

# Programmation Orientée Objet (POO)

## Langage Java



# Concept d'objet

- Qu'est-ce qu'un objet ?
  - Le monde qui nous entoure est composé d'objets
  - Ces objets ont tous deux caractéristiques
    - un état
    - un comportement
- Exemples d'objets du monde réel
  - chien
    - état : nom, couleur, race, poids....
    - comportement : manger, aboyer, renifler...
  - Bicyclette
    - état : nombre de vitesses, vitesse courante, couleur
    - comportement : tourner, accélérer, changer de vitesse

# L'approche objet

- L'approche objet :
  - **Programmation dirigé par les données** et non par les traitements
  - Notion d'encapsulation
    - les données et les procédures qui les manipulent (on parle de méthodes) sont regroupés dans une même entité (la classe).
- Un objet informatique
  - maintient son état dans des variables (appelées *attributs*)
  - implémente son comportement à l'aide de méthodes

objet informatique = regroupement logiciel de variables et de méthodes

- Cycle de vie
  - construction (en mémoire)
  - Utilisation (changements d'état par affectations, comportements par exécution de méthodes)
  - destruction

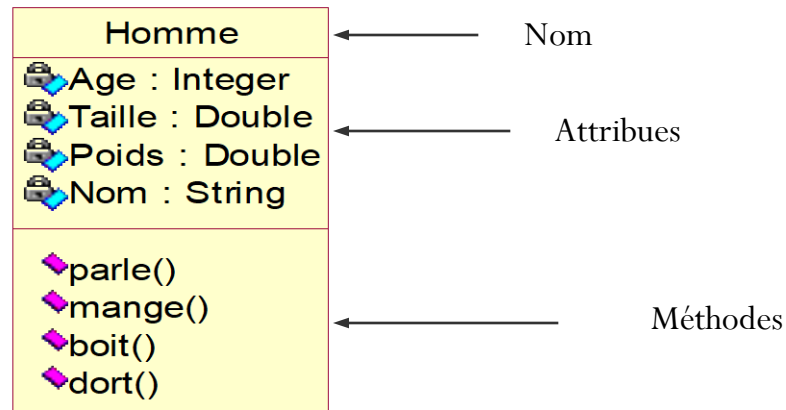
# Concept de classe

- En informatique, la **classe est un modèle** décrivant les caractéristiques communes et le comportement d'un ensemble d'objets : la classe est un moule et l'objet est ce qui est moulé à partir de cette classe
- Mais l'état de chaque objet est indépendant des autres
  - Les objets sont des représentations dynamiques (appelées instances) du modèle défini au travers de la classe
  - Une classe permet d'instancier plusieurs objets
  - Chaque objet est une instance d'une seule classe

# La classe (1) : définition

- **Classe** : description d'une famille d'objets ayant une même structure et un même comportement. Elle est caractérisée par :
  - Un nom
  - Une composante statique : des **champs** (ou **attributs**) nommés ayant une valeur. Ils caractérisent l'état des objets pendant l'exécution du programme
  - Une composante dynamique : des **méthodes** représentant le comportement des objets de cette classe. Elles manipulent les attributs des objets et caractérisent les actions pouvant être effectuées par les objets.

# La classe (2) : représentation graphique



Une classe représentée avec la notation UML  
(Unified Modeling Language)

# Modificateurs d'accès

Un modificateur indique si les autres classes de l'application pourront accéder ou non à la classe/méthode/attribut (qualifié par la suite d'« item »).

Ces modificateurs sont au nombre de quatre :

**public** : toutes les classes peuvent accéder à l'item

**protected** : seules les classes dérivées et les classes du même package peuvent accéder à l'item

**private** : l'item est seulement accessible depuis l'intérieur de la classe où il est défini.

**default** : sans modificateur d'accès, seules les classes du même package peuvent accéder à l'item.

# Syntaxe de définition d'une classe

## Exemple : Une classe définissant un point

```
Class Point  
{  
    double x; // abscisse du point  
    double y; // ordonnée du point  
    // translate de point de (dx,dy)  
    void translate (double dx, double dy) {  
        x = x+dx;  
        y = y+dy;  
    }  
    // calcule la distance du point à l'origine  
    double distance() {  
        return Math.sqrt(x*x+y*y);  
    }  
}
```

← Nom de la Classe

← Attributs

← Méthodes

} 8



# L'instanciation (1)

- Instanciation : concrétisation d'une classe en un objet « *concret* ».
  - Dans nos programmes Java nous allons définir des classes et *instancier* ces classes en des objets. Le fonctionnement du programme résultera de l'interaction entre ces objets « *instanciés* ».
  - En Programmation Orientée Objet, on décrit des classes et l'application en elle-même va être constituée des objets instanciés, à partir de ces classes, qui vont communiquer et agir les uns sur les autres.

# L'instanciation (2)

- Instance
  - Un objet constituant un exemplaire de la classe
  - obtenu par moulage du dictionnaire des variables et détenant les valeurs de ces variables.
  - Son comportement est défini par les méthodes de sa classe
  - Par abus de langage « instance » = « objet »
- Exemple :
  - si nous avons une classe voiture, alors votre voiture est une instance particulière de la classe voiture.
  - Classe = concept, description
  - Objet = représentant *concret* d'une classe

# Les constructeurs (1)

- L'appel de **new** pour créer un nouvel objet déclenche, dans l'ordre :
  - L'allocation mémoire nécessaire au stockage de ce nouvel objet et l'initialisation par défaut de ces attributs,
  - L'initialisation explicite des attributs
  - L'exécution d'un constructeur.
- Un constructeur est une méthode d'initialisation.

```
public class Application
{
    public static void main(String args[])
    {
        Personne jean = new Personne()
        jean.setNom("Jean") ;
    } }
```

Le constructeur est ici celui par défaut (pas de constructeur défini dans la classe Personne)

# Les constructeurs (2)

- Le constructeur est une méthode :
  - de même nom que la classe,
  - sans type de retour.
- Toute classe possède au moins un constructeur. Si le programmeur ne l'écrit pas, il en existe un par défaut, sans paramètres, de code vide.

