Chapitre 11 : Programmation Orientée Objet

Analyse et conception Orientée Objets
Héritage
Polymorphisme
Classes abstraites
Interfaces

Développement Orienté Objet

- Analyse OO
- Conception OO
- Programmation OO

Analyse Orientée Objet

- Identifier les objets du domaine d'application
- Décrire les différents attributs
- Établir les relations entre les différents objets
- Former les groupes d'objets

Diagramme de classe

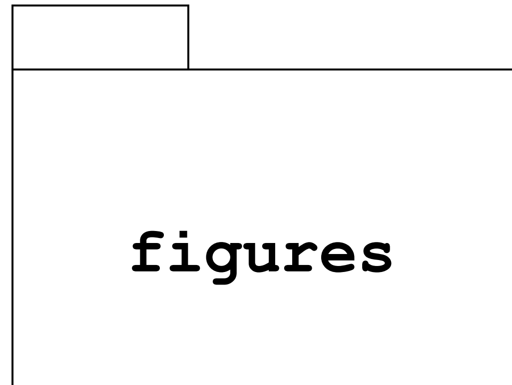
PolygoneRegulier
-nbCotes:int -longueur:double
+perimetre():double +surface():double

Diagramme d'objet

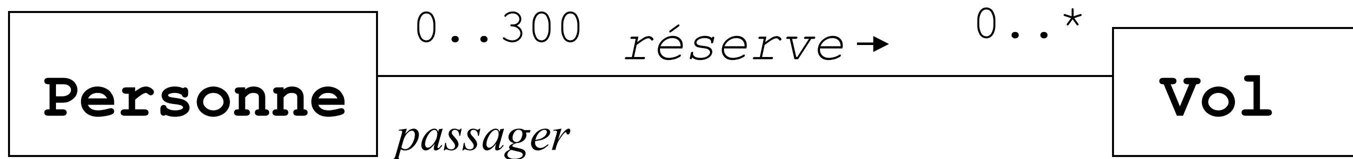
pr1:PolygoneRegulier
nbCotes=6 longueur=8.7

pr1:PolygoneRegulier
nbCotes=4 longueur=12.8

Diagramme de package



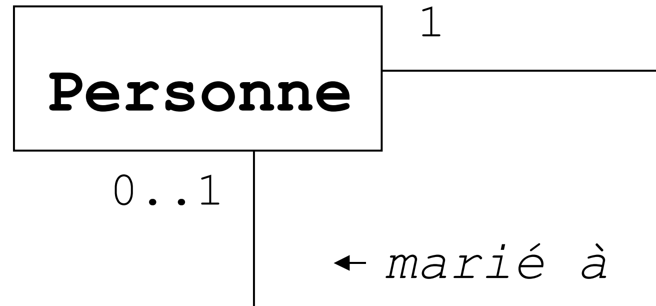
Association



Association : traduction Java

```
public class Personne
{
  private String nom;
  private Vector<Vol> lesVols;
  public Personne( String nom ){
       lesVols = new Vector<Vol>();
       this.nom = nom;
  }
  public void reserve( Vol v ){
       lesVols.addElement( v ); }
  public boolean passager( Vol v ){
       return lesVols.contains( v );}
  ...
}
```

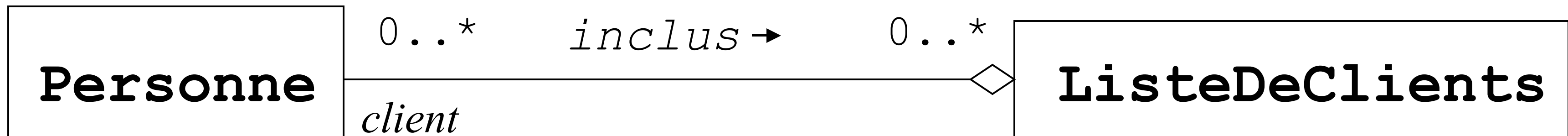
Association d'une classe sur elle-même



Association : traduction Java

```
public class Personne{
  private String nom;
  private Personne epoux = null;
  private boolean marie = false;
  public Personne( String nom ){
       this.nom = nom; }
  public void marie( Personne p ){
       epoux = p; marie = true; }
  public boolean marie(){
       return marie; }
  ...
}
```

Agrégation



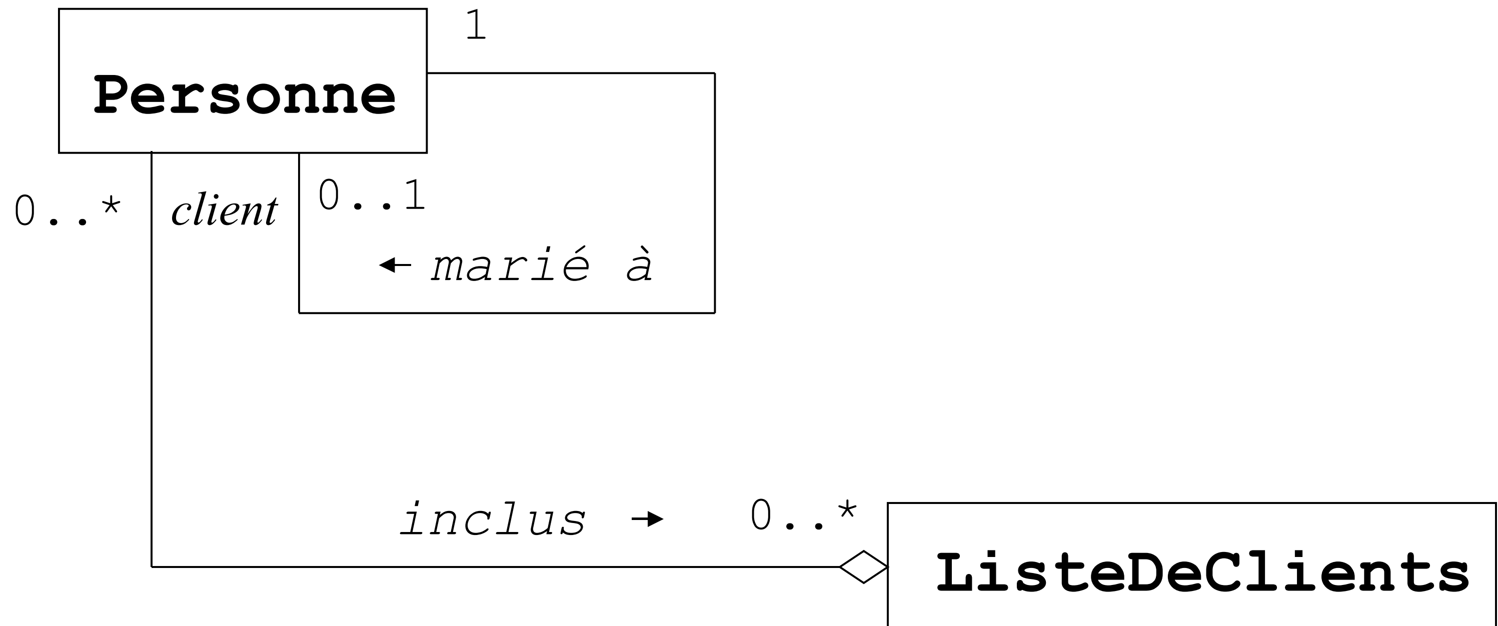
Agrégation : traduction Java

```
public class ListeClients{
  private Vector<Personne> laListe;

  public ListeClients(){
      laListe = new Vector<Personne>(); }

  public void inclus( Personne p ){
      laListe.add( p );
  }

  public Vector<Personne> laListe(){
      return laListe; }
}
```

Agrégation : traduction Java

```
public class Personne{
  private String nom;

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

  public boolean client( ListeClients<Personne> l ){
      return (l.laListe()).contains( this );
  }
}
```

Association+agrégation



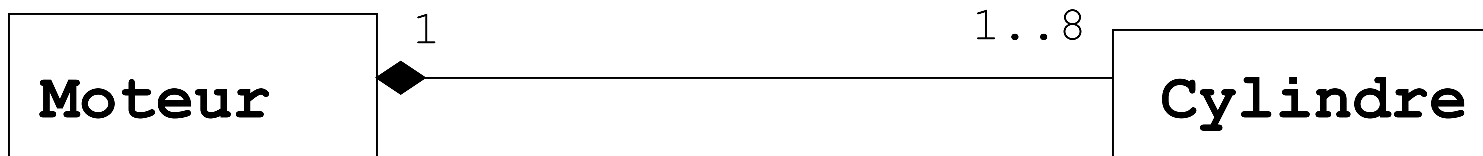
Traduction Java

```
public class Personne
{
  private String nom;
  private Personne epoux = null;
  private boolean marie = false;
  public Personne( String nom ){
      this.nom=nom; }
  public void marie( Personne p ){
      epoux = p; marie = true; }
  public boolean marie(){
      return marie; }
  public boolean client( ListeClients<Personne> l ){
      return (l.laListe()).contains( this );}
}
```

