



02 Le paradigme objet

Chapitre

1I2AC1 : Génie logiciel et Conception orientée objet

Régis Clouard, ENSICAEN - GREYC

« N'importe quel programmeur peut écrire du code que l'ordinateur comprend. Les bons programmeurs écrivent du code que les humains peuvent comprendre. »

Martin Fowler

Objectif du chapitre

- Initiation au paradigme objet pour la conception logicielle.
 - Première approche de la conception orientée objet.
- À l'issue de ce chapitre :
 - Vous serez capable de concevoir une modélisation qui tire profit des concepts introduits par le paradigme objet.
 - Vous saurez relire une modélisation objet.
 - Vous mesurerez l'intérêt du paradigme objet pour concevoir des logiciels complexes.

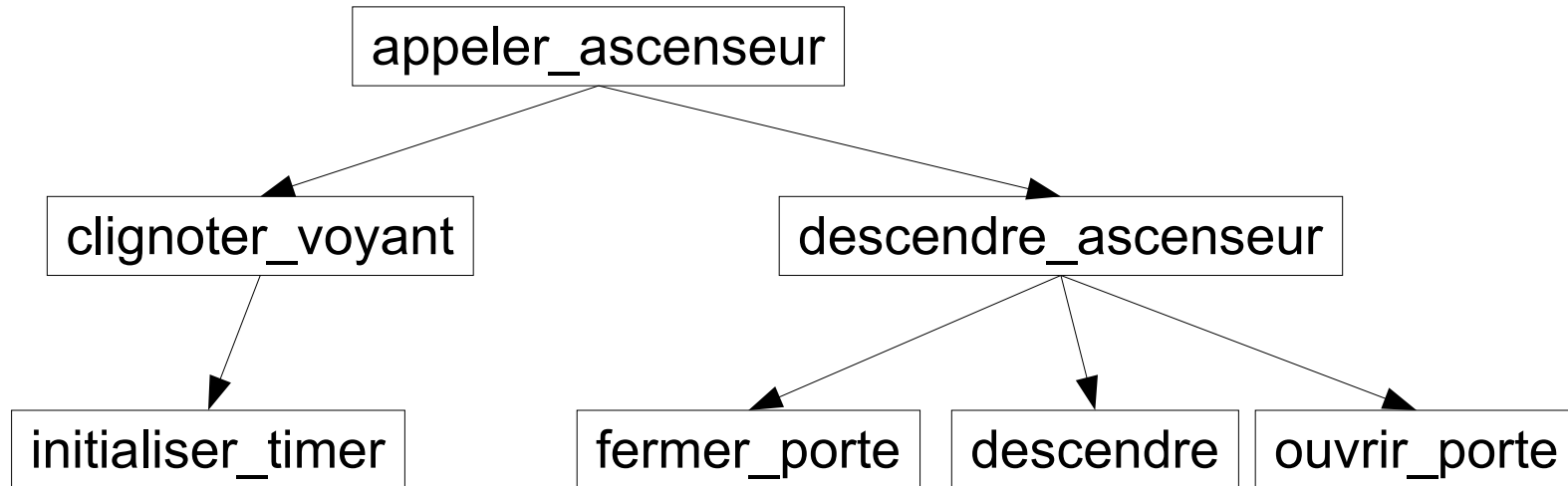
Plan du chapitre

1

Le paradigme
objet

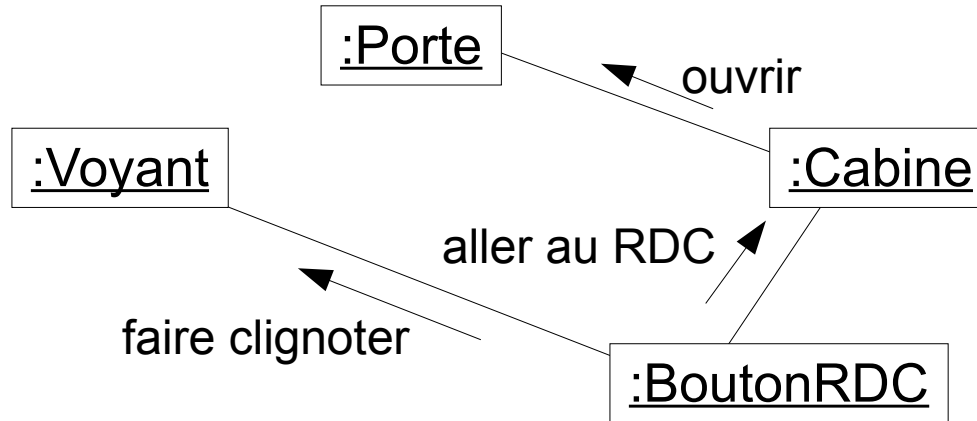
Paradigme procédurale

- On s'intéresse aux traitements : actigramme.
 - Programme : enchaînement de procédures s'échangeant des données.
 - Les données sont **inertes** : elles sont modifiées par les procédures.



Paradigme objet

- On s'intéresse aux données : datagramme.
 - Programme : ensemble d'objets s'échangeant des services.
 - Les données sont **animées** : ce sont elles qui exécutent les procédures.



Conception procédurale

- Méthode de conception : **Algorithmique**.
 - Trouver la séquence d'appels de procédures.
- Avantages
 - Modélisation proche du fonctionnement de la machine.
 - Calculs de complexité.
 - Preuves de programmes dans certains cas.
- Limites
 - Modélisation inaccessible aux clients.
 - Inadaptée aux gros logiciels.
 - ▶ Nécessite de savoir résoudre le problème globalement pour l'implémenter.
 - ▶ Mélange conception et implémentation.
 - Enchaînement figé dans le temps.
 - Maintenance et réutilisabilité difficiles dues à l'interdépendance des procédures entre elles et avec les données.

Conception orientée objet

7

- Méthode de conception : **Modélisation**
 - Trouver les objets du domaine d'application à utiliser.
 - « Si je disposais d'un chapeau magique, quels types de données voudrais-je voir sortir du chapeau pour m'aider à résoudre le problème ? ».
- Avantages
 - Permet d'appréhender la complexité (gros logiciels).
 - ▶ On peut repousser le plus tard possible l'implémentation.
 - ▶ La conception est basée sur un raffinement progressif.
 - Maintenance et réutilisabilité facilitées : objet = bloc indépendant.
- Limites
 - Vision fractionnée du fonctionnement du logiciel.
 - Calculs de complexité et preuves difficiles (p.ex. à cause du polymorphisme).

Langages de programmation objet

- Foisonnement de langages de programmation orientés objet.
 - Simula (Ole-Johan Dahl & Kristen Nygaard, 1963).
 - SmallTalk (Alan Kay, 1972), Eiffel (Bertrand Meyer ,1986) .
 - **C++** (Bjarne Stroustrup, 1983), Objective C, D.
 - **Java** (Sun MicroSystem, 1991), *C# : basés sur UML*.
 - **PHP**, **Python**, Perl, Clojure.
 - Ruby, Scala.
- La différence entre langage procédural et langage orienté objet :
 - Ne concerne que quelques mots clés et types de données.
 - **Mais** ce sont deux paradigmes totalement différents.
- Conséquence :
 - Le paradigme objet ne s'apprend pas par le langage.

Un formalisme de modélisation

- UML : Unified Modeling Language (normalisé).
 - **Visuel** (à base de diagrammes).
 - **Indépendant** de tout langage de programmation.
 - Débarrassé des **contraintes syntaxiques**.
 - **Correspondance** forte avec les langages de programmation.
UML ↔ Java, C++, Php, Python...
- Il sera vu plus en détail dans le chapitre 3.

Concepts de la conception objet

- La conception orientée objet s'appuie sur 5 concepts :
 - Objet
 - Classe
 - Association
 - Héritage
 - Polymorphisme

Plan du chapitre

1

Le paradigme
objet

2

Les objets

Notion d'objet

- Prosaïquement
 - **Objet = structure en C + pointeurs sur des procédures.**
- Conceptuellement
 - **Objet = attributs + méthodes.**
 - **Attribut : donnée propre.**
 - ▶ Possédant une valeur.
 - ▶ Évoluant au cours du temps.
 - **Méthode : procédure associée.**
 - ▶ Utilisant potentiellement les attributs pour fonctionner.
 - ▶ Déclenchée par appel explicite.

