

Programmation Orientée Objet

Lamine BOUGUEROUA
Lamine.bougueroua@esigetel.fr



Bibliographie

- Brad J. Cox, Andrew J. Novobilski (1986)
« *Object-Oriented Programming: An Evolutionary Approach* », [ISBN 0201548348](#).
 - Bertrand Meyer (2000)
« *Conception et programmation orientées objet* », [ISBN 2-212-09111-7](#).
 - Grady Booch, James Rumbaugh, Ivar Jacobson (2000)
« *Le processus unifié de développement logiciel* » [ISBN 2-212-09142-7](#).
 - G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre (1997)
« *Les langages à Objets* » [ISBN 2729602755](#)
 - De Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1999)
« *Design Patterns* » [ISBN 2-7117-8644-7](#)
-



Caractéristiques d'un logiciel

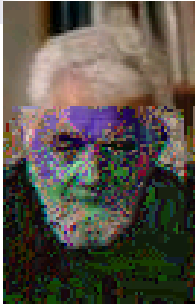




Langage de programmation (1)

- Un langage de programmation est un langage de communication qui permet à un utilisateur de dialoguer avec une machine
- Il existe des milliers de langages de programmation
- Les langages peuvent être classés en fonction de plusieurs critères:
 - Généraliste / Spécialisé
 - Bas niveau / Haut niveau
 - Compilé / interprété
 - Avec / sans gestion de la mémoire automatique
 -

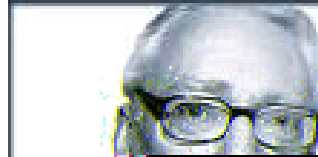
Langage de programmation (2)



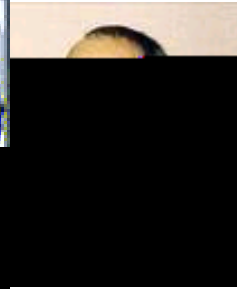
John McCarthy

John Backus

Kristen Nygaard



Ole-Johan Dahl



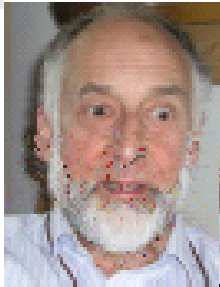
Jean Ichbiah

Niklaus Wirth

Alan Kay

- 1954 : **Fortran**, premier langage de programmation.
- 1958 : **LISP**, langage fonctionnel de **John McCarthy**.
- 1958 : **Algol** amélioration de Fortan (Equipe **Backus**).
- 1967 : **Simula**, langage orienté objet de **Ole-Johan Dahl** et **Kristen Nygaard**.
- 1970 : **PASCAL**, Niklaus Wirth.
- 1976 : **Smalltalk-72** et **Smalltalk-80**, langage orienté objet (**Alan Kay**).
- 1977-83 : **ADA**, premier langage des systèmes embarqués (**Jean Ichbiah**).

Langage de programmation (3)



Robin Milner



Ken Thompson–Dennis Ritchie



Bjarne Stroustrup



Bertrand Meyer



James Gosling



Anders Hejlsberg

- 1972 : **C**, langage de **Dennis Ritchie** et **Ken Thompson**.
- 1980 : **ML (Meta Language)**, langage fonctionnel basé sur une théorie des types (**Robin Milner**).
- 1980 : **C++** (C with classes) par **Bjarne Stroustrup**.
- 1985 : **Eiffel**, par **Bertrand Meyer** and co.
- 1996 : **Java**, (**James Gosling** and co.)
- 1996 : **C#** par **Anders Hejlsberg**
-



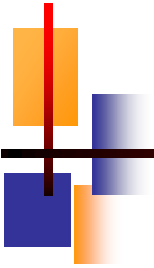
Classification des langages (2)

- Langages spécialisés pour la communication avec des bases de données
 - SQL, SAS, 4GL,...
- Langages pour les technologies WEB
 - Perl, PHP, JavaScript, Python,....
- Langage avec balises
 - HTML, XML,...
- Langages de programmation théorique
 - Machine de Turing,...
- Langages de définition de données
 - XML Schéma,...
- Langages de commandes
 - Shell,...



Classification des langages (3)

- Les langages de programmation sont classés aussi par générations:
 - 1^{ère} génération : langage binaires
 - 2^{ème} génération : langage assembleur
 - 3^{ème} génération : procéduraux et orienté objet
 - 4^{ème} génération : souvent bases de données

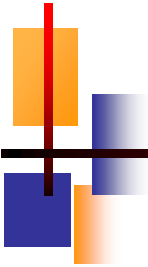


Programmation orientée objet



La programmation objet

- Les notions d'objets sont apparues ultérieurement, de la nécessité de structurer encore plus les programmes informatiques, au fur et à mesure que leur complexité augmentait
- Les langages à objets permettent en effet de décomposer un problème en un certain nombre d'entités indépendantes les unes des autres, qui représentent chacune un intervenant dans le système à gérer
- Ces entités, que l'on appelle communément les objets, sont capables de se gérer elles-mêmes et de communiquer avec les autres pour leur offrir un certain nombre de services



Concepts de base - Plan

- Nouvelle terminologie
- Qu'est-ce qu'un objet?
- Représentation graphique
- Comment utiliser un objet?
- Encapsulation
- Relations entre objets
- Héritage
- Polymorphisme



Nouvelle terminologie

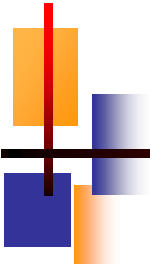
- Dans le modèle à objet on voit apparaître une nouvelle terminologie:
 - Attribut
 - Méthode
 - Classe, Objet, instance
 - Encapsulation
 - Abstraction
 - Héritage
 - Agrégation
 - ...
- Nombre de ses termes sont plus ou moins utilisés dans d'autres langages non objets précédent
- Les langages à objets, quant à eux, les regroupent tous

Qu'est-ce qu'un objet?

- Nous sommes entourés d'objets dans le monde réel : une table, une voiture, une télévision,....



- Le concept d'objet en informatique a pour but de reprendre cette idée de structuration en objet au sein des programmes informatiques.



Qu'est-ce qu'un objet?

- Les objets du monde réel partagent trois caractéristiques fondamentales:
 - Un état
 - Un comportement
 - Et une interface avec le monde qui les entoures

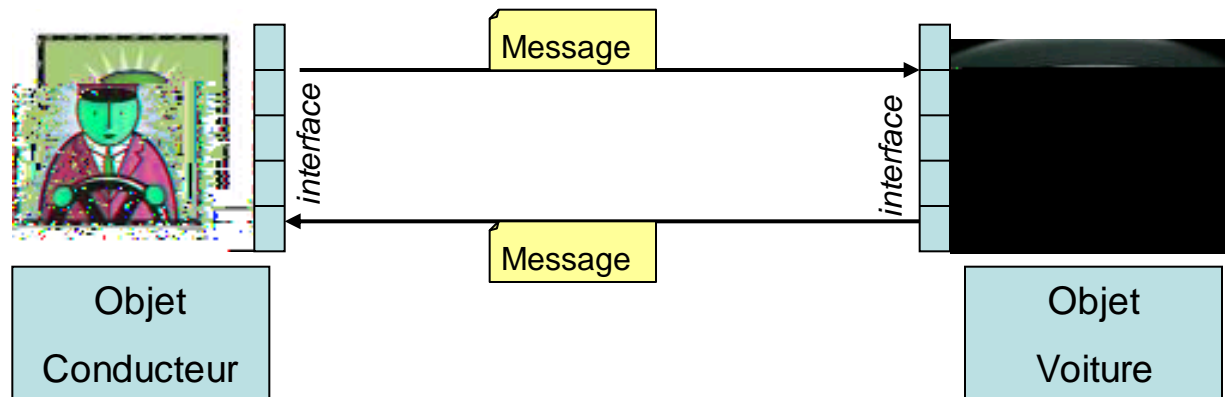


Qu'est-ce qu'un objet?

- Les tables ont :
 - Un état : nombre de pieds, couleur, forme,
 - Mais pas de comportement.
- Les voitures ont :
 - Un état : nombre de roues, nombre de portes, vitesse courante,...
 - Un comportement : accélérer, freiner, changer de vitesse,...
 - Interface : le conducteur qui interagit avec la voiture peut accélérer,...
 - En revanche, l'injection d'essence dans le moteur est un comportement interne de la voiture. Il ne fait pas parti de l'interface.
- Les êtres vivants!!!!

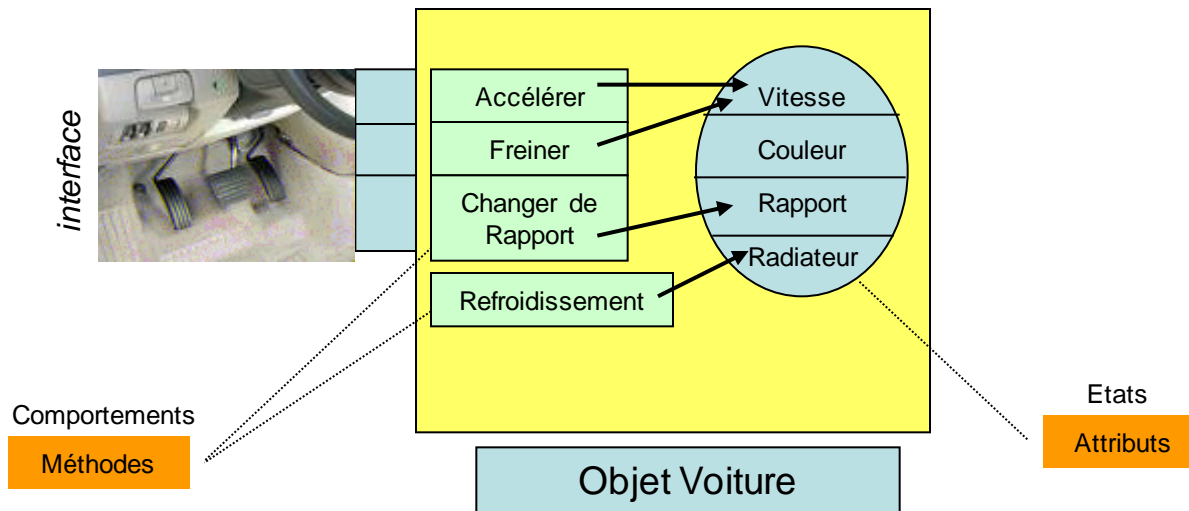
Qu'est-ce qu'un objet?

- Les objets ne sont pas complètement isolés, ils communiquent entre eux par l'envoi de messages (appels) en invoquant les services des autres objets.



Qu'est-ce qu'un objet?

- En POO la séparation entre données et procédures disparaît
 - On retrouve les données (attributs) et les procédures (méthodes) dans la même structure.



- Nous distinguons deux types de comportements : internes et externes.



