Programmation Orientée Objet

Amélie Lambert

USSE09 - Java

Analyse, conception et programmation Orientée Objets

Analyse et conception 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

Les objets et leurs attributs

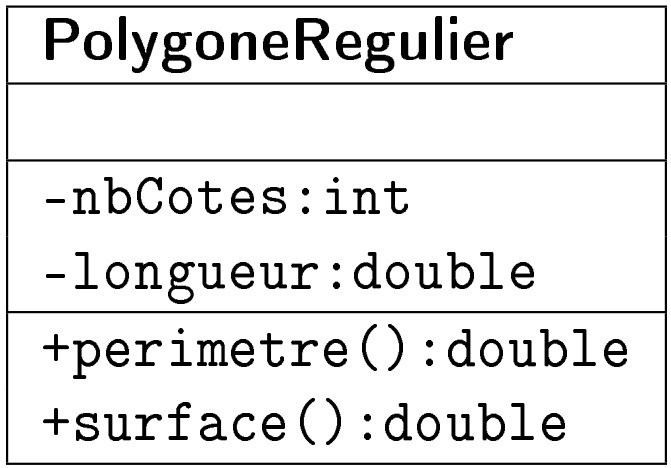
Pour décrire les objets et leurs attributs, on utilise différents diagrammes :

- Diagrammes de classe
- Diagrammes d'objet

Diagrammes de classe

Ils sont utilisés pour présenter les classes de l'application ainsi que les différentes relations entre celles-ci.

Ils décrivent **le comportement et le type d'un ensemble d'objets**.



Diagrammes d'objet

Ils permettent de représenter les instances des classes, c'est-à-dire les objets.

En plus des diagrammes de classes, **ils expriment l'état des objets**, ce qui permet d'exprimer des contextes d'exécution.

pr1 : PolygoneRegulier
`nbCotes=6`
`longueur=8.7`

pr2 : PolygoneRegulier
`nbCotes=4`
`longueur=12.8`

Les relations entre les différents objets

- L'association
- L'agrégation
- La composition simple ou multiple
- L'héritage

L'association

Une **association exprime une connexion sémantique entre deux classes**.

Elle décrit un groupe de liens ayant une structure et une sémantique commune. Elle exprime une relation structurelle entre deux classes.

Exemple : Une `Personne` peut reserver 1 ou plusieurs `Vol`

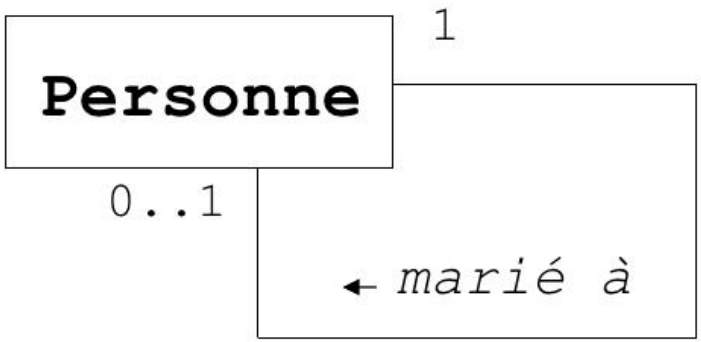
L'association (traduction en Java)

```
public class Personne
{
  private String nom;
  private nbVol = 0;
  private Vol[] lesVols;
  // l'association est représentée par le vecteur lesVols

  public Personne( String nom ){
    lesVols = new Vol[10];
    this.nom = nom;
  }

  public void reserve( Vol v ){
    lesVols[nbVol] = v;
    nbVol++;
  }
 ...
}
```

L'association d'une classe sur elle-même

Il également est possible de définir une association entre une classe et elle-même.

