

Programmation orienté objet JAVA

Première partie

JAVA
Programmation orientée objet

Hafidi Imad

imad.hafidi@gmail.com

Approche Objet

Exemple

- Nombres complexes
 - En C, faire une procédure permettant de faire la somme de 2 nombres complexes

```
void Somme(double a1, double b1,  
           double a2, double b2, double *a, double *b)  
{  
    *a = a1 + a2;  
    *b = b1 + b2;  
}
```

$$z_1 = 1 + j$$
$$z_2 = 2 - 4j$$

```
int main(void)  
{  
    double re, im;  
    Somme( 1, 1, 2, -4, &re, &im);  
    printf(« %lf +j %lf », re, im);  
    return 0;  
}
```

Exemple

- Nombres complexes, C avec structure
 - En C, faire une procédure permettant de faire la somme de 2 nombres complexes

```
typedef struct
{ double Reel;
  double Imag;
} Complexe;
```



Complexe est maintenant un nouveau type

```
void Somme(Complexe z1, Complexe z2, Complexe *z3)
{
    z3->Reel = z1.Reel + z2.Reel;
    z3->Imag = z1.Imag + z2.Imag;
}
```

```
void main()
{ Complexe z1, z2, z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  Somme( z1, z2, &z3);
}
```

Exemple

- Nombres complexes, C avec structure (*bis*)
 - En C, faire une procédure permettant de faire la somme de 2 nombres complexes

```
typedef struct
{ double Reel;
  double Imag;
} Complexe;
```



Complexe est maintenant un nouveau type

```
Complexe Somme(Complexe z1, Complexe z2)
{
  Complexe s;
  s.Reel = z1.Reel + z2.Reel;
  s.Imag = z1.Imag + z2.Imag;
  return s;
}
```

```
void main()
{ Complexe z1, z2, z3;
  z1.Reel = 1; z1.Imag = 1;
  z2.Reel = 2; z2.Imag = -4;
  z3 = Somme( z1, z2);
}
```

Exemple

- Nombres complexes, du C au JAVA

Langage C

Données

```
typedef struct  
{ double Reel;  
  double Imag;  
} Complexe;
```

Fonctions

```
Complexe Somme(Complexe z1, Complexe z2)  
{  
  Complexe s;  
  s.Reel = z1.Reel + z2.Reel;  
  s.Imag = z1.Imag + z2.Imag;  
  return s;  
}
```

Langage JAVA

```
class Complexe  
{
```

```
  double Reel;  
  double Imag;
```

Champs

```
  Complexe() {Reel=0;Imag=0;}  
  Complexe Plus(Complexe z)  
  {  
    Complexe s;  
    s.Reel = Reel + z.Reel;  
    s.Imag = Imag + z.Imag;  
    return s;  
  }  
};
```

Méthodes

constructeur

L'approche objet (1)

- L'approche objet :
 - **Programmation dirigé par les données** et non par les traitements
 - les procédures existent toujours mais on se concentre d'abord sur les entités que l'on va manipuler avant de se concentrer sur la façon dont on va les manipuler
 - Notion d'encapsulation
 - les données et les procédures qui les manipulent (on parle de méthodes) sont regroupés dans une même entité (la classe).

L'approche objet (2)

- L'approche objet (suite):
 - Notion de classe et d'objets : la **classe est un modèle** décrivant les caractéristiques communes et le comportement d'un ensemble d'objets (on parle d'instance d'une classe) : la classe est un moule et l'objet est ce qui est moulé à partir de ce moule
 - Ces caractéristiques communes sont les variables (on parle d'attributs) qui caractérisent les objets et les opérations que l'on peut effectuer sur ces objets

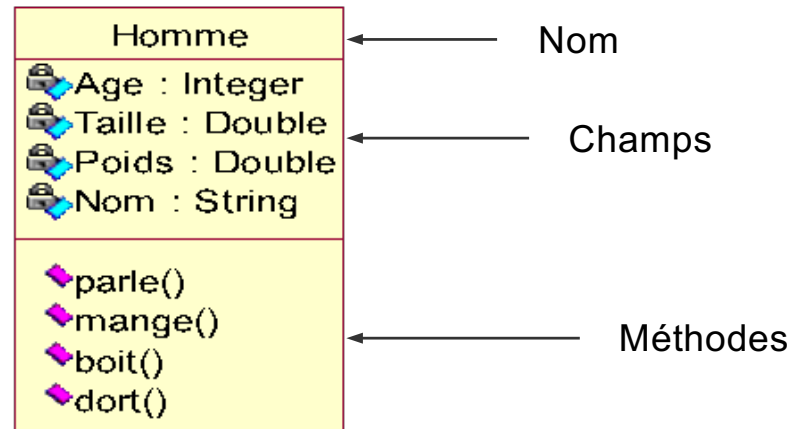
Pourquoi la programmation objet

- Abstraction
 - Séparation entre la définition et son implantation
 - Réutilisation de code car basé sur la définition
- Unification
 - Les données et le code sont unifiés en un seul modèle
- Réutilisation
 - La conception par classe conduit à des composant réutilisable
 - Cache les détails d'implantation
- Spécialisation (peu vrai dans la vrai vie)
 - Le mécanisme d'héritage permet une spécialisation pour des cas particulier