# Les constructeurs (3)

Personne.java

```
public class Personne
{
    public String nom;
    public String prenom;
    public int age;
    public Personne(String unNom,
                     String unPrenom,
                     int unAge)

    {
        nom=unNom;
        prenom=unPrenom;
        age = unAge;
    }
}
```

Va donner une erreur à la compilation

Définition d'un Constructeur. Le constructeur par défaut (Personne() )n'existe plus. Le code précédent occasionnera une erreur

```
public class Application
{
    public static void main(String args[])
    {
        Personne jean = new Personne()
        jean.setNom("Jean") ;
    }
}
```

# Les constructeurs (4)

- Pour une même classe, il peut y avoir plusieurs constructeurs, de signatures différentes (surcharge).
- L'appel de ces constructeurs est réalisé avec le **new** auquel on fait passer les paramètres.
  - **p1 = new Personne("Pierre", "Richard", 56);**
- Déclenchement du "bon" constructeur
  - Il se fait en fonction des paramètres passés lors de l'appel (nombre et types). C'est le mécanisme de "lookup".
- Attention
  - Si le programmeur crée un constructeur (même si c'est un constructeur avec paramètres), le constructeur par défaut n'est plus disponible. Attention aux erreurs de compilation !

# Les constructeurs (5)

Personne.java

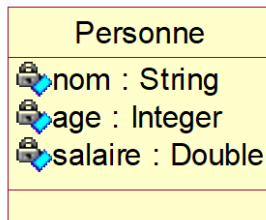
```
public class Personne
{
    public String nom;
    public String prenom;
    public int age;
    public Personne()
    {
        nom=null; prenom=null;
        age = 0;
    }
    public Personne(String unNom,
                    String unPrenom, int unAge)
    {
        nom=unNom;
        prenom=unPrenom; age = unAge;
    } }
}
```

Redéfinition d'un  
Constructeur sans paramètres

On définit plusieurs constructeurs  
qui se différencient uniquement  
par leurs paramètres (on parle  
de leur signature)

# Classe et objet en Java

Du modèle à ...



... la classe Java et  
de la classe à ...

```
class Personne
{
    String nom;
    int age;
    float salaire;
}
```

... des instances  
de cette classe

```
Personne jean, pierre;
jean = new Personne ();
pierre = new Personne ();
```

L'opérateur d'instanciation en Java est **new** :

```
MaClasse monObjet = new MaClasse();
```

En fait, **new** va réserver l'espace mémoire nécessaire pour créer l'objet

« monObjet » de la classe « MaClasse »

Le **new** ressemble beaucoup au **malloc** du C



# Objets, tableaux, types de base

- Lorsqu'une variable est d'un type objet ou tableau, ce n'est pas l'objet ou le tableau lui-même qui est stocké dans la variable mais une *référence* vers cet objet ou ce tableau (on retrouve la notion d'adresse mémoire ou du pointeur en C).
- Lorsqu'une variable est d'un type de base, la variable contient la valeur.

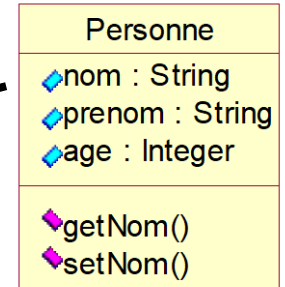
# Références

- La référence est, en quelque sorte, un pointeur pour lequel le langage assure une manipulation transparente, comme si c'était une valeur (pas de déréférencement).
- Par contre, du fait qu'une référence n'est pas une valeur, c'est au programmeur de prévoir l'allocation mémoire nécessaire pour stocker effectivement l'objet (utilisation du **new**).

# Accès aux attributs d'un objet (1)

Personne.java

```
public class Personne
{
    public String nom;
    public String prenom;
    public int age;
    public void setNom(String unNom)
    {
        nom = unNom;
    }
    public String getNom()
    {
        return (nom);
    }
}
```



# Accès aux attributs d'un objet (2)

Application.java

```
public class Application
{
    public static void main(String args[])
    {
        Personne jean = new Personne()
        jean.nom = "Jean" ;
        jean.prenom = "Pierre" ;
    }
}
```

## Remarque :

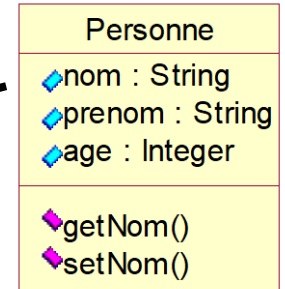
Contrairement aux variables, les attributs d'une classe, s'ils ne sont pas initialisés, se voient affecter automatiquement une valeur par défaut.

Cette valeur vaut : **0** pour les variables numériques, **false** pour les booléens, et **null** pour les références.