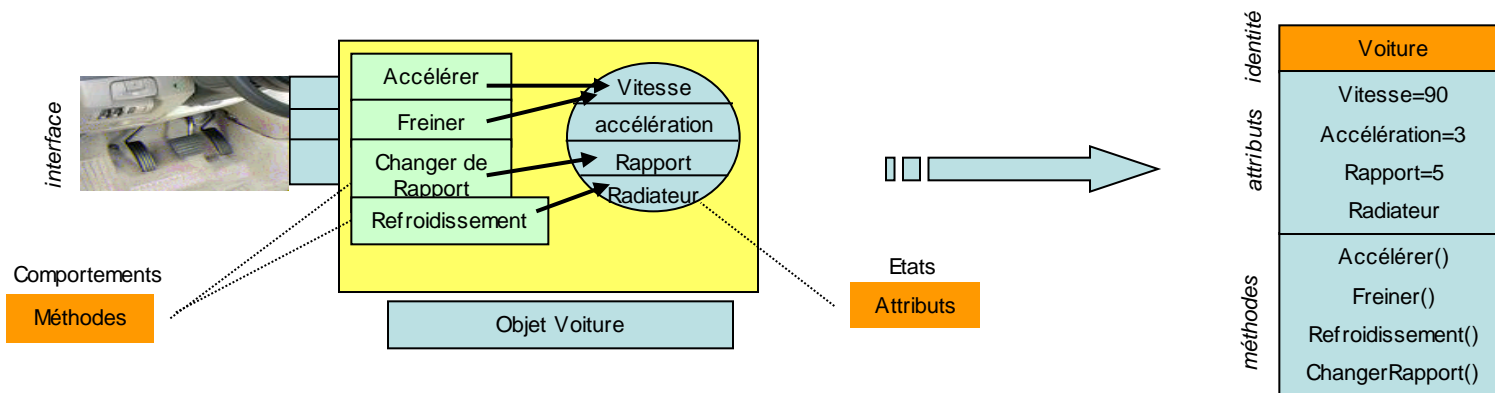
Qu'est-ce qu'un objet?

- Un objet représente une entité du monde réel, ou du monde virtuel dans le cas d'objets immatériels, qui se caractérisent par une **identité**
- Un objet informatique maintient son **état** dans une ou plusieurs variables (attributs). Une variable est un élément contenant la valeur caractéristique de l'état et identifié par un nom.
- Un objet informatique met en œuvre son **comportement** à l'aide de méthodes. Une méthode s'applique sur les variables d'un objet.
- En résumé :

Objet = identité + état + comportement

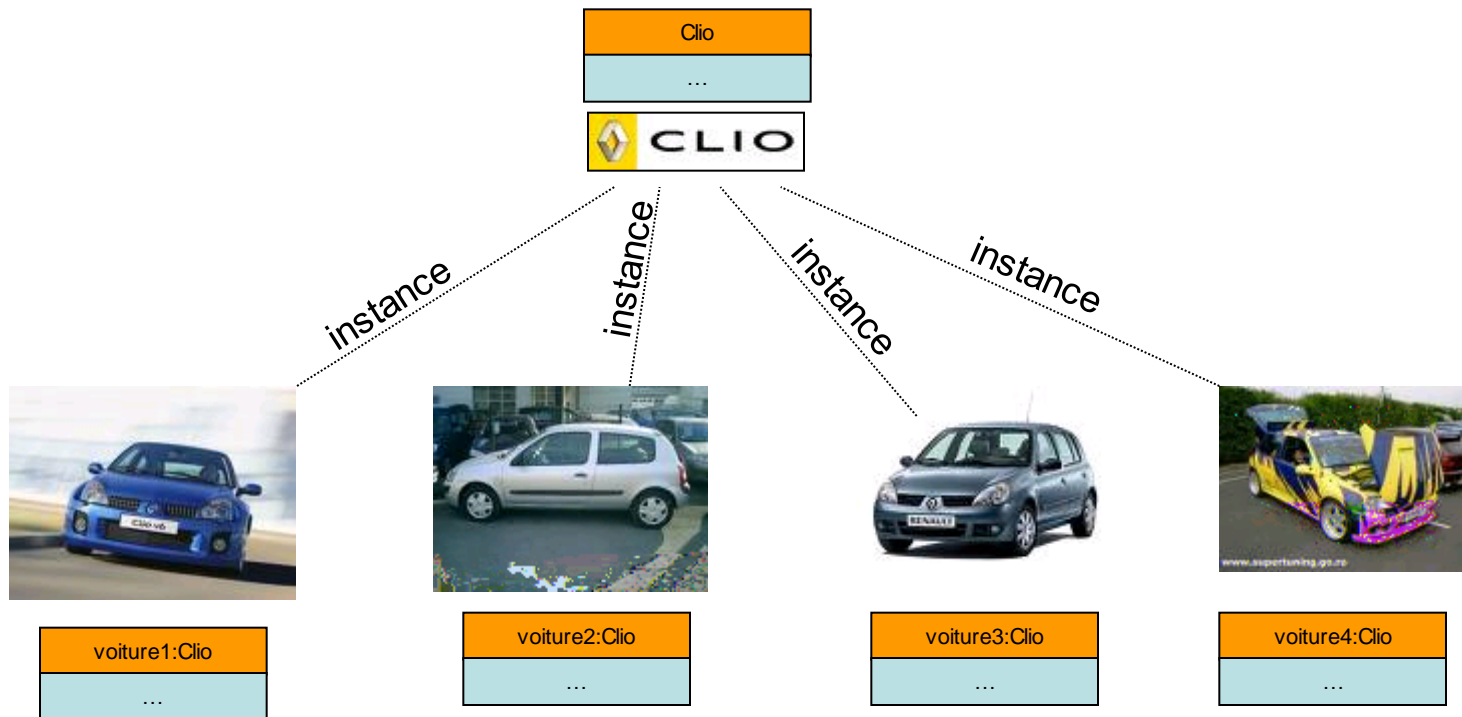
Représentation graphique

- On veut modéliser une voiture par un objet informatique. La voiture est en marche : sa vitesse est égale à 90km/h, avec une accélération de 3 m/s² et un rapport de transmission en 5^{ème} vitesse.
- En plus, le conducteur peut accélérer, freiner ou changer de vitesse.



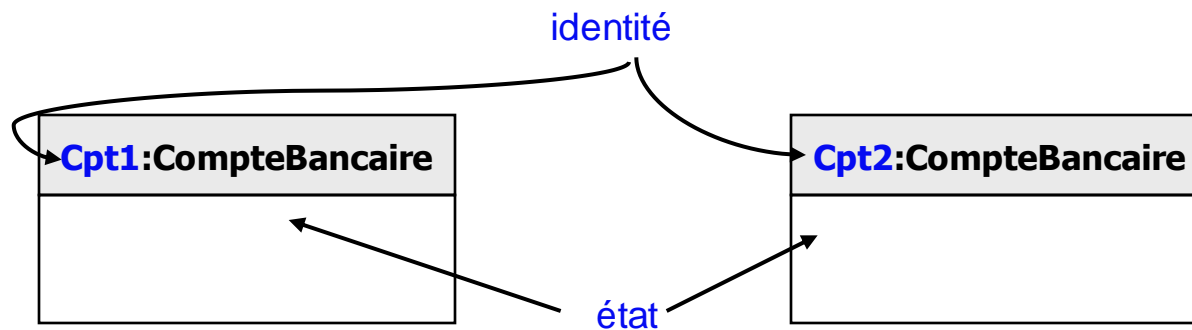
La notion d'instance

- Dans le monde réel, nous avons souvent plusieurs objets du même type (le même modèle de voiture que son voisin par exemple une Clio).
 - **En terminologie objet** : on dit que votre voiture est une **instance** de la classe des objets Voiture Clio.



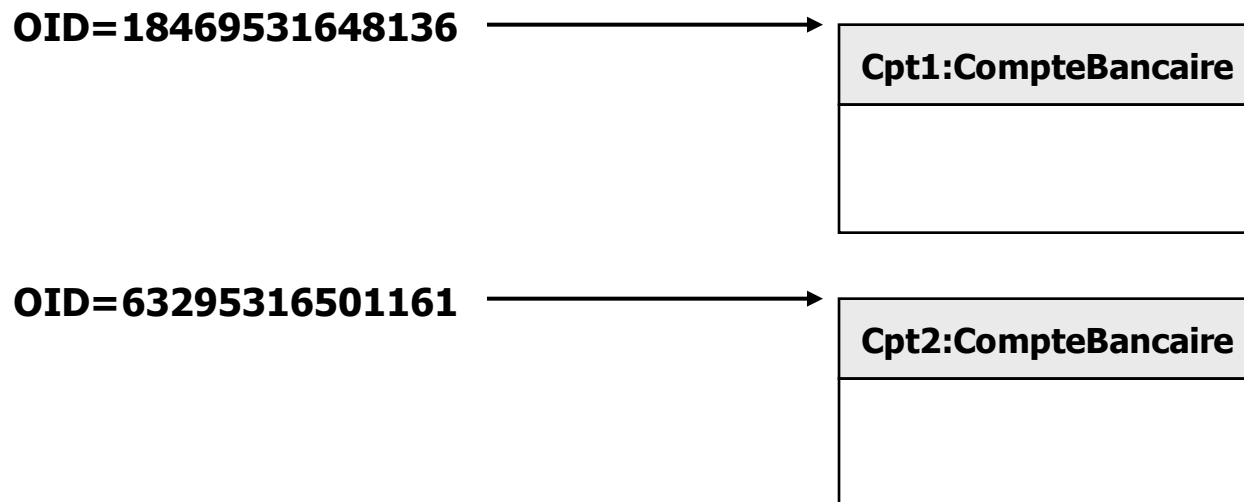
Identité d'un objet (1/2)

- L'*identité* d'un objet permet de distinguer les objets les uns par rapport aux autres
- Pour le programmeur *les noms des objets* permettent de les distinguer:
 - voiture1, voiture2 et voiture3
- L'identité d'un objet est indépendante de son état (ici c'est 1000)



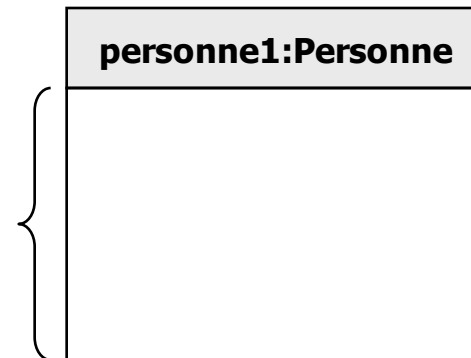
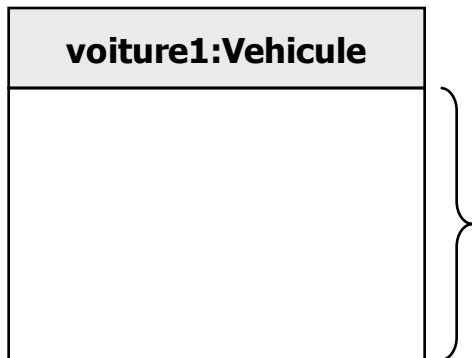
Identité d'un objet (2/2)

- Pour le système l'identité est représentée par un identifiant (Object Identifier ou **OID**) unique et invariant qui permet de référencer l'objet indépendamment des autres objets
 - L'OID est affecté automatiquement par le système à l'objet lors de sa création
- Exemple:



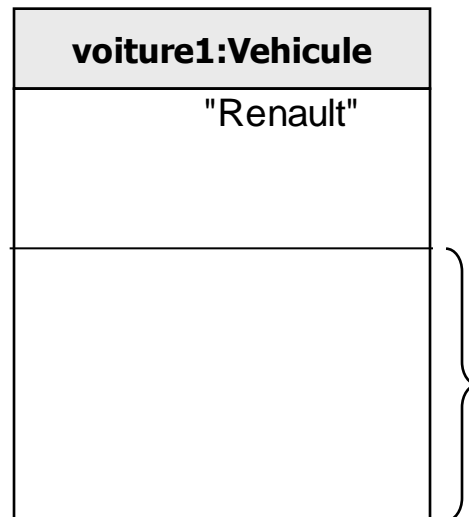
Etat d'un objet

- L'*état* d'un objet correspond aux valeurs de tous ses attributs à un instant donné
- La notion du temps est très importante si l'objet dispose d'un état dynamique
- **Exemple:**
 - L'état de l'objet voiture1 englobe les propriétés statiques (marque, puissance, vitesseMax) et dynamiques (vitesseCourante)
 - L'état de voiture1 change suivant la valeur de la vitesse courante



Comportement d'un objet

- Le *comportement* d'un objet se définit par l'ensemble des actions qu'il effectue et les réactions qu'il manifeste suite aux messages envoyés
 - un message = demande d'exécution d'une opération
 - une action est une opération qu'un objet effectue sur un autre pour provoquer une réaction





La notion de classe (1/2)

Définition : La classe est un **prototype** qui définit les variables (même ou différents états) et les méthodes communs à tous les objets d'un même type.