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 champs 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)

Les attributs

- Une classe définie de la structure d'un objet en mémoire
- Une classe des champs (ou attributs) dont le type indique la façon dont la mémoire est structurée
- Contrairement au C, en Java, l'ordre des champs en mémoire n'est pas l'ordre des champs lors de la déclaration

```
class Point {  
    int x;  
    int y;  
}
```

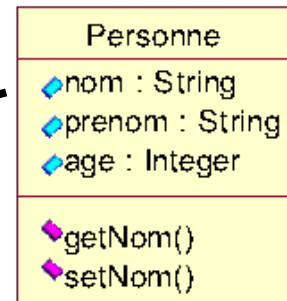


champs

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 p = new Personne()
        p.nom = "jabri" ;
        p.prenom = "ahmed" ;
    }
}
```

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.

Variables 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.
- Mot réservé : **static**
- 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.

Variables de classe (2)

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

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

Variable de classe

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


Utilisation de la variable de classe
compteur dans une autre classe

Les méthodes

- En plus de définir les champs, une classe définit le code qui va pouvoir manipuler les champs dans des méthodes
- Une méthode est une fonction liée à une classe

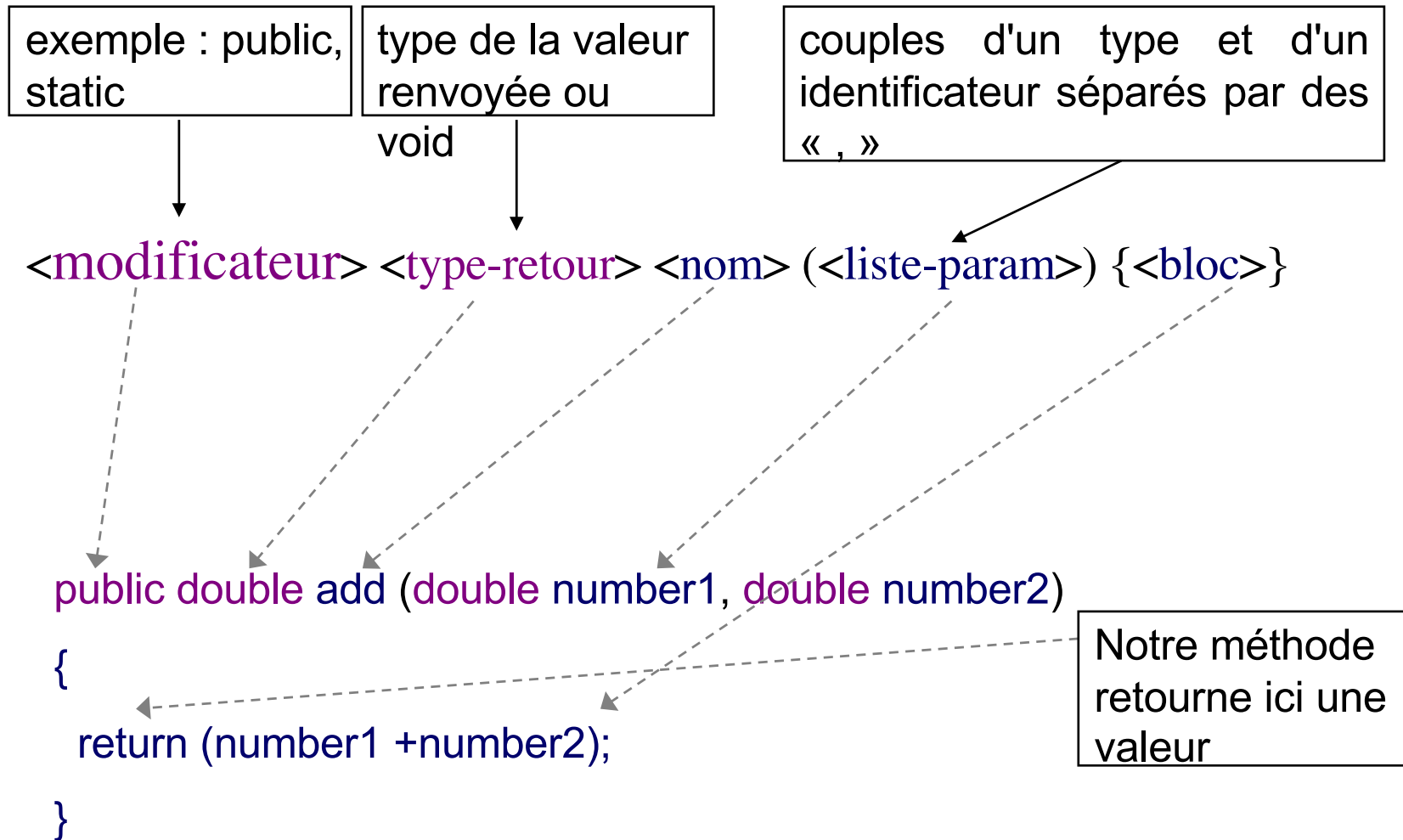
```
class Point {  
    int x, y;  
  