Traduction Java

```
public class ListeClients
{
  private Vector<Personne> laListe;

  public ListeClients(){
     laListe = new Vector<Personne>();
  }

  public void inclus( Personne p ){
laListe.add( p ); }

  public Vector<Personne> laListe(){
     return laListe;}
}
```

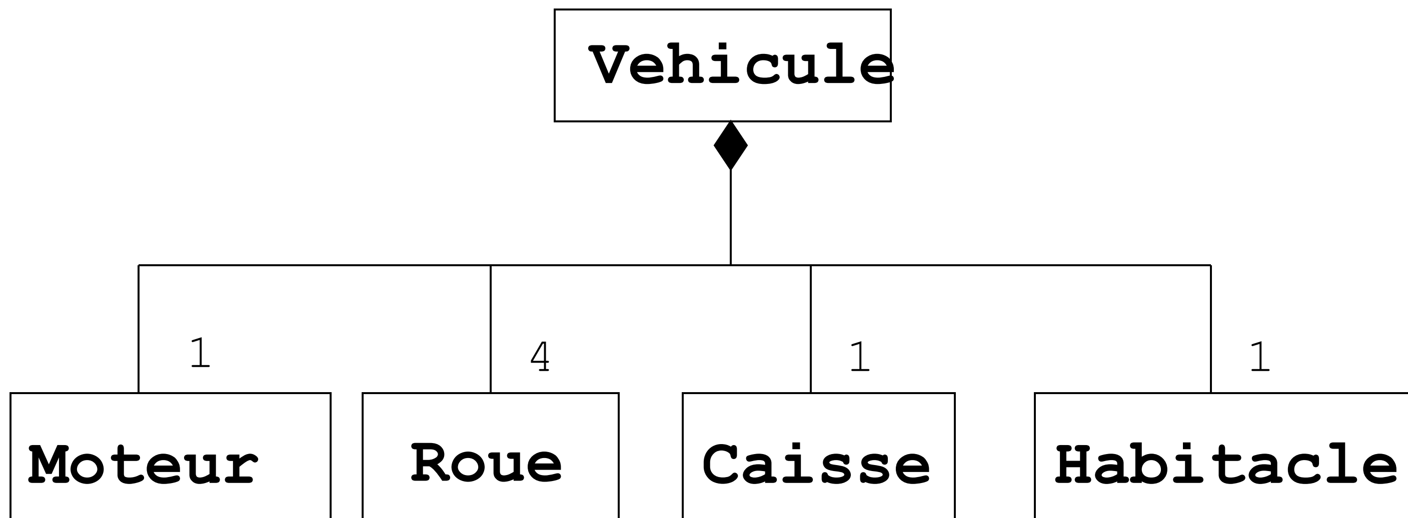
Composition



Composition : traduction java

```
public class Moteur
{
  private Cylindre[] lesCylindres;
  public Moteur()
  {
    lesCylindres = new Cylindre[8];
  }
...
}
```

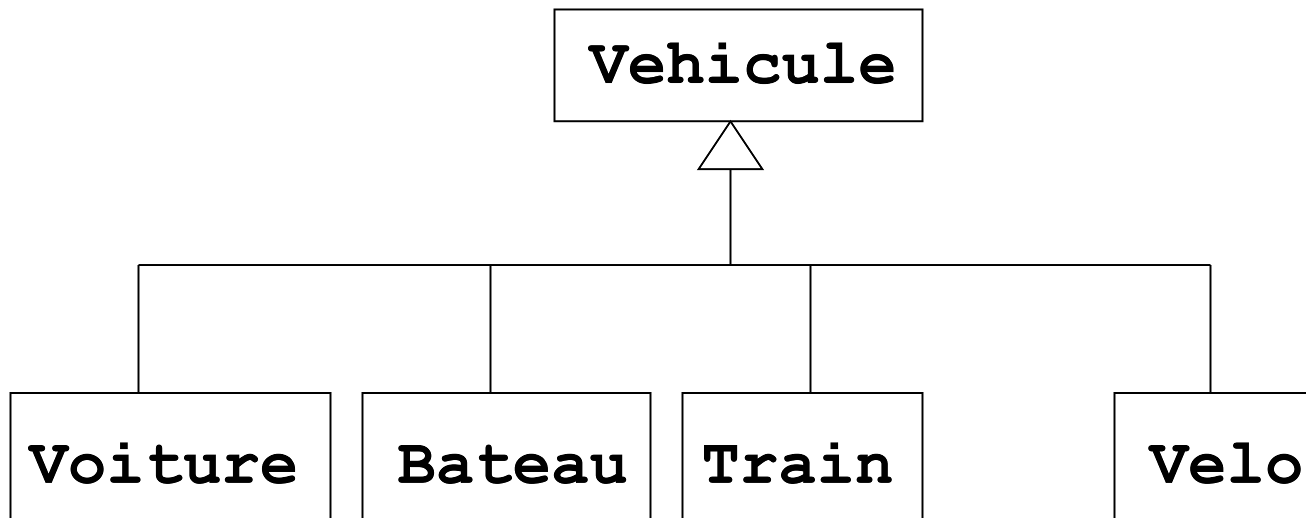
Composition multiple



Composition multiple : traduction Java

```
public class Vehicule
{
   private Moteur moteur;private Roue[] roues;
   private Caisse caisse;private Habitacle habitacle;
   public Vehicule(){
    moteur = new Moteur();roues = new Roue[4];
    caisse = new Caisse();habitacle = new Habitacle();}
   public Vehicule( Moteur moteur,Roue[] roues,
                    Caisse caisse,Habitacle habitacle){
    this.moteur = moteur;
    this.roues = roues;
    this.caisse = caisse;
    this.habitacle = habitacle;
   }
   ...
}
```

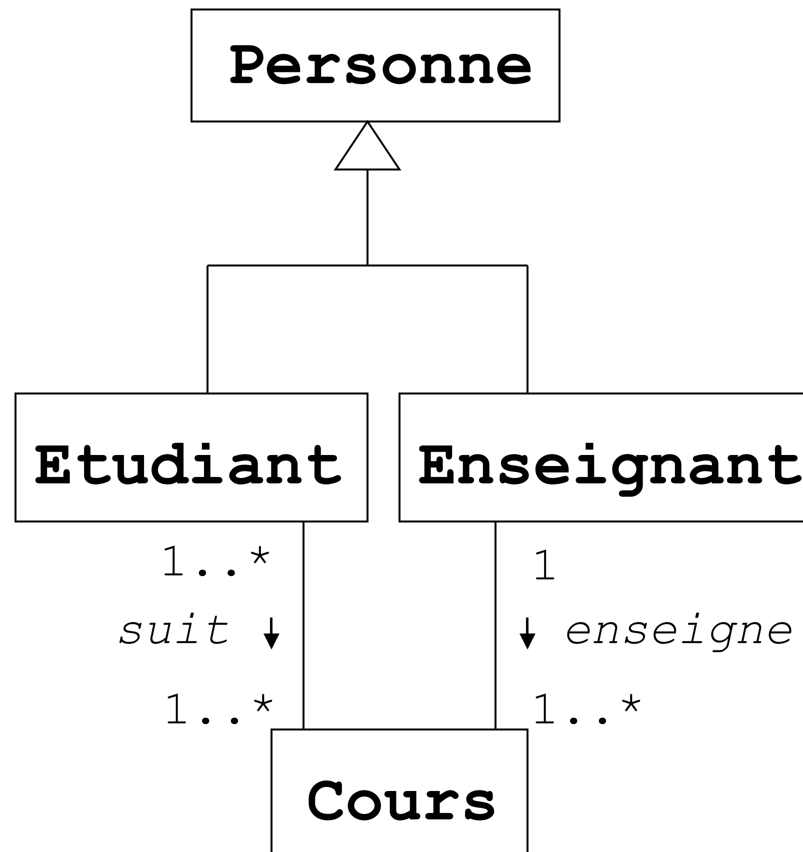
Héritage



Héritage : traduction Java

```
public class Voiture extends Vehicule
{...}
public class Bateau extends Vehicule
{...}

public class Train extends Vehicule
{...}

public class Velo extends Vehicule
{...}
```