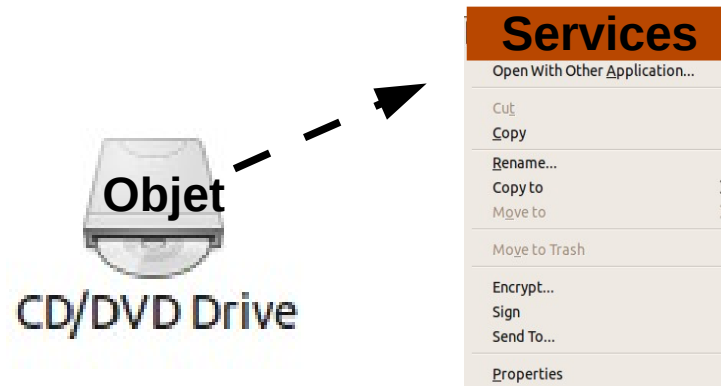
AT-013-SR: Car

color = blue
weight= 979 kg
power = 100 hp

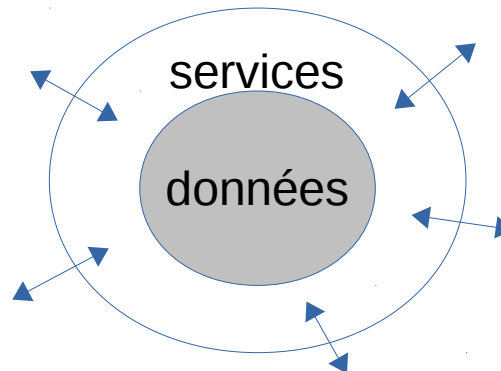
move()
stop()
refuel()

Encapsulation

- Penser un objet comme un **fournisseur de services** et pas comme une structure qui stocke des valeurs de données.

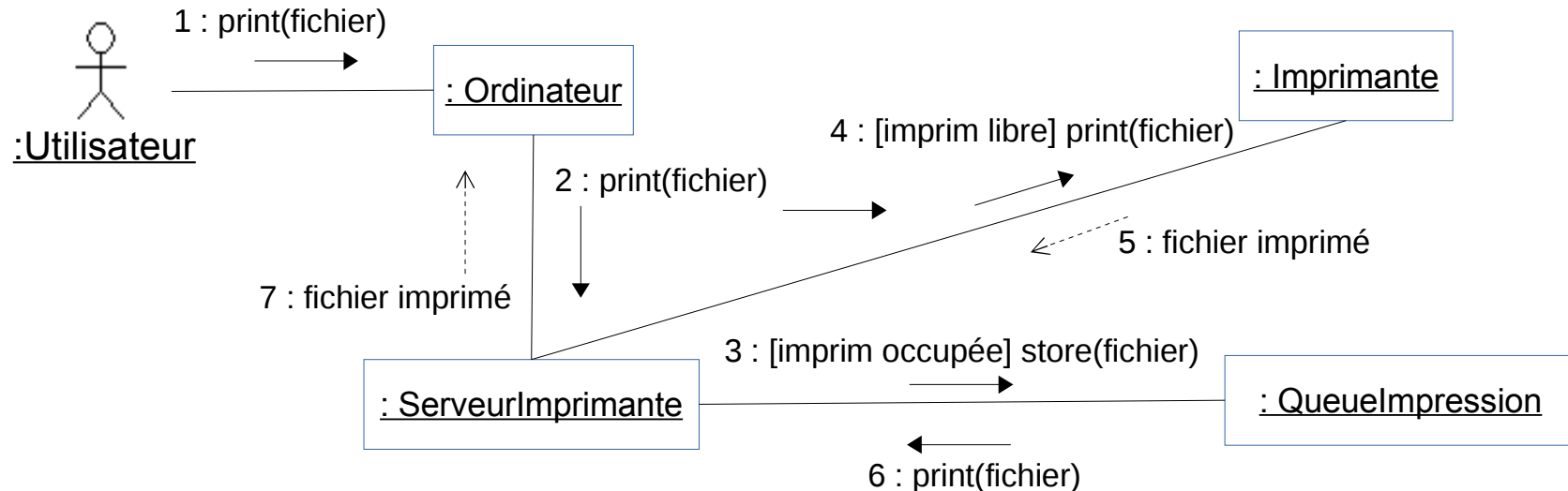


- Les données sont **cachées et protégées**, elles ne servent qu'à assurer les services.



Communication entre objets

- L'unité de communication est le **message**.
 - Correspond à l'appel d'une procédure d'un objet.
 - Relation de communication dynamique.
- Interactions pour réaliser les fonctionnalités du logiciel.
 - Séquence de messages échangés entre les objets.



Plan du chapitre

1

Le paradigme
objet

2

Les objets

3

Les classes

Classe

- Représente un **concept** du problème.
 - Concept : *une entité ayant une représentation dans le monde de la modélisation.*
- Une classe est une génératrice d'objets qui définit pour chaque objet :
 - La liste des attributs.
 - La liste des méthodes.
 - La liste des relations avec les autres classes.
- Notation UML
 - *Rectangle.*
 - *Nom un substantif au **singulier**.*
 - *Casse : première lettre en majuscule et CamelCase.*



Attribut

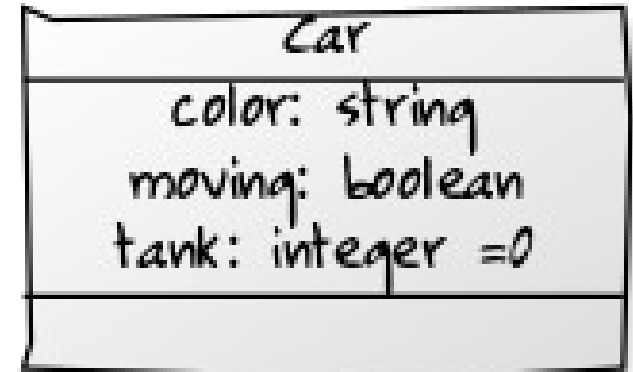
- Donnée que possède tout objet de la classe.
 - Possède une valeur unique pour chaque objet.

■ Important

 **Les attributs ne peuvent être que d'un type primitif (ou assimilé) :** float, double, boolean, int, char, enumerate, string, int[], date, time.

■ Notation UML

- *Placé dans un premier compartiment.*
- *Nom : un substantif.*
- *Casse : 1ere lettre en minuscule et camelCase.*
- *On peut ajouter :*
 - ▶ *le type.*
 - ▶ *une valeur par défaut.*
 - ▶ *la multiplicité (pour les tableaux, ensembles, ...).*



Méthode

- Procédure qui définit le comportement des objets.

- Procédure classique avec paramètres et valeur de retour.

```
type_retour nom( paramètre_formel* ) { }
```

- La signature d'une méthode :

- ▶ nom + paramètres.

- L'appel se fait par l'intermédiaire d'un objet de la classe.

- Notation UML

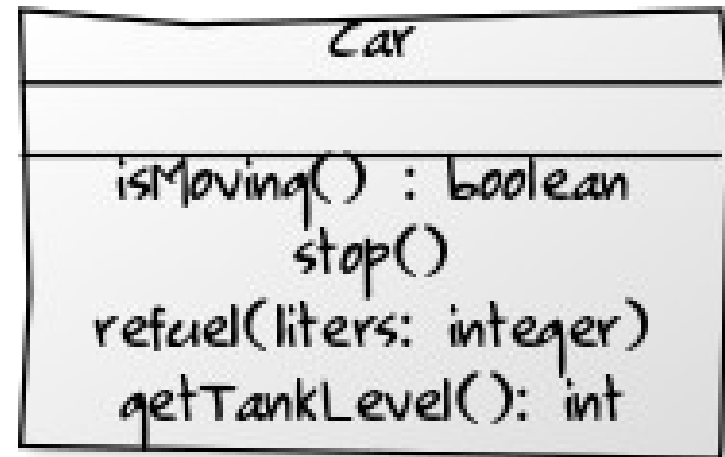
- Placée dans un deuxième compartiment.

- Nom : un verbe.

- Casse : 1ere lettre en minuscule et camelCase.

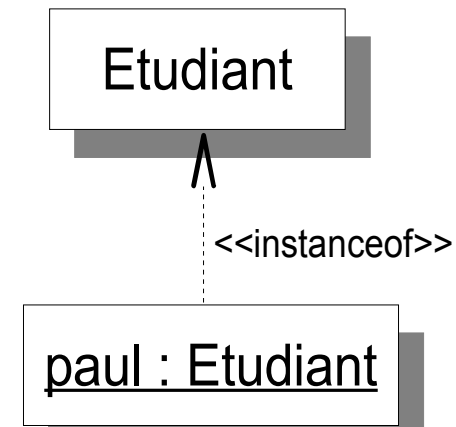
- On peut ajouter :

- ▶ La signature + la valeur de retour.