Autrement dit :

- Une classe décrit la structure et le comportement des objets
 - C'est la représentation statique de l'objet avant sa création
 - C'est un **moule** qui nous permet de créer des objets
 - On peut créer plusieurs objets différents d'une même classe
 - C'est le « **négatif** » **du film** qui nous permet de tirer la photo
- Comme pour l'objet la classe est caractérisée par trois aspects:

Classe = instantiation + attributs + [méthodes]

La notion de classe (2/2)

Exemple :

```
class Livre{
```

```
.....
```

```
}
```

Classe Livre
-titre, auteur

Classe Journal
-titre

Classe Employé
-nom, prénom, statut

Classe Lecteur
-nom, prénom



Germinal
E. Zola



Le seigneur des anneaux
J.R.R. Tolkien



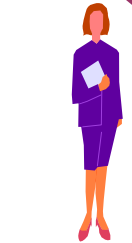
Le Monde



Michel Martin
Bibliothécaire



Alice Dupont
Directrice



Anne Durand

Arsène Deschamps



Les attributs

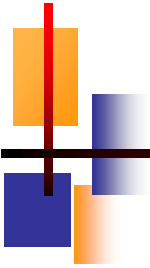
- Les attributs représentent les caractéristiques de l'objet
 - Elle sont définis dans la classe mais sont réellement créés lors de la création de l'objet
 - On retrouve exactement la même notion de variables dans la programmation procédurale (le langage C par exemple)
 - Les attributs peuvent être des variables de type simple (entier, réel, constante..) ou même des objets
- Attribut de classe?
 - Cet attribut contient des informations qui sont partagées par l'ensemble des objets de la classe (exemple : nombre de roues d'une voiture)
 - Ce qui est équivalent aux variables globales dans les langages procéduraux (langage C)



Les attributs

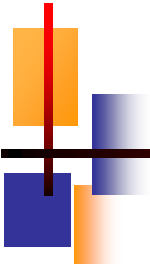
- Exemple en Java:

```
class Compte{  
    private int solde;  
    String nom;                // attribut  
    String static banque="BNP"; //attribut de classe  
    .....  
}  
  
void main(String args[]){  
    Compte C1=new Compte();  
    C1.banque;                 //affiche BNP  
    Compte.banque;             //affiche BNP  
}
```



Les méthodes

- Qu'est-ce qu'une méthode?
 - Les méthodes sont les actions, tâches ou opérations que l'objet peut exécuter
 - Elles agissent généralement sur les attributs de cet objet
- L'exécution d'une méthode peut conduire à :
 - La création ou la destruction d'un objet
 - La modification de l'état de l'objet (exemple : les **accesseurs**)
 - L'envoi d'un message à un autre objet
- Une méthode peut avoir des arguments ou non.



Les méthodes

- Qu'est-ce qu'une méthode de classe?
 - Une méthode de classe est une méthode que nous pouvons invoquer directement sans passer par l'intermédiaire d'un objet.
 - Une méthode de classe ne peut lire ou modifier que **les variables de classe**

- Remarque :
 - Une méthode de classe peut être invoquée en envoyant un message soit à la classe qui l'a définie, soit à n'importe quelle instance de cette classe.



Les méthodes

■ Exemple :

```
class Compte{  
    private int solde;  
    String nom;  
    String static banque="BNP"; //attribut de classe  
  
    void debiter(int montant){    // méthode  
        solde = solde - montant;  
    }  
    static void getName(){        // méthode de classe  
        // affiche la variable banque  
        System.out.println(banque);  
    }  
}
```



Les méthodes

- Les méthodes publiques sont communément appelées **accesseurs** : elles permettent d'accéder aux champs d'ordre *privé*.
- Il existe des accesseurs en *lecture*, destinés à récupérer la valeur d'un champ;
- et des accesseurs en *écriture* destinés pour leur part à la modification d'un champ.
- En général : accesseurs / modificateurs, getter / setter



Les méthodes

- Il n'est pas nécessaire d'avoir un accesseur par champ privé, car ceux-ci peuvent n'être utilisés qu'à des fins internes.
- Très souvent, les accesseurs en *lecture* verront leur nom commencer par *Get*
- quand leurs homologues en *écriture* verront le leur commencer par *Set* ou *Put*



Les méthodes

- Exemple en Java:

```
class CompteBNP{  
    private int solde;  
    String nom;  
    String banque="BNP"; //attribut de classe  
  
    int getsolde() { //accesseur - lecture  
        return solde;}  
  
    int setsolde(int montant){ //modificateur - écriture  
        solde = montant;}  
  
    void afficheBanque() { //méthode de classe  
        System.out.println(banque);}  
}
```



La création d'un objet

- La création d'un objet se fait à partir d'une classe (prototype)
- La création d'un objet se déroule en 3 étapes :
 - **La déclaration** : avant de créer un objet, il est nécessaire de déclarer une variable de type objet.
 - **L'instanciation** : l'objet est créé et un espace mémoire est alloué pour le stockage de l'objet. L'état de l'objet est à ce moment inconnu ou indéterminé; ses attributs ne sont pas initialisés.
 - **L'initialisation** : après la création de l'objet, son état est indéterminé. Il existe une fonction particulière appelée constructeur qui a pour objectif d'initialiser l'état de l'objet. Cette opération consiste à affecter des valeurs aux attributs de l'objet créé.



La déclaration d'un objet

- En programmation objet, les classes sont considérées comme des types à part entière.
- Déclarer un objet revient donc simplement à déclarer une variable ou une constante, dont le type est l'identificateur de la classe.
- La syntaxe est donc :
 <identificateur de classe> <identificateur d'instance>
 type *variable*
- Exemple :
 - `Personne P1; //déclare une instance nommée P1 de la classe Personne`
 - `Véhicule V1; //déclare une instance nommée V1 de la classe Véhicule`



L'initialisation d'un objet

- Il est important qu'un objet soit doté d'un mécanisme d'initialisation efficace.
- Un moyen simple est de définir une méthode dédiée à cette tâche.

- Exemple :

```
class véhicule {  
    int puissance;  
    int nombre_roue;  
  
    void init(int p){                // méthode initialisation  
        puissance =p;  
        nombre_roue=4;  
    }  
}
```

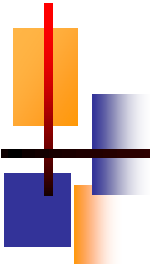


L'initialisation d'un objet

- Remarque :

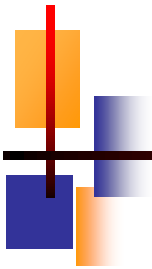
Les langages comme C++ et Java fournissent un mécanisme d'initialisation des instances, par le biais d'une méthode particulière : *le constructeur*.

- Chaque classe implémente par défaut deux méthodes particulières indispensables permettant la *création* puis la *destruction* de l'objet à la fin de son utilisation



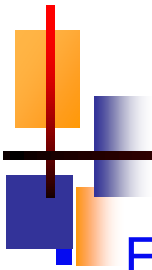
Construction & destruction d'objet

- Le constructeur de l'objet:
 - Invoqué lors de l'instanciation de l'objet par l'opérateur « *new* » (dans la plupart des langages sauf CLOS qui utilise make-instance)
 - Une opération qui crée l'objet (allocation de l'espace mémoire) et initialise ses états (attributs)
 - La méthode invoqué lors de la création peu être redéfini par le programmeur pour effectuer d'autres initialisations
 - Porte le *même nom* que la classe dans laquelle il est défini.
 - Un constructeur n'a pas *de type de retour* (même pas un void)
 - Un constructeur peut avoir des arguments



Construction & destruction d'objet

- Un objet peut avoir **plusieurs constructeurs**
- La définition d'un constructeur n'est pas obligatoire lorsqu'il n'est pas nécessaire.
- Dans ce cas, c'est le compilateur qui se charge de créer de manière statique les liens entre champs et méthodes.
- Chaque classe possède un constructeur par défaut.



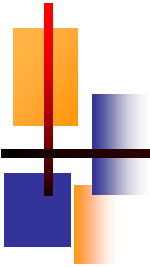
Construction & destruction d'objet

Exemple :

```
class Personne{  
    // constructeurs  
    Personne() {...}  
    Personne(String nom, String prenom) {...}  
    Personne(String nom, int age) {...}  
    Faux --> Personne(String nom, String adresse) {...}  
    Faux --> int Personne(String nom) {...}  
}
```

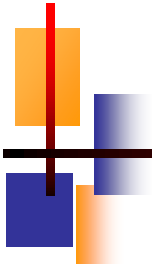
//appel du constructeur

```
Personne p1= new Personne("Martin","Pierre");
```



Construction & destruction d'objet

- La destruction d'un objet se fait par une méthode particulière : le «*destructeur*» de l'objet.
 - Le destructeur est une opération permettant la destruction de l'objet et la récupération de son espace mémoire
 - La destruction peut être manuelle en appelant une méthode: *delete()*, *release()* ou *free()* (selon les langages)
 - Ou automatique grâce à un ramasse-miette (*garbage collector*) dans le cas du langage Java



Construction & destruction d'objet

- La destruction d'un objet – suite :
 - La méthode qui permet de détruire l'objet peut être réécrite pour rajouter un traitement particulier avant sa destruction
 - Un destructeur ne peut pas avoir d'argument
 - La définition d'un destructeur n'est pas obligatoire lorsque celui-ci n'est pas nécessaire



Destruction d'objet

- Exemple dans le langage C++:
 - La syntaxe de déclaration d'un destructeur en C++, pour une classe nommée Voiture, est :

```
~Voiture() {  
    //opérations  
}
```
 - Le destructeur est une méthode sans argument, même nom que celui de la classe, préfixé du signe «~» (tilde)
 - Toute classe admet donc exactement un destructeur; s'il n'est pas défini par le programmeur, c'est le compilateur qui se chargera d'en générer une version minimaliste.