Diagramme mixte



Traduction Java

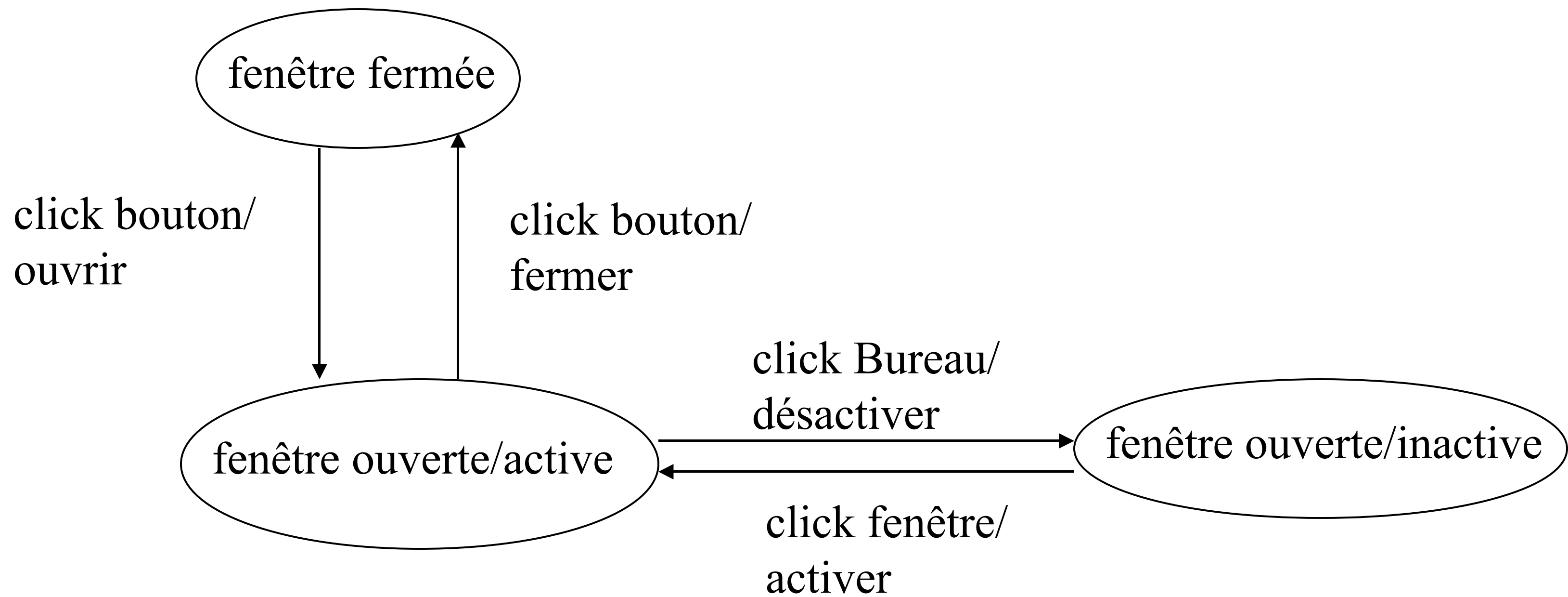
```
public class Personne {
      protected String nom;
   ...
}

public class Cours{ ... }

public class Etudiant extends Personne{
   private Vector<Cours> lesCours;
   public Etudiant( String nom ){
       lesCours = new Vector<Cours>();
       this.nom = nom;}
   public void suit( Cours cours ){
lesCours.addElement( cours ); }
}
```

Traduction Java

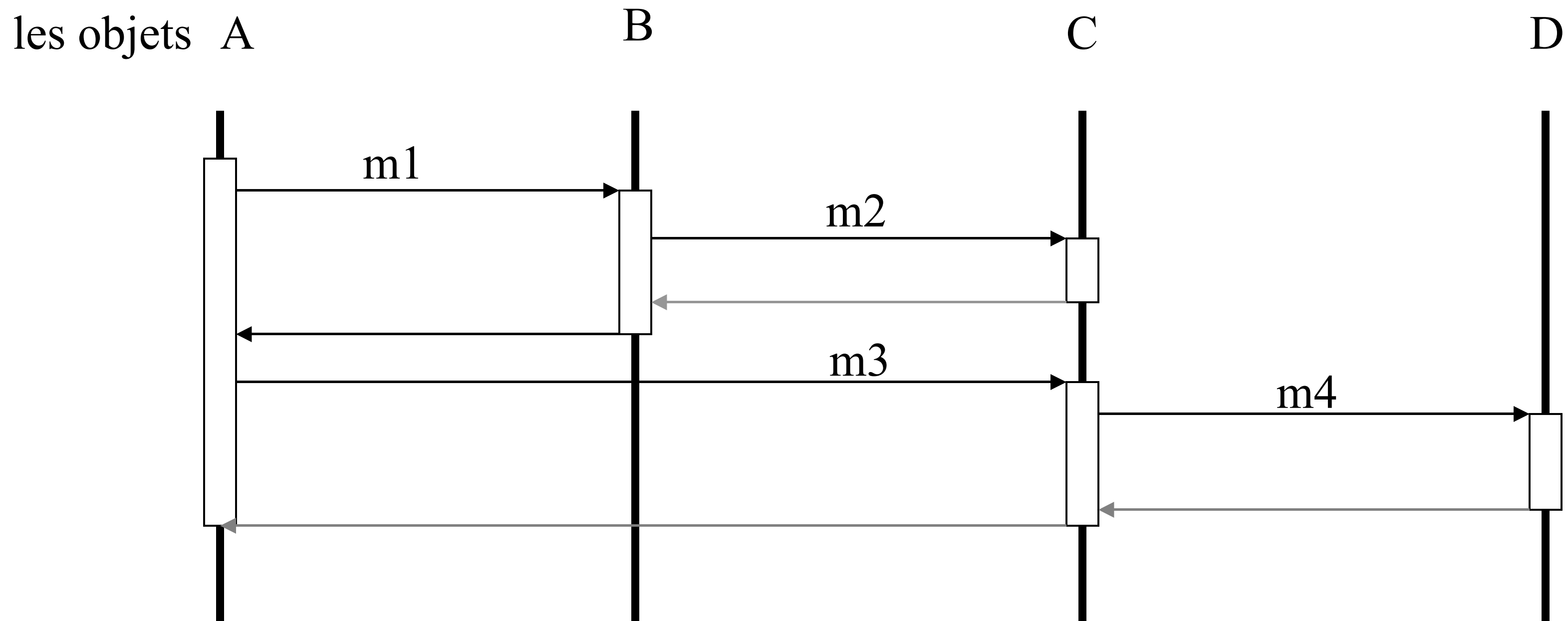
```
public class Enseignant extends Personne{
  private Vector<Cours> lesCours;
  public Enseignant( String nom ){
      lesCours = new Vector<Cours>();
      this.nom = nom;}
  public void enseigne( Cours cours ){
lesCours.addElement( cours ); }
}
```