    double distance() {  
        return Math.sqrt(x * x + y * y);  
    }  
}
```

méthode



-
- En Java, le code est toujours dans une méthode
 - En java on a 2 grands types de méthodes les méthodes de classe (défini avec le mot clé `static` comme pour la méthode `main`) et les méthodes d'instance :
 - une méthode est un message que l'on envoie à une classe (méthode de classe) ou à un objet (méthode d'instance)

méthodes



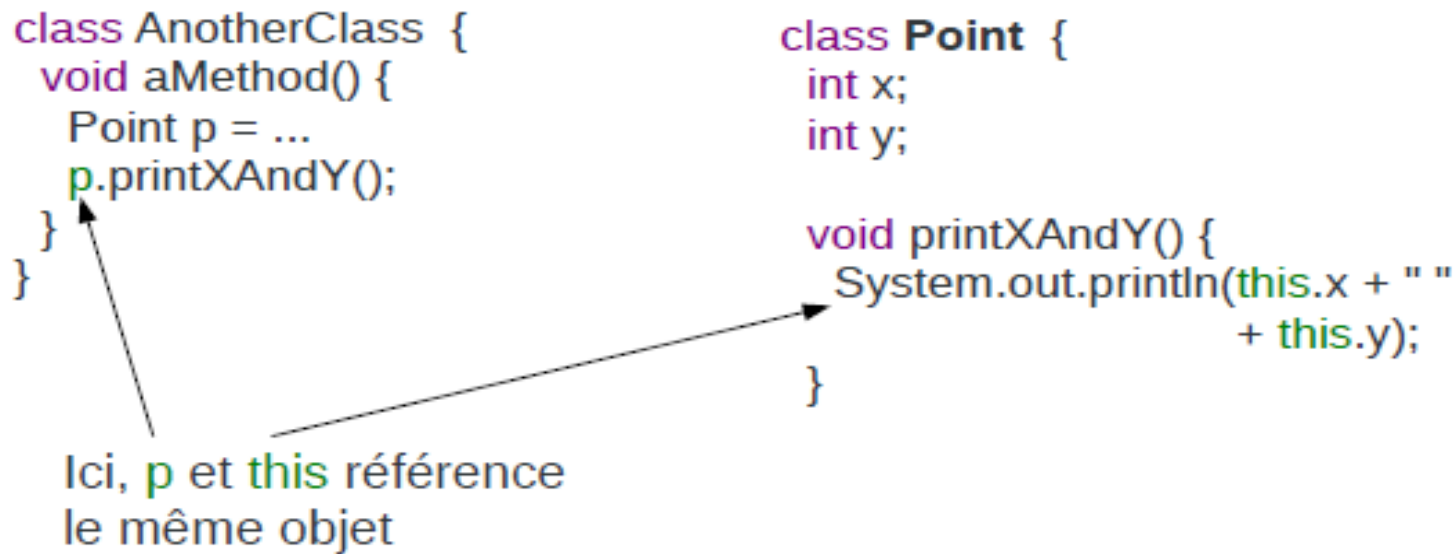
Méthode & fonction

- Comme une méthode est liée à une classe, il n'est pas possible d'appeler une méthode sans envoyé un objet de cette classe
- Une méthode est une fonction avec un paramètre caché

```
class AnotherClass {  
    void aMethod() {  
        Point p = ...  
        p.printXAndY();  
    }  
}
```

```
class Point {  
    int x;  
    int y;  
  
    void printXAndY() {  
        System.out.println(this.x + " "  
                             + this.y);  
    }  
}
```

Ici, **p** et **this** référence
le même objet



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.

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 la plus 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;
    }
}
```



Méthodes de classe (1)

- Il peut être nécessaire de disposer d'une méthode qui puisse être appelée sans instance de la classe. C'est une méthode de classe.
- On utilise là aussi le mot réservé **static**
- Puisqu'une méthode de classe peut être appelée sans même qu'il n'existe d'instance, une méthode de classe ne peut pas accéder à des attributs non statiques. Elle ne peut accéder qu'à ses propres variables et à des variables de classe.

Méthodes 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 de classe

Autres exemples de méthodes de classe courantes

```
Math.sin(x);
```

```
String String.valueOf (i);
```

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 qui vont interagir. 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
 - représentant physique d'une 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
- 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, s'il y a lieu,
 - L'exécution d'un constructeur.
- Un constructeur est une méthode d'initialisation.

