la main mise sur ses attributs privé.
Possibilité de donner des droits grâce à des opérations de getter et setter.

La visibilité

La visibilité d'un attribut (ou une opération) désigne son accessibilité par les autre objets. Principalement, on a :

+	Public	:	accessible par tous
#	Protégé	:	accessibles aux sous-classes seulement
−	Privé	:	restreint à la classe seule





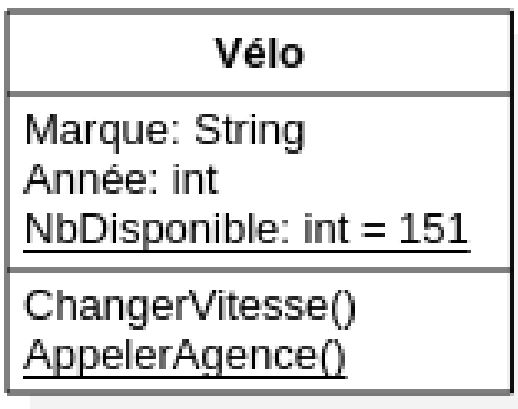
Quelle différence entre ces deux classes ?

→ Avec les getter/setter, on peut contrôler comment N est modifié

ex: $N > 0$

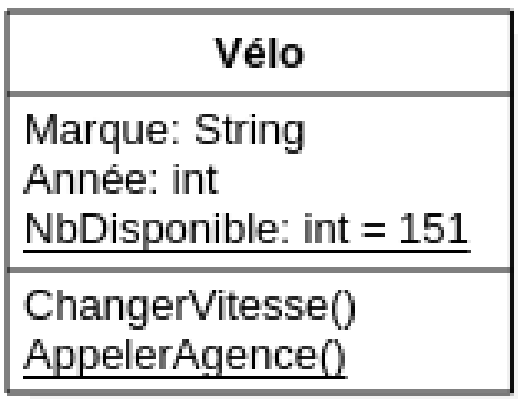
Attributs et Opérations de Classe

Les attributs et opérations de classes sont des paramètres **constants** à travers toutes les instances de la classe.



Attributs et Opérations de Classe

Les attributs et opérations de classes sont des paramètres **constants** à travers toutes les instances de la classe.
→ Le code exécuté ne dépend pas de l'objet mais de la classe !

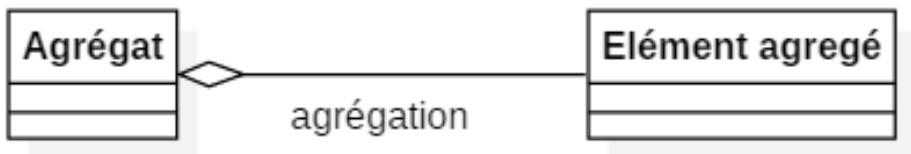


En Java, cela donne :

```
Vélo V1= new Vélo();
x= V1.Année;
y=Vélo.NbDisponible;
Vélo.AppelerAgence();
```

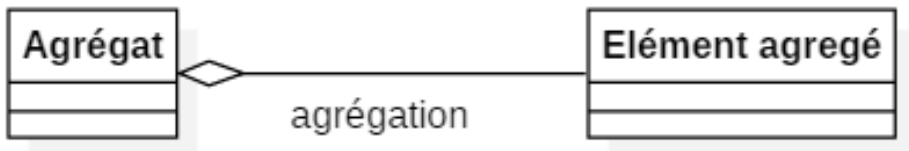
Agrégation

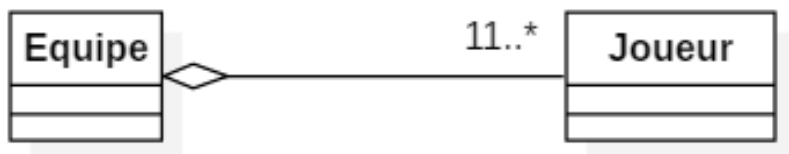
L'agrégation est un type particulier d'association. Elle est *non symmétrique*, et modélise le fait qu'une des parties joue un rôle plus fort. On parle de *contenance faible*.



Agrégation

L'agrégation est un type particulier d'association. Elle est *non symmétrique*, et modélise le fait qu'une des parties joue un rôle plus fort. On parle de *contenance faible*.

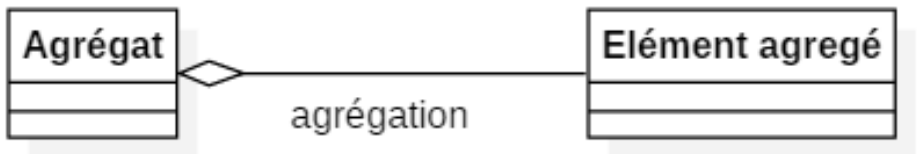




Le joueur est lié à son (ou ses) équipe(s), mais existe en dehors.

Agrégation

L'agrégation est un type particulier d'association. Elle est *non symmétrique*, et modélise le fait qu'une des parties joue un rôle plus fort. On parle de *contenance faible*.



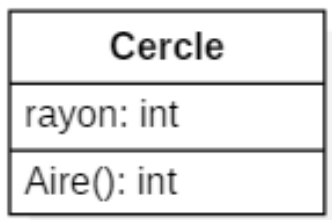
L'agrégation ne suppose pas d'implémentation particulière, c'est un outil de *modélisation*. Les deux ci-dessous sont valables (mais pas en même temps !).

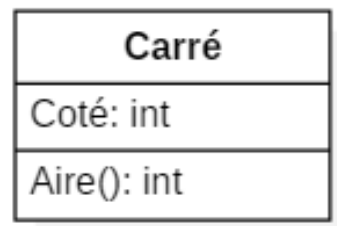
Généralisation

La généralisation est un type de *relation*, comme l'agrégation par exemple. Il s'agit d'une relation de *classification*.
$\rightarrow$ Elle permet de regrouper des comportements similaires sous un même chapeau.

Généralisation

La généralisation est un type de *relation*, comme l'agrégation par exemple. Il s'agit d'une relation de *classification*.
→ Elle permet de regrouper des comportements similaires sous un même chapeau.