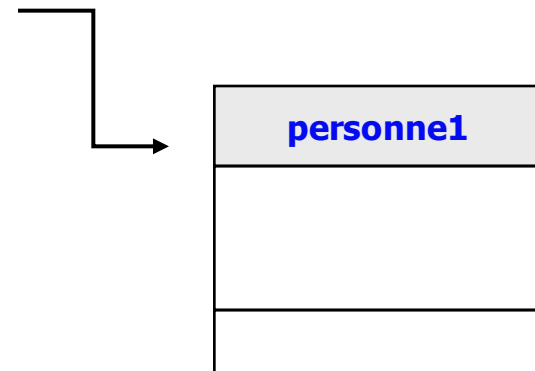
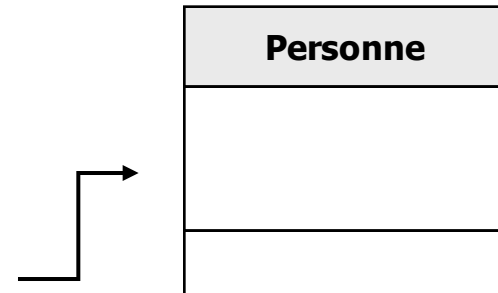
Destruction d'objet

- Tout comme pour les constructeurs, un objet peut **ne pas avoir de destructeur**.
- Une fois encore, c'est le compilateur qui se chargera de la destruction de l'objet.
- Un objet peut posséder **plusieurs destructeurs**. Leur rôle commun reste identique, mais peut s'y ajouter la destruction de certaines variables internes pouvant différer d'un destructeur à l'autre.
- La plupart du temps, à un constructeur distinct est associé un destructeur distinct.

Classe vs Objet: Exemple

```
class Personne
{
    String nom;
    private String conjoint ;
    private boolean mariee ;
    public void SeMarrier( String newName )
    {
        conjoint = new String( newName ) ;
        mariee = true ;
    }
}

public static void main( String[] args )
{
    Personne personne1 = new Personne() ;
    personne1.nom= "Nicolas";
    personne1.SeMarrier( "Christine" ) ;
}
```





Expression de chemin

- L'accès aux attributs et aux méthodes de l'objet se fait en écrivant le nom de l'objet et le nom de l'attribut ou la méthode séparés par un point
- Donc on utilise:
 - *voiture1.puissance* pour accéder à l'attribut puissance
 - *voiture1.demarrer ()* pour invoquer la méthode de démarrage

| voiture1 :Voiture | |
|-------------------|--------------------|
| | 1991 |
| | 12 |
| | 710 |
| | Honda1 |
| | écurie = "McLaren" |
| | |

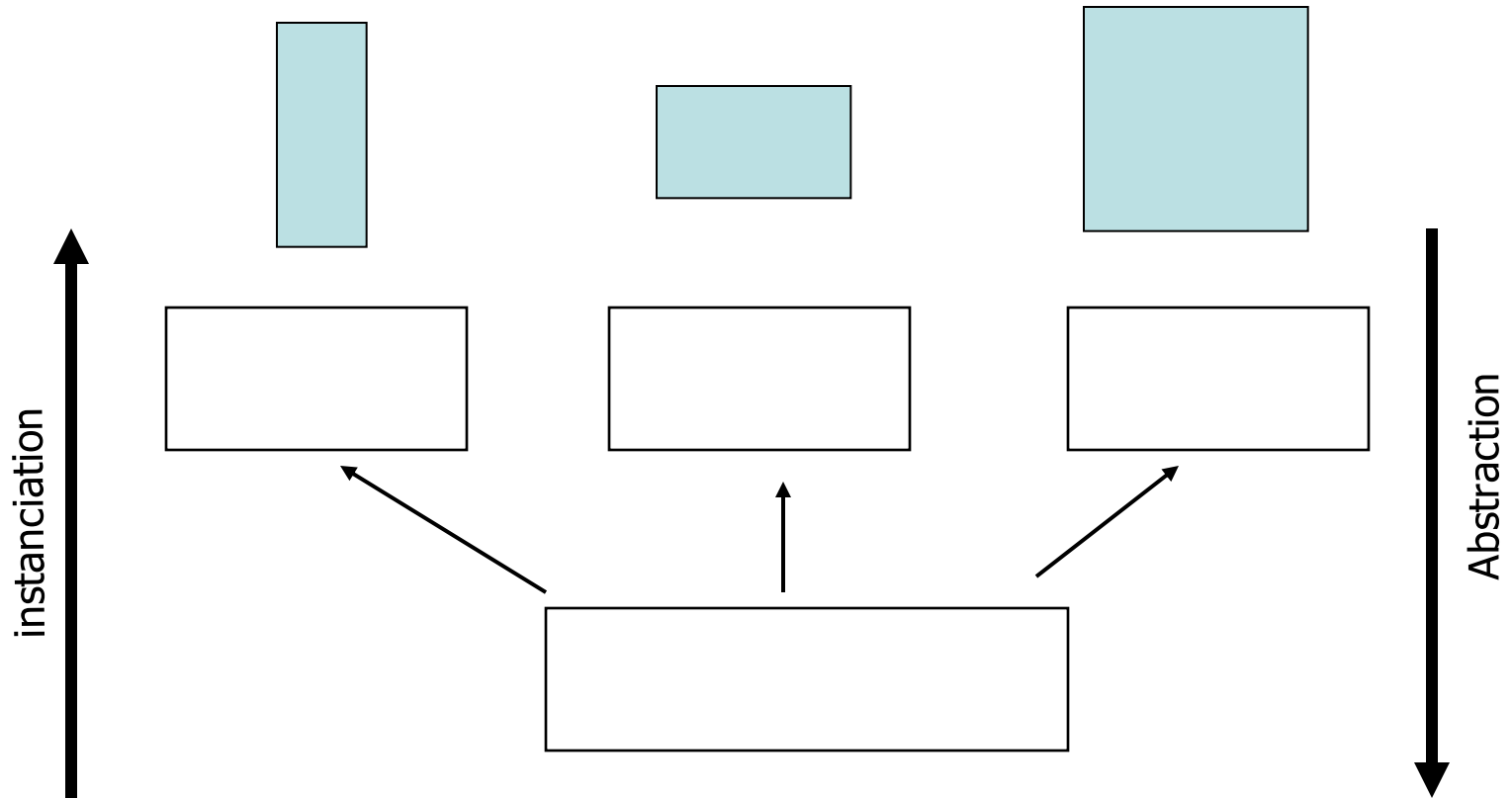
Expression de chemin

- Si les attributs sont eux même des objets on peut accéder à leur attributs de la même façon
 - *pilote1.vehicule.nbcylindre,*
 - *pilote1.vehicule.accelerer()*
 -
 - *pilote1.vehicule.moteur. ...*

| pilote1:Pilote |
|-----------------------|
| "Senna" voiture1 |
| |

| voiture1 :Voiture |
|---|
| 1991 12 710 Honda1 écurie = "Mclaren" |
| |

Abstraction





Abstraction

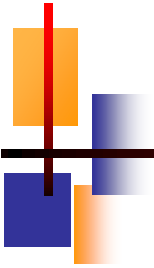
- *Définition :*

« Une abstraction fait sortir les *caractéristiques essentielles* *objet* (qui le distingue de tous les autres genres tout en ignorant les autres caractéristiques non essentielles et donc procure des frontières conceptuelles rigoureusement définies, relativement au point de vue de »



Abstraction

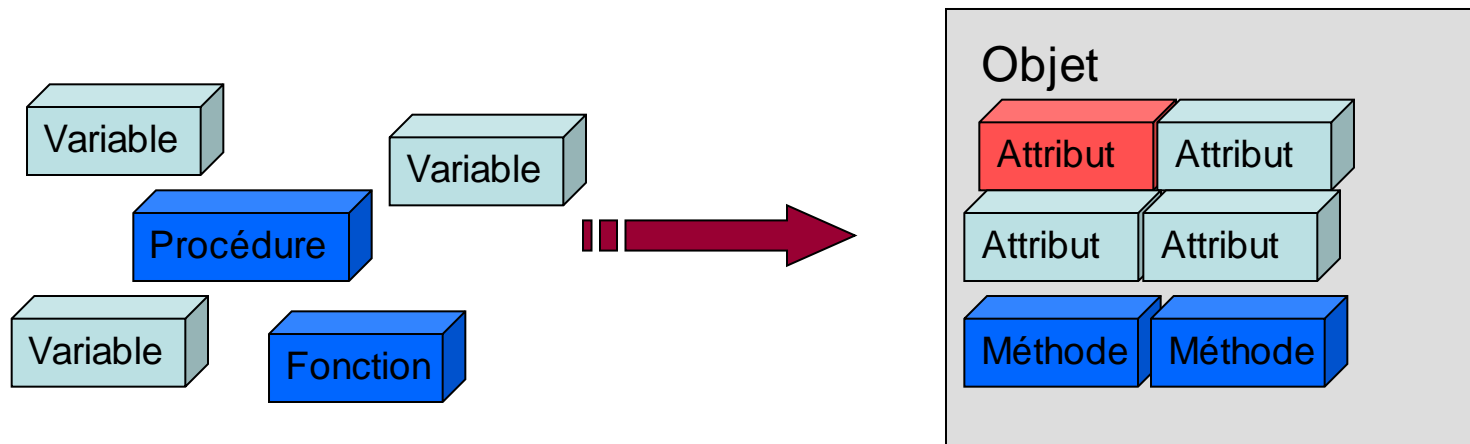
- On constate que la plupart des gens qui pense "abstraction" tendent à croire que les caractéristiques non essentielles se retrouvent *cachées* (d'où "abstraction" = "information hiding") à l'intérieur de la classe qui représente le concept (d'où abstraction = encapsulation)
- Ceci est faux!! Puisque les caractéristiques **non essentielles** sont **IGNORÉES**.
- On ne les masque pas, on ne les encapsule pas



Encapsulation

Encapsulation

- L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une même structure en cachant l'implémentation de l'objet
- Pouvoir masquer l'accès aux données par un autre moyen que les services proposés (méthodes)





Encapsulation

- En interdisant l'utilisateur de modifier directement les attributs, et en l'obligeant à utiliser les fonctions définies pour les modifier (appelées interfaces), on s'assure de l'intégrité des données (type de données conforme à nos attentes, intervalle respecté)
- En plus:
 - L'encapsulation permet de définir des **niveaux de visibilité** des éléments de la classe. Ces niveaux définissent les droits d'accès aux données selon que l'on y accède par une méthode de la classe elle-même, d'une classe héritière, ou bien d'une classe quelconque
 - L'utilisateur d'une classe n'a pas forcément à connaître comment sont structurées les données dans l'objet, cela signifie qu'un utilisateur n'a pas à connaître l'implémentation