# Accès aux méthodes d'un objet (1)

Personne.java

```
public class Personne
{
    public String nom;
    public String prenom;
    public int age;
    public void setNom(String unNom)
    {
        nom = unNom;
    }
    public String getNom()
    {
        return (nom);
    }
}
```



# Accès aux méthodes d'un objet (2)

Application.java

```
public class Application
{
    public static void main(String args[])
    {
        Personne jean = new Personne()
        jean.setNom("Jean") ;
    }
}
```

# Notion de méthodes et de paramètres (1)

La notion de méthodes dans les langages objets

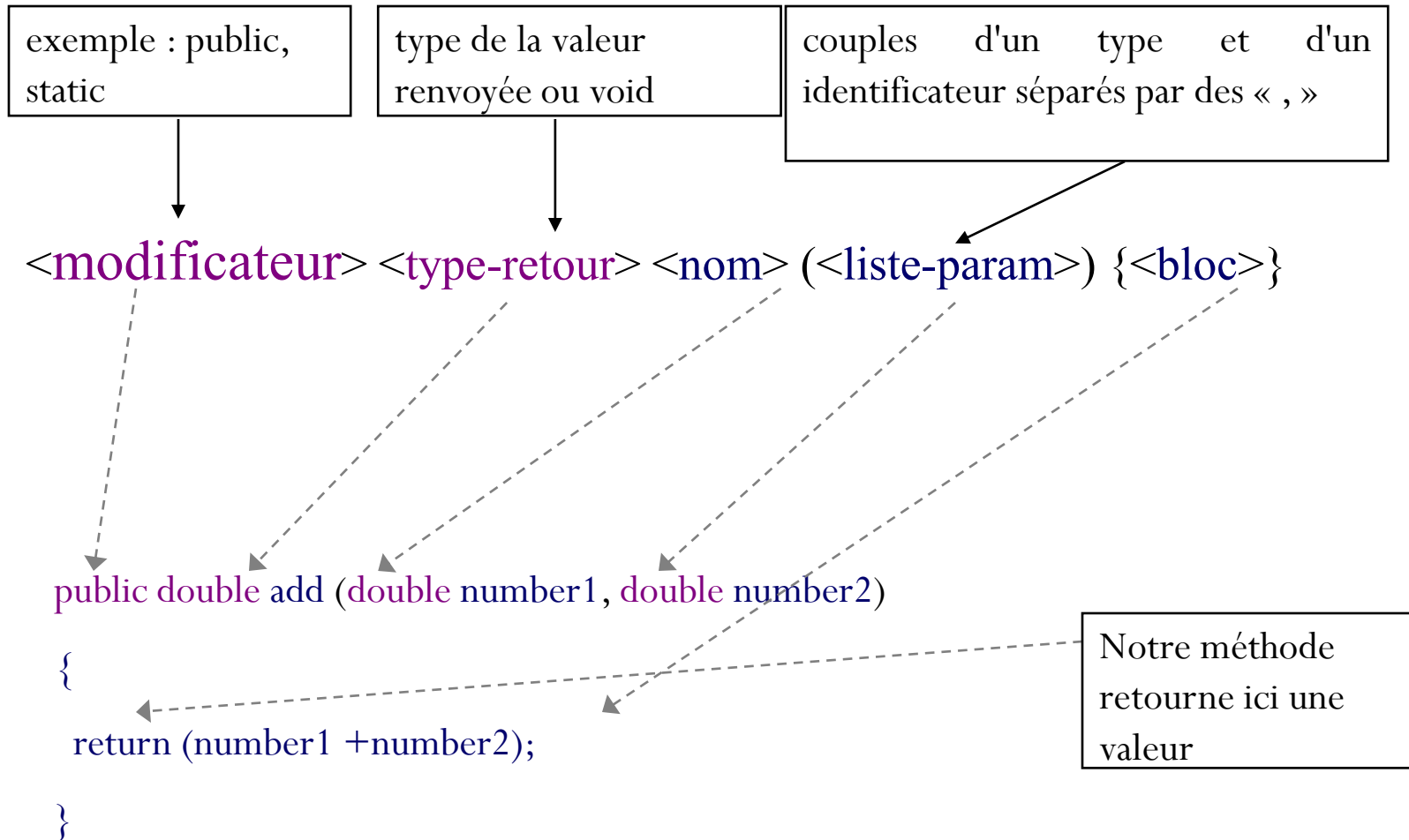
- Proches de la notion de procédure ou de fonction dans les langages procéduraux.
- La méthode c'est avant tout le regroupement d'un ensemble d'instructions suffisamment générique pour pouvoir être réutilisées
- Comme pour les procédures ou les fonctions (au sens mathématiques) on peut passer des paramètres aux méthodes et ces dernières peuvent renvoyer des valeurs (grâce au mot clé `return`).

# Mode de passage des paramètres

- Java n'implémente qu'un seul mode de passage des paramètres à une méthode : le passage par valeur.
- Conséquences :
  - l'argument passé à une méthode ne peut être modifié,
  - si l'argument est une instance, c'est sa référence qui est passée par valeur. Ainsi, le contenu de l'objet peut être modifié, mais pas la référence elle-même.



# Notion de méthodes et de paramètres (2)



# Portée des variables (1)

- Les variables sont connues et ne sont connues qu'à l'intérieur du bloc dans lequel elles sont déclarées

```
public class Bidule
{
    int i, j, k;
    public void truc(int z)
    {
        int j, r;
        r = z;
    }
    public void machin()
    {
        k = r;
    }
}
```

Ce sont 2 variables différentes

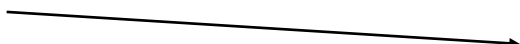
k est connu au niveau de la méthode machin() car déclaré dans le bloc de la classe. Par contre r n'est pas défini pour machin(). On obtient une erreur à la compilation

# Portée des variables (2)

- En cas de conflit de nom entre des variables locales et des variables globales, c'est toujours la variable locale qui est considérée comme étant la variable désignée par cette partie du programme
  - Attention par conséquent à ce type de conflit quand on manipule des variables *globales*.