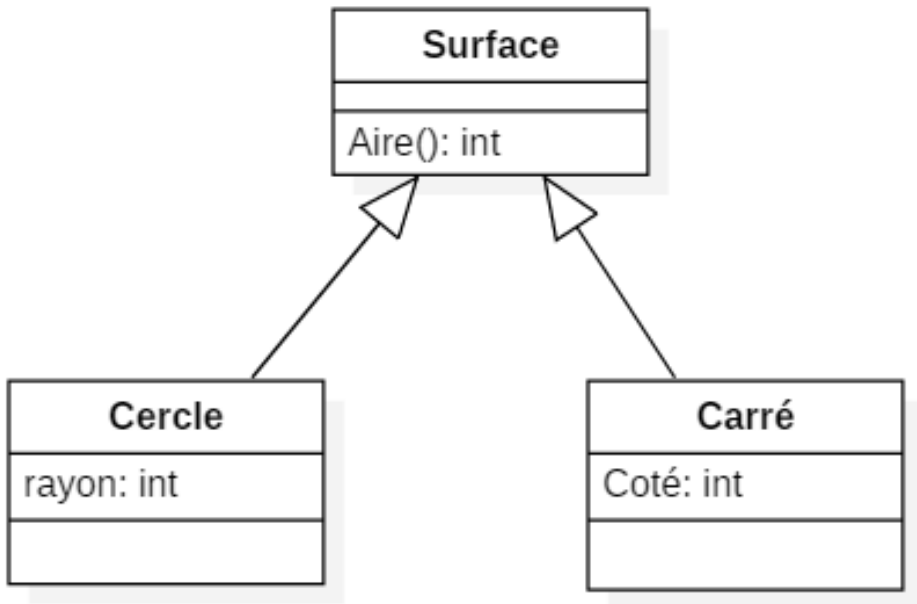




Le cercle et le carré ont des similitudes : ce sont des surfaces possédant une Aire().

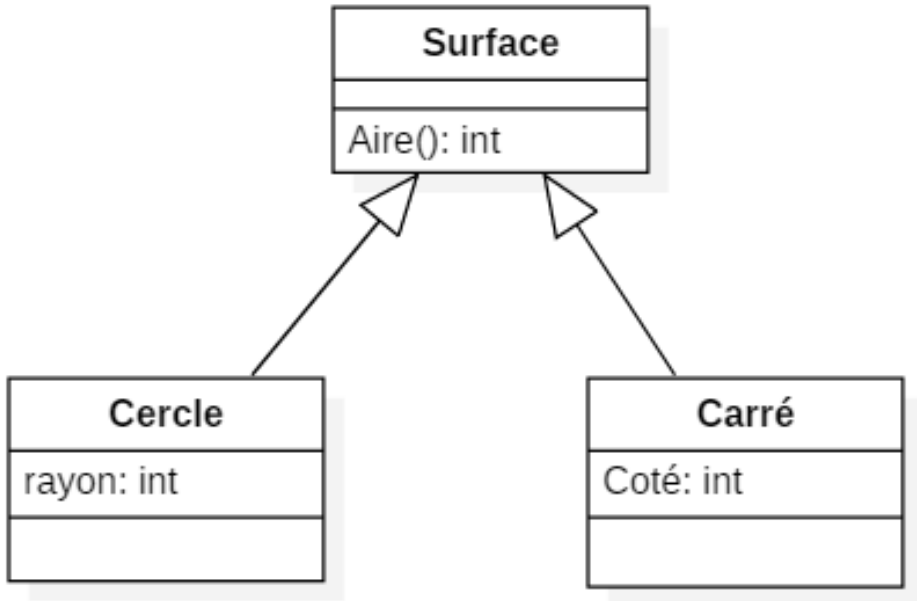
Généralisation

La généralisation est un type de *relation*, comme l'agrégation par exemple. Il s'agit d'une relation de *classification*.
→ Elle permet de regrouper des comportements similaires sous un même chapeau.



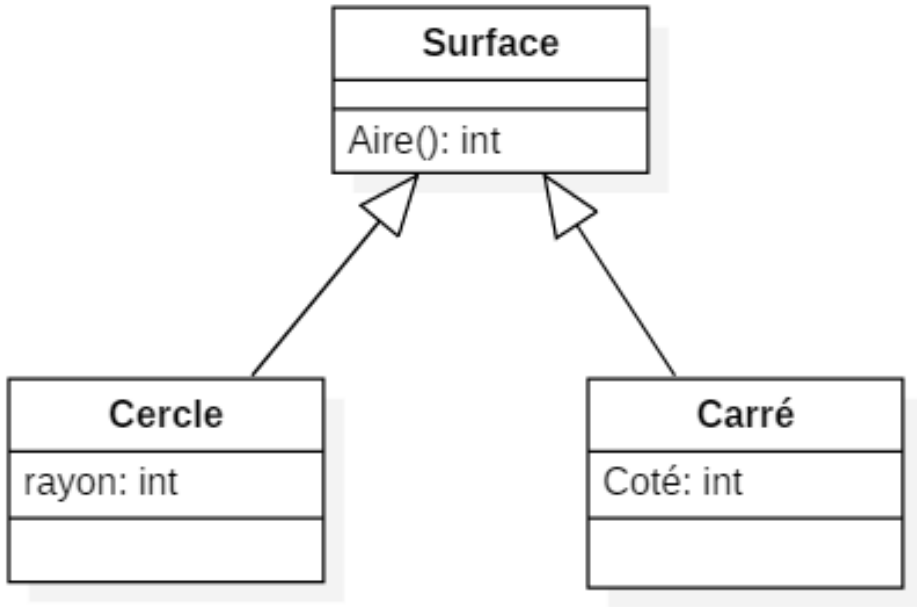
La classe **Surface** *généralise* le cercle et le carré. Inversement, le cercle *est une* surface.