Objet : instance d'une classe

- Instance de classe :
 - Manifestation concrète d'une abstraction.
 - ▶ qui propose des méthodes comme un ensemble de services.
 - ▶ qui possède des attributs capables de mémoriser les effets des méthodes.
- Notation UML
 - *Un rectangle avec un nom souligné.*
 - *Nom : un substantif.*
 - *Casse : première lettre en minuscule et camelCase.*



Objet

- Caractère statique de l'instance.
 - une instance ne peut pas changer de classe.
- Accès à un attribut ou à une méthode.
 - Notation pointée :
 - ▶ instance.attribut
 - ▶ instance.operation()
 - ▶ p.ex. pierre.calculeCotisation()
- L'auto-référence : **'this'** ou **'self'**.

toto:Chenille

toto:Papillon

pierre : AssureSocial

calculeCotisation(): float

Implémentation en Java

■ Déclaration de la classe

```
class Car {  
    String color = "green";  
    boolean moving;  
    int tank = 0;  
    boolean isMoving(){ return moving; }  
    void stop(){ moving = false; }  
    void refuel(int liters){ tank = liters; }  
    int getTankLevel(){ return tank; }  
}
```

■ Création d'un objet :

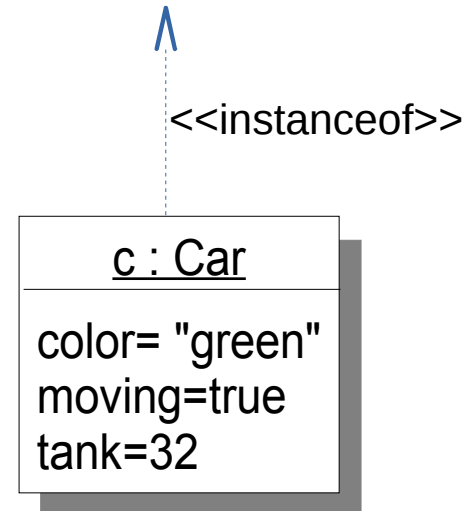
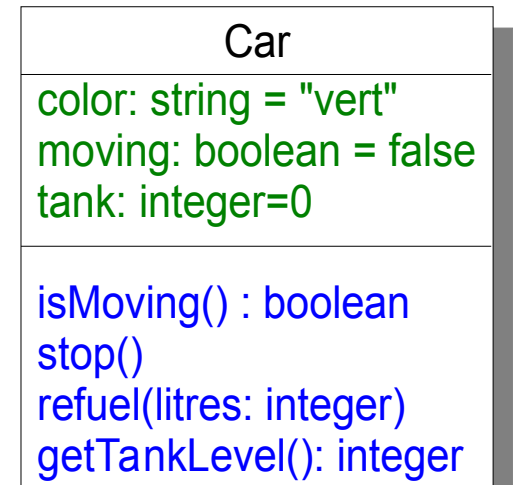
- `Car c = new Car();`

■ Appel de méthodes :

- `c.stop(); c.refuel(32);`

■ Destruction d'un objet :

- `c = null;`



Plan du chapitre

1

Le paradigme
objet

2

Les objets

3

Les classes

4

Associations
entre
classes

Association

- Relation organisationnelle.
 - Permet d'accéder aux services des instances de la classe associée.
- Notation UML



- Cas particulier : association réflexive.



Navigabilité des associations

- Restreindre l'accessibilité à un sens unique.

- Notation UML

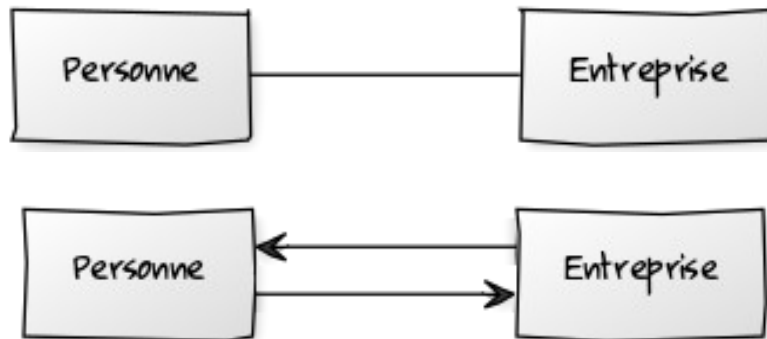
- *une flèche ouverte.*



- Par défaut

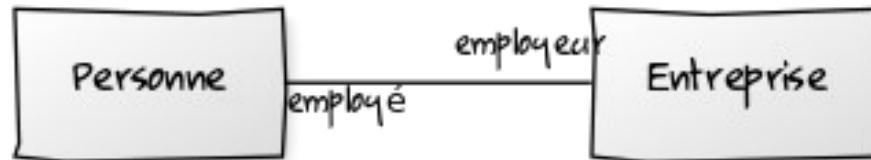
- Navigable dans les deux sens.
- Notation UML.
 - ▶ *un trait entre les deux classes participantes.*

- ▶ *équivalent à :*



Rôle des associations

- Spécifier la fonction de la classe associée.
- Notation UML
 - *Placé sur le bout de flèche de l'association.*
 - *Casse : en minuscule.*
 - *En pratique : un substantif.*



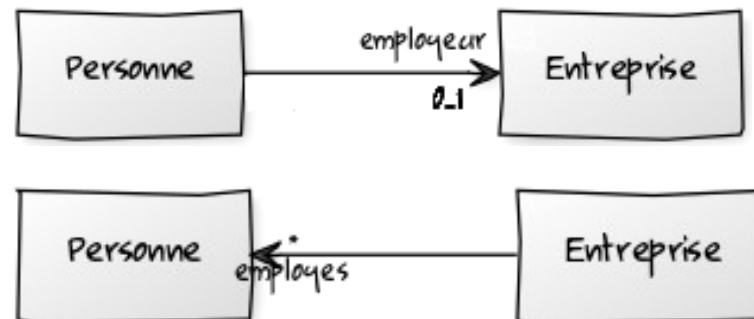
Multiplicité des associations

- Définir le nombre d'instances de l'association pour une instance de la classe.
 - Un nombre entier ou un intervalle de valeurs.

1	Une et une seule instance (par défaut)
0..1	Zéro ou une instance
M..N	De M à N instances
*	De zéro à plusieurs instances
1..*	De 1 à plusieurs instances
N	Exactement N instances

- Notation UML

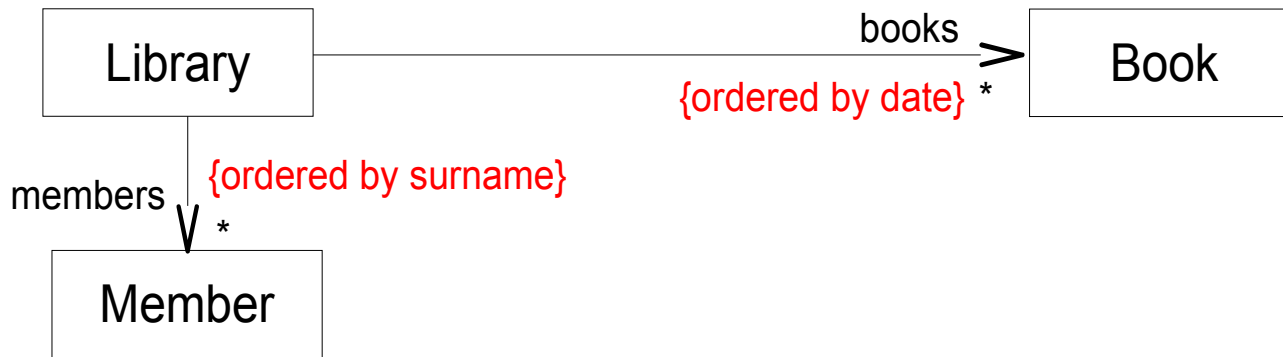
- Notée avec le rôle.
- Par défaut 1 (non notée).



Contraintes sur les associations

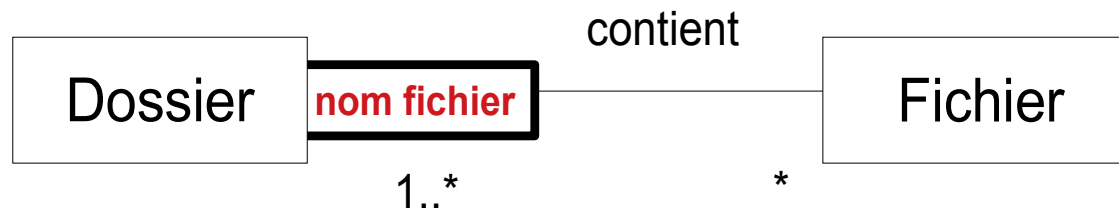
■ Association **ordonnée**.

- unordered (c'est la valeur par défaut).
- ordered : les associés sont stockés dans un ordre précis.



■ Association **qualifiée** : tableau associatif, table de hachage, dictionnaire.

- Les éléments sont accessibles par une clé (p.ex, *le nom du fichier*).

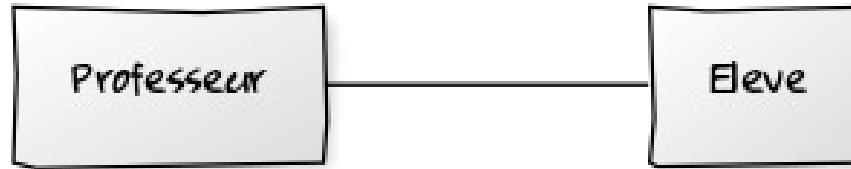


Types d'association

- 4 types d'association :
 - standard
 - agrégation
 - composition
 - dépendance

Association standard

- C'est la relation par défaut.
 - Relation de type « connaît ».
- Notation UML



Association de type agrégation

- Relation ensembliste non symétrique de type « possède ».
 - Une classe joue le rôle d'ensemble et une autre classe le rôle d'élément.
 - Relation à portée essentiellement sémantique.