C'est le j défini en local  
qui est utilisé dans  
la méthode truc()

```
public class Bidule
{
    int i, k, j;
    public void truc(int z)
    {
        int j,r;
        j = z;
    }
}
```



# Destruction d'objets (1)

- Java n'a pas repris à son compte la notion de destructeur telle qu'elle existe en C++ par exemple.
- C'est le ramasse-miettes (ou Garbage Collector - GC en anglais) qui s'occupe de collecter les objets qui ne sont plus référencés.
- Le ramasse-miettes fonctionne en permanence dans un thread de faible priorité (en « *tâche de fond* »). Il est basé sur le principe du compteur de références.

# Destruction d'objets (2)

- Ainsi, le programmeur n'a plus à gérer directement la destruction des objets, ce qui était une importante source d'erreurs (on parlait de fuite de mémoire ou « *memory leak* » en anglais).
- Le ramasse-miettes peut être "désactivé" en lançant l'interpréteur `java` avec l'option `-noasyncgc`.
- Inversement, le ramasse-miettes peut être lancé par une application avec l'appel `System.gc();`

# Destruction d'objets (3)

- Il est possible au programmeur d'indiquer ce qu'il faut faire juste avant de détruire un objet.
- C'est le but de la méthode `finalize()` de l'objet.
- Cette méthode est utile, par exemple, pour :
  - fermer une base de données,
  - fermer un fichier,
  - couper une connexion réseau,
  - etc.

# L'encapsulation (1)

- Notion d'encapsulation :
  - les données et les procédures qui les manipulent sont regroupées dans une même entité, l'objet.
  - Les détails d'implémentation sont cachés, le monde extérieur n'ayant accès aux données que par l'intermédiaire d'un ensemble d'opérations constituant l'**interface** de l'objet.
  - Le programmeur n'a pas à se soucier de la représentation physique des entités utilisées et peut raisonner en termes d'abstractions.

# L'encapsulation (2)

- Programmation dirigée par les données :
  - pour développer une application, le programmeur commence par définir les classes d'objets appropriées, avec leurs opérations spécifiques.
  - chaque entité manipulée dans le programme est un représentant (ou instance) d'une de ces classes.
  - L'univers de l'application est par conséquent composé d'un ensemble d'objets qui détiennent, chacun pour sa part, les clés de leur comportement.



# Contrôle d'accès (1)

- Chaque attribut et chaque méthode d'une classe peut être :
  - visible depuis les instances de toutes les classes d'une application. En d'autres termes, son nom peut être utilisé dans l'écriture d'une méthode de ces classes. Il est alors **public**.
  - visible uniquement depuis les instances de sa classe. En d'autres termes, son nom peut être utilisé uniquement dans l'écriture d'une méthode de sa classe. Il est alors **privé**.
- Les mots réservés sont :
  - **public**
  - **private**

# Contrôle d'accès (2)

- En toute rigueur, il faudrait toujours que :
  - les attributs ne soient pas visibles,
    - Les attributs ne devraient pouvoir être lus ou modifiés que par l'intermédiaire de méthodes prenant en charge les vérifications et effets de bord éventuels.
  - les méthodes "utilitaires" ne soient pas visibles,
  - seules les fonctionnalités de l'objet, destinées à être utilisées par d'autres objets soient visibles.
  - C'est la notion d'encapsulation

# Contrôle d'accès (3)

```
public class Parallelogramme
{
    private int longueur = 0; // déclaration + initialisation explicite
    private int largeur = 0; // déclaration + initialisation explicite
    public int profondeur = 0; // déclaration + initialisation explicite
    public void affiche ( )
    {System.out.println("Longueur= " + longueur + " Largeur = " + largeur +
        " Profondeur = " + profondeur);
    }}
}
```

```
public class ProgPpal
{
    public static void main(String args[])
    {
        Parallelogramme p1 = new Parallelogramme();
        p1.longueur = 5; // Invalide car l'attribut est privé
        p1.profondueur = 4; // OK
        p1.affiche( ); // OK
    } }
}
```

# Attributs de classe (1)

- Il peut s'avérer nécessaire de définir un attribut dont la valeur soit partagée par toutes les instances d'une classe. On parle de variable de classe.
- Ces variables sont, de plus, stockées une seule fois pour toutes les instances d'une classe.
- **Accès :**
  - depuis une méthode de la classe comme pour tout autre attribut,
  - via une instance de la classe,
  - à l'aide du nom de la classe.

# Attributs de classe (2)

```
public class UneClasse
{
    public static int compteur = 0;
    public UneClasse ()
    {
        compteur++;
    }
}
```

Variable de classe


Utilisation de la variable de classe  
compteur dans le constructeur de  
la classe

```
public class AutreClasse
{
    public void uneMethode()
    {
        int i = UneClasse.compteur;
    }
}
```

Utilisation de la variable de classe  
compteur dans une autre classe

# Méthodes static de classe (2)

```
public class UneClasse
{
    public int unAttribut;
    public static void main(String args[])
    {
        unAttribut = 5; // Erreur de compilation
    }
}
```



La méthode main est une méthode de classe donc elle ne peut accéder à un attribut non lui-même attribut static de classe

Autres exemples de méthodes de classe courantes

Math.sin(x);