Super classe et classe enfant



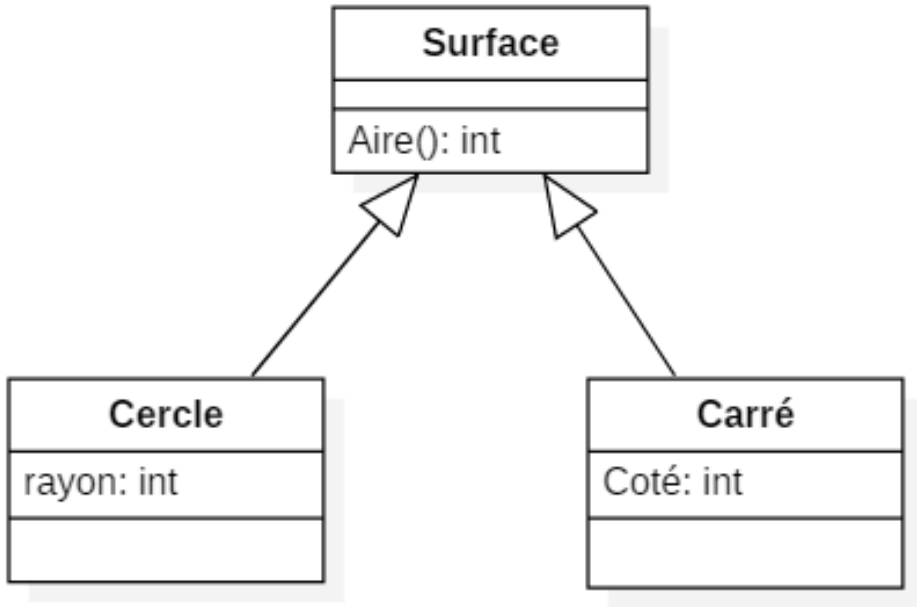
La *superclasse*, ou *classe parent*, est l'élément générique. Elle comporte les éléments communs à toutes ses classes enfants.

Super classe et classe enfant



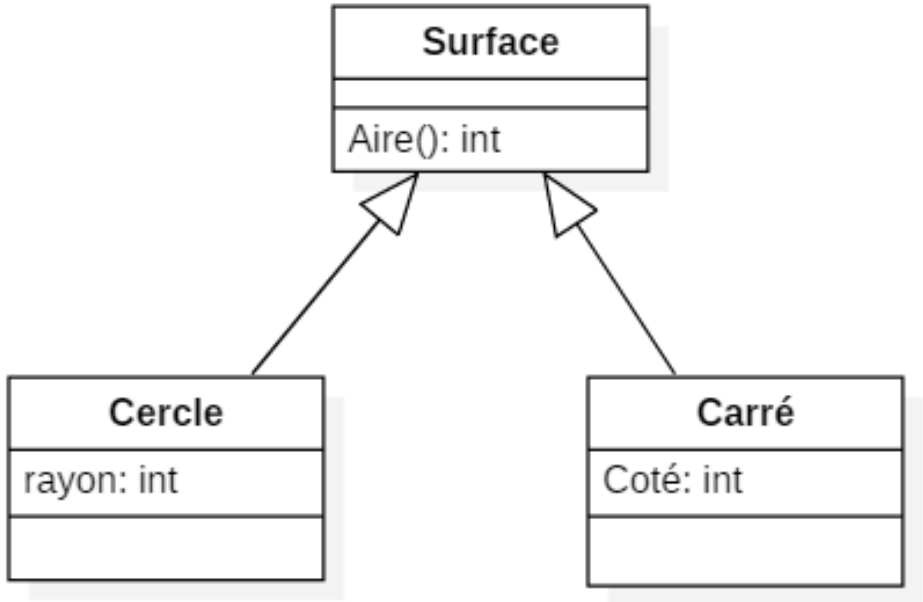
La *sous-classe*, ou *classe enfant*, est la spécialisation. Elle hérite des attributs, opérations et relations de la classe parent.
Elle est un cas particulier de la superclasse, un sous-ensemble.

Super classe et classe enfant



D'un point de vue notationnel, les éléments (attributs, opérations...) de la superclasse existent dans la sous-classe, mais ne sont pas représentés dans le diagramme.

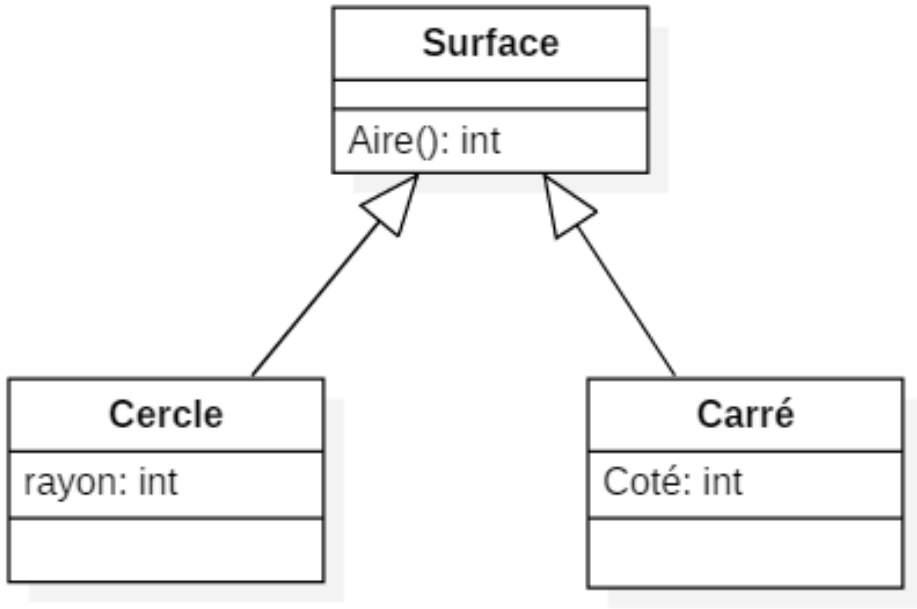
Super classe et classe enfant



Principe de Substitution

Dans un programme, on doit *toujours* pouvoir remplacer un objet d'une superclasse par un objet d'une sous-classe sans en changer la sémantique !

Super classe et classe enfant



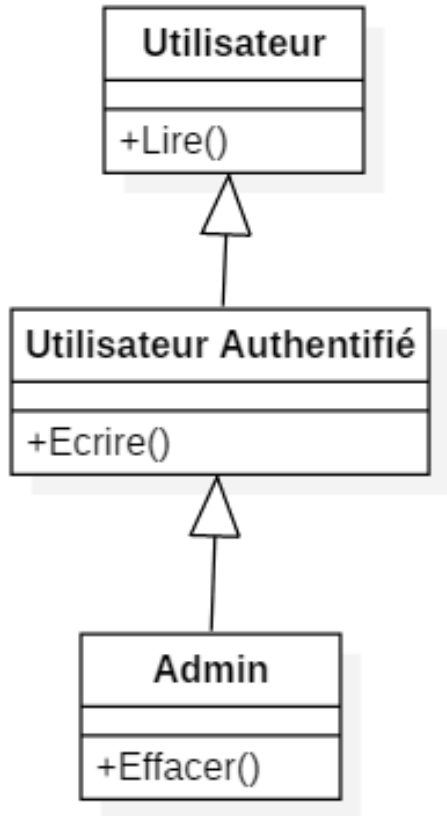
Différence avec l'agrégation :