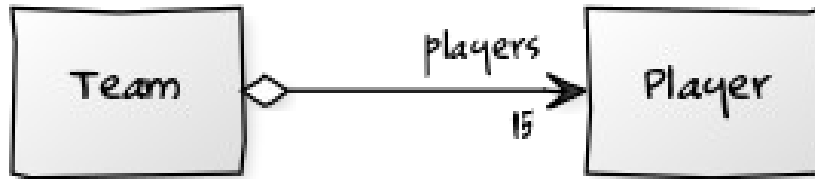
Relation	Exemple
Composé / Composant	<i>Voiture / Roues</i>
Collection / Élément	<i>Forêt / Arbres</i>
Espace / Position	<i>Desert / Oasis</i>
Événement / Étape	<i>Document / Chapitre</i>

- Conséquences visibles sur le code :
 - Des méthodes dans l'agrégat pour ajouter ou supprimer des éléments.

Association de type agrégation

31

- Notation UML :
 - *Un losange vide du côté de l'agrégat.*



- Notation Java : tableau, liste, vecteur ...

```
class Team {
    Player[] _players = new Player[15];
    void add( Player p, int i ) {
        _players[i]= p;
    }
}
class Player { }
```

Association de type composition

- Relation de **subordination** non symétrique de type « *est constitué de* ».
 - S'interprète comme si la classe composant était une partie intrinsèque et privée de la classe composite.

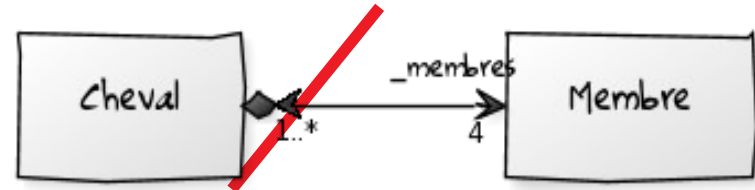
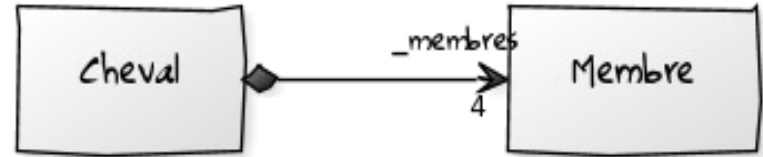
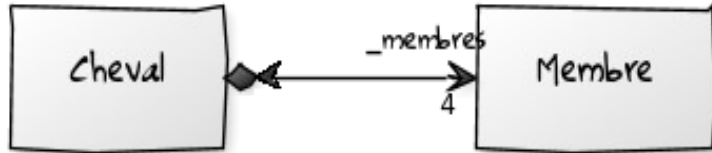
Relation	Exemple
Corps / Portion	<i>Corps / Tête</i>
Matière / Substance	<i>Eau / Hydrogène</i>
Activité / Phase	<i>Achat / Paiement</i>

- Durée de vie
 - Si le composite est détruit (ou copié), les composants le sont aussi.
- Exclusivité
 - Une classe ne peut être le composant que d'une classe composite.
- Conséquences visibles sur le code :
 - Le composé crée et détruit les instances du composant dans ses méthodes.

Association de type composition

■ Notation UML

- *Un losange plein du côté du composite.*
- L'objet ne peut pas être partagé.



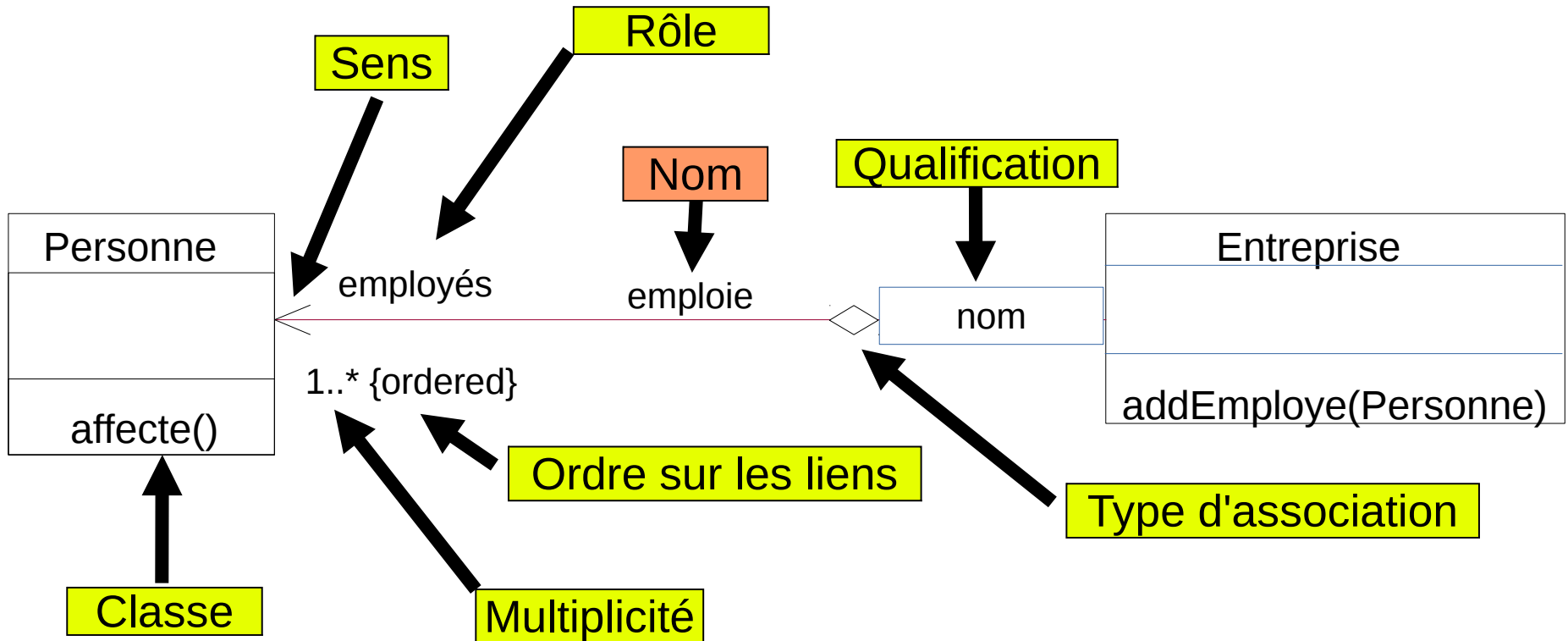
■ Notation Java : tableau, liste, vecteur ...

- Typiquement, le composé est une classe interne du composite.

```
class Cheval {
    Membre[] _membres = new Membre[4];
    Cheval ( ) {
        for (int i = 0; i < _membres.length; i++) {
            _membres[i] = new Membre();
        }
    }
}
class Membre { }
```

Résumé de la notation

34



- Importance des décorations d'une association :
 - Forte implication forte dans le code : construction, destruction et choix de la structure de données pour représenter l'association.

Les associations en Java

- Association avec une multiplicité connue a priori.

- Multiplicité 0 ou 1 : une référence.

```
class Personne {  
    Entreprise employeur;  
}
```

- Multiplicité de N : un tableau de N références.

```
class Entreprise {  
    Personne[10] employes;  
}
```

- Association normale avec une multiplicité de '*'.

- Vecteur ou liste : `ArrayList` ou `LinkedList`

- Association ordonnée.

- Ensemble ou tableau associatif ordonnés : `TreeSet` ou `TreeMap`

- Association qualifiée.

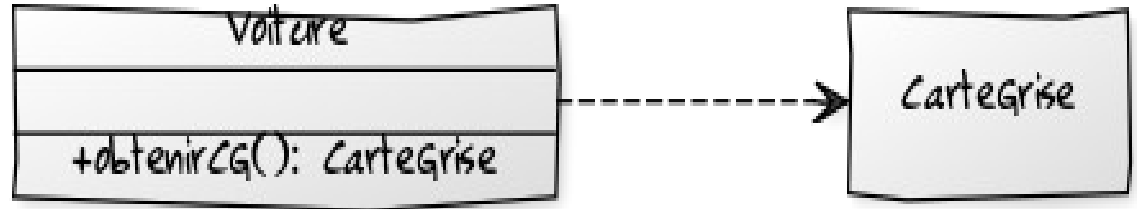
- Tableau associatif (table de hachage) : `HashMap`

Dépendance

- Relation lâche entre objets qui n'est **pas** de nature **structurelle**.
 - Marquer un lien dynamique.
 - ▶ p. ex. paramètre ou valeur de retour ou variable locale d'une classe.

- Notation UML

- Une flèche pointillée.