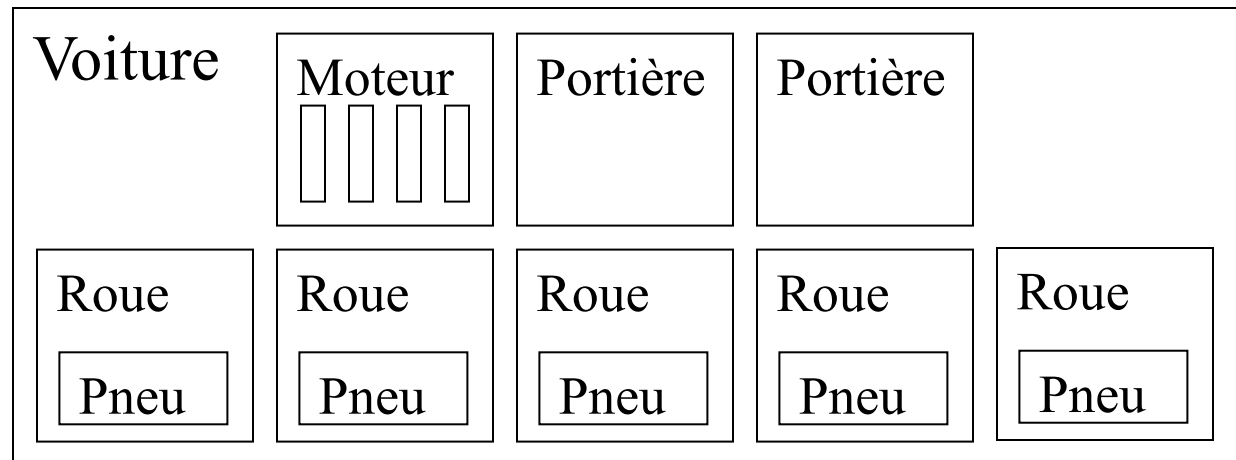
String String.valueOf(i);

# Composition d'objets (1)

- Un objet peut être composé à partir d'autres objets

Exemple : Une voiture composée de

- 4 roues ,chacune composée
  - d'un pneu
- d'un moteur composé
  - de plusieurs cylindres
- de portières
- etc...



Chaque composant est un attribut de l'objet composé

# Composition d'objets (2)

## Syntaxe de composition d'objets

```
class Pneu {  
    private float pression ;  
    void gonfler();  
    void degonfler();}
```

```
class Roue {  
    private float diametre;  
    Pneu pneu ;}
```

```
class Voiture {  
    Roue roueAVG, roueAVD, roueARG, roueARD;  
    Portiere portiereG, portiereD;  
    Moteur moteur;}
```



# Composition d'objets (3)

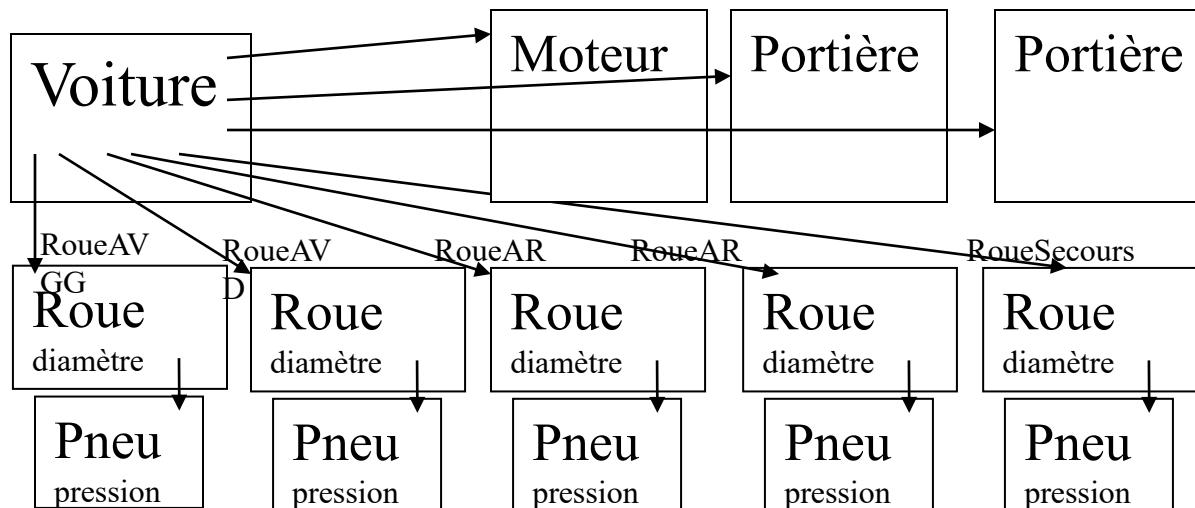
- Généralement, le constructeur d'un objet composé doit appeler le constructeur de ses composants

```
public Roue () {  
    pneu = new Pneu();}
```

```
public Voiture () {  
    roueAVG = new Roue();  
    roueAVD = new Roue();  
    roueARG = new Roue();  
    roueARD = new Roue();  
    portiereG = new Portiere();  
    portiereD = new Portiere();  
    moteur = new Moteur();}
```