Modèle dynamique

Diagramme état-transition



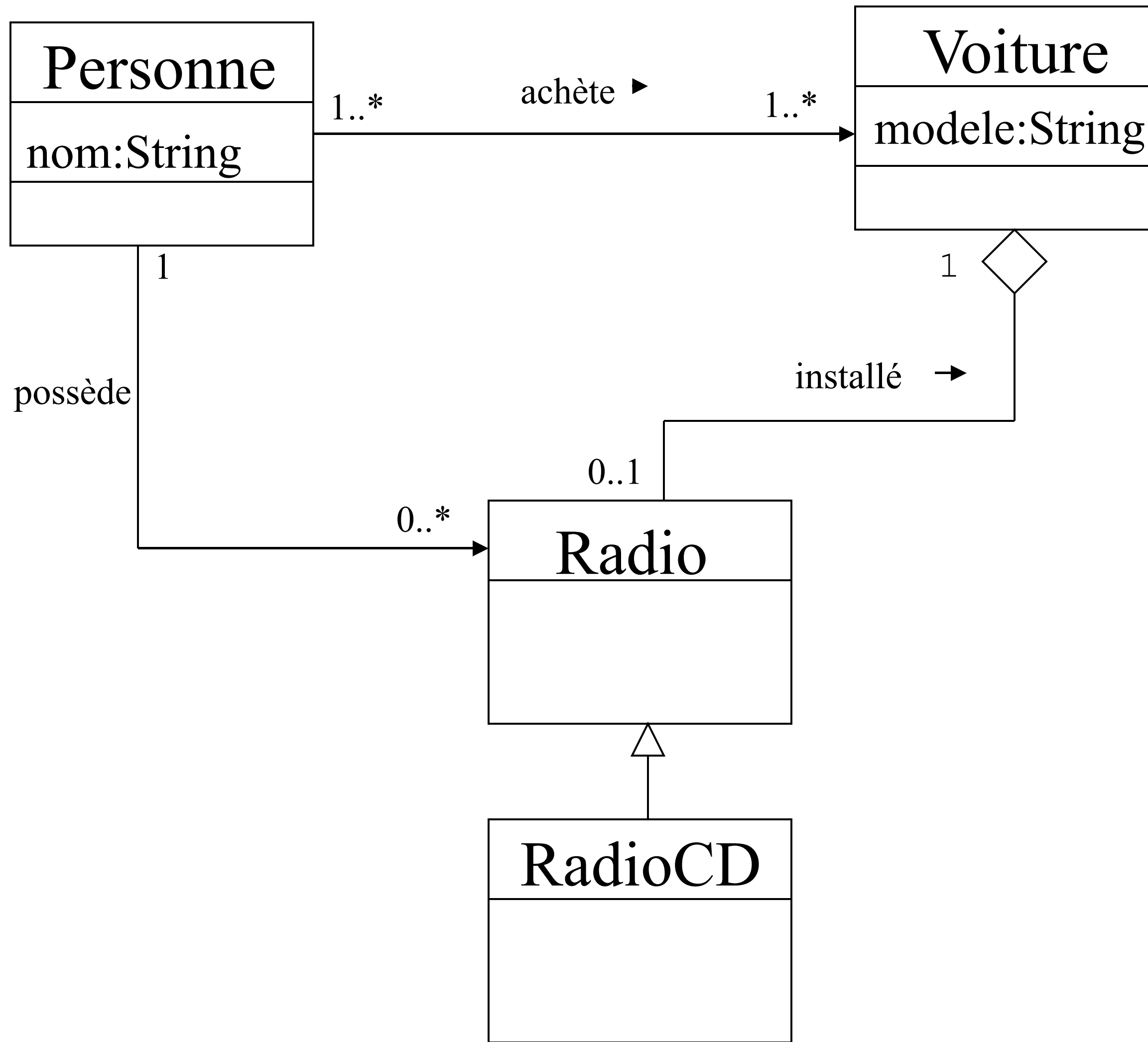
Diagrammes de séquences



ED n°5

Des personnes définies par leur nom peuvent acheter une ou plusieurs voitures (définies par leur modèle). Chaque voiture peut posséder un autoradio (avec ou sans lecteur de CD). Cet autoradio est extractible. Son propriétaire est une personne qui n'est pas obligatoirement celui de la voiture.

- Tracer le diagramme UML correspondant
- Implanter ce diagramme en Java


Personne
nom:String
1..*
achète
1..*
Voiture
modele:String
1
1
possède
installé
0..1
0..*
Radio
RadioCD

Exercice (1/3)

```
public class Personne
{
  private String nom;
  private Vector<Radio> radios;
  private boolean possede = false;
  private Vector<Voiture> voitures;

  public Personne( String nom )
  { this.nom = nom; voitures = new Vector<Voiture>(); }

  public void possede( Radio radio )
  { radios.addElement(radio); possede = true;}

  public boolean possedeRadio(){ return possede;}

  public void achete( Voiture voiture )
   { voitures.addElement( voiture ); }
}
```

Exercice (2/3)

```
public class Voiture
{
    private String modele;
    private Radio radio = null;
    private boolean radioDispo = false;

    public Voiture( String modele )
    { this.modele = modele; }

    public void installe( Radio radio )
    { this.radio = radio; radioDispo = true; }
    public boolean radioDispo() { return radioDispo; }
    public Radio getRadio(){ return radio; }
}
```

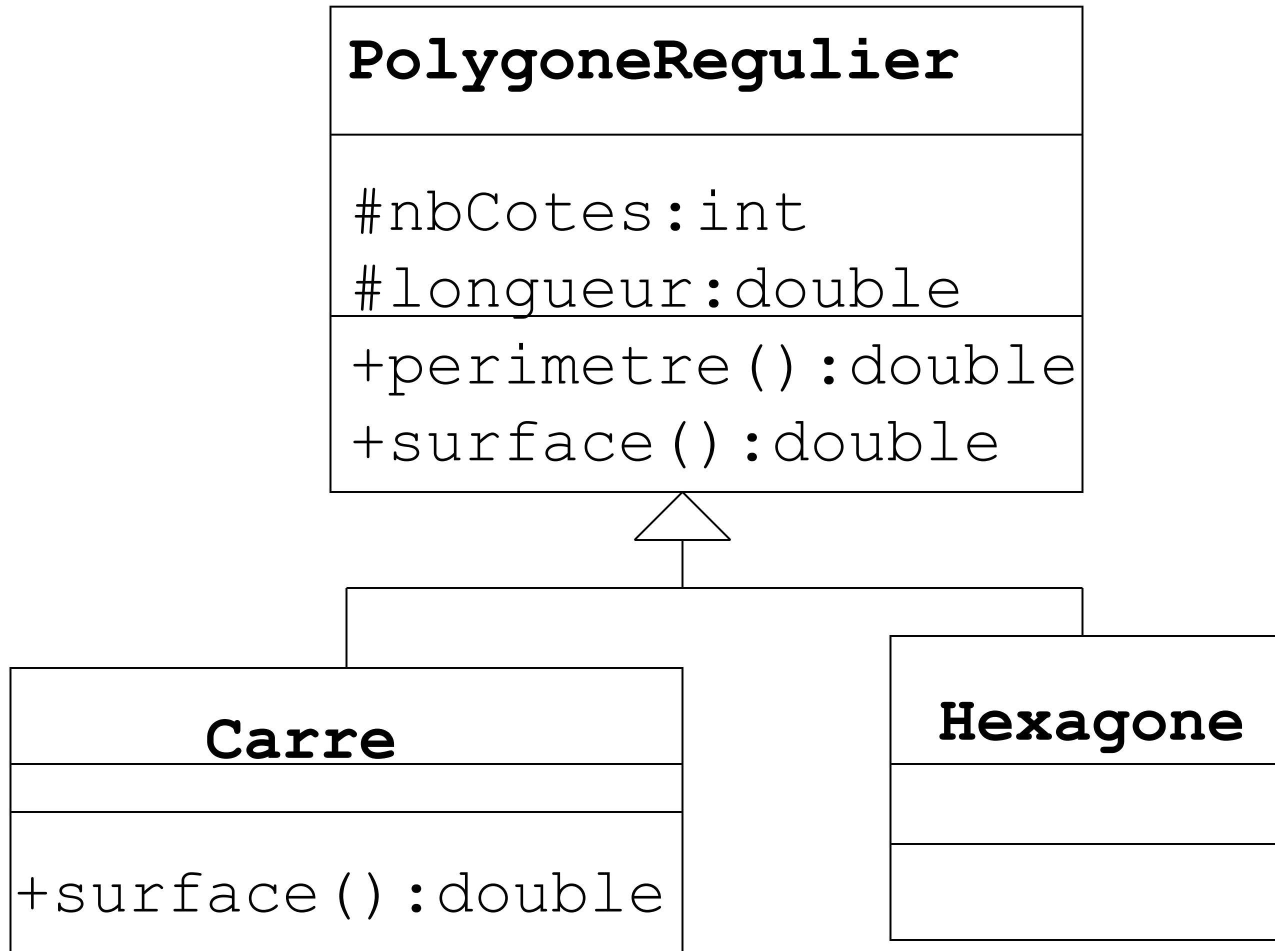
Exercice (3/3)

```
public class Client{
  public static void main ( String[] args ){
    Personne personne = new Personne( "Ledoux" );
    Voiture voiture   = new Voiture( "Renault" );
    Radio radio       = new Radio();
    voiture.installe( radio );
    personne.achete( voiture );
    personne.possede( voiture.getRadio() );
  }
}
```

Héritage

- Une classe peut être créée par dérivation d'un classe existante : la super-classe
- la nouvelle classe possède obligatoirement tous les attributs et méthodes de la super-classe
- Elle se différencie par la possibilité :
  - d'ajouter de nouvelles variables d'instance
  - d'ajouter de nouvelles méthodes
  - de redéfinir des méthodes de la super-classe