Encapsulation

Exemple:

```
class Personne{
    String nom, conjoint ;
    private boolean mariee ;

    public void SeMarrier( String newName ){
        conjoint = new String( newName ) ;
        mariee = true ;
    }
}

public static void main( String[] args ){
    Personne personne1= new Personne();
    personne1.nom= "Nicolas"
    personne1.conjoint="Christine" ;
    personne1.mariee=TRUE ; -----> génère une erreur de compilation!
}
```



Niveaux de visibilité

- Il existe trois niveaux de visibilité (la portée d'une variable/classe):
 - **Publique** (*public*): Les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité public. Il s'agit du plus bas niveau de protection des données
 - **Protégée** (*protected*): l'accès aux données est réservé aux fonctions des classes héritières, c'est-à-dire par les fonctions membres de la classe ainsi que des classes dérivées
 - **Privée** (*private*): l'accès aux données est limité aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé



Niveaux de visibilité

- Il existe un autre niveau de visibilité : **Ami (friend)**
- Il peut prendre deux sens distincts selon le langage de programmation :
- **Java** : consiste à dire que tout membre non publique, privé ou protégé est un membre ami (**niveau de visibilité par défaut**).
 - Il est accessible par tout objet issu d'une classe appartenant au même package (même répertoire).
- **C++** : consiste à définir des relations d'amitié entre les classes ou entre une classe et une fonction définie en dehors de la classe.
 - Cette relation permet aux instances de la classe déclarée ami d'accéder à tous les membres de la classe (qu'ils soient publiques, privés ou protégés)



Niveaux de visibilité

- Exemple :
- C++

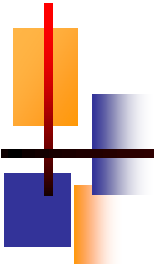
```
class Cercle{  
    // on déclare que Manager est une classe amie  
    friend class Manager;  
    .....}  
  
class Manager{  
    .... // a accès aux variables des instances Cercle  
}
```

- Java

```
class Cercle{  
    int rayon; //attribut ami par défaut  
}
```



Modificateurs des méthodes - JAVA



Relations entre classes



Relations entre classes

- Dans le cadre de la programmation Objet, chaque classe doit être responsable de tâches précises, laissant à d'autres classes le soin de

cm

de



Relations entre classes

- La communication interclasse est effectivement primordiale : la manière dont les messages sont échangés entre les objets est un aspect important de la POO.
- Elles définissent le fonctionnement du programme : une application Objet est définie en tant qu'ensemble d'objets reliés par des relations, qui précisent les comportements des objets les uns par rapport aux autres.
- Ces relations mettent en avant une méthode de programmation basée sur le comportement plutôt que sur les données brutes.

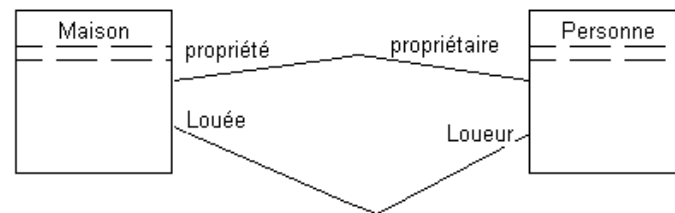
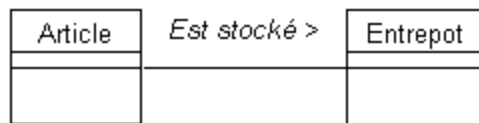


Relations entre classes

- Cinq types de relation interclasse sont couramment utilisés (il en existe d'autres) :
 - ✓ Association
 - ✓ Héritage
 - ✓ Dépendance
 - ✓ Agrégation
 - ✓ Composition

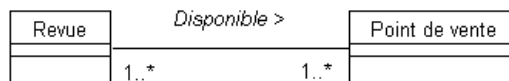
Relations entre classes

- **Association** : c'est la relation la plus simple entre deux classes.
- Elle existe à partir du moment où l'une des deux classes sert de type à un attribut de l'autre, et que cet autre envoie des messages à la première (condition nécessaire pour une association).
- Simplement, une association indique que deux classes communiquent entre elles (dans un sens ou dans les deux sens).
- En représentation graphique (ex. UML), une association est représentée par une ligne entre deux classes, possiblement accompagnée d'une flèche si l'association n'est pas bidirectionnelle.

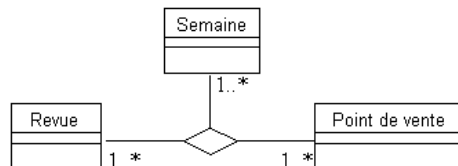


Relations entre classes

- Association (multiplicité):
 - Si l'une des deux classes a une association multiple avec l'autre classe on parlera alors de cardinalité (multiplicité).
 - L'association peut relier deux classes (binaire) ou plusieurs classes
 - En représentation graphique (ex. UML), la multiplicité est représentée par des valeurs : 0..1, 0..*(*), 1..1 (1), 1..*, m..n



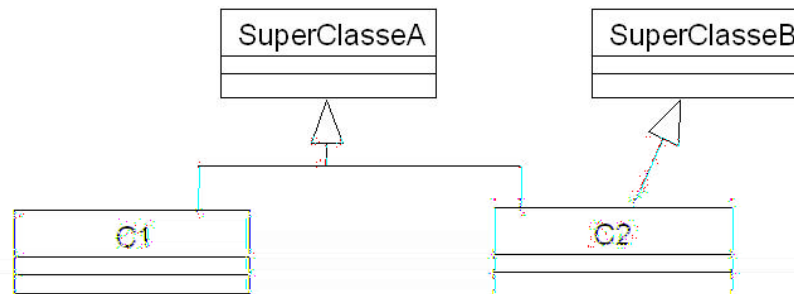
Une revue est disponible dans un ou plusieurs points de vente. Un point de vente distribue de une à plusieurs revues.



Association ternaire pour suivre les disponibilités hebdomadaires

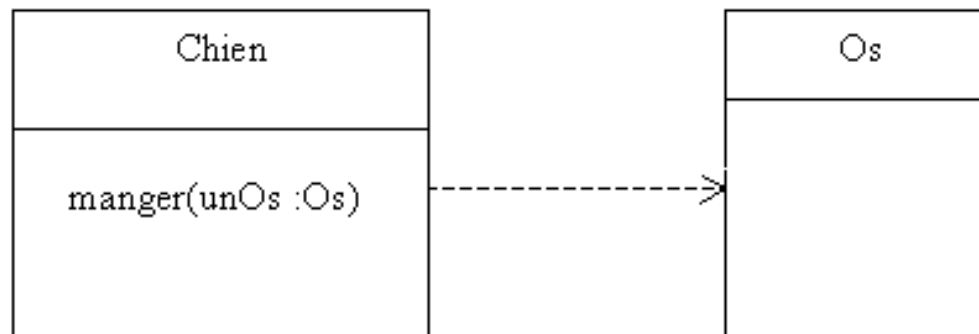
Relations entre classes

- **Héritage** : elle présente une classe spécifique comme descendante d'une classe plus générique. Cette classe spécifique propose des méthodes dont la classe générique ne dispose pas, tout en conservant la plupart des méthodes de cette classe "parente".
- Par exemple, une classe EtudiantVA et une sous classe de la classe Etudiant.
- En représentation graphique (ex. UML), une dépendance est représentée par une ligne, terminée par une flèche évidée.



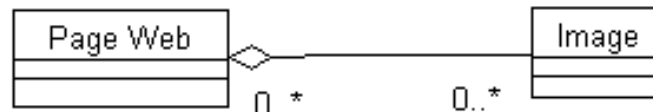
Relations entre classes

- **Dépendance** : elle présente l'utilisation que fait une classe d'une autre. Une classe dépend d'une autre si ses méthodes manipulent l'objet de cette classe.
- Par exemple, une classe Réservation ne pourrait exister que si la classe Compte indique les coordonnées de la personne...
- En représentation graphique (ex. UML), une dépendance est représentée par une ligne en tirets, terminée par une simple flèche.



Relations entre classes

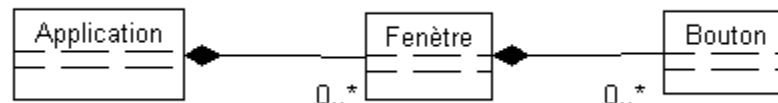
- **Agrégation** : cette relation indique un principe de subordination entre l'agrégat (classe qui regroupe les classes agrégées) et les agrégées.
- Concrètement, elle indique une "possession" : l'agrégat peut contenir plusieurs objets d'un type.
- Par exemple, une classe Réservation peut contenir un ou plusieurs objets de type Place de Cinéma.
- En représentation graphique (ex. UML), une agrégation est représentée par une ligne entre deux classes, terminée par un losange vide ("diamant") du côté de l'agrégat.



- Une page peut contenir des images mais celles-ci peuvent appartenir à d'autres pages.
- La destruction d'une page n'entraîne pas celle de l'image mais seulement la suppression du lien.

Relations entre classes

- **Composition** "agrégation forte" ou "agrégation par valeur" : il s'agit en fait d'une agrégation à laquelle on impose des contraintes internes
- Un seul objet peut faire partie d'un composite (l'agrégat de la composition), et celui-ci doit gérer toutes ses parties. En clair, les composants sont totalement dépendants du composite.
- En représentation graphique (ex. UML), la composition est représentée de la même manière que l'agrégation, mais le diamant est plein..

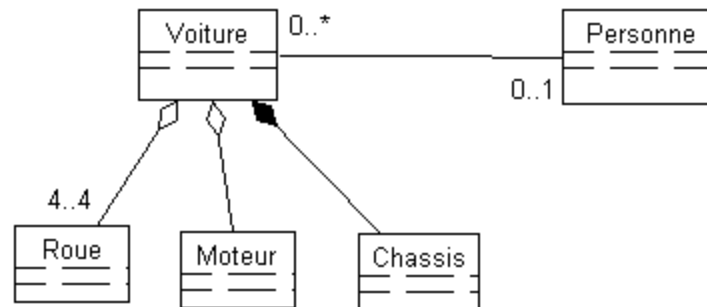


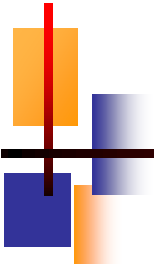
- Une application contient de 0 à n fenêtres qui contiennent de 0 à n boutons.
- La fermeture de l'application entraîne la destruction des fenêtres qui entraîne la destruction des boutons.
- La non-présence des valeurs de multiplicités est synonyme de 1..1.

Relations entre classes

- Exemple :

- Le châssis est un élément indissociable d'une voiture, d'où la composition.
- On estime que le moteur et les roues peuvent être utilisés dans d'autres voitures.
- Les valeurs 4..4 caractérisent plus précisément les valeurs de multiplicité.
- Les absences de cardinalité sont assimilable à 1..1.
- L'association entre *Voiture* et *Personne* n'est pas nommée, cela est conseillé lorsque son nom est trivial : "appartient", "concerne"... afin de ne pas alourdir le modèle, sans rien apporter à la sémantique.