Exemple : Une `Personne` peut être mariée à une autre `Personne`



L'association d'une classe sur elle-même (en Java)

```
public class Personne{
  private String nom;
  private boolean marie = false;
  private Personne epoux = null;
  // l'association est représentée par la Personne epoux.

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

  public void seMarie( Personne p ){
    epoux = p;
    marie = true; }

 public boolean estMarie(){
   return marie;
   }
  ...
}
```

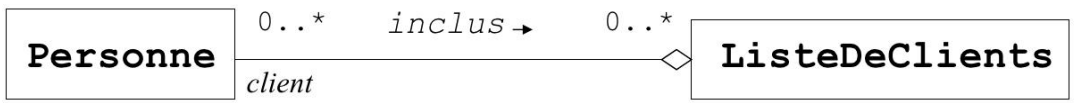
L'agrégation

L'**agrégation exprime une association avec relation de subordination**.

Elle est représentée par un trait reliant les deux classes et dont l'origine (classe contenante) se distingue par un losange de l'autre extrémité (classe subordonnée).

La classe contenante "regroupe" la classe subordonnée.

Exemple : L'objet `ListeDeClients` utilise 1 ou plusieurs instances de la classe `Personne`.



L'agrégation (traduction en Java) (1/2)

```
public class ListeClients{
  private Personne[] laListe;
  // l'agrégation est représentée par  le vecteur de Personne laListe
  private int nbClients=0;

  public ListeClients(){
    laListe = new Personne[10];
  }

  public void ajouter( Personne p ){
    laListe[nbClient] =p;
    nbClients++;
  }

  public Personne[] getListe(){
    return laListe;
  }
}
```

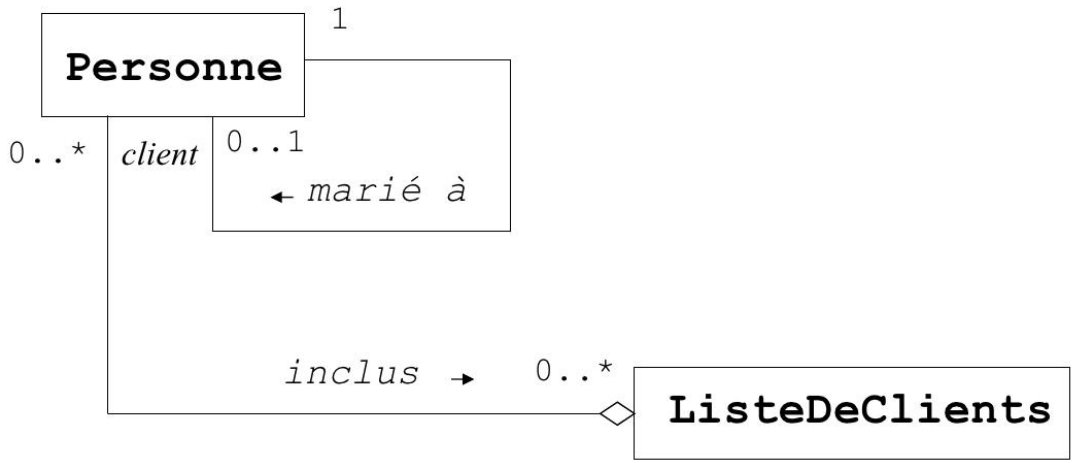
L'agrégation (traduction en Java) (2/2)

```
public class Personne{
  private String nom;

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

  public boolean estClient( ListeClients l ){
   for (int i=0;i<l.lenght;i++){
       if (l[i] ==this)
          return true;
   }

   return false;
  }
}
```

Combinaison Association et Agrégation

Il est possible de combiner l'association et l'agrégation.

Exemple : L'objet `ListeDeClients` utilise 1 ou plusieurs instances de la classe `Personne` ET une `Personne` peut être mariée à une autre `Personne`



Association + Agrégation (traduction en Java) (1/2)

```
public class ListeClients{
  private Personne[] laListe;
  // l'agrégation est représentée par le vecteur de Personne laListe
  private int nbClients=0;

  public ListeClients(){
    laListe = new Personne[10];
  }

  public void ajouter( Personne p ){
    laListe[nbClient] =p;
    nbClients++;
  }

  public Personne[] getListe(){
    return laListe;
  }
}
```