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

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

Les constructeurs (2)

- Lorsque l'initialisation explicite n'est pas possible (par exemple lorsque la valeur initiale d'un attribut est demandée dynamiquement à l'utilisateur), il est possible de réaliser l'initialisation au travers d'un constructeur.
- 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 p = new Personne()
        p.setNom(" ahmed");
    }
}
```

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(" ahmed", " jabri", 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

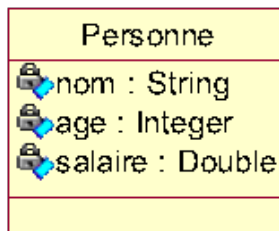
```
public class Personne
{
    public String nom;
    public String prenom;
    public int age;
    public Personne()
    {
        nom=null; prenom=null;
        age = 0;
    }
    public Personne(String unNom,
                    String unPrenom, int unAge)
    {
        nom=unNom;
        prenom=unPrenom; 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 p1, p2;
p1 = new Personne ();
p2 = 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

Ordre des membres

- En Java, une classe est analysée par le compilateur en plusieurs passes on peut donc déclarer les méthodes, les champs etc dans n'importe quel ordre
- L'ordre usuel à l'intérieur d'une classe est: champs, constructeurs, getter/setter, méthode, méthode statique

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.

Exercice : initialisation JAVA

- Créez une classe `Personne` avec les méthodes et les attributs suivants:
 - `String nom`,
 - `String Prenom`,
 - `int age`
 - `String getNom()`
 - `String getPrenom()`
 - `int getAge();`
 - `void afficher()` : Affiche les informations de la personne

Programmation Orientée Objet

- Les paradigmes « objet »
 - Encapsulation
 - Regrouper données et opérations
 - Héritage
 - Généralisation de classes
 - Polymorphisme
 - Découle de l'héritage, permet aux classes les plus générales d'utiliser les spécifications des classes plus spécifiques
 - Généricité (extension de l'encapsulation)
 - Modèle d'encapsulation, quelque soit le type des données

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 traiter 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
- Nous verrons dans la suite que l'on peut encore affiner le contrôle d'accès

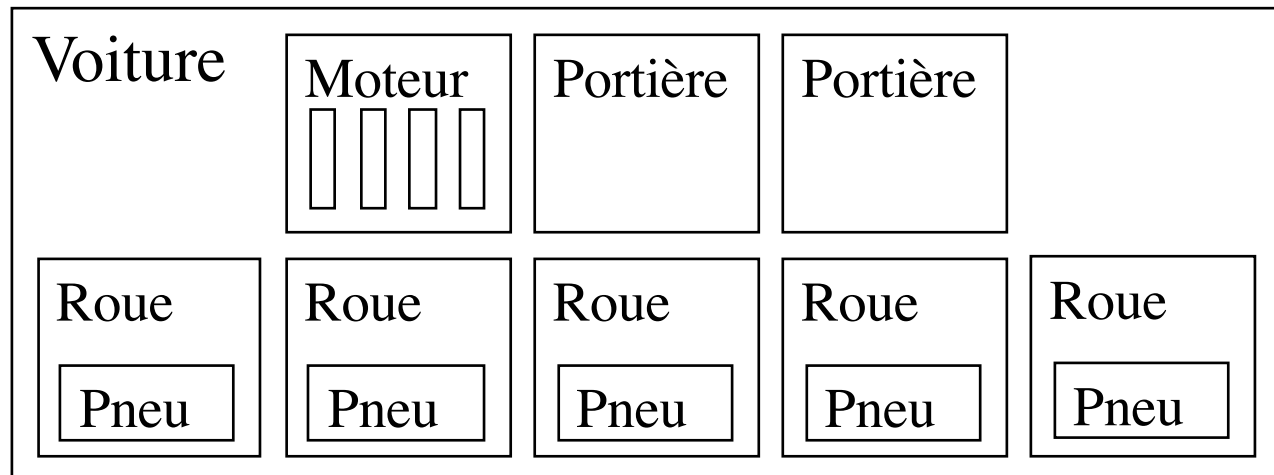
Contrôle d'accès (3)

```
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
    }
}
```

Composition d'objets (1)