- Remarque

- À noter sur le diagramme que si cela est réellement nécessaire.

- Notation Java

```
class Voiture {
    CarteGrise obtenirCG() {
        cg = new CarteGrise();
        cg.setImmatriculation("AB-123-CD");
        return cg;
    }
}
```

Plan du chapitre

1

Le paradigme
objet

2

Les objets

3

Les classes

4

Associations
entre
classes

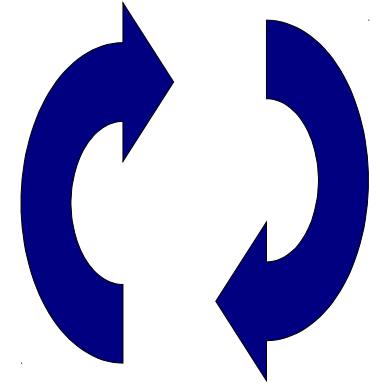
5

Héritage
et
polymorphisme

Généralisation et spécialisation

- Deux points de vue menant au mécanisme d'héritage.
 - **Généralisation**
 - ▶ Factoriser les éléments communs d'un ensemble de classes.
 - ▶ Une super-classe est une abstraction de ses sous-classes.
 - **Spécialisation**
 - ▶ Étendre les méthodes et les attributs.
 - ▶ Une sous-classe est un cas particulier de sa super-classe.

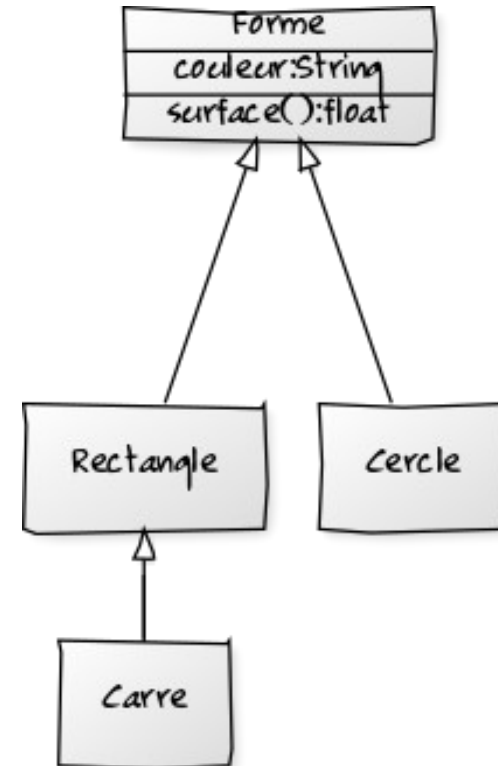
Généraliser



Spécialiser

Héritage

- Relation très forte et **statique**.
 - Permet de construire une nouvelle classe à partir de classes existantes.
 - La classe dérivée contient par héritage tous les attributs et toutes les méthodes de la classe parent.
- Notation UML
 - Une flèche triangulaire anonyme.
- Vocabulaire
 - Super-classe (*super-class*) ou classe de base.
 - Sous-classe (*subclass*) ou classe dérivée (*derived class*).



Notation Java

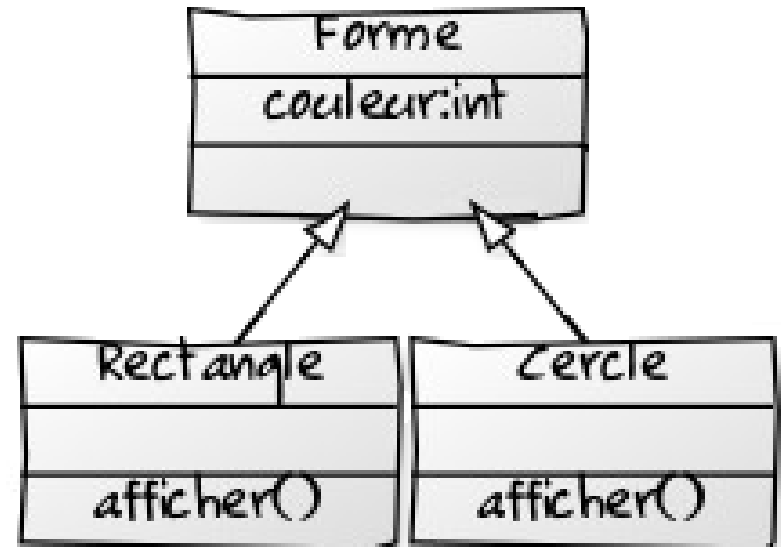
- L'héritage utilise le mot clé 'extends'.

```
class Forme {
    int couleur;
}

class Cercle extends Forme {
    int rayon
    void afficher() { ... }
}

class Rectangle extends Forme {
    void afficher() { ... }
}

static void main(String args[]) {
    Cercle cercle = new Cercle();
    cercle.couleur = 1;
    cercle.afficher();
}
```

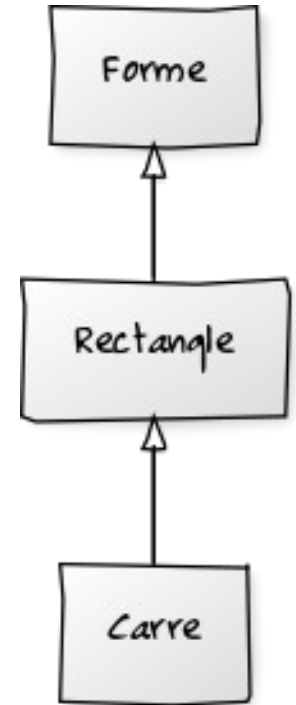


Transtypage

■ Sur-classement : **Upcasting**

- Un objet d'une classe dérivée peut se faire passer pour un objet de sa super-classe.
- La liste de ses services utilisables s'en trouve réduite à celle de la super-classe.
- Le sur-classement consiste à référencer un objet d'une classe dérivée B héritant d'une classe A dans une variable de type A.

```
Forme c1 = new Carre();  
//accès aux seuls services de Forme  
Rectangle c2 = new Carre();  
// accès aux services de Forme et Rectangle  
Carre c3 = new Carre();  
// accès aux services de Forme, Rectangle et Carre
```

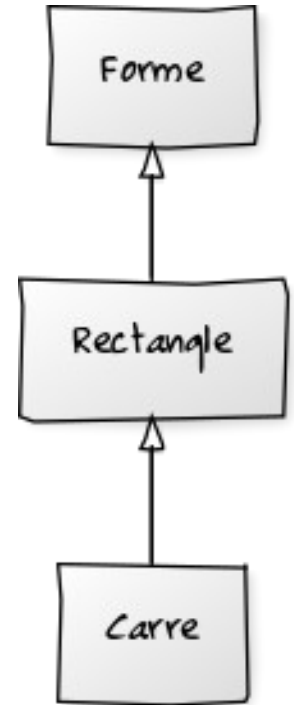


Transtypage

■ Déclassement : **Downcasting**

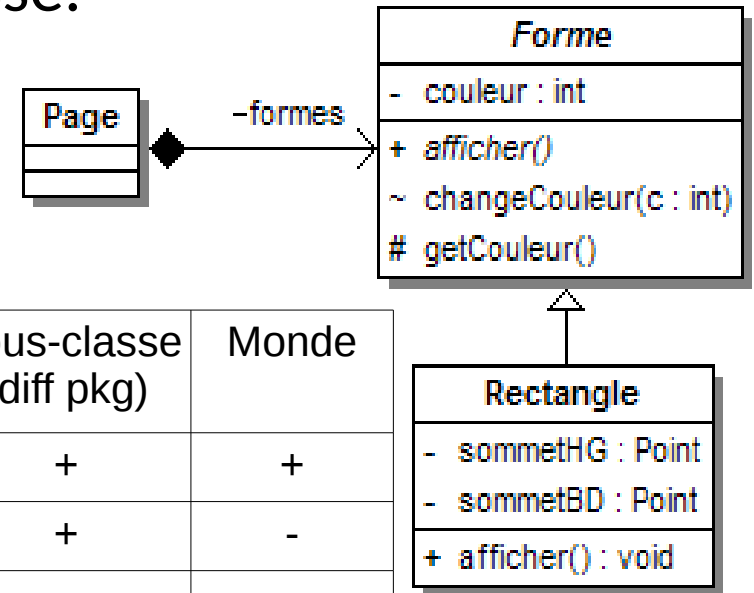
- Le déclassement est le fait de convertir une référence sur une classe de base A vers une de ses classes dérivées B.
- Un objet référencé comme un objet de la super-classe est ainsi promu en tant qu'objet de la classe dérivée.
- Cette opération n'est possible que si l'objet est effectivement du type de la classe dérivée.

```
Forme r = new Rectangle();  
Rectangle r1 = (Rectangle)r; // OK  
Carre c = (Carre)r; // Erreur à l'exécution
```



Visibilité des membres

- Restreindre l'accès aux membres de la classe.
 - Attributs
 - Méthodes
 - Associations