Héritage



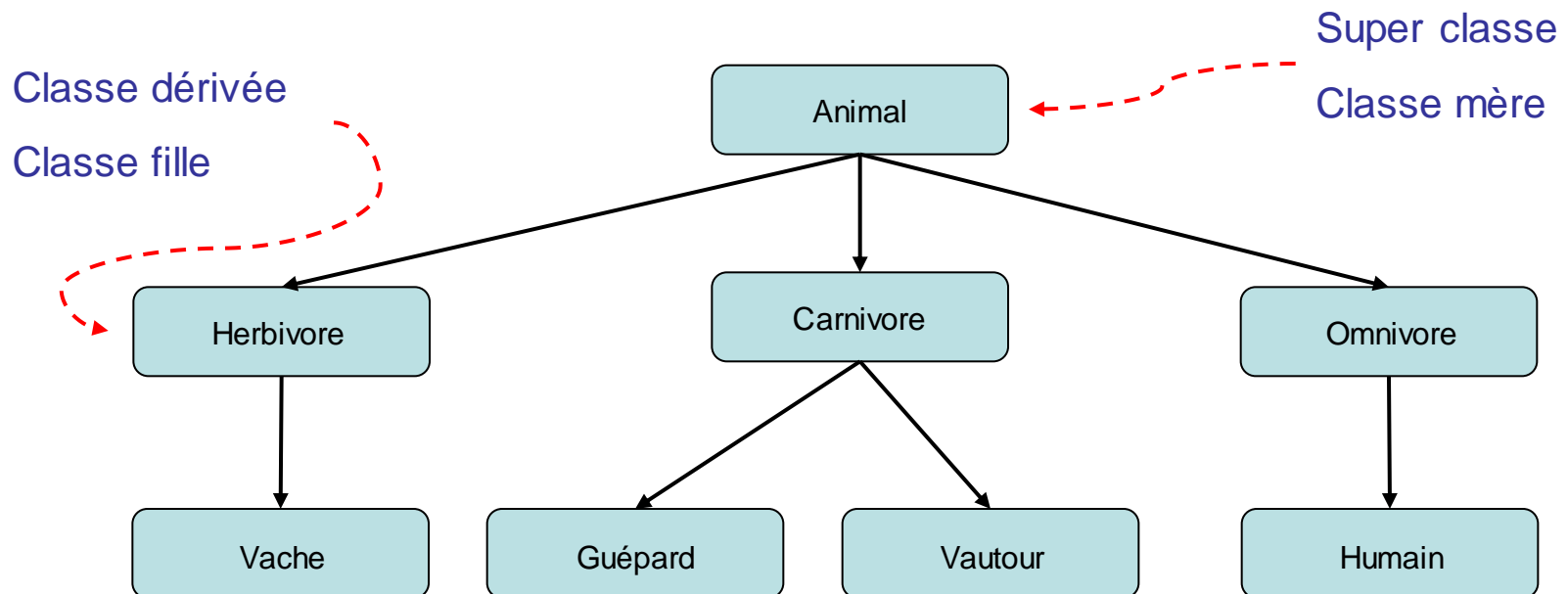
Héritage

- **L'héritage** (en anglais *inheritance*) est un principe propre à la programmation orientée objet.
- permettant de créer une nouvelle classe à partir d'une classe existante
- Le nom *héritage* est parfois appelé dérivation de classe
- La classe dérivée (la classe nouvellement créée) contient les attributs et les méthodes de sa superclasse (la classe dont elle dérive)

Héritage

- Exemple :

- La classe Humain hérite de la classe Omnivore



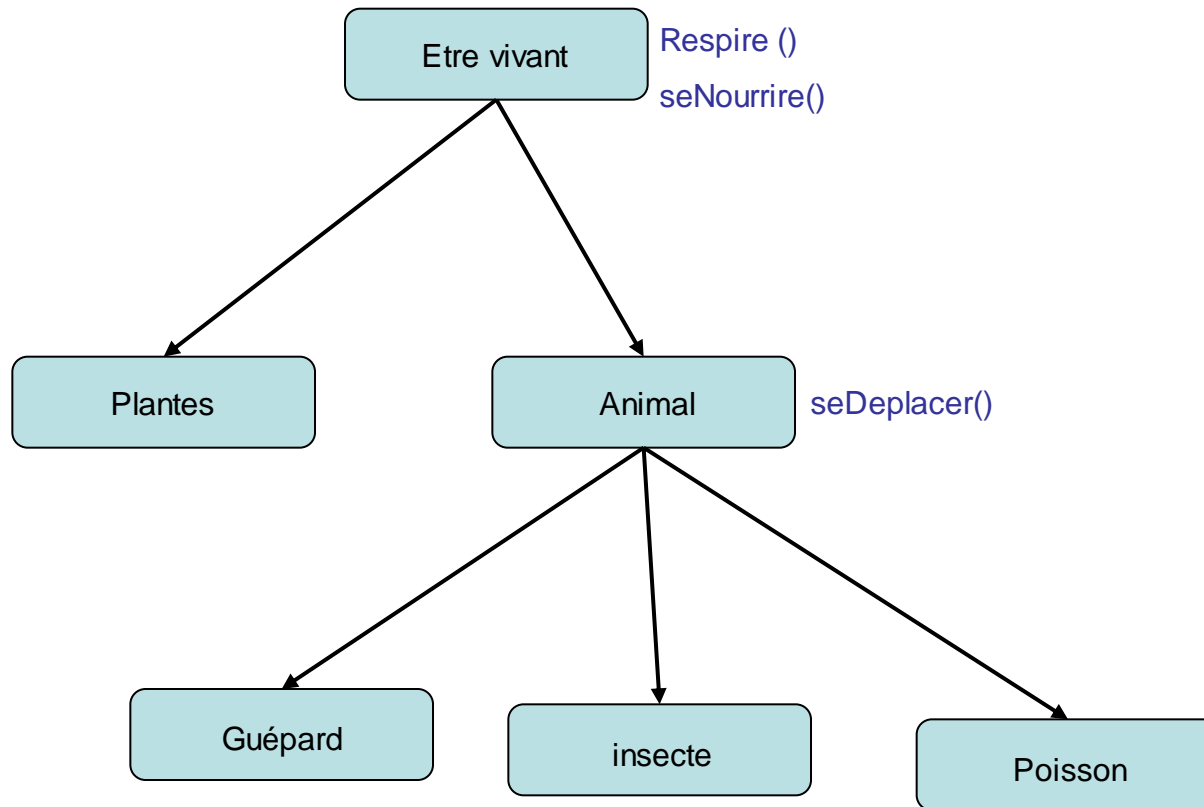


Héritage

- L'intérêt majeur de l'héritage est de pouvoir **définir** ou **redéfinir** des attributs et des méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées
- Avec l'héritage on crée une hiérarchie de classes de plus en plus **spécialisées**
- Cela a comme avantage majeur de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante
- Grâce à l'héritage il est possible d'acheter dans le commerce des librairies de classes, qui constituent une base, pouvant être spécialisées à loisir (on comprend encore mieux l'intérêt pour l'entreprise qui vend les classes de protéger les données membres grâce à l'encapsulation...)

Héritage

Exemple:





Héritage: Hiérarchie des classes

- Il est possible de représenter sous forme de **hiérarchie de classes**, parfois appelée arborescence de classes, la relation de parenté qui existe entre les différentes classes
- L'arborescence commence par une classe générale appelée **superclasse** (parfois classe de base, classe parent, classe ancêtre, classe mère ou classe père, les métaphores généalogiques sont nombreuses)
- Les classes dérivées (classe fille ou sous-classe) deviennent de plus en plus spécialisées. On peut généralement exprimer la relation qui lie une classe fille à sa mère par la phrase "**est un/une**" (de l'anglais "**is a**")

Ex. : un chien **est un** animal



Héritage: Hiérarchie des classes

- Exemple (dans Java):

animator4

Class Animator

java.lang.Object

|

+--java.awt.Component

|

+--java.awt.Container

|

+--java.awt.Panel

|

+--java.applet.Applet

|

+--[edu.davidson.tools.SApplet](#)

|

+--animator4.Animator



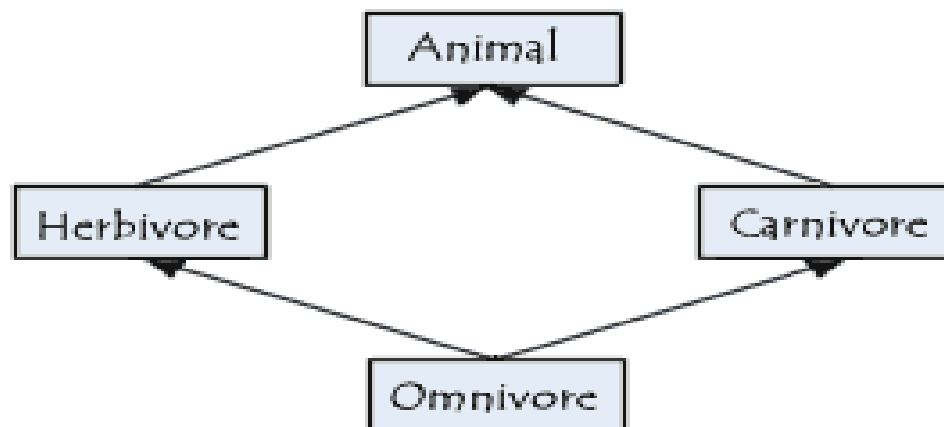
Héritage multiple

- En plus de l'héritage Simple, certains langages orientés objet, tels que C++, permettent de faire de l'héritage multiple: possibilité de faire hériter une classe de deux superclasses
- D'autres langage, comme Java, proposent d'autres solutions de remplacement de l'héritage multiple
- Cette technique permet de regrouper au sein d'une seule et même classe les attributs et méthodes de plusieurs classes

Héritage multiple

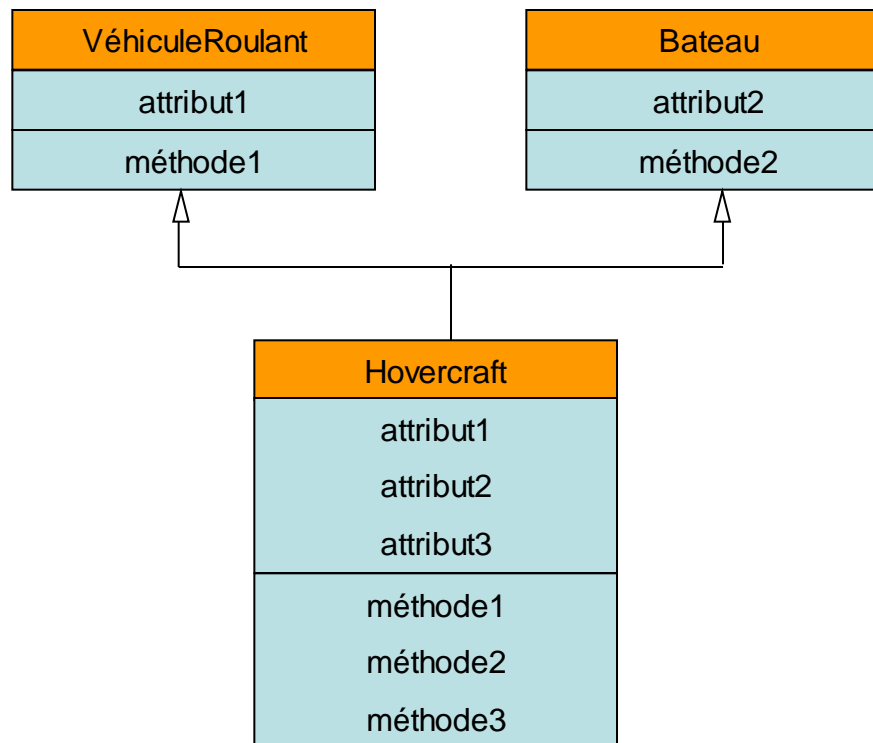
- Exemple :