le chat *est* un animal → généralisation

le chat *a* un maître → agrégation

Plusieurs couches de généralisation

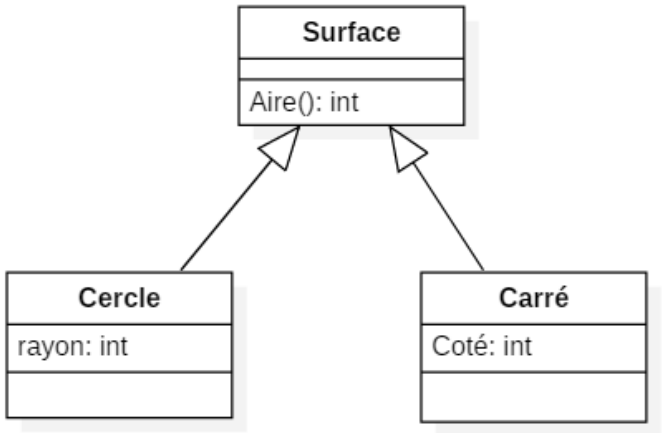
La généralisation est transitive, et une super-classe peut être une sous-classe !



→ Un Admin peut Effacer, Ecrire ET Lire !

Implémentation en Java

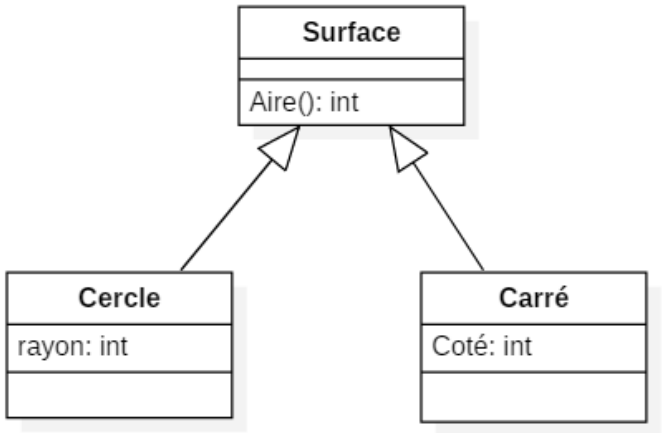
La notion de *généralisation* est très proche de la notion d'*héritage*, au sens de Java.




```
public class Surface {
public int Aire(){...}
...}
public class Cercle extends
Surface {...}
```

Implémentation en Java

La notion de *généralisation* est très proche de la notion d'*héritage*, au sens de Java.



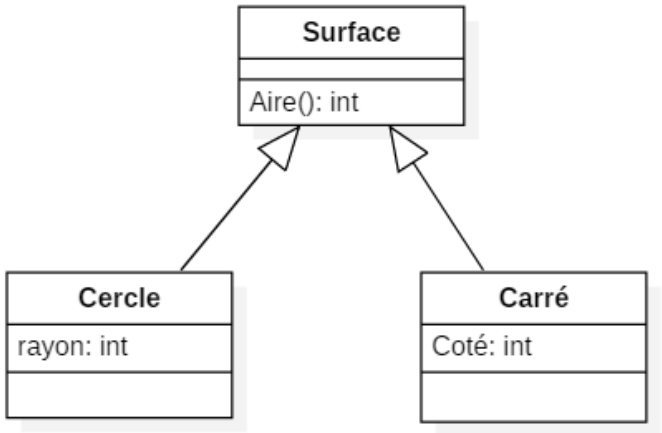

```
public class Surface {
public int Aire(){...}
...}
public class Cercle extends
Surface {...}
```

La classe Cercle hérite de Aire() de Surface. On peut **redéfinir** Aire() dans la classe Cercle avec le mot-clé Override :

```
 @Override
public int Aire(){
return Math.pow(this.rayon,2)*Math.PI;}
```

Implémentation en Java

La notion de *généralisation* est très proche de la notion d'*héritage*, au sens de Java.




```
public class Surface {
public int Aire(){...}
...}
public class Cercle extends
Surface {...}
```

La classe Cercle hérite de Aire() de Surface. On peut **redéfinir** Aire() dans la classe Cercle avec le mot-clé Override :

```
 @Override
public int Aire(){
return Math.pow(this.rayon,2)*Math.PI;}
```

Attention !

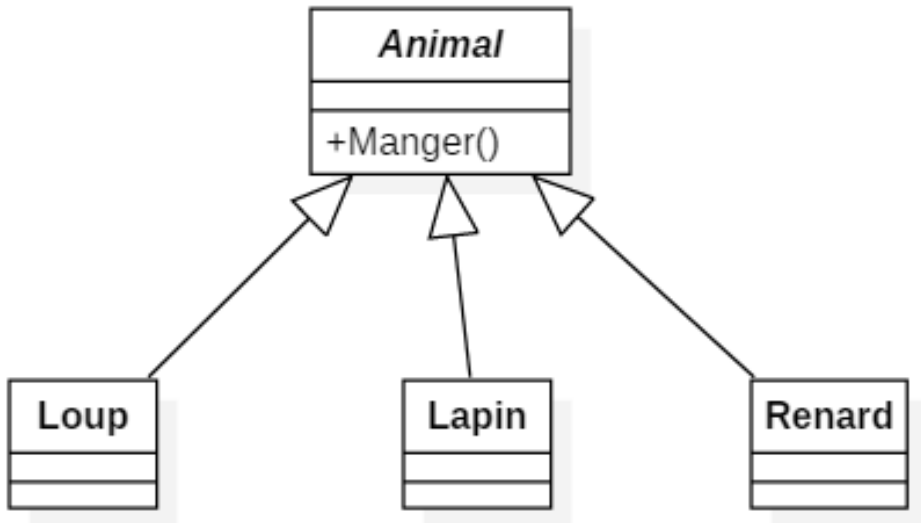
Une classe ne peut hériter que d'une seule classe !

Classe abstraite

Une classe abstraite est une classe *non instanciable*, càd qu'aucun objet ne l'instancie. Autrement, elle est définie comme une classe normale, avec ses attributs et ses opérations.