- Un objet peut être composé à partir d'autres objets
Exemple : Une voiture composée de
 - 5 roues (roue de secours) 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 , roueSecours ;  
    Portiere portiereG, portiereD;  
    Moteur moteur;}
```

Composition d'objets (3)

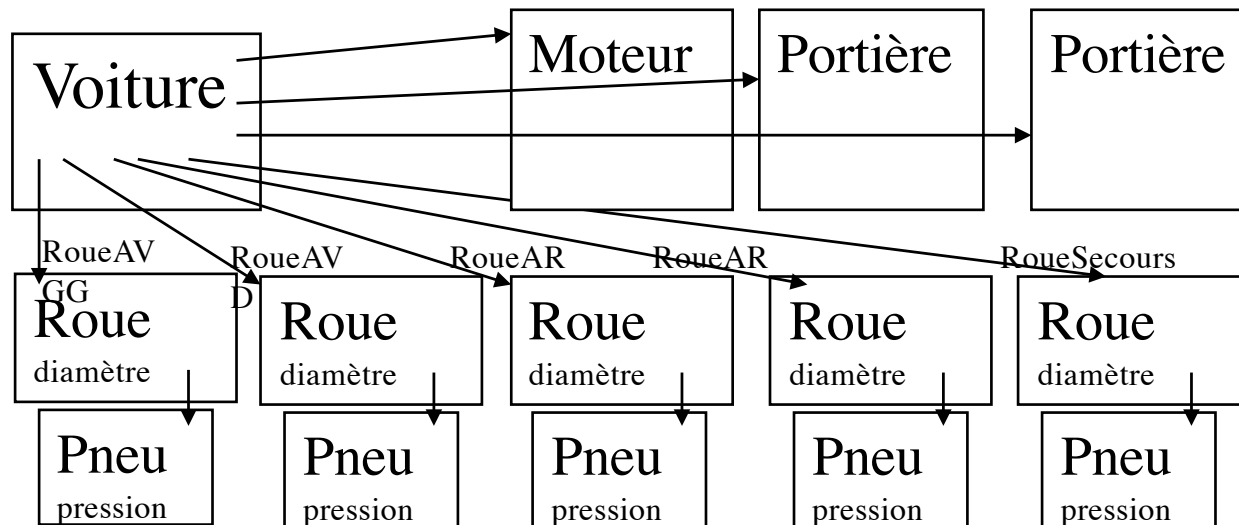
- Généralement, le constructeur d'un objet composé doit appeler le constructeur de ses composants