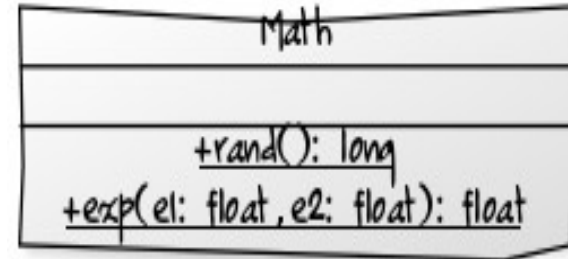
	Notation UML	Classe	Package	Sous-classe (même pkg)	Sous-classe (diff pkg)	Monde
public	+	+	+	+	+	+
protected	#	+	+	+	+	-
package	~	+	+	+	-	-
private	-	+	-	-	-	-

La visibilité des membres

- Remarques :
 - Un objet d'une classe peut accéder aux membres privés d'un objet de la même classe.
 - Les classes dérivées **ne peuvent pas diminuer la visibilité** d'une méthode redéfinie.
 - ▶ Contrainte due à la compilation qui n'a pas moyen de vérifier le respect d'une visibilité plus restrictive.
 - Pour des questions de sécurité et de maintenance de code, il est important de limiter le plus possible la visibilité des membres.
 - ▶ La visibilité par défaut doit être **privée**.
 - ▶ En particulier, **tous les attributs autres que les constantes doivent toujours être privés** (cf. encapsulation).

La portée des membres

- Portée syntaxique pour les attributs
 - **Niveau instance** : valeur distincte pour chaque instance.
 - **Niveau classe** : valeur partagée par toutes les instances.
- Portée syntaxique pour les méthodes
 - Une méthode de classe ne peut utiliser que des attributs de classe.
 - Ne nécessite pas d'instance pour appeler la méthode.
- Notation UML
 - *Nom de l'attribut ou de la méthode en souligné.*
- Notation Java : le mot clé 'static'.



```
class Math {
    public static int PI = 3.141592;
    public static float exp(float e1, float e2){}
}
```

Utilisation : Math.PI; Math.exp(5.0f, 2.0f);

Classe paramétrée

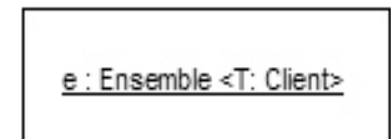
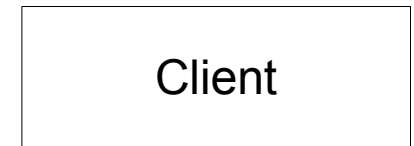
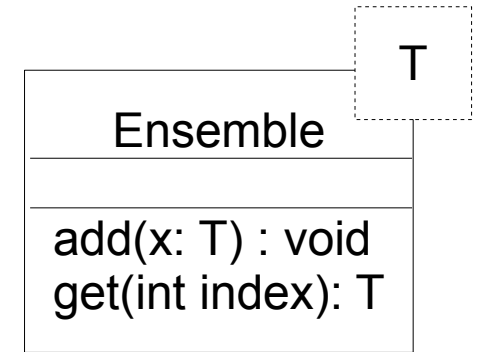
- Nommée *generics* en Java (*template* en C++).
 - Un type générique à instancier.
- Notation Java

```
public class Ensemble<T> {  
    public Ensemble() {}  
    public void add(T x) { .. }  
    public T get(int index) { .. }  
}
```

- Classe paramétrée

- Instanciation au moment de la déclaration.
- En Java :

```
Ensemble<Client> e = new Ensemble<Client>();  
e.add(new Client());  
Client c = e.get(0);  
Ensemble<Integer> s = new Ensemble<Integer>();
```



La portée des noms

- Il est possible de définir des méthodes de même nom dans des **classes** sans rapport entre elles.
 - Permet de garder une écriture locale des classes sans se soucier des conflits de nom avec d'autres classes.
 - Exemple

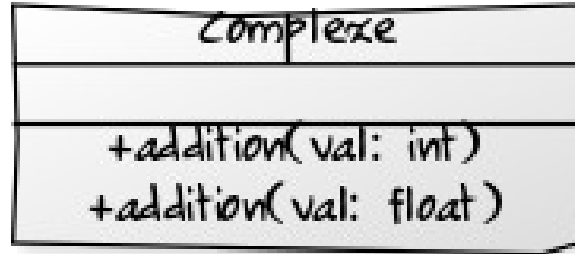


- Le choix de la méthode **est fait à la compilation**.
 - À partir de la classe de l'objet appelant.
 - Liaison statique (*early binding*)

La surcharge

- Surcharge (*overloading*)

- Dans une même portée, donner un même nom à des méthodes de signatures différentes.
- Permet de définir des procédures adaptées aux paramètres.
- Exemple :

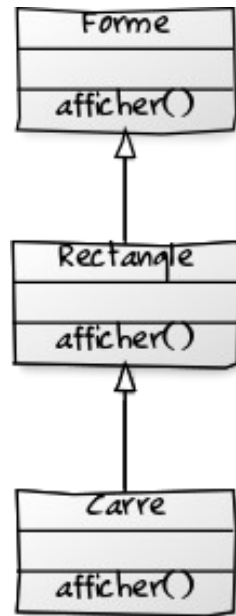


- Le choix de la méthode **est fait à la compilation**.
 - À partir de la signature (liste des paramètres effectifs).
 - Liaison statique (*early binding*).

Le polymorphisme

- Redéfinition (*overriding*)

- Dans une même hiérarchie, donner le même nom à des méthodes de même **signature**.
- Exemple :



- Le choix de la méthode **est fait à l'exécution**.

- À partir du type de l'objet appelant.
- Liaison dynamique (*late binding*).

Classe abstraite

- Une classe dont on ne peut pas créer d'instance.

- Conséquence

- Une classe abstraite doit forcément posséder des classes dérivées.

- Notation UML

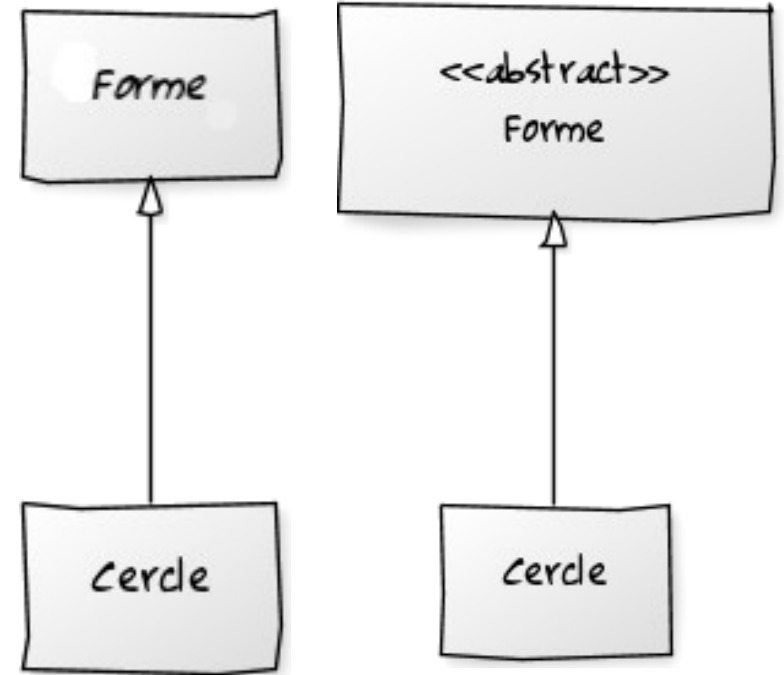
- *Nom en italique*
- *ou stéréotype <<abstract>>*

- Notation Java : le mot clé 'abstract'

```
public abstract class Forme () {  
}
```

- Si on ne peut pas créer d'instance directe, on peut créer des références de ce type par upcasting :

```
Forme f = new Cercle ();
```



Méthode abstraite

- Méthode sans implantation (sans code).
- Notation UML :
 - *Nom en italique ou stéréotype <<abstract>>*.
- Notation Java : le mot clé 'abstract'

```
abstract public void afficher();
```

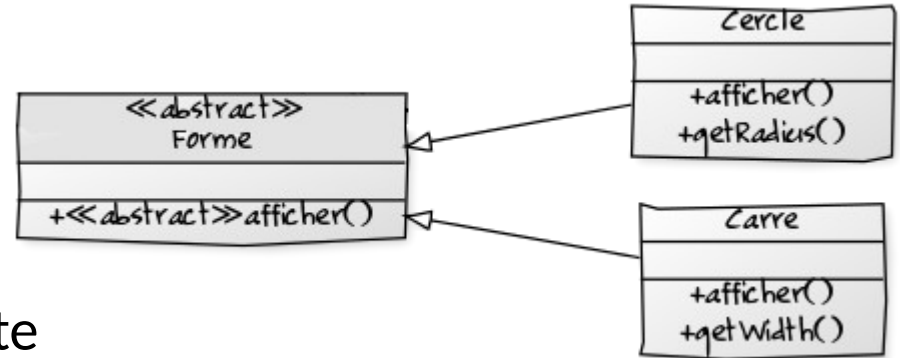
- Rôle : obliger les classes dérivées à redéfinir la méthode.