Classe abstraite

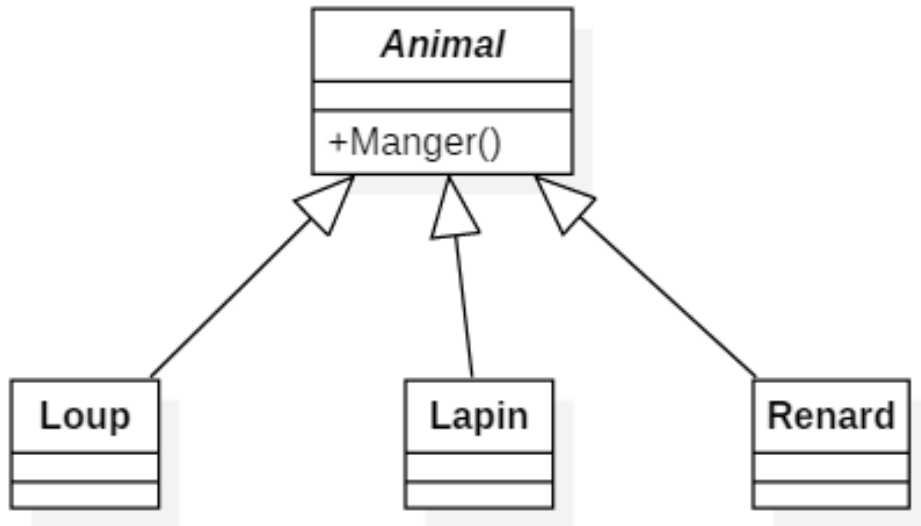
Une classe abstraite est une classe *non instanciable*, càd qu'aucun objet ne l'instancie. Autrement, elle est définie comme une classe normale, avec ses attributs et ses opérations.



Un animal n'existe pas en tant que tel, il appartient à une espèce.

Classe abstraite

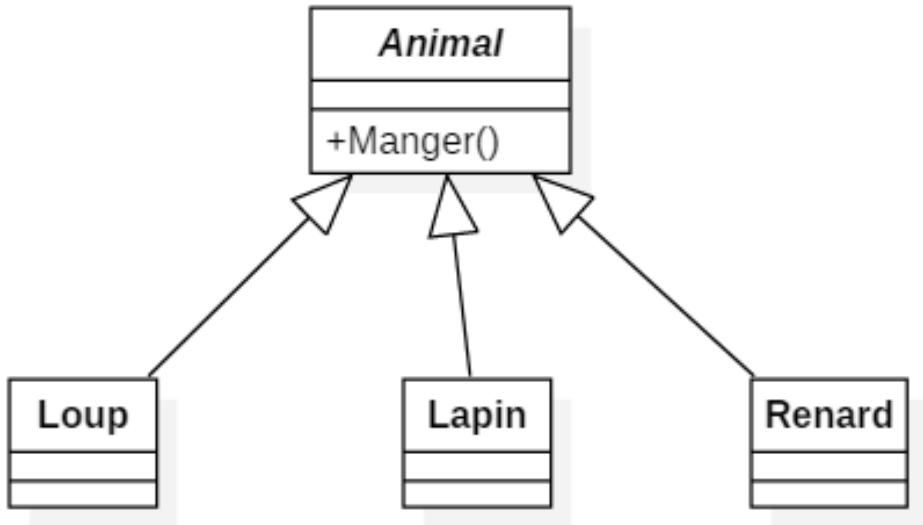
Une classe abstraite est une classe *non instanciable*, càd qu'aucun objet ne l'instancie. Autrement, elle est définie comme une classe normale, avec ses attributs et ses opérations.



Un animal n'existe pas en tant que tel, il appartient à une espèce. Quelle est l'utilité d'une telle classe ?

- Comme une superclasse quelconque, elle généralise et encapsule plusieurs classes similaires
- Permet un regroupement des comportements

Classe abstraite, implémentation Java

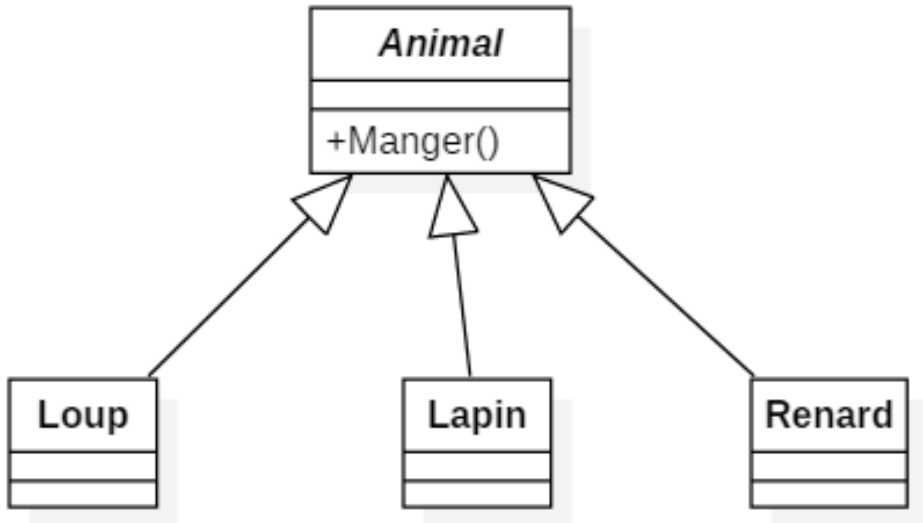


Les classes abstraites sont présentes en Java. On les note (ici, Manger est une opération abstraite) :

```
abstract class Animal {

        }
```

Classe abstraite, implémentation Java



Les classes abstraites sont présentes en Java. On les note (ici, Manger est une opération abstraite) :

```
abstract class Animal {
    abstract void Manger();
        }
```

Attention : Une classe abstraite doit généraliser une classe concrète !

Exercice

Une classe abstraite peut contenir des attributs ou des opérations (abstraites ou non).

Pour représenter les véhicules motorisés, nous avons besoin entre autre des classes *Véhicule*, *Terrestre*, *Maritime*, *Camion* et *Barque*.
Lesquelles sont des classes abstraites ? Représentons-les.

Exercice

Une classe abstraite peut contenir des attributs ou des opérations (abstraites ou non).

Pour représenter les véhicules motorisés, nous avons besoin entre autre des classes *Véhicule*, *Terrestre*, *Maritime*, *Camion* et *Barque*.
Lesquelles sont des classes abstraites ? Représentons-les.
Comment et où représenter le nombre de passagers ? Le nombre de roues ?