Association + Agrégation (traduction en Java) (2/2)

```
public class Personne{
  private String nom;
  private Personne epoux = null;
  // association représentée par epoux.
  private boolean marie = false;

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

  public void seMarie( Personne p ){
    epoux = p;
    marie = true; }

  public boolean estMarie(){
   return marie; }

 public boolean estClient( ListeClients l ){
   for (int i=0;i<l.lenght;i++){
      if (l[i] ==this)
         return true;}
   return false;}
}
```

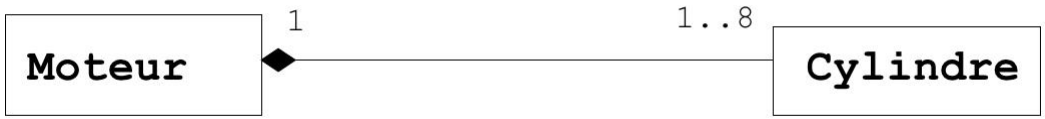
La composition

La **composition est une variante plus forte de l'agrégation**.

Dans une composition, le cycle de vie des deux classes en relation est dépendant. Si la classe contenante est détruite, la classe subordonnée est détruite.

Elle est représentée par un trait reliant les deux classes et dont l'origine (classe contenante) se distingue par un losange noir de l'autre extrémité (classe subordonnée).

Exemple : L'objet `Moteur` utilise de 1 à 8 instances de la classe `Cylindre`, ET un `Cylindre` n'existe qu'au sein d'un `Moteur`



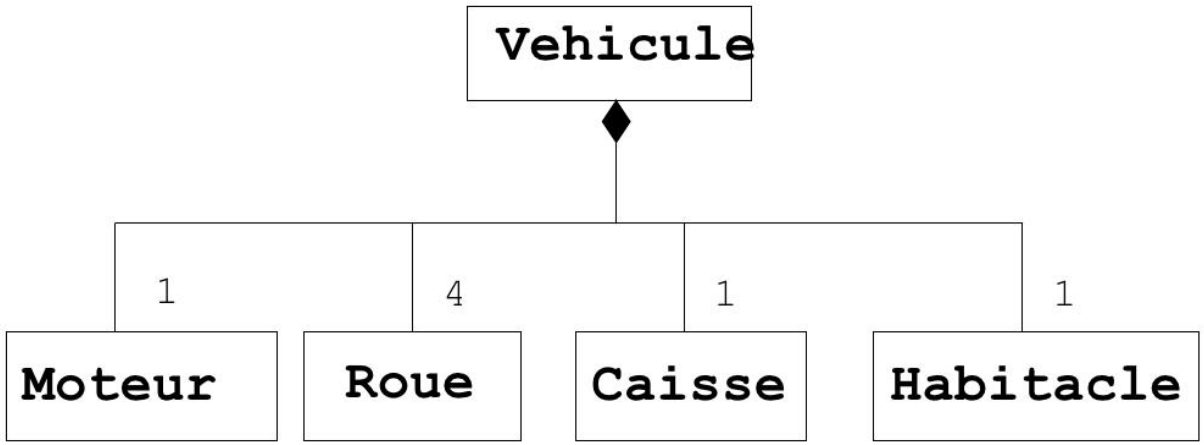
La composition (traduction en java)

```
public class Moteur
{
  private Cylindre[] lesCylindres;
  // la composition est représentée par le vecteur lesCylindres
  public Moteur()
  {
    lesCylindres = new Cylindre[8];
  }
...
}
```

La composition multiple

La composition peut être multiple : **la classe contenante a alors plusieurs classes subordonnées**.

Exemple : L'objet `Vehicule` utilise de 1 instance de la classe `Moteur`, 4 de la classe `Roue`, 1 de la classe `Caisse`, et 1 de la classe `Habitacle`.