# Composition d'objets (4)

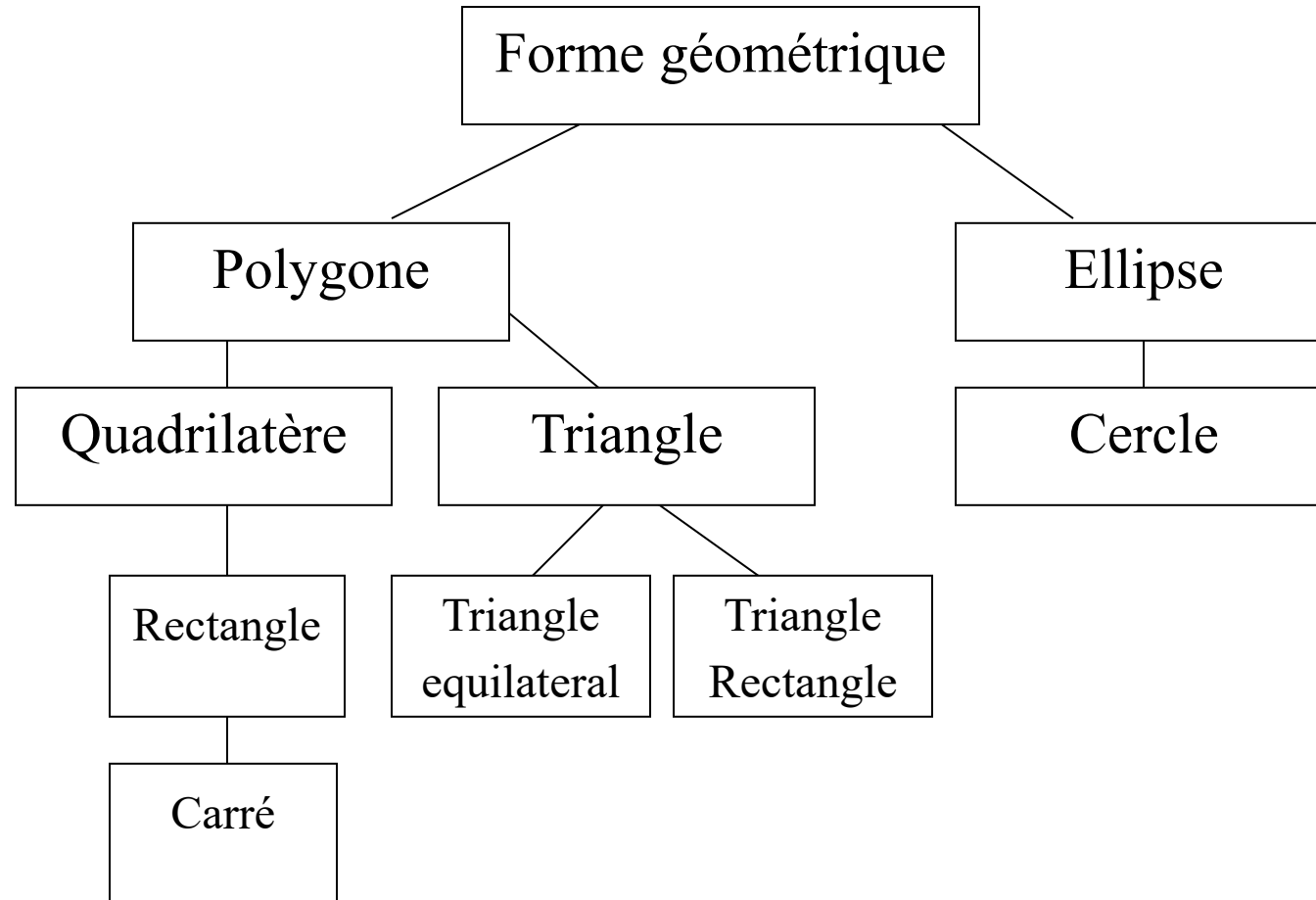
- L'instanciation d'un objet composé instancie ainsi tous les objets qui le composent



# L'héritage (1) : Concept

- La modélisation du monde réel nécessite une classification des objets qui le composent
- Classification = distribution systématique en catégories selon des critères précis
- Classification = hiérarchie de classes
- Exemples :
  - classification des éléments chimiques
  - classification des êtres vivants

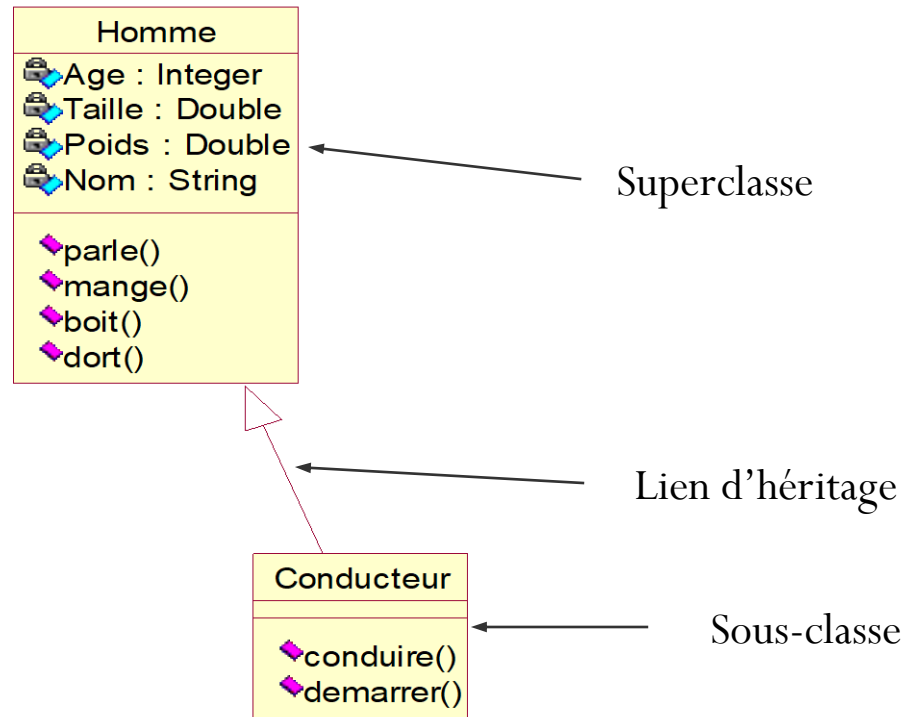
# L'héritage (2) : exemple



# L'héritage (3) : définition

- **Héritage** : mécanisme permettant le partage et la réutilisation de propriétés entre les objets. La relation d'héritage est une relation de généralisation / spécialisation.
- La classe parente est la **superclasse**.
- La classe qui hérite est la **sous-classe**.