- Exemple en Java

```
Forme c1 = new Cercle();  
c1.afficher();
```

- Conséquences

- Une classe avec une méthode abstraite doit être une classe abstraite.
- Une classe abstraite peut ne contenir que des méthodes concrètes.

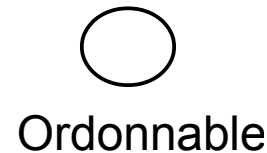
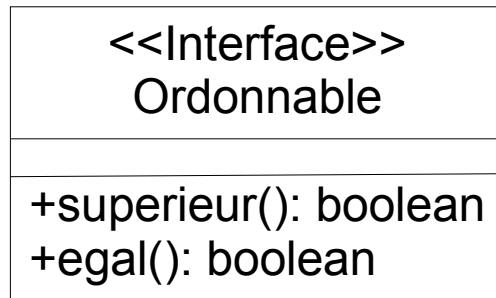


Notion de type

- Type
 - **Définit une liste de services** que doit posséder une classe pour être du type.
 - Oblige une classe de ce type à implémenter ces services.
 - Le but est de garantir que le service existe quand une classe utilise une classe de ce type.
- **Un objet ne peut appartenir qu'à une classe mais peut être de plusieurs types.**
- Exemple
 - Objets de la classe Carré.
 - et de type :
 - ▶ Imprimable (service *print()*)
 - ▶ Ordonnable (service *sort()*)
 - ▶ Affichable (service *display()*), etc

Interface

- La notion de type est implémentée en COO par une interface.
 - Une structure dont toutes les **méthodes sont abstraites**.
 - Une structure qui **ne possède pas d'attribut**.
- Une classe peut implémenter plusieurs interfaces.
- Notations UML : deux alternatives.
 - *Une classe normale stéréotypée* `<<interface>>`
 - *Un rond nommé*.



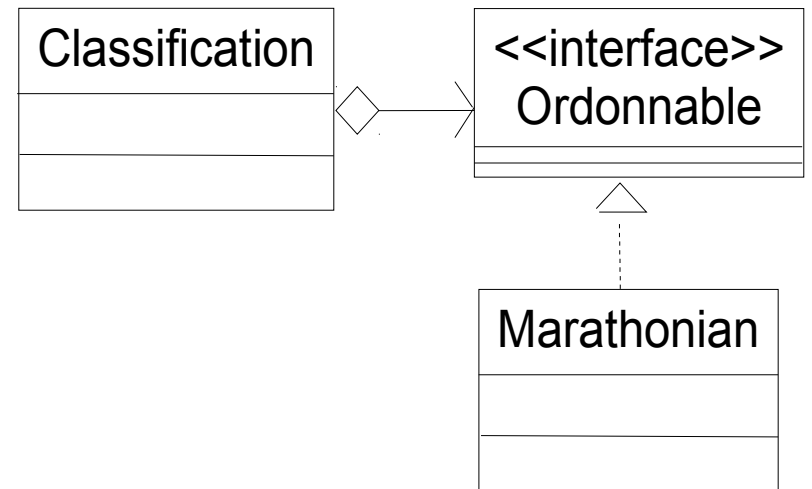
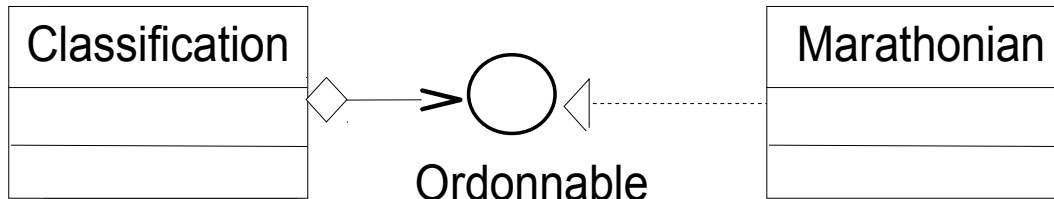
Interface

■ Relation d'implémentation

- Une classe qui implémente une interface doit définir toutes les méthodes de l'interface.
- On peut toujours définir une référence sur un type.
 - ▶ `Ordonable o = new Marathonien();`
- Par contre, on ne peut pas créer d'objet d'un type donné.
 - ▶ ~~`Ordonable o = new Ordonable();`~~

■ Notation UML

- *Flèche fermée avec trait en pointillé.*



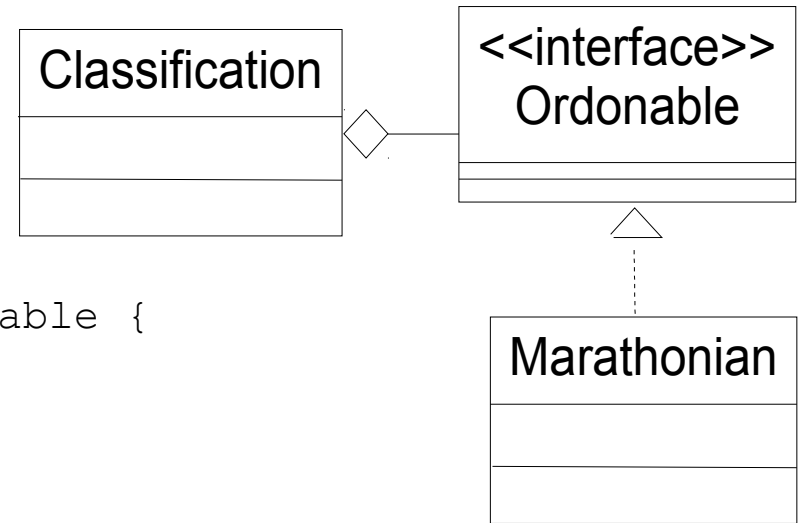
Interface

■ Notation Java :

```
public interface Ordonnable {  
    public boolean superieur();  
    public boolean egal();  
}
```

```
public class Marathonian implements Ordonnable {  
    public boolean superieur() {...}  
    public boolean egal() {...}  
    // méthodes spécifiques  
}
```

```
public class Classification {  
    ArrayList<Ordonnable> coureurs;  
    ...  
    coureurs.get(1).superieur(coureurs.get(2));  
}
```



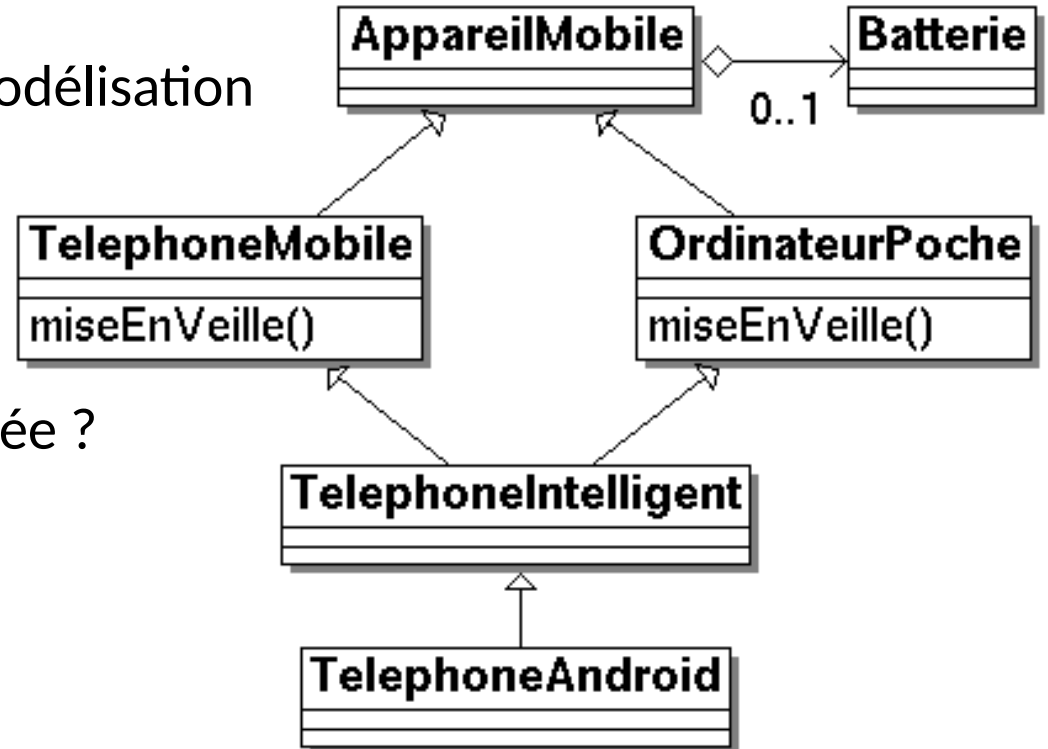
Cas de l'héritage multiple

- L'héritage multiple est possible en UML
 - Mais son emploi est déconseillé pour les difficultés qui peuvent apparaître lors de l'implémentation avec un langage de programmation.
 - Problème du diamant.