Diagrammes de classe



```
public class PolygoneRegulier{
  protected int nbCotes;
  protected double longueur;

  public PolygoneRegulier( int n,double l ){
    nbCotes = n;longueur = l;
  }

  public double perimetre(){
    return nbCotes*longueur;
  }

  public double surface(){
 return nbCotes*longueur*longueur/4*Math.tan(3.14/nbCotes);
  }
  ...
}
```

```
public class Carre extends PolygoneRegulier{
  public Carre( double l ){
    nbCotes = 4;longueur = l;
  }
  public double surface(){
    return longueur*longueur;
  }
  ...
}
```

```
public class Hexagone extends PolygoneRegulier{
  public Hexagone( double l ){
    nbCotes = 6;longueur = l;
  }
...
}
```

```
Carre carre         = new Carre(5.7);
double p            = carre.perimetre();
// perimetre est héritée de PolygoneRegulier
carre.surface();
// surface est redéfinie dans la classe Carre
Hexagone hexagone = new Hexagone(3.9);
double surface = carre.surface()+hexagone.surface();
// la méthode surface() de Carre
// puis celle de Hexagone sont invoquées
```

Protection

- les membres (variables d'instance, constantes, méthodes) déclarées **`public`** sont accessibles par tout objet
- les membres déclarés **`private`** ne sont accessibles que dans les méthodes de la classe
- les membres **`protected`** sont accessibles seulement par :
  - les membres de la classe
  - les membres de chacune des sous classes quel que soit le package d'appartenance
- Si aucun modificateur de protection n'est spécifié pour une classe, ses membres sont accessibles par tous les membres des autres classes du même package

Remarque : si toutes les sous-classes appartiennent au même package, accès protected ou par défaut sont équivalents

Règles de visibilité

visibilité	`private`	par défaut	`protected`	`public`
autre classe du même package	non	oui	oui	oui
sous classe d'un autre package	non	non	oui	oui
classe d'un autre package	non	non	non	oui

Exercice 11

Retour sur la classe `Radio` (voir diapositive suivante).

On souhaite créer un nouveau modèle de radio capable de lire des CDs.

Concevoir une nouvelle classe `RadioCD` à partir de la classe existante `Radio`.

La classe `Radio` est-elle assez générale pour devenir super-classe de `RadioCD` ? sinon comment faut-il la modifier ?

Le programme principal crée une instance de cette nouvelle classe avec les stations préréglées, écoute la station 4 puis 89.9 puis insère une CD l'écoute, l'enlève et écoute la station 93.5

```
class Radio{
  protected double[] stations = new double[5];

  public Radio(){}
  public Radio(double[] stations){
       this.stations = stations;
  }
  public void preregler(int index, double frequence){
       stations[index] = frequence;
  }
  public void ecouter(int index){
    System.out.println
       ("Ecoute de la station :  "+stations[index]+" Mhz");
  }
  public void ecouter(double frequence){
    System.out.println
       ("Ecoute de la station :  " + frequence+ " Mhz");
  }
}
```

```
class RadioCD extends Radio{
  private boolean cdIn = false;
  public RadioCD(double[] stations){
    super(stations);
  }
  public void insererCD(){
    cdIn = true;
    System.out.println("\t--->CD inséré");
  }
  public void enleverCD(){
    cdIn = false;
    System.out.println("\t--->CD enlevé");
  }
  public void ecouter(){
    if(cdIn)
      System.out.println("\t--->lecture du CD");
    else
      System.out.println("\t--->insérer un CD d'abord");
  }
}
```

```
public class Exercice11{
  public static void main(String[] args){
    double[] stations = {101.5, 87.9, 105.1, 95, 101.1};

    Radio radio = new Radio(stations);
    radio.ecouter(3);
    radio. ecouter(93.5);
    RadioCD radioCD = new RadioCD(stations);
    radioCD. ecouter(2);
    radioCD. ecouter(89.9);
    radioCD.insererCD();
    radioCD. ecouter();
    radioCD.enleverCD();
    radioCD. ecouter(93.5);
  }
}
```

Surcharge et redéfinition

- plusieurs méthodes d'une même classe peuvent avoir le même nom à condition d'avoir des **signatures différentes** (nombre et/ou types des paramètres différents). On dit qu'elles sont **surchargées**.

```
void afficher(String s);
void afficher(int i,Object o);
```

- Le compilateur s'appuie sur la signature des méthodes pour choisir sans ambiguïté la méthode à appliquer
- La **surcharge** est un mécanisme qui s'applique aussi aux constructeurs
- En revanche, une méthode **redéfinie** possède la **même signature** et le même type de retour que la méthode de la super classe.
- Une variable d'instance, une variable ou une méthode `static` ne peuvent pas être redéfinies (mais peuvent être cachées)

Surcharge : exemples

```
class Surcharge{
  void afficherValeur( ){
    System.out.println("aucune valeur definie"); }
  void afficherValeur(long x){
    System.out.println("valeur entiere egale a"+x); }
  void afficherValeur(double x){
    System.out.println("valeur flottante egale a"+x); }
  void afficherValeur(String s, int x){
    System.out.println(s+x); }
}
class Test{
  public static void main(String[ ] args){
    Surcharge sRef = new Surcharge( ) ;
    sRef.afficherValeur( ) ;
    sRef.afficherValeur(3.14159) ;
  }
}
```

aucune valeur définie
valeur flottante égale à 3.14159

Surcharge : ambiguïté

Il n'y a pas surcharge si 2 méthodes ne se différencient que par le type de la valeur retournée

```
class A {
  double g(){....}
  int g(){....}
}
```

le compilateur décèle une ambiguïté

Il y a surcharge si 2 méthodes ont même nombre et même type de paramètres mais dans un ordre différent

```
class A {
  int g(double d, int i){....}
  int g(int i, double d){....}
}
```

compilation OK

Mais

```
A a = new A();
a.g(5,5);
```

ambiguïté : le compilateur ne sait pas quelle méthode choisir

Sous-classement

```
// supposons les variables suivantes
PolygoneRegulier poly = new PolygoneRegulier(5,2.5);
Hexagone hexa = new Hexagone(4.67);

// Que dire de l'affectation
poly = hexa;
```

Sous-classement

```
// Cette affectation est légale.
// après affectation, poly contient une référence à
// un objet, instance d'une sous-classe de
// PolygoneRegulier
// Or un hexagone est un polygone régulier

// Pour la même raison, il est tout aussi légal
// d'écrire
Object o = new Object();
o = hexa;
// puisque tout est objet
```

Sous-typage

Définition :

Soit ST un sous-type de T, alors toute valeur de ST peut-être utilisée en lieu et place d'une valeur du type T.

si SC est sous-classe de T, alors SC est sous-type de T

Exemple :

si `poly.surface()` est valide, alors `hexa.surface()` le sera aussi

Les classes `PolygoneRegulier` et `Hexagone` :

- sont en relation de **sous-classement**
- sont en relation de **sous-typage**

Sous-classement et sous-typage

```
// On ajoute la méthode dessiner à Hexagone
public void dessiner()
{
   ...
}

// supposons maintenant le morceau de programme
poly = hexa;
poly.dessiner();

// Que se passe-t-il ?
// une erreur surviendra à la compilation
// en effet, le type de poly (PolygoneRegulier)
// ne possède pas de méthode dessiner()
```

Sous-classement et sous-typage

```
// on ajoute la méthode suivante à la classe PolygoneRegulier
  public boolean plusGrand(PolygoneRegulier p)
  {
    return this.surface()>p.surface();
  }
  PolygoneRegulier poly = new PolygoneRegulier( 5,2.5 );
        Hexagone hexa = new Hexagone( 4.67 );
  if( poly.plusGrand(hexa) )
     System.out.println( "poly>hexa" );
  else
     System.out.println( "poly<hexa" );
// erreur PolygoneRegulier plusGrand(Hexagone) n'existe
// pas limitation du sous-typage
```

Transtypage

```
// autre cas
hexa = poly;
// Cette affectation est illégale
// Erreur de typage détectée : types incompatibles

poly = hexa;
// comment rendre possible un tel message
poly.dessiner(); // message impossible
// la méthode dessiner n'appartient pas à la
// classe PolygoneRegulier → erreur à la compilation

// vérifier si poly contient une référence sur une instance
// d'hexagone
if ( poly instanceof Hexagone ){
   Hexagone h = (Hexagone)poly; // conversion de type
   h.dessiner();
}
```

Transtypage

```
Carre c = new Carre(5);
Hexagone h = new Hexagone(3);
c = h; // types incompatibles
c = (Carre)h; // types inconvertibles
```