```
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



TP a faire

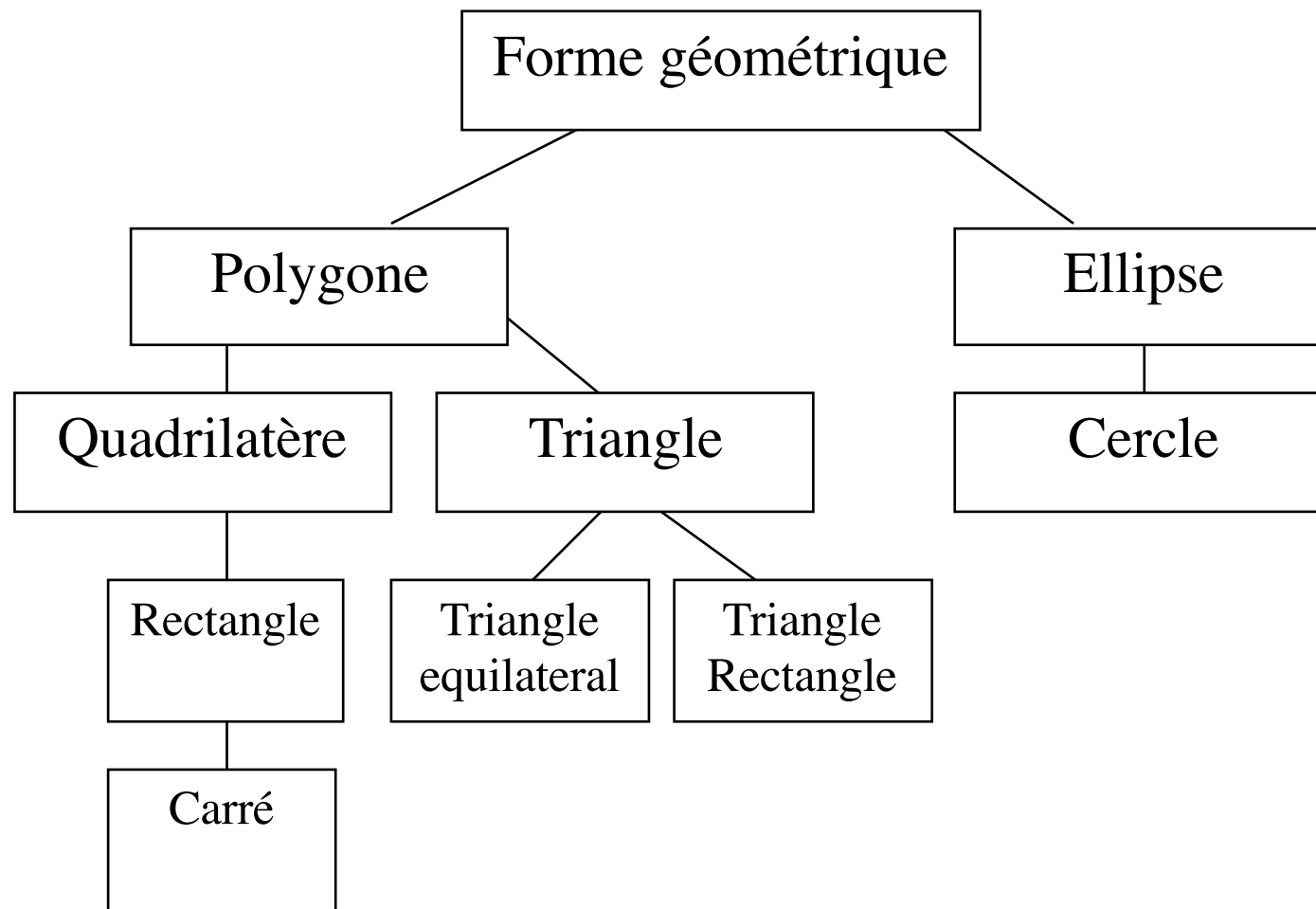
- Partie 1 : TP 1

Héritage en Java

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

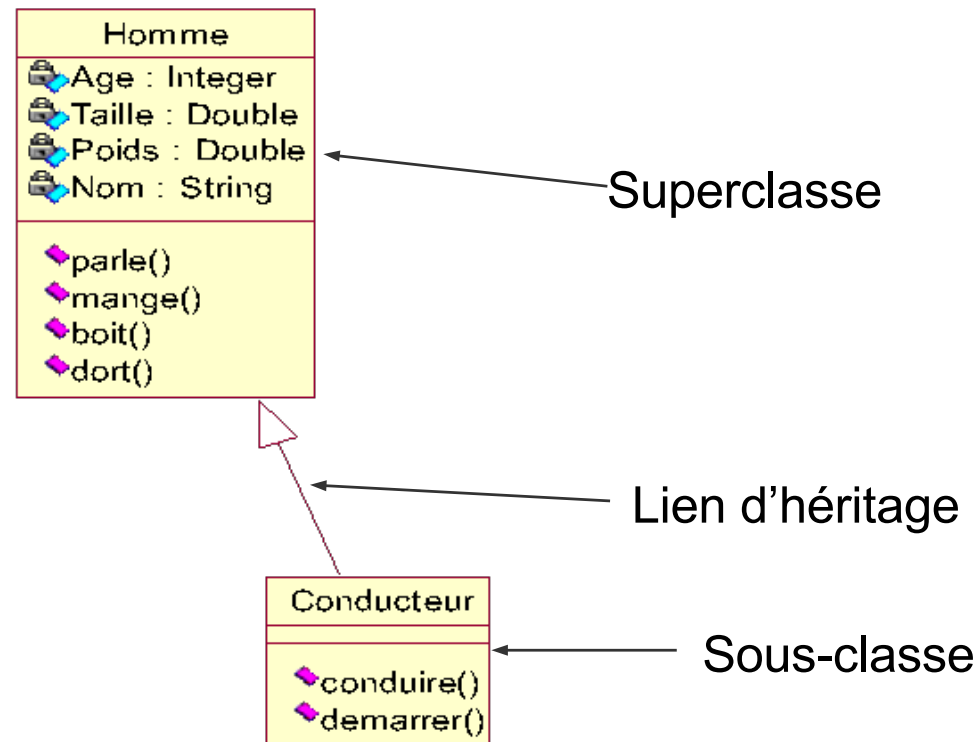
L'héritage (2) : exemple



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" de 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
```

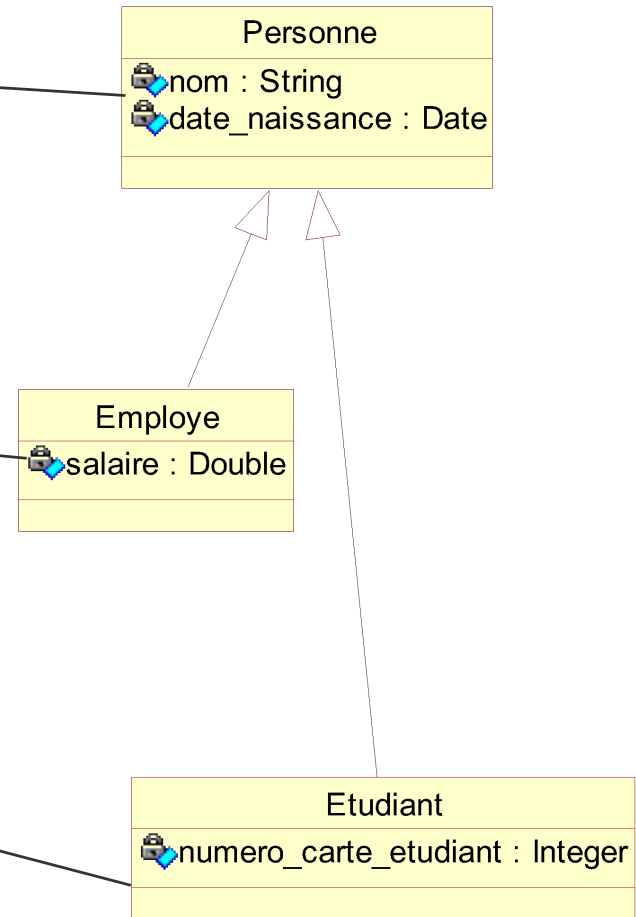
```
{  
    private String nom;  
    private Date date_naissance;  
    // ...  
}
```

```
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. Attention donc dans ce cas que le constructeur sans paramètre existe toujours dans 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
```