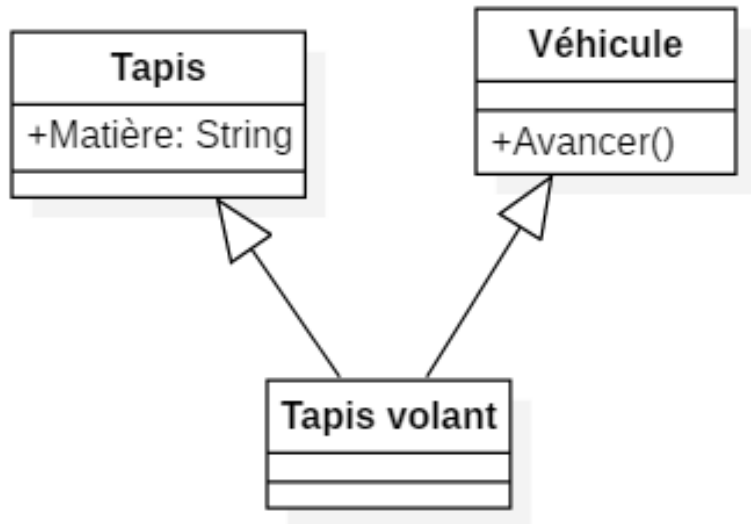
Exercice

Une classe abstraite peut contenir des attributs ou des opérations (abstraites ou non).

Pour représenter les véhicules motorisés, nous avons besoin entre autre des classes *Véhicule*, *Terrestre*, *Maritime*, *Camion* et *Barque*.
Lesquelles sont des classes abstraites ? Représentons-les.
Comment et où représenter le nombre de passagers ? Le nombre de roues ?
Un Camion peut contenir d'autres véhicules, telles que des Voitures ou des Barques. Comment le représenter ?

Généralisation multiple

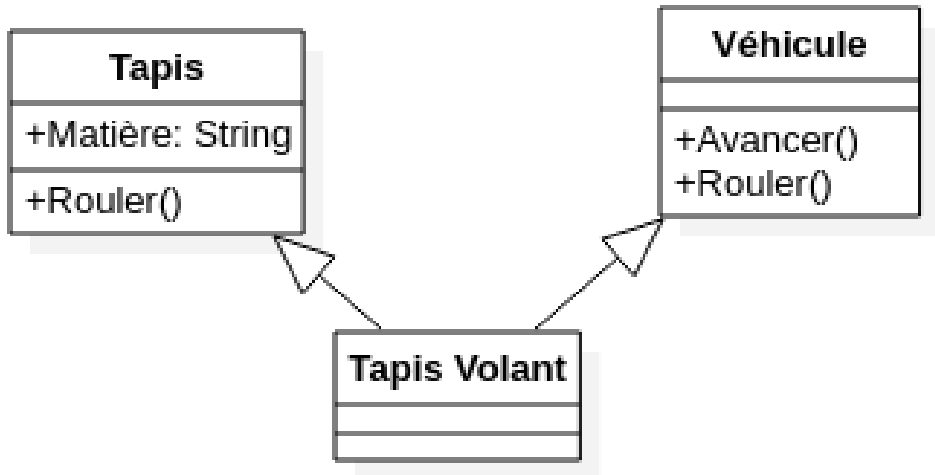
Contrairement à Java, rien n'interdit la généralisation multiple en UML !



Un tapis volant hérite des propriétés d'un tapis ainsi que d'un vehicule ! Il possède une Matière ainsi qu'une Vitesse.

Généralisation multiple

Cependant Java interdit la généralisation multiple !



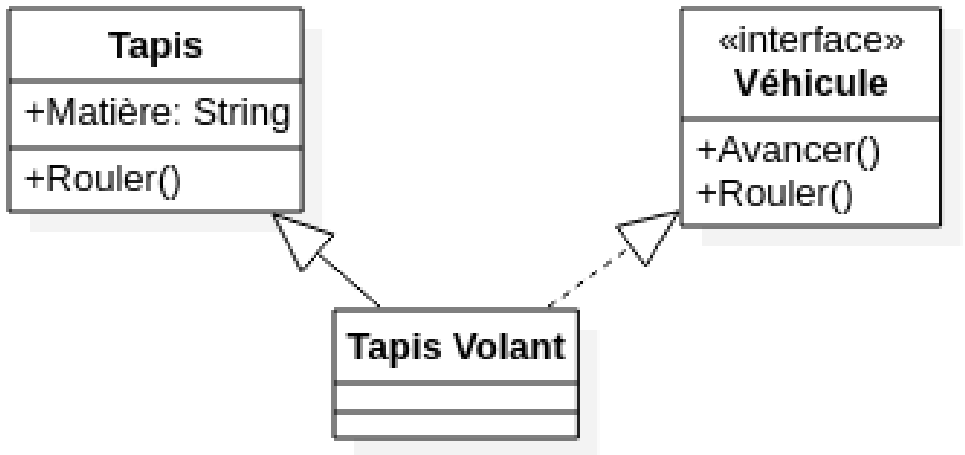
Car que se passe-t-il si j'appelle Rouler() sur un Tapis Volant ?
Quel code est executé ?

Généralisation multiple : Implémentation Java ?

Si Java n'autorise pas la généralisation multiple, comment faire ?

Généralisation multiple : Implémentation Java ?

Si Java n'autorise pas la généralisation multiple, comment faire ?
`public class TapisVolant extends Tapis implements Vehicule`

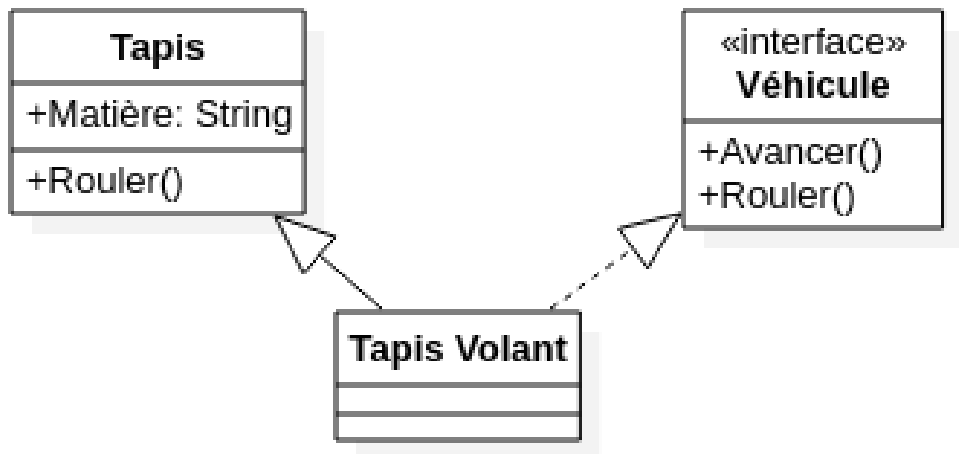


Interface

Une *interface* est un ensemble d'opérations **publiques** (pas d'attributs) → Il ne s'agit que d'une liste, pas d'une implémentation. Une interface est donc une classe abstraite ne contenant que des méthodes abstraites.