- Les ours noirs sont omnivores : herbes, fruits (noisettes, baies, pignons, glands, ...), insectes (fourmis, abeilles, termites), rongeurs ...saumons, truites, miel...
- Ils héritent des classes herbivore et carnivore en même temps



Héritage multiple

- L'héritage multiple est une extension au modèle d'héritage simple.
- Une classe peut posséder plusieurs classes mères afin de modéliser une généralisation multiple
- Exemple : «Hovercraft» est à la fois un bateau et un véhicule terrestre





Héritage multiple

- En plus de l'héritage Simple, certains langages orientés objet, tels que C++, permettent de faire de l'héritage multiple: possibilité de faire hériter une classe de deux superclasses
- Cette technique permet de :
 - Regrouper au sein d'une seule et même classe les attributs et méthodes de plusieurs classes
 - Mieux organiser les objets en évitant la redondance des définition des attributs et des opérations



Héritage multiple

- Situation :

- Nombreuses bibliothèques de classes utilisent une classe abstraite de base (méta-classe) afin de bénéficier de mécanismes polymorphiques (sur le constructeur et le destructeur par exemple)...

- Donc:

- Nous risquons d'être confrontés au problème de l'héritage à répétition dès que nous utilisons de l'héritage multiple

- Mais:

- Certains langages de programmation (tels que Java, .NET) bannissent l'héritage multiple.

- Ils proposent une alternative intéressante et particulièrement agréable à l'héritage multiple : **les interfaces**



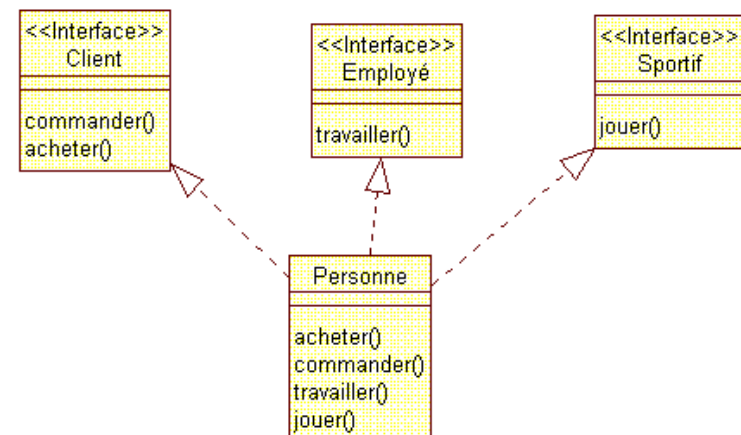
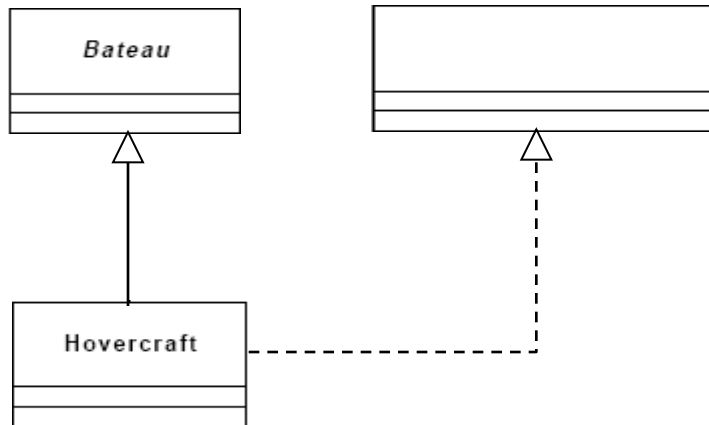
La notion d'interface

- Qu'est-ce qu'une interface?
 - Une interface est semblable à une classe sans attributs (mais pouvant contenir des **constantes**) dont toutes les méthodes sont **abstraites** (sans instructions)
- Qu'est-ce qu'une méthode abstraite?
 - C'est une méthode qui n'a pas d'instructions
- Une classe implémente une interface si elle propose une implémentation pour chacune des méthodes décrites en interface
- Exemple d'une interface en Java:

```
public interface Vehicule {  
    protected final static boolean MARCHE = true;  
    protected final static boolean ARRET = false;  
    public abstract void demarrerMoteur();  
    public abstract void afficheAttrib();  
}
```

La notion d'interface

- Les méthodes décrites dans les interfaces sont, par définition, polymorphes puisqu'elles sont implémentées de **façon indépendante** dans chaque classe implémentant une même interface
- Ce mécanisme est particulièrement puissant car il crée des relations fortes entre différentes classes implémentant les mêmes interfaces sans que ces dernières aient une relation de parenté

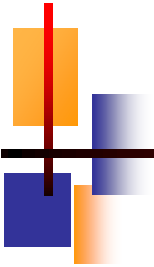




La notion d'interface

- Chaque classe peut implémenter autant d'interfaces qu'elle le désire.
- Exemple:

```
public class classe1 implements interface1, interface2 {  
    ...  
};
```
- Si la classe n'implémente pas toutes les méthodes abstraites, c'est une *classe abstraite* (i.e. elle ne peut avoir d'instances)
- RQ: Faire la différence entre héritage et utilisation d'interfaces n'est pas toujours aisé et dépend du point de vue du concepteur mais également de l'environnement du logiciel à construire



Polymorphisme



Polymorphisme

- Le nom « *polymorphisme* » vient du grec et signifie « qui peut prendre plusieurs formes »
- C'est une caractéristique essentielle de la programmation orientée objet, elle donne la possibilité de définir plusieurs méthodes de même nom mais possédant des paramètres différents (en nombre et/ou en type),
- si bien que la bonne méthode sera choisie en fonction de ses paramètres lors de l'appel
- Le polymorphisme rend possible le choix automatique de la bonne méthode à adopter en fonction du type de donnée passée en paramètre.
- Types de polymorphisme: par surcharge et par héritage(redéfinition),...



Polymorphisme - surcharge

- On peut, par exemple, définir plusieurs méthodes homonymes `addition()` effectuant une somme de deux valeurs:
 - **La méthode 1:** `int addition(int, int)` pourra retourner la somme de deux entiers
 - **La méthode 2:** `float addition(float, float)` pourra retourner la somme de deux flottants
 - **La méthode 3:** `char addition(char, char)` pourra définir au gré de l'auteur la somme de deux caractères
 - Etc.



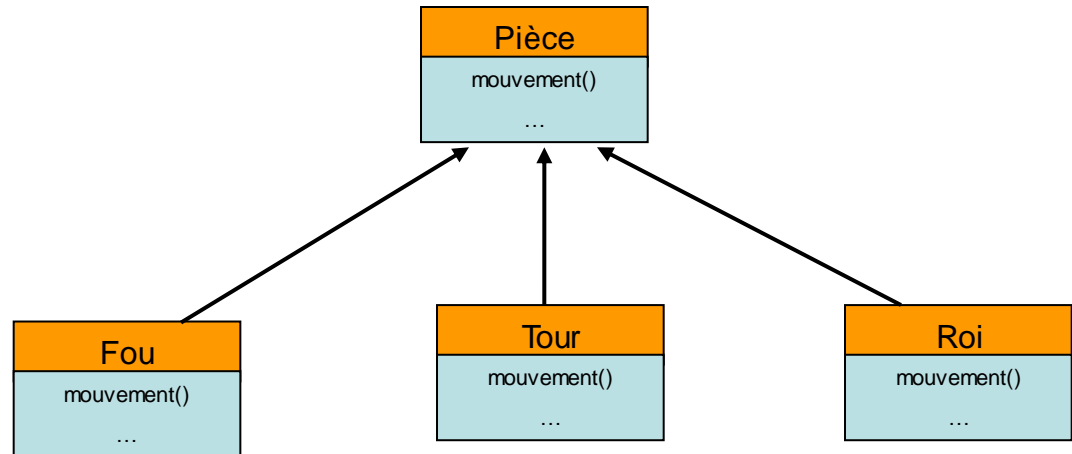
Polymorphisme - héritage

- Imaginons un jeu d'échec comportant:
 - Des objets fou, roi, tour,...
 - La méthode mouvement() pour le déplacement de chaque pièce
- Cette méthode pourra, grâce au polymorphisme, effectuer le mouvement approprié d'une pièce grâce au type de pièce qui lui sera fourni en paramètre:
 - mouvement(fou) fera appel à une méthode définie au préalable et qui spécifie la manière dont se déplace le fou,
 - mouvement(tour) fera appel à une méthode définie au préalable et qui spécifie la manière dont se déplace la tour,
 - etc...

Polymorphisme

- Exemple:

```
Roi roi=new Roi();  
Tour tour=new Tour();  
Fou fou=new Fou();  
Piece p= new Piece();
```



```
p=roi;  
p.mouvement();  
p=tour;  
p.mouvement();  
p=fou;  
p.mouvement();
```

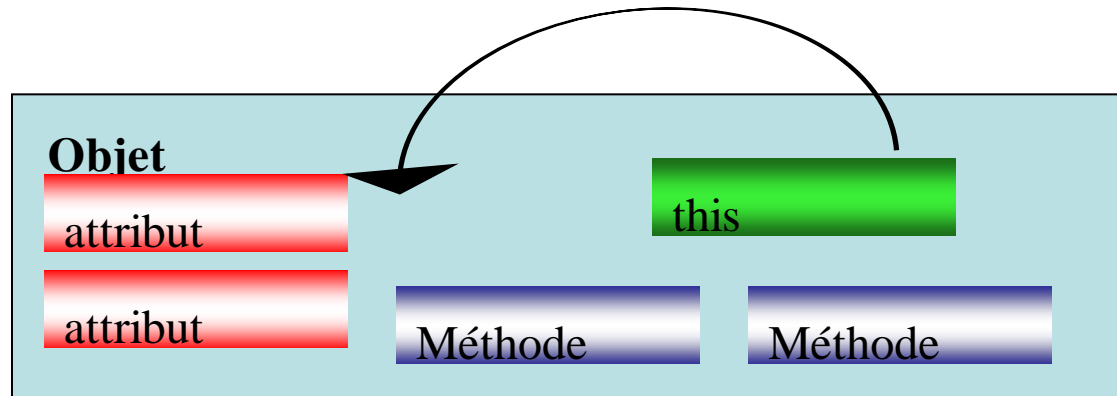
Même variable (p) mais des
mouvement différents



Pointeur interne - « this »

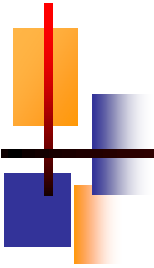
- Très souvent, les objets sont utilisés de manière **dynamique**, et ne sont donc créés que lors de l'exécution.
- Si les méthodes sont toujours communes aux instances d'un même type objet, il n'en est pas de même pour les données.
- Il peut donc se révéler indispensable pour un objet de pouvoir se référencer lui-même. Pour cela, toute instance dispose d'un **pointeur interne** vers elle-même.
- Ce pointeur peut prendre différentes appellations. En **Pascal**, il s'agira du pointeur **Self**. D'autres langages pourront le nommer **this**, comme le C++, java...