# L'héritage (3) : représentation graphique



Représentation avec UML d'un héritage (simple)

# L'héritage avec Java (1)

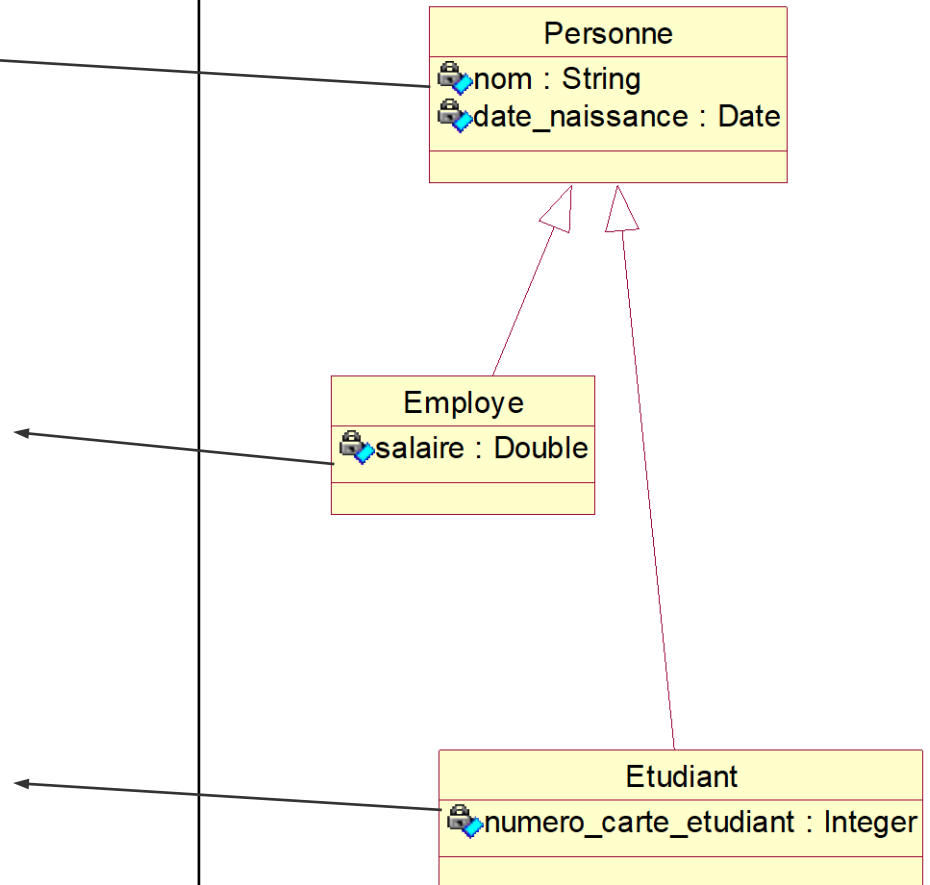
- Java implémente le mécanisme d'héritage simple qui permet de "factoriser" l'information grâce à une relation de généralisation / spécialisation entre deux classes.
- Pour le programmeur, il s'agit d'indiquer, dans la sous-classe, le nom de la superclasse dont elle hérite.
- Par défaut toutes classes Java hérite de la classe **Object**
- L'héritage multiple n'existe pas en Java.
- Mot réservé : **extends**

# L'héritage avec Java (2)

```
class Personne
{
    private String nom;
    private Date date_naissance;
    // ...
}

class Employe extends Personne
{
    private float salaire;
    // ...
}

class Etudiant extends Personne
{
    private int numero_carte_etudiant;
    // ...
}
```





# L'héritage en Java (3)

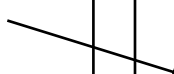
- Constructeurs et héritage
  - par défaut le constructeur d'une sous-classe appelle le constructeur "par défaut" (celui qui ne reçoit pas de paramètres) de la superclasse.
  - Pour forcer l'appel d'un constructeur précis, on utilisera le mot réservé **super**. Cet appel devra être la **première instruction** du constructeur.

# L'héritage en Java (4)

```
public class Employe extends Personne
{
    public Employe () {}
    public Employe (String nom,
                    String prenom,
                    int anNaissance)
    {
        super(nom, prenom, anNaissance);
    }
}
```

Appel explicite à ce constructeur  
avec le mot clé super

```
public class Personne
{
    private String nom;
    private String prenom;
    private int anNaissance;
    public Personne()
    {
        nom=""; prenom="";
    }
    public Personne(String nom,
                    String prenom,
                    int anNaissance)
    {
        this.nom=nom;
        this.prenom=prenom;
        this.anNaissance=anNaissance;
    }
}
```



# L'héritage en Java (5)

```
public class Personne
{
    private String nom, prenom;
    private int anNaissance;
    public Personne()
    {
        nom=""; prenom="";
    }
    public Personne(String nom,
                    String prenom,
                    int anNaissance)
    {
        this.nom=nom;
        this.prenom=prenom;
        this.anNaissance=anNaissance;
    }
}
```

```
public class Object
{
    public Object()
    {
        ... / ...
    }
}
```

Appel par défaut dans le constructeur de Personne au constructeur par défaut de la superclasse de Personne, qui est Object