- ▶ Sur le code ci-dessous et la modélisation ci-à-droite :

```
t = new TelephoneAndroid()  
t.miseEnVeille();
```

- Combien de batterie ?
- Quelle méthode est appelée ?

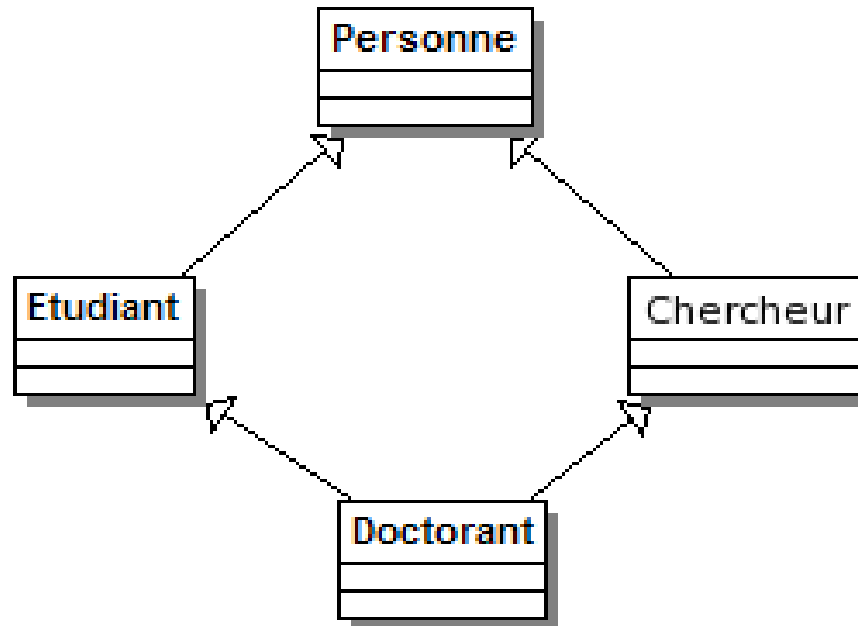


Avons nous besoin de l'héritage multiple ?

57

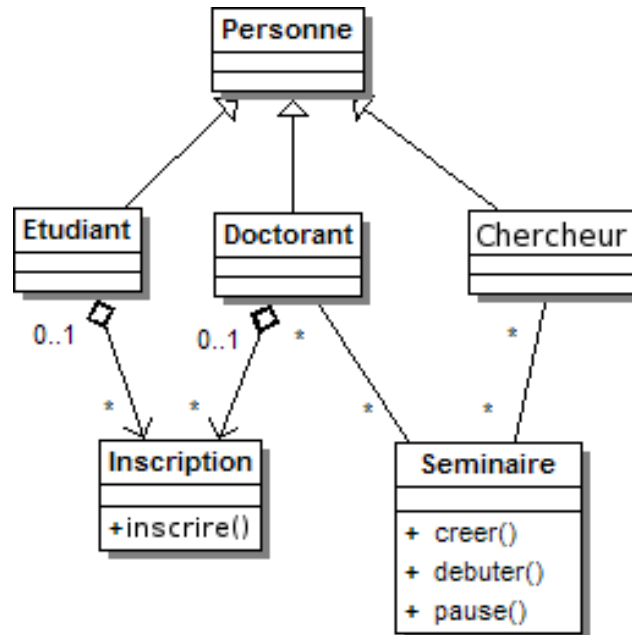
■ Héritage

- L'erreur consiste à utiliser l'héritage pour le partage de ressources communes.
- Par exemple un doctorant :
 - ▶ possède une procédure d'inscription comme un étudiant.
 - ▶ peut assister à des séminaires de recherche comme un chercheur.



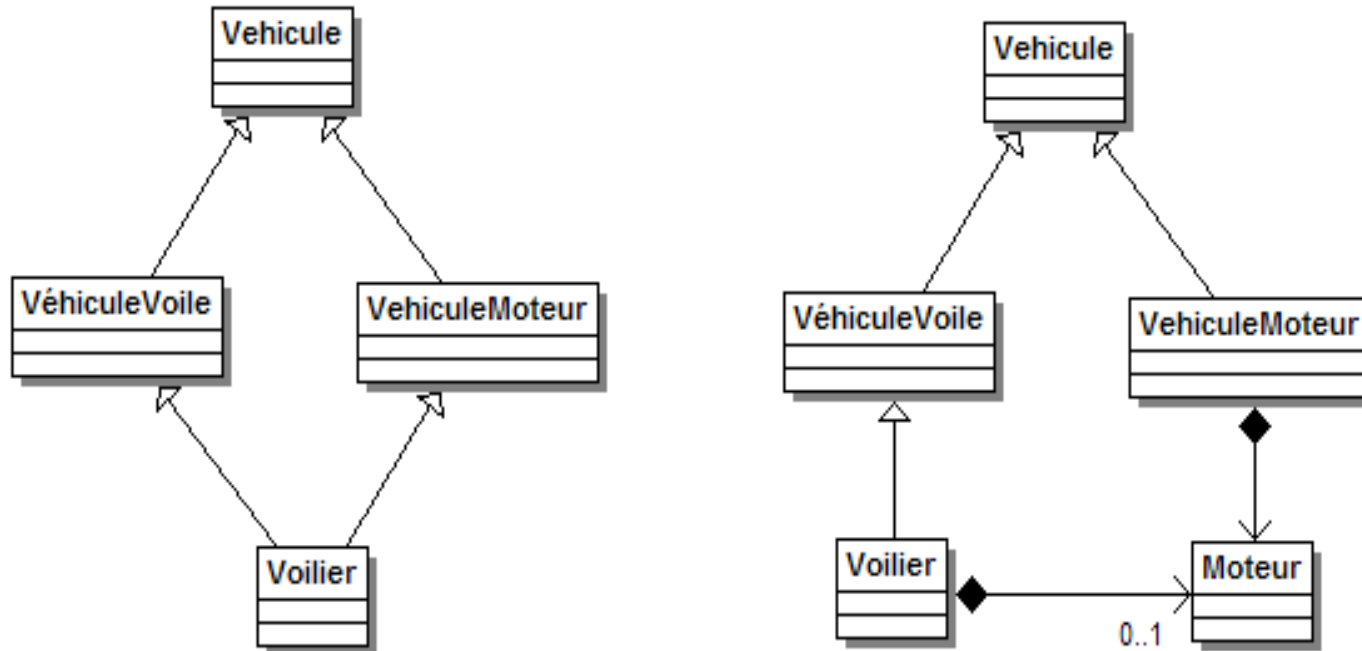
Solution

- L'héritage ne doit être utilisé que pour la catégorisation.
 - Par exemple, un doctorant n'est pas un chercheur à part entière.
- Solution
 - Faire des sous-classes strictement exclusives.
 - Utilisation d'une agrégation pour le partage de ressources (*abstraction hypostatique*).



Solution

- L'abstraction hypostatique est aussi utilisée pour le partage d'attributs.
 - Par exemple, le voilier est d'abord un véhicule à voile qui possède un moteur comme les véhicules à moteur. Le moteur est externalisé et partagé par les deux classes.



Les traits

- L'abstraction hypostatique peut facilement s'implémenter par un **trait**.
 - Un trait est une interface qui encapsule un ensemble cohérent de méthodes à caractère transverse et réutilisable.
- Implémentation Java.
 - Une interface avec une méthode par défaut (*default method*).

```
public interface Inscrivable {  
    default void inscrire( ) { ... }  
}  
public class Etudiant implements Inscrivable {  
    // peut utiliser directement inscrire() de Inscrivable.  
}  
public class Doctorant implements Inscrivable {  
    // peut utiliser directement inscrire() de Inscrivable.  
}
```

Remarque

- La notion d'héritage multiple n'est ni un concept fondateur ni même nécessaire à la conception orientée objet.
 - Tous les langages à objets n'implémentent pas l'héritage multiple (eg. notamment les langages orientés développement de logiciels les plus récents : Java, C#, Scala, Ruby).
 - L'héritage multiple est utilisé pour d'autres principes que la généralisation.
 - ▶ p. ex. implémenter la notion d'interface en C++.

Que retenir de ce chapitre ?

- Le paradigme objet définit plusieurs concepts importants :
 - Classe et objet.
 - Encapsulation.
 - Relations.
 - ▶ Association.
 - Standard, Agrégation, Composition, Dépendance.
 - ▶ Héritage et polymorphisme.
 - Classe et Type.
- Dans l'utilisation de ces concepts, le développeur doit respecter deux principes fondamentaux :
 - Restreindre le plus possible la visibilité des membres pour respecter le principe d'encapsulation.
 - Exprimer le plus de choses statiquement pour qu'elles soient vérifiables par le compilateur.