Pointeur interne



- Exemple :

```
class Personne{  
    String nom;  
    Personne(String nom1){  
        this.nom = nom1;  
    }  
}
```



Abstraction

Méthodes et Classes



Méthode / classe abstraite

- Les méthodes abstraites :

- Une méthode abstraite est une méthode qui peut posséder un ou plusieurs arguments
- Une méthode abstraite ne possède pas de bloc d'instructions encadré par des accolades, lequel est remplacé par un séparateur point-virgule (;).
- Une méthode abstraite est identifiée par le modificateur : *abstract*

```
[ClassModifiers] abstract type_de_retour methodName ([type arg1]);
```

- Exemples :

```
abstract public void affichage ();
```

```
abstract int addition (int a, int b);
```



Méthode / classe abstraite

- Les classes abstraites :

- Une classe abstraite est une classe qui possède au moins une méthode abstraite
- *classes dérivées* en proposant un ensemble de méthodes que l'on retrouvera tout au long de l'arborescence
- Une classe abstraite est identifiée par le modificateur : *abstract*

En java

```
[ClassModifiers] abstract class className { // corps de la classe }
```

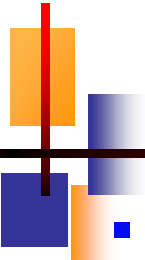
- *new*) puisqu'elle ne contient que des informations incomplètes
- On ne peut pas utiliser le modificateur *final* avec *abstract* sinon on ne pourra pas étendre la classe et redéfinir les méthodes



Méthode / classe abstraite

- Les méthodes / classes abstraites exemple JAVA :

```
abstract class Animal {  
    int age;  
  
    abstract void seNourrir();  
  
    int getAge(){  
        return age;  
    }  
  
    public static void main (String arg[]){  
        Animal a = new Animal(); //Faux  
        // on a pas le  
    }  
}
```

Méthode - surcharge

- La surcharge des méthodes :
 - Si deux méthodes possèdent un *nom semblable* mais différentes *signatures*, le nom de ces méthodes est dit surchargé.
 - La signature est composée :

Des types de paramètres

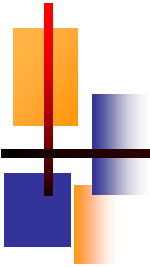
- Attention : le type de retour ne fait pas partie de la signature

- Exemple :

void *debiter* (int montant, String commentaire,int numero)

int *debiter* (int montant, int numero)

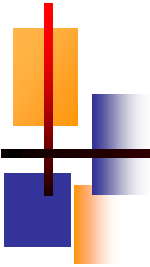
String *debiter* (int montant, int nb) ///erreur : méthode déjà défini



Classe abstraite: Utilité

Le but des classes abstraites est de:

- Définir un cadre de travail pour les classes dérivées en proposant un ensemble de méthodes que l'on retrouvera tout au long de l'arborescence
- Ce mécanisme est fondamental pour la mise en place du polymorphisme:
 - Si on considère la classe Véhicule, il est tout à fait naturel qu'elle ne puisse avoir d'instance : un véhicule ne correspond à aucun objet concret mais plutôt au concept d'un objet capable de démarrer, ralentir, accélérer ou s'arrêter, que ce soit une voiture, un camion ou un avion



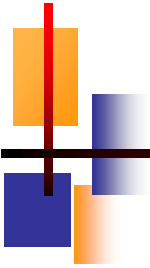
Classe abstraite: quand les utiliser?

- Déterminer si une classe sera abstraite ou concrète dépend souvent du cahier des charges du logiciel
- Au travers de l'étude de celui-ci, il est possible de déterminer quelles sont les classes concrètes : ce sont celles qui possèdent des instances
- A partir de ces dernières, il est possible, par une démarche de généralisation, de trouver les classes abstraites qui permettront de bénéficier du polymorphisme.



Surcharge

- La surcharge est un mécanisme fréquemment proposé par les langages de programmation orientés objet et qui permet d'associer au même nom de méthode différentes signatures (en nombre et/ou en type), si bien que la bonne méthode sera choisie en fonction de ses paramètres lors de l'appel
- Le polymorphisme et rend possible le choix automatique de la bonne méthode à adopter en fonction du type de donnée passée en paramètre
- C'est une forme faible du polymorphisme!!



Surcharge

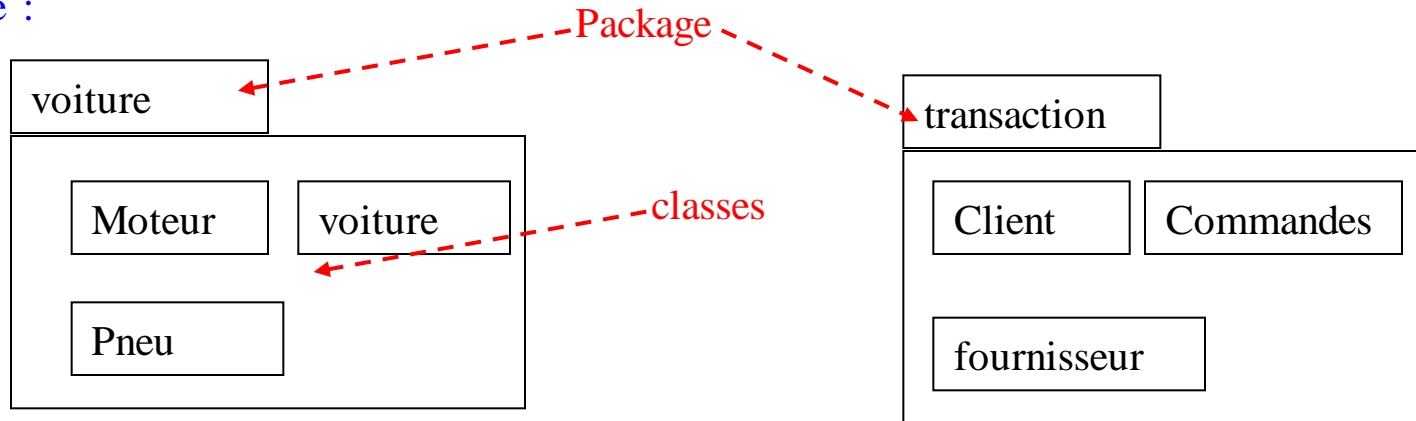
Exemple:

- On peut, par exemple, définir plusieurs méthodes homonymes `addition()` effectuant une somme de deux valeurs:
 - La méthode 1: `int addition(int, int)` pourra retourner la somme de deux entiers
 - La méthode 2: `float addition(float, float)` pourra retourner la somme de deux flottants
 - La méthode 3: `char addition(char, char)` pourra définir au gré de l'auteur la somme de deux caractères
 - Etc.

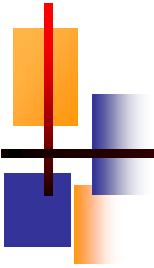
Package – création

- Visibilités - JAVA :

Exemple :



| Modificateur de la méthode | Même package | | Autre package | |
|----------------------------|--------------|--------------|---------------|--------------|
| | Classe mère | Autre classe | Classe mère | Autre classe |
| Public | ✓ | ✓ | ✓ | ✓ |
| Protected | ✓ | ✓ | ✓ | |
| Friend | ✓ | ✓ | | |
| Private protected | ✓ | | ✓ | |
| private | | | | |



Exemple



Exemple: Classe Voiture

- La classe la plus simple que l'on peut créer est une classe vide:

```
class Voiture {  
};
```

- Définition des attributs ou variables d'instance de la classe voiture:

```
class Voiture {  
    String marque; // chaîne de caractère  
    String couleur; // chaîne de caractère  
    boolean etatMoteur; // état Marche ou Arrêt du moteur  
    ...  
}
```




Exemple: Classe Voiture

- Définition d'une méthode demararMoteur:

```
void demararMoteur() { // teste l'état du moteur et le démarre si besoin
    if (etatMoteur == true)
        System.out.println("Le moteur est déjà en marche.");
    else {
        etatMoteur = true;
        System.out.println("Le moteur a été mis en marche.");
    }
}
```



Exemple: définition complète

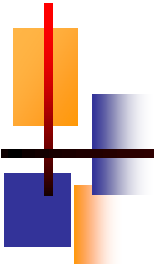
```
class Voiture {  
    String marque; // chaîne de caractère  
    String couleur; // chaîne de caractère  
    boolean etatMoteur; // état Marche ou Arrêt du moteur  
    public Voiture() { } // constructeur. Peut être omis  
    void demararMoteur() { // teste l'état du moteur et le démarre si besoin  
        if (etatMoteur == true)  
            System.out.println("Le moteur est déjà en marche.");  
        else {  
            etatMoteur = true;  
            System.out.println("Le moteur a été mis en marche.");  
        }  
    }  
}
```



Exemple: Ajout d'une méthode

- La méthode "afficheAttrib" affiche les attributs d'un objet:

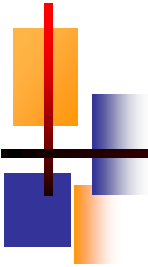
```
void afficheAttrib() {  
    System.out.println("Cette voiture est une " + marque + " " + couleur);  
    if (etatMoteur == true) System.out.println("Moteur en marche.");  
    else System.out.println("Moteur arrêté.");  
}
```



Exemple: utilisation de la classe

- Pour une application Java indépendante, il faut une méthode "main« (méthode que Java cherche à exécuter en premier)

```
public static void main (String args[]) {  
    Voiture v = new Voiture(); // crée une instance de voiture et place  
    v.marque = "Renault"; // positionne les variables membres  
    v.couleur="blanche";  
    v.afficheAttrib(); // appel méthode sur référence 'v'  
    System.out.println("On démarre le moteur...");  
    v.demarrerMoteur();  
    v.afficheAttrib();  
    System.out.println("On redemarre le moteur...");  
    v.demarrerMoteur();  
}
```



Exemple: utilisation de la classe

- Le source de la classe doit être mis dans un fichier *Voiture.java* (même nom que la classe + extension '.java')
- Après compilation, on obtient un fichier *Voiture.class*
- L'exécution de cette classe donne le résultat suivant à l'écran:
 - Cette voiture est une Renault blanche
 - Moteur arrêté
 - On démarre le moteur
 - Le moteur a été mis en marche
 - Cette voiture est une Renault blanche
 - Moteur en marche
 - On redémarre le moteur
 - Le moteur est déjà en marche