Généralisation multiple : Implémentation Java ?

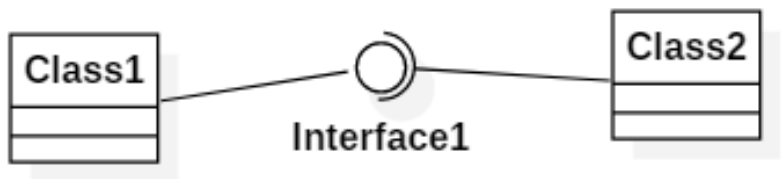
Si Java n'autorise pas la généralisation multiple, comment faire ?
`public class TapisVolant extends Tapis implements Vehicule`



Si j'appelle Rouler() sur un Tapis Volant, le code sera celui de Tapis car Véhicule ne propose pas d'implémentation ! Il n'y a plus d'ambiguité !

Interface

Une interface décrit donc un ensemble d'opérations, implémentées par certaines classes et utilisées par d'autres.
On spécifie ainsi les échanges entre Class1 et Class2.



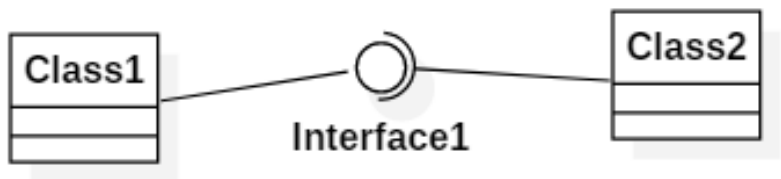
Interface

Une interface décrit donc un ensemble d'opérations, implémentées par certaines classes et utilisées par d'autres.
On spécifie ainsi les échanges entre Class1 et Class2.

Relation de réalisation : Les classes les *réalisant* (ici Class1) implémentent les opérations.
`public Class1 implements Interface1`
Relation de dépendance : Les classes *dépendantes* (ici Class2) utilisent les opérations de l'interface, mais ne les implémentent pas (par exemple, dans l'opération main d'un fichier Test.java).

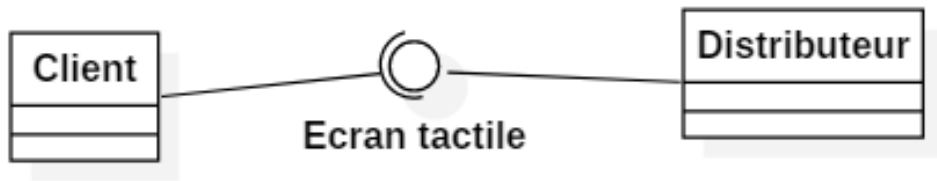
Interface

Une interface décrit donc un ensemble d'opérations, implémentées par certaines classes et utilisées par d'autres.
On spécifie ainsi les échanges entre Class1 et Class2.



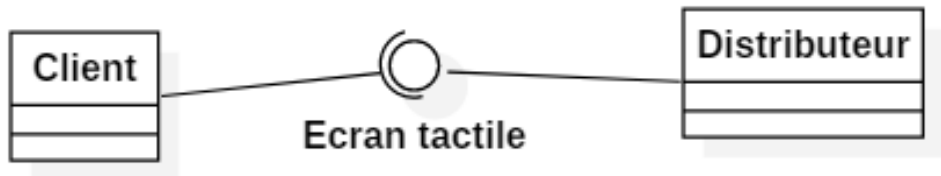
Interface

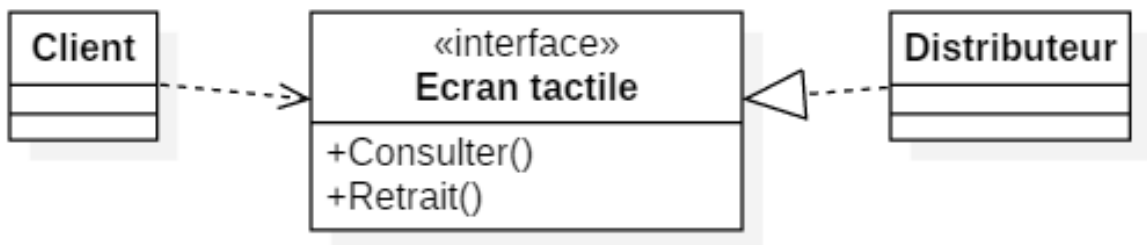
Représentations dans un Diagramme de Classe :



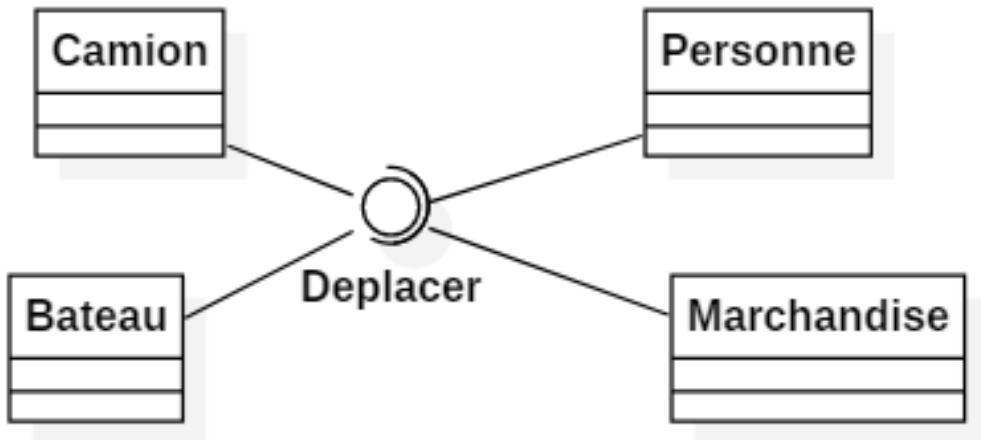
Interface

Représentations dans un Diagramme de Classe :