```
{
    public String nom, prenom;
    public 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
{
    public String nom, prenom;
    public 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 existant dans une de ses superclasses (directe ou indirectes), à des fins de spécialisation.
 - Le terme anglophone est "overriding". On parle aussi de masquage.
 - 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

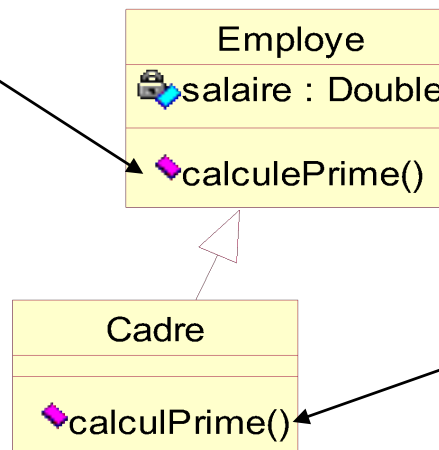
(1)

- 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 e = new Employe();  
e.calculPrime();
```



```
Employe e = new Cadre();  
e.calculPrime();
```

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.
 - Le choix de la méthode à utiliser est fonction des paramètres passés à l'appel.
 - Ce choix est réalisé de façon statique (c'est-à-dire à la compilation).
 - 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 p = new Etudiant();  
else  
    Personne p = new Employe();  
  
//...  
  
if (p instanceof Employe)  
    // discuter affaires  
else  
    // proposer un stage
```


Forçage de type / transtypage (1)

- Lorsqu'une référence du type d'une classe contient 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 p = new Employe ();
float i = p.salaire; // Erreur de compilation
float j = ( (Employe) p ).salaire; // OK
```

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

On « force » le type de la variable « p » 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**, utilisé dans une méthode, 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 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);
    }
}
```

Interfaces

Interfaces : comment classifier ?

- Java ne permet pas l'héritage multiple
- Or, il existe parfois différentes classifications possibles selon plusieurs critères