La pseudo variable `super`

- Une variable d'instance d'une sous-classe peut avoir le même nom qu'une variable d'instance de sa super-classe
- Comment atteindre celle de la super-classe à partir d'une méthode de la sous-classe ?
- Exemple :
  - on ajoute la variable d'instance `String nom` aux classes `PolygoneRegulier`, `Hexagone` et `Carre`
  - on ajoute la méthode `void println()` (qui affiche le nom de la figure) aux classes `Hexagone` et `Carre`

```
// code de println()
public void println(){
  System.out.println( this.nom+super.nom );
}
```

Constructeurs et héritage

```
public class Point{
  protected int x,y;
  public Point( int x,int y ){
    this.x = x;this.y = y;
  }
  ...
}

public class Cercle extends Point{
  private int rayon;
  public Cercle( int x,int y,int r ){
    super( x,y );rayon = r;
  }
  ...
}
```

Polymorphisme

```
// Reprenons la hiérarchie PolygoneRegulier, et ajoutons
la // classe LiaisonRetardee
public class LiaisonRetardee{
  public static void main( String[] args )throws IOException{
    PolygoneRegulier p;
    char reponse;
    Scanner in = new Scanner(System.in);
    System.out.println( "Polygone régulier(p,c,h) ?" );
    reponse = in.next().charAt( 0 );
    switch ( reponse ){
      case 'c':p = new Carre(...);
      case 'h':p = new Hexagone(...);
      case 'p':p = new PolygoneRegulier(...);
    }
    System.out.println( "surface de p="+p.surface() );
}
```

Polymorphisme

- On ne connaît qu'à l'exécution la classe d'appartenance de l'objet dont la référence est affectée à `p`.
- Le polymorphisme trouve son intérêt lorsque l'on veut construire dynamiquement des structures de données dont les éléments sont des références d'objets d'une même hiérarchie
- En effet, la déclaration d'une telle structure impose un type identique à tous les objets qu'elle est susceptible de contenir

Polymorphisme : exemple

```
PolygoneRegulier[] p = new PolygoneRegulier[3];
Hexagone h           = new Hexagone();
Carre c              = new Carre();
PolygoneRegulier r   = new PolygoneRegulier();
p[0] = h;
p[1] = c;
p[2] = r;
for ( int i=0;i<3;i++ )
{
  System.out.println( p[i].surface() );
}
// la méthode surface() effectivement invoquée n'est connue
// qu'à l'exécution
// si la méthode à invoquer était choisie à la compilation,
// c'est la méthode surface() de PolygoneRegulier qui serait
// systématiquement choisie
```

Exercice 13

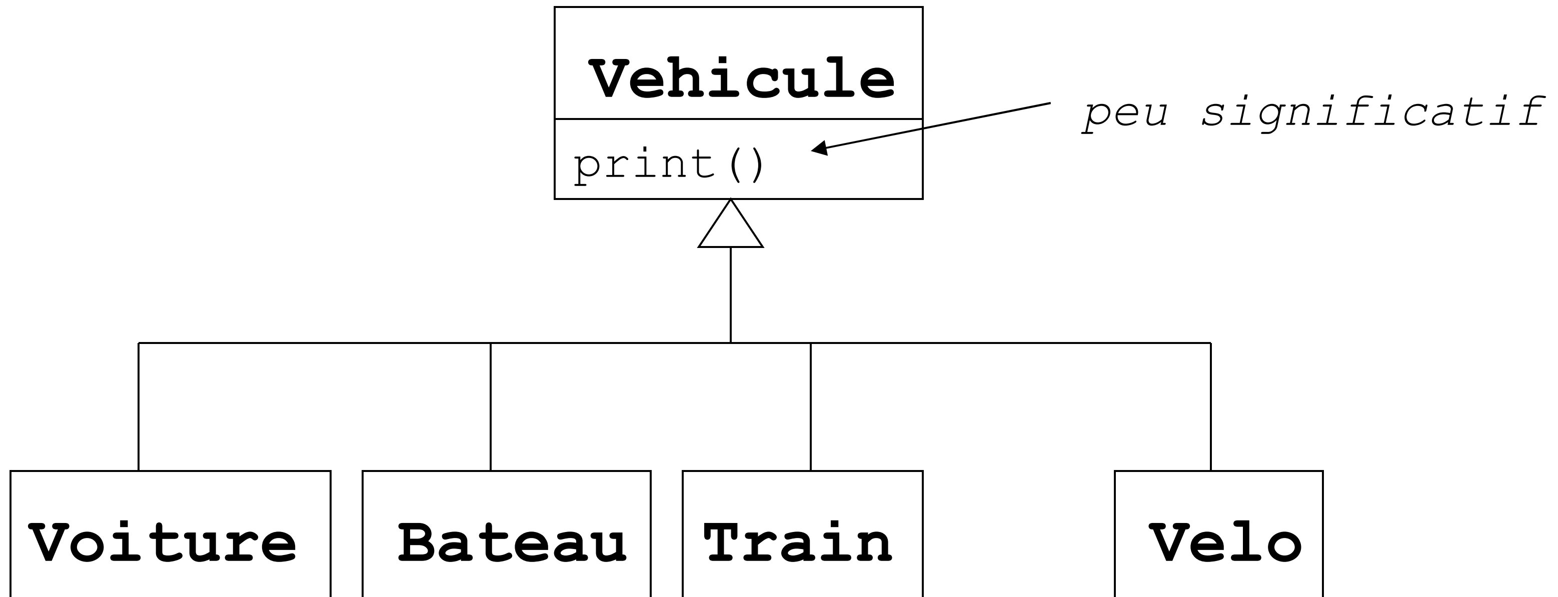
```
class A extends Object{
  public void m(){
    System.out.println("m de la classe A");
}}
class B extends A{
  public void m(){
    System.out.println("m de la classe B");
}}
public class Poly03{
  public static void main( String[] args ){
    Object var = new B();
    ((B)var).m();
    ((A)var).m();
     var.m();
     var = new A();
    ((A)var).m();
}}
```

Quel est le résultat de chaque ligne de code ?

Exemple : résultat (2/2)

```
Object var = new B();
((B)var).m();      // downcast          m in class B
((A)var).m();      // Polymorphisme     m in class B
 var.m();
// échec à la compilation
//      Poly03.java:40: cannot resolve symbol
//      symbol   : method m   ()
//      location: class java.lang.Object
//      var.m();
var = new A();
((A)var).m(); // downcast               m in class A
```

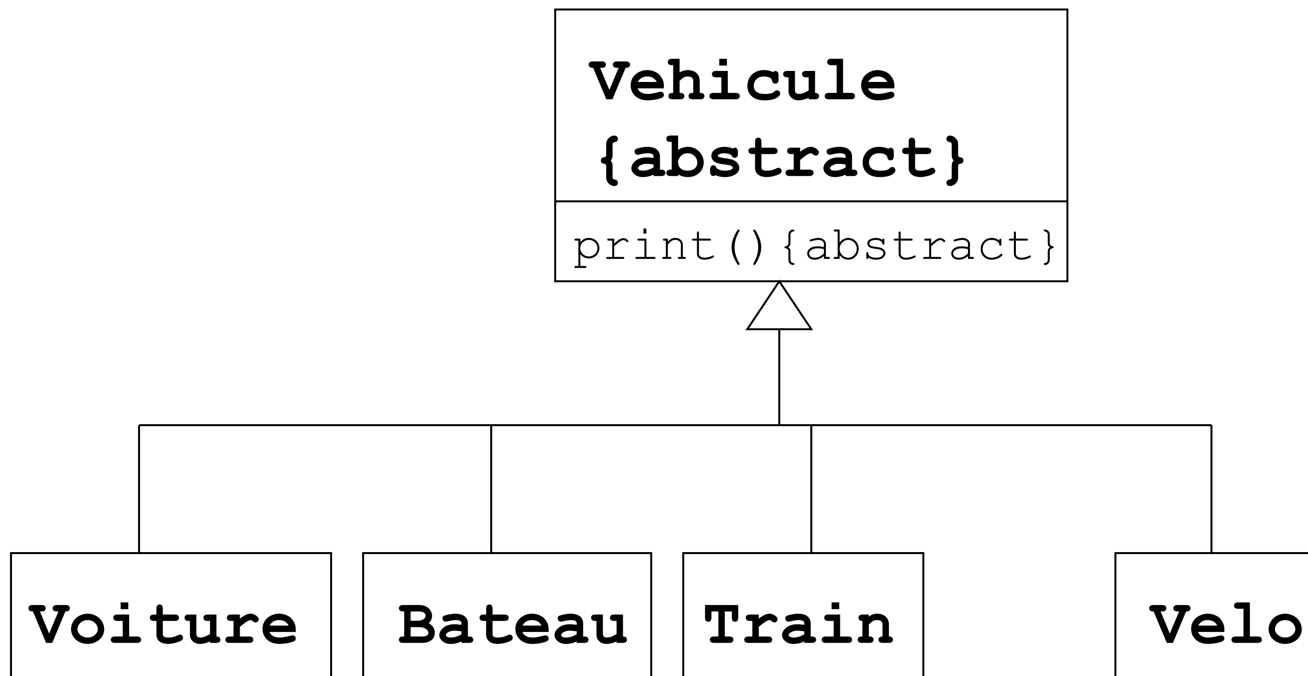
Méthodes et classes abstraites



– Au niveau de la classe `Vehicule`, la méthode `print()` n'a pas beaucoup de sens