# Redéfinition de méthodes

- Une sous-classe peut redéfinir des méthodes de ses superclasses (directe ou indirectes), à des fins de spécialisation.
  - Le terme anglophone est "overriding".
  - La méthode redéfinie **doit avoir la même signature**.

```
class Employe extends Personne
{
    private float salaire;
    public calculePrime( )
    {
        // ...
    }
}
```

redéfinition

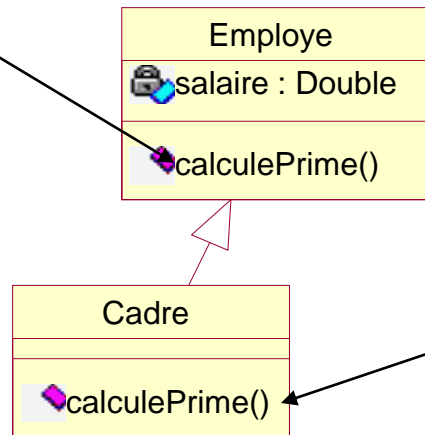
```
class Cadre extends Employe
{
    public calculePrime()
    {
        // ...
    }
    // ...
}
```

# Recherche dynamique des méthodes (1)

- Le polymorphisme
  - Capacité pour une entité de prendre plusieurs formes.
  - En Java, toute variable désignant un objet est potentiellement polymorphe, à cause de l'héritage.
  - Polymorphisme dit « d'héritage »
- le mécanisme de "lookup" dynamique :
  - déclenchement de la méthode la plus spécifique d'un objet, c'est-à-dire celle correspondant au type réel de l'objet, déterminé à l'exécution uniquement (et non le type de la référence, seul type connu à la compilation, qui peut être plus générique).
  - Cette dynamicité permet d'écrire du code plus générique.

# Recherche dynamique des méthodes (2)

```
Employe jean = new Employe();  
jean.calculerPrime();
```



```
Employe jean = new Cadre();  
jean.calculerPrime();
```

# Surcharge de méthodes (1)

- Dans une même classe, plusieurs méthodes peuvent posséder le même nom, pourvu qu'elles diffèrent en nombre et/ou type de paramètres.
  - On parle de surdéfinition ou surcharge, on encore en anglais d'overloading en anglais.
  - Très souvent les constructeurs sont surchargés (plusieurs constructeurs prenant des paramètres différents et initialisant de manières différentes les objets)

# Opérateur instanceof

- L'opérateur **instanceof** confère aux instances une capacité d'introspection : il permet de savoir si une instance est instance d'une classe donnée.
  - Renvoie une valeur booléenne

```
if ( ... )  
    Personne jean = new Etudiant();  
else  
    Personne jean = new Employe();  
  
// ...  
  
if (jean instanceof Employe)  
    // discuter affaires  
else  
    // proposer un stage
```



# Forçage de type / transtypage (1)

- Lorsqu'une référence du type d'une classe désigne une instance d'une sous-classe, il est nécessaire de forcer le type de la référence pour accéder aux attributs spécifiques à la sous-classe.
- Si ce n'est pas fait, le compilateur ne peut déterminer le type réel de l'instance, ce qui provoque une erreur de compilation.
- On utilise également le terme de transtypage
- Similaire au « cast » en C

# Forçage de type / transtypage (2)

```
class Personne
{
    private String nom;
    private Date date_naissance;
    // ...
}
```

```
class Employe extends Personne
{
    public float salaire;
    // ...
}
```

```
Personne jean = new Employe ();
float i = jean.salaire; // Erreur de compilation
float j = (Employe) jean .salaire; // OK
```

A ce niveau pour le compilateur dans la variable « jean » c'est un objet de la classe Personne, donc qui n'a pas d'attribut « salaire »

On « force » le type de la variable « jean » pour pouvoir accéder à l'attribut « salaire ». On peut le faire car c'est bien un objet Employe qui est dans cette variable

# L'autoréférence : this (1)

- Le mot réservé **this**, désigne la référence de l'instance à laquelle le message a été envoyée (donc celle sur laquelle la méthode est « exécutée »).
- Il est utilisé principalement :
  - lorsqu'une référence à l'instance courante doit être passée en paramètre à une méthode,
  - pour lever une ambiguïté,
  - dans un constructeur, pour appeler un autre constructeur de la même classe.

# L'autoréférence : this (2)

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

Pour lever l'ambiguïté sur le mot « nom »  
et déterminer si c'est le nom du paramètre  
ou de l'attribut

```
public MaClasse(int a, int b) {...}
public MaClasse (int c)
{
    this(c,0);
}
public MaClasse ()
{
    this(10);
}
```

Appelle le constructeur  
MaClasse(int a, int b)

Appelle le constructeur  
MaClasse(int c)

# Référence à la superclasse

- Le mot réservé **super** permet de faire référence au constructeur de la superclasse directe mais aussi à d'autres informations provenant de cette superclasse.

```
class Employe extends Personne
{
    private float salaire;
    public float calculePrime()
    {
        return (salaire * 0,05);
    }
    // ...
}
```

Appel à la méthode calculPrime()  
de la superclasse de Cadre

```
class Cadre extends Employe
{
    public float calculePrime()
    {
        return (super.calculePrime() / 2);
    }
    // ...
}
```

# Classes abstraites (1)

- Il peut être nécessaire au programmeur de créer une classe déclarant une méthode sans la définir (c'est-à-dire sans en donner le code). La définition du code est dans ce cas laissée aux sous-classes.
- Une telle classe est appelée classe abstraite.
- Elle doit être marquée avec le mot réservé **abstract**.
- Toutes les méthodes de cette classe qui ne sont pas définies doivent elles aussi être marquées par le mot réservé **abstract**.
- Une classe abstraite ne peut pas être instanciée.

# Classes abstraites (2)

- Par contre, il est possible de déclarer et d'utiliser des variables du type de la classe abstraite.
- Si une sous-classe d'une classe abstraite ne définit pas toutes les méthodes abstraites de ses superclasses, elle est abstraite elle aussi.

```
public abstract class Polygone
{
    private int nombreCotes = 3;
    public abstract void dessine (); // methode non définie
    public int getNombreCotes()
    {
        return(nombreCotes);
    }
}
```