La composition multiple (traduction Java)

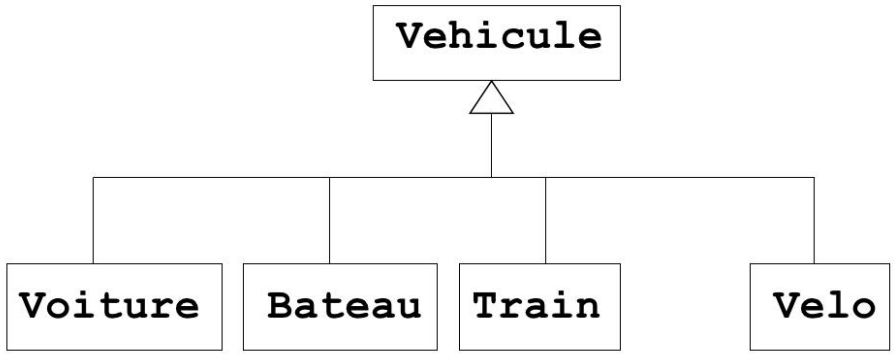
```
public class Vehicule
{
    private Moteur moteur;
    private Roue[] roues;
    private Caisse caisse;
    private Habitacle habitacle;
    // la composition multiple est représentée par moteur, le tableau
    // roues, caisse et  habitacle
    public Vehicule(){
      moteur = new Moteur();
      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;
    }
    ...
}
```

L'héritage

La **relation d'héritage indique qu'une "sous-classe" (classe fille) est une spécification de la "super-classe" (classe mère)**.
La classe fille hérite de tous les attributs et méthodes de la classe mère, on va du général au particulier.

Elle est représentée par un trait reliant les deux classes et dont l'origine (classe mère) se distingue de l'autre extrémité (classe fille) par un triangle.

Exemple : La classe mère `Vehicule` regroupe toutes les caractéristiques communes aux véhicules, et les classes filles `Voiture`, `Bateau`, `Train`, et `Velo` héritent de ces caractéristiques ET se spécialisent à leurs fonctions.



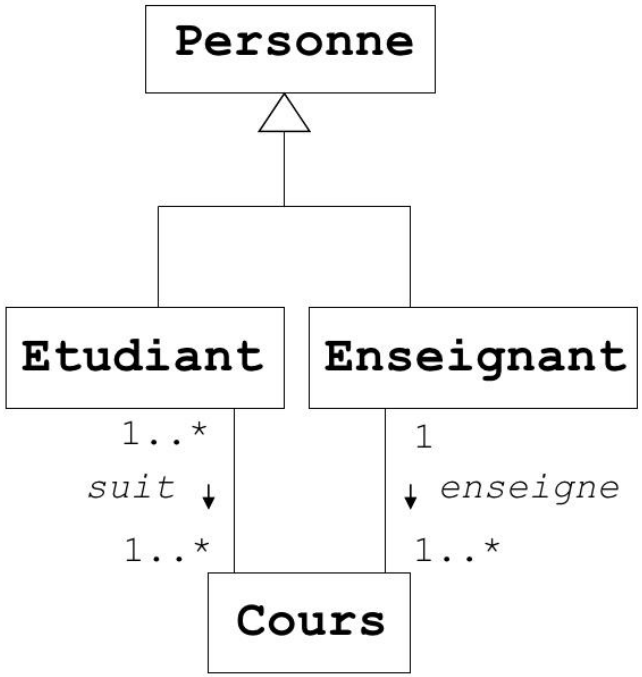
L'héritage (traduction Java)

```
public class Vehicule{
    ...
}

public class Voiture extends Vehicule{
// l'heritage est représenté par le mot clé extends
    ...}

public class Bateau extends Vehicule{
    ...}

public class Train extends Vehicule{
    ...}

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

Combinaison Association et Héritage

Il est possible de combiner l'association et l'héritage.

Exemple : La classe mère `Personne` regroupe toutes les caractéristiques communes aux personnes, et les classes filles `Etudiant` et `Enseignant` héritent de ces caractéristiques ET se spécialisent à leurs fonctions.
Les objets `Etudiant` peuvent suivre 1 ou plusieurs instances de la classe `Cours`, et les objets `Enseignant` peuvent ensiegner 1 ou plusieurs instances de la classe `Cours`.



Association + Héritage (traduction en Java) (1/2)

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

public class Cours{
    ... }

public class Etudiant extends Personne{
// l'heritage est représenté par le mot clé extends
  private Cours[] coursSuivis;
  private int nbCours =0;
// l'association est représentée par le vecteur lesCours

  public Etudiant( String nom ){
     coursSuivis = new Cours[10];
     this.nom = nom; }

  public void suit( Cours cours ){
     coursSuivis[nbCours] = cours;
     nbCours++;}
}
```