– En revanche, au niveau des sous-classes, elle permet d'afficher toutes les informations utiles concernant un type de véhicule particulier

Méthodes et classes abstraites



Méthodes et classes abstraites

```
public abstract class Vehicule{
        protected Position position;
        public abstract void print();
        public void deplacer( Position position ){
                this.position = position;
        }
        ...
}

public class Voiture extends Vehicule{
        public void print(){
                System.out.print( position.toString() );
        }
}
```

Méthodes et classes abstraites

- En déclarant **abstraite** une méthode au niveau de la super-classe, on impose à toute sous-classe **présente** ou **future** d'implanter cette méthode
- Intérêt : on impose un comportement commun à toute classe de la hiérarchie
- On préserve la relation de **sous-typage** et le bon fonctionnement de la **liaison dynamique**

Méthodes et classes abstraites

- Une classe abstraite ne peut pas être instanciée, mais on peut définir des constructeurs qui seront invoqués dans les sous-classes

- Si une sous-classe d'une classe abstraite n'implémente pas toutes les méthodes abstraites, cette sous-classe doit être déclarée abstraite.

- Une sous-classe peut être abstraite même si sa super-classe est concrète (exemple la super-classe `Object`)

Classe abstraite et polymorphisme

```
Vehicule v;
Train t  = new Train();
Bateau b = new bateau();
boolean presse;
...
if (presse)
   v = t;
else
   v = b;
v.print();
```

Une classe abstraite sert de modèle. Elle ne peut pas être instanciée

quelle que soit la référence contenue dans dans v, on a la garantie que la méthode print() est implantée

Interfaces

- En java, **pas d'héritage multiple**. Il serait donc impossible de faire hériter les classes `Bateau`, `Train`, etc.. de la classe `Canvas` pour leur ajouter une méthode `draw()`
- Le concept d'interface permet de remédier à cette limitation
- Une classe Java n'hérite que d'une seule classe, mais peut implanter plusieurs interfaces

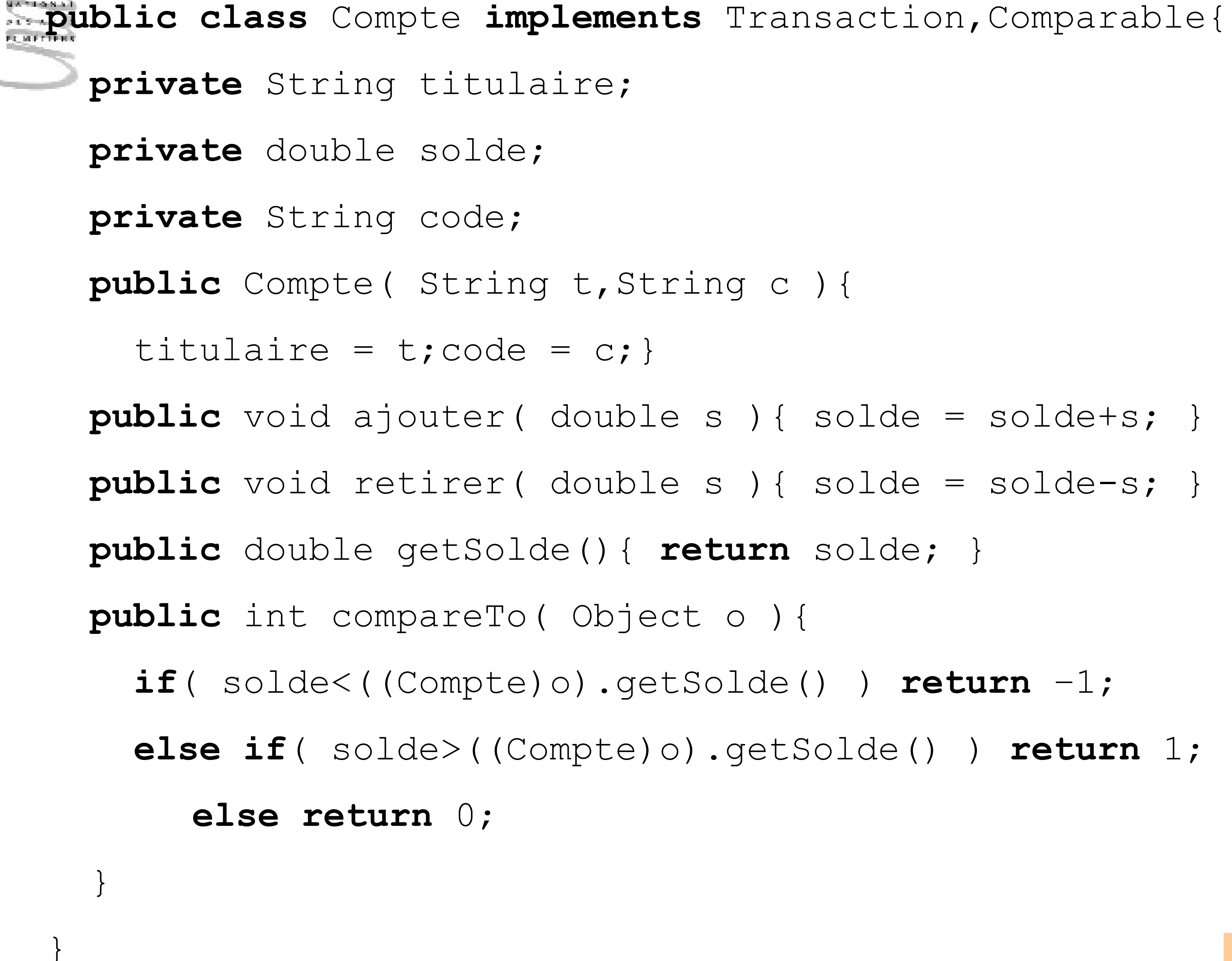
Interfaces

- Une interface est une sorte de classe abstraite qui contient uniquement :
  - des déclarations de méthodes
  - des constantes `static`
- Une interface ne peut pas être instanciée
- Une interface peut hériter d'une autre interface

Interfaces : exemples

```
public interface Transaction{
  public void ajouter( double s );
  public void retirer( double s );
}

public interface Comparable{
      /**
       * @param o objet à comparer
       * @return <0 si l'objet est inférieur à O
       *         =0                   égal
       *         >0                   supérieur
       */
      public int compareTo( Object o );
}
```

java.lang.Comparable



```
public class Compte implements Transaction,Comparable{
  private String titulaire;
  private double solde;
  private String code;
  public Compte( String t,String c ){
    titulaire = t;code = c;}
  public void ajouter( double s ){ solde = solde+s; }
  public void retirer( double s ){ solde = solde-s; }
  public double getSolde(){ return solde; }
  public int compareTo( Object o ){
    if( solde<((Compte)o).getSolde() ) return -1;
    else if( solde>((Compte)o).getSolde() ) return 1;
      else return 0;
  }
}
```

```
public class Reservoir implements Transaction,Comparable{
  private double contenu;
  public Reservoir( double c ){ contenu = c; }
  public void ajouter( double s ){ contenu = contenu+s; }
  public void retirer( double s ){ contenu = contenu-s; }
  public double getContenu(){ return contenu; }
  public int compareTo( Object o ){
    if(contenu<((Reservoir)o).getContenu()) return -1;
    else if(contenu>((Reservoir)o).getContenu()) return 1;
       else return 0;
  }
}
```

Interfaces et polymorphisme

```
Transaction[] t = new Transaction[2];
t[0] = new Compte( "Toto","4567" );
t[1] = new Reservoir(600);
...
for ( int i=0;i<t.length;i++ ){
  if (t[i] != null)
    t[i].ajouter( 300 );
}
...
Comparable[] t = new Comparable[3];
t[0] = new Reservoir(950);
t[1] = new Reservoir(600);
...
if(t[0].compareTo(t[1])<0) t[2]=t[1];
else t[2]=t[0];
```