Exemple de classification

- Selon la forme Solides convexes

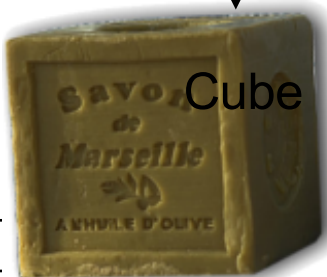
Polyèdres



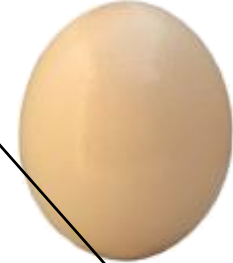
Parallélépipède



Cube



Solides de révolution



Cylindres

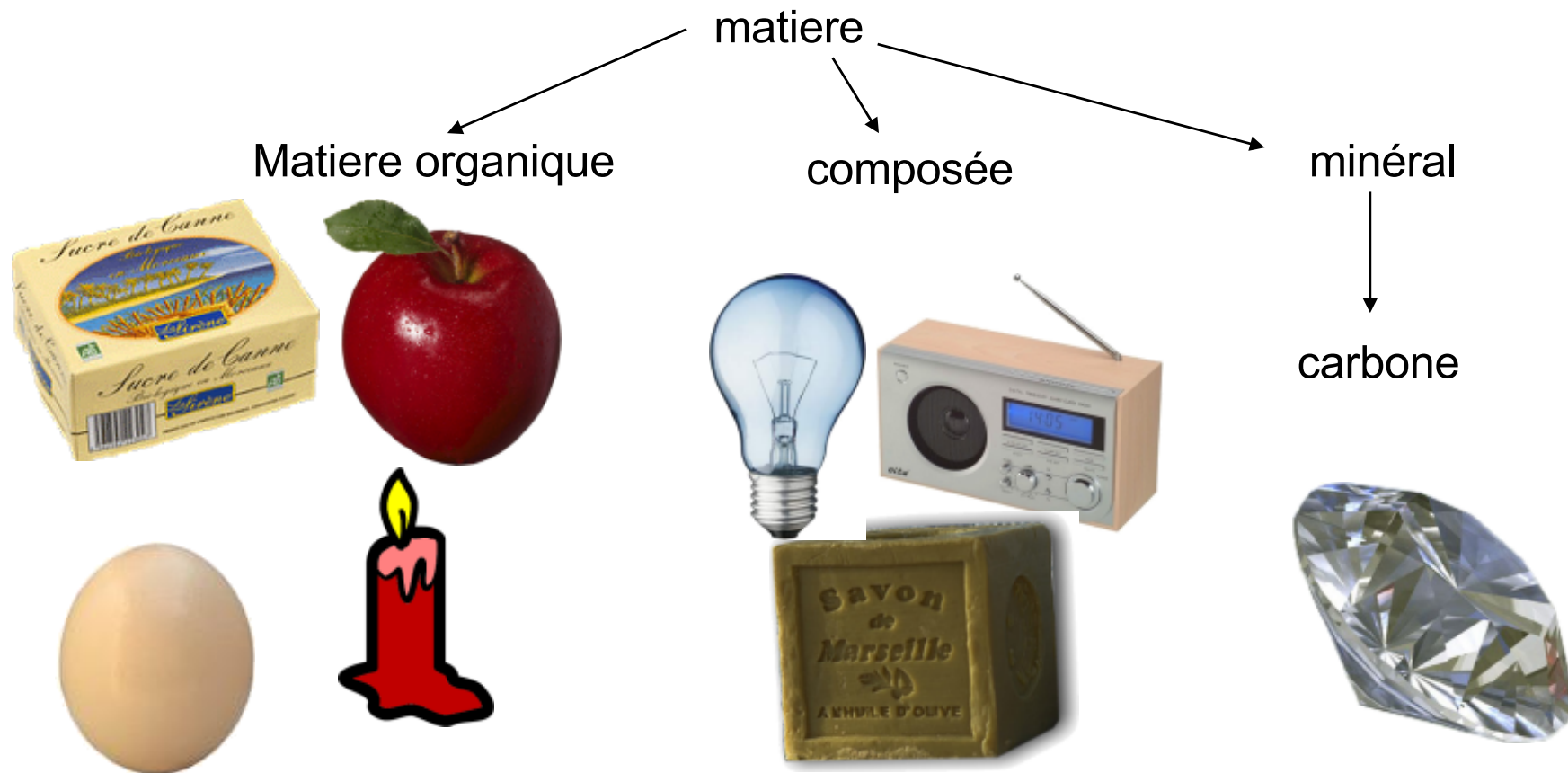


Sphères










Autre classification

- Selon la matière



Autres critères

- D'autres critères qui pourraient servir à réaliser une classification décrivent des comportements ou des capacités
 - « électrique »  
 - « comestible »   
 - « lumineux »  
- Or ces « mécanismes » peuvent être commun à différentes classes non reliées entre elles par une relation d'héritage

Notion d' « Interfaces »

- Pour définir qu'une certaine catégorie de classes doit implémenter un ensemble de méthodes, on peut regrouper les déclarations de ces méthodes dans une interface.
- Le but est de décrire le fait que de telles classes pourront ainsi être manipulées de manière identique.
- Exemple :
 - Tous les appareils électriques peuvent être allumés ou éteints
 - Tous les objets comestibles peuvent être mangés
 - Tous les objets lumineux éclairent

Définition d'Interface

- Une interface est donc la description d'un ensemble des procédures (méthodes) que les classes Java peuvent mettre en oeuvre.
- Les classes désirant appartenir à la catégorie ainsi définie
 - déclareront qu'elles implémentent cette interface,
 - fourniront le code spécifique des méthodes déclarées dans cette interface.
- Cela peut être vu comme un contrat entre la classe et l'interface
 - la classe s'engage à implémenter les méthodes définies dans l'interface

Codage d'une interface en Java

- Mot réservé : `interface`
- Dans un fichier `nom_interface.java`, on définit la liste de toutes les méthodes de l'interface

```
interface nomInterface {  
    type_retour methode1(paramètres);  
    type_retour methode2(paramètres);  
    ... }  

```

- Les méthodes d'une interface sont abstraites : elles seront écrites spécifiquement dans chaque classe implémentant l'interface
- Le modificateur **abstract** est facultatif.

Implémentation d'une interface dans une classe

- Mot réservé : `implements`
- La classe doit expliciter le code de chaque méthode définie dans l'interface

```
class MaClasse implements nomInterface
{
...
    type_retour methode1(paramètres)
    { code spécifique à la methode1 pour cette classe };
    ...
}
```


Exemple d'Interface (1)

```
interface Electrique
{
    void allumer();
    void eteindre();
}
```

```
class Radio implements Electrique
{
    // ...
    void allumer() {System.out.println(« bruit »);}
    void eteindre()
    {System.out.println(« silence »);}
}
```

```
class Ampoule implements Electrique
{
    // ...
    void allumer() {System.out.println(« j'éclaire »);}
    void eteindre() {System.out.println(« plus de lumière»);}
}
```

Exemple d'Interface (2)

```
// ...  
  
Ampoule monAmpoule = new Ampoule();  
Radio maRadio = new Radio();  
Electrique c;  
Boolean sombre;  
  
// ...  
  
if(sombre == true)  
    c = monAmpoule;  
else  
    c = maRadio;  
  
c.allumer();  
...  
c.eteindre();  
  
// ...
```

Utilisation des interfaces

- Une variable peut être définie selon le type d'une interface
- Une classe peut implémenter plusieurs interfaces différentes
- L'opérateur `instanceof` peut être utilisé sur les interfaces

Exemple :

```
interface Electrique
...
interface Lumineux
...
class Ampoule implements Electrique, Lumineux
...
Electrique e;
Object o = new Ampoule();
if (o instanceof Electrique) {e=(Electrique)o;e.allumer();}
```

Conclusion sur les interfaces

- Un moyen d'écrire du code générique
- Une solution au problème de l'héritage multiple
- Un outil de concevoir d'applications réutilisables

TP a faire

- Partie 2 : TP 1