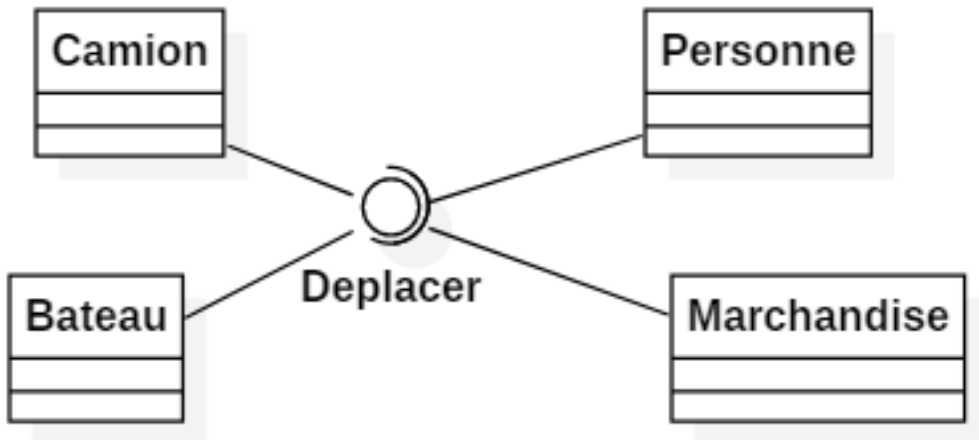


Interface



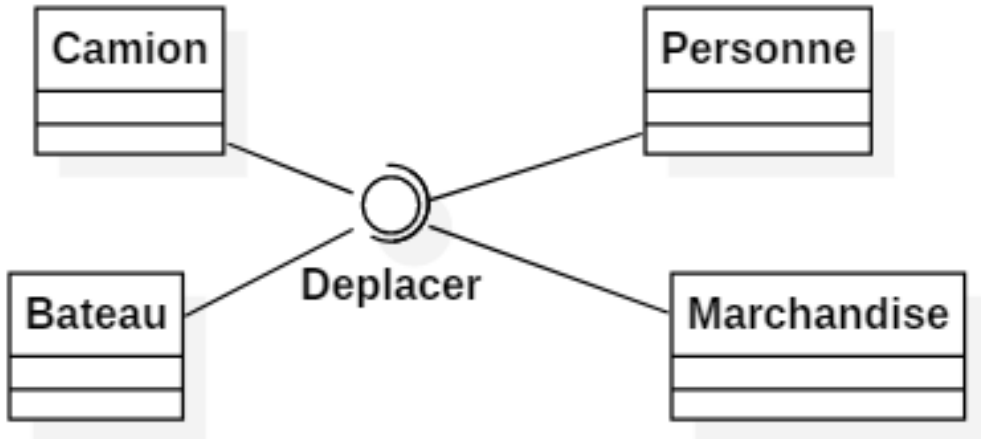
Camion et Bateau n'ont pas à se déplacer de la même façon ! Ils implémentent l'interface Deplacer de façon différentes.

Interface



Personne et marchandises, bien que différentes, vont interagir avec les moyens de transport de la même façon !

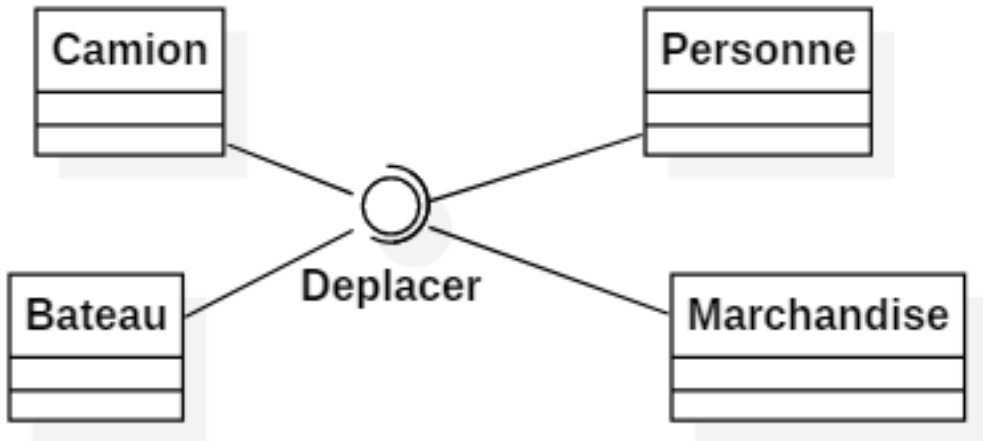
Interface



Déclaration d'une interface :

```
public interface Deplacer{
    public void Avancer();
    public void Reculer();
}
```

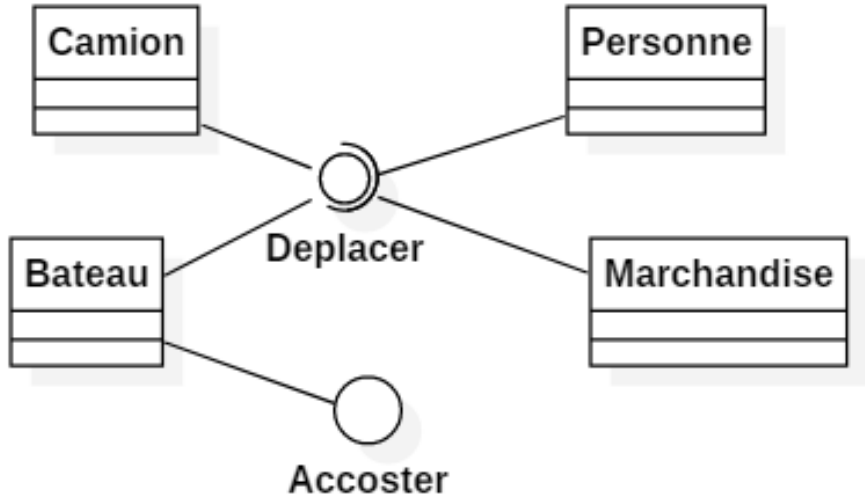
Interface



Implémentation de l'interface :

```
public class Bateau implements Deplacer
    public void Avancer(){...}
    public void Reculer(){...}
```

Interface



Implémentation d'interfaces multiples :

```
public class Bateau implements Deplacer,Accoster
    public void Avancer(){...}
    public void Reculer(){...}
    public void Amarrer(){...}
```