Conflit de noms

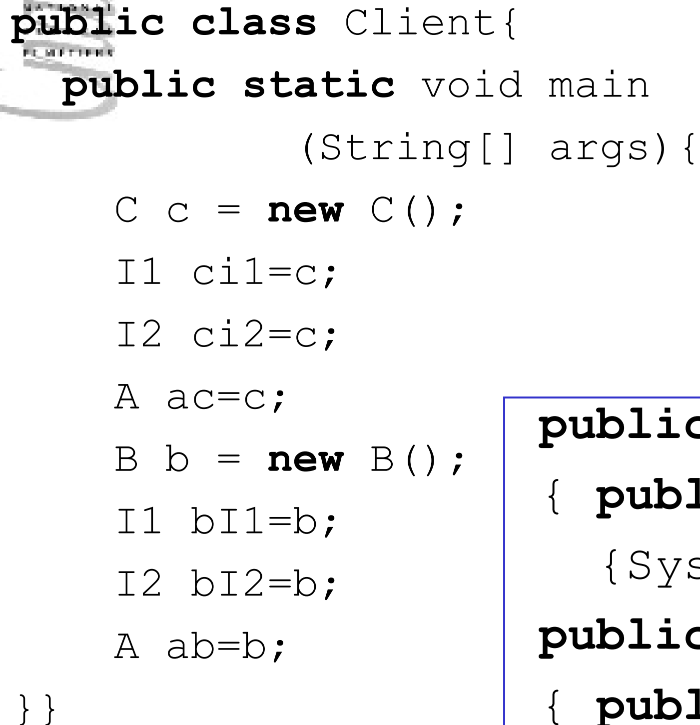
```
interface I1{
        void m(double d);
        void f();
        void g(int i);
}
```

```
interface I2{
        void m(int i);
        void f();
        float g(int i);
}
```

```
class A implements I1,I2{
//m de I1 et m de I2 doivent être définies
      public void m(double d){....}
      public void m(int i){....}
// f est présente dans les 2 interfaces, pas de conflit de noms
// car il suffit de définir une méthode f pour satisfaire
// les besoins d'implémentation imposés par les 2 interfaces
      public void f(){....}
// les méthodes float g(int i) de I2 et void g(int i) de I1 sont
// bien  distinctes. Elles ne peuvent pas pourtant pas être
// définies dans la classe A puisqu'elles ont le même nom et
// les mêmes arguments.L'ambiguïté est décelée par le compilateur
```

Types, classes, interfaces

- Un objet peut avoir plusieurs types (celui de sa classe et de son ou ses interfaces
- Différents objets, instances de classes implémentant la même interface peuvent être considérés comme ayant le même type.



```
public class Client{
  public static void main
           (String[] args){
    C c = new C();
    I1 ci1=c;
    I2 ci2=c;
    A ac=c;
    B b = new B();
    I1 bI1=b;
    I2 bI2=b;
    A ab=b;
}}
```

```
public interface I1
{ void mI1() }
public interface I2
{ void mI2() }
```

```
public class A
{ public void p()
  {System.out.println("A-->p");}}
public class B extends A implements I1,I2
{ public void mI1()
  { System.out.println("B-->mI1");}
  public void mI2()
  { System.out.println("B-->mI2");}}
public class C extends A implements I1,I2
{ public void mI1()
{ System.out.println("C-->mI1");}
  public void mI2()
{ System.out.println("C-->mI2");}}
```

ED n°7

On veut manipuler dans un même programme des objets caractérisés par leur couleur et leur position (respectivement de type `Color` et `Position`) et que l'on pourra déplacer et afficher.

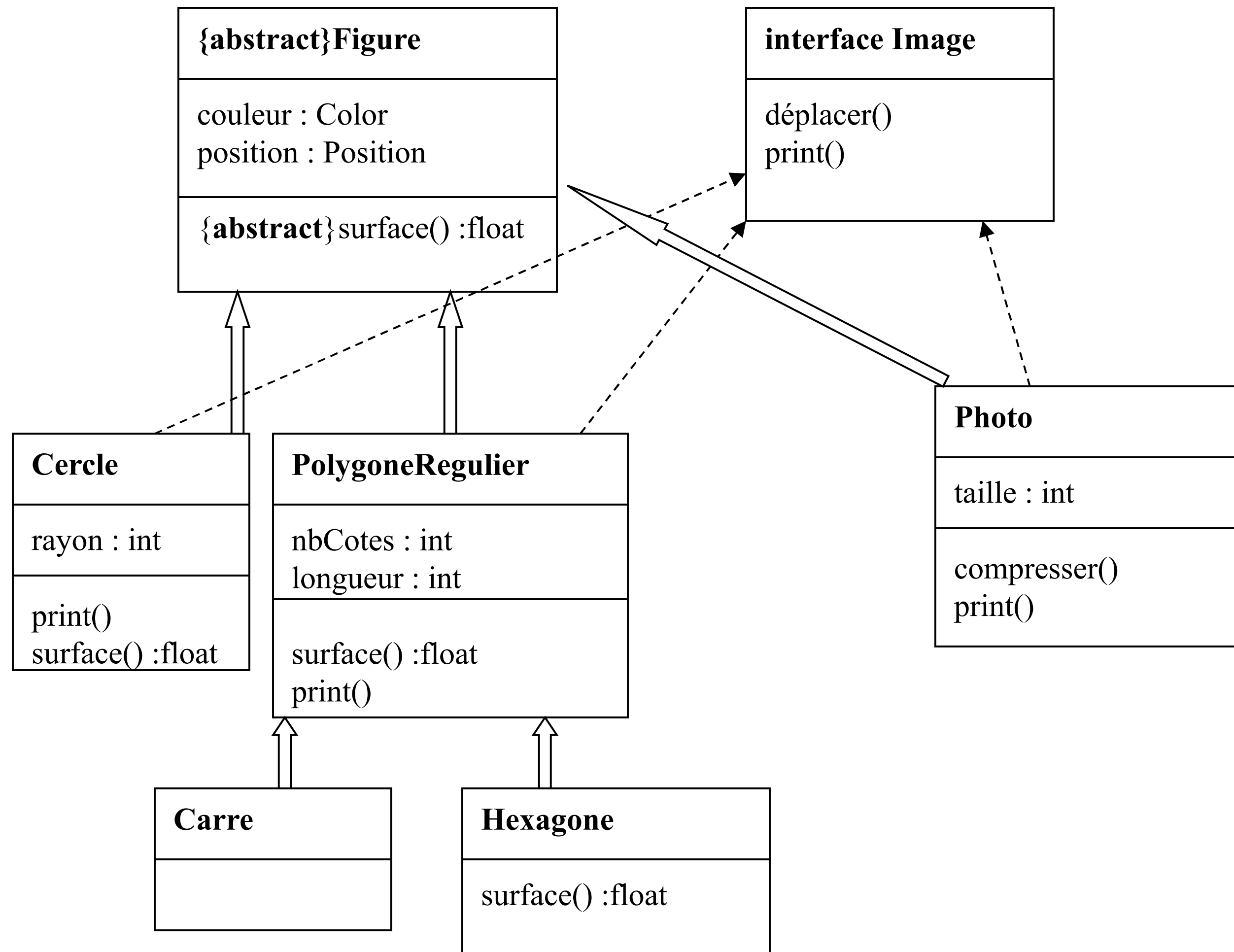
Parmi ces objets, on trouve des :

- photos, caractérisées par leur taille et que l'on pourra compresser
- des figures géométriques, carrés et des hexagones et plus généralement des polygones réguliers définis par le nombre et la longueur de leurs cotés et des cercles définis par leur rayon dont on pourra calculer la surface.

Notons que le calcul de la surface d'une photo ne nous intéresse pas non plus la compression d'une figure géométrique quelconque.

On souhaite pouvoir à l'avenir ajouter d'autres figures géométriques possédant les mêmes caractéristiques que tout objet et qu'on pourra afficher à l'écran, déplacer et dont on pourra aussi calculer surface.

Proposer l'architecture qui vous paraît la plus adaptée. Elle sera exprimée en UML. On ne s'intéressera donc pas au codage des méthodes mais seulement à leur déclaration.


{abstract}Figure
couleur : Color
position : Position
{abstract}surface() :float
interface Image
déplacer()
print()
Cercle
rayon : int
print()
surface() :float
PolygoneRegulier
nbCotes : int
longueur : int
surface() :float
print()
Photo
taille : int
compresser()
print()
Carre
Hexagone
surface() :float