Association + Héritage (traduction en Java) (1/2)

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

public class Cours{
    ... }

public class Enseignant extends Personne{
  private Cours[] coursEnseignes;
  private int nbCours =0;

  public Enseignant( String nom ){
    coursEnseignes = Cours[8];
    this.nom = nom;}

  public void enseigne( Cours cours ){
    coursEnsignes[nbCours] = cours;
     nbCours++;}
}
```

Exemple (1/6)

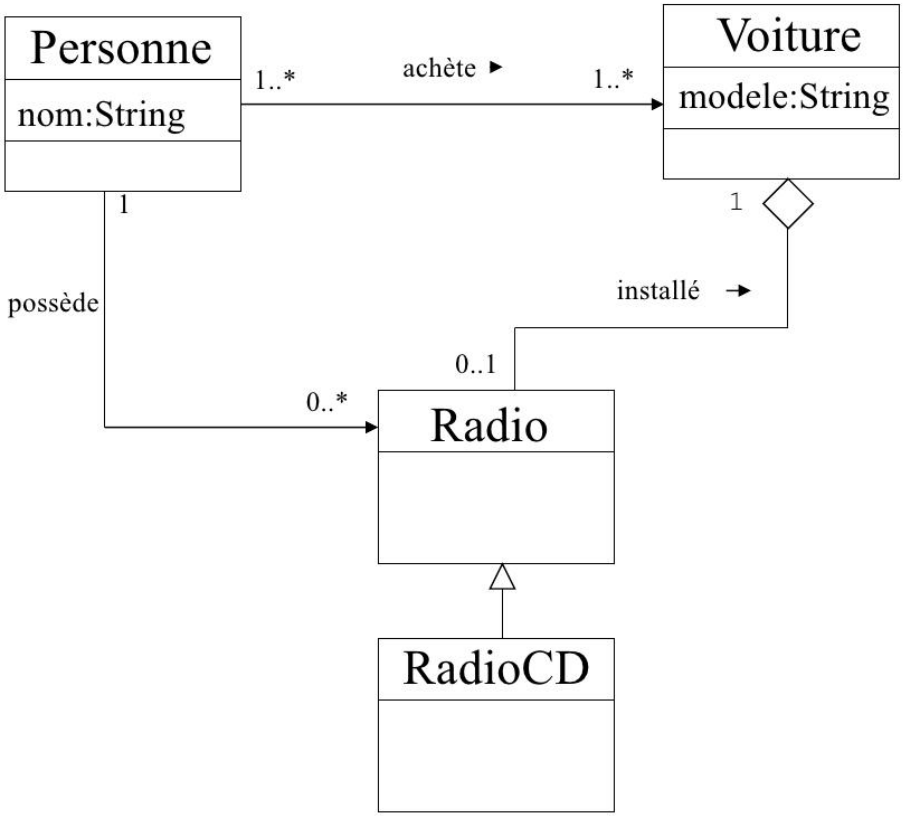
Une personne est définie par son nom et peut acheter une ou plusieurs voitures qui sont définies par leur modèle.

Chaque voiture peut posséder une radio (avec ou sans lecteur de CD).

Cet autoradio est extractible. Le propriétaire de l'autoradio est une personne qui n'est pas obligatoirement celui de la voiture.

1. Tracer le diagramme UML correspondant
2. Implanter ce diagramme en Java

Exemple (2/6)


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

Exemple (3/6)

```
public class Radio
{
  protected boolean installe;
  // l'agregation est representee par le boolean installe

  public Radio(){
    this.installe=false; }

  public void setInstalle(boolean b){
    this.installe = b; }

  public boolean getInstalle(){
    return this.installe; }
}

public class RadioCD extends Radio
{
  private boolean CDOK=true;
  public RadioCD(){
    this.installe=false; }
  public void casse(){
    this.CDOK = false;
  }
}
```

Exemple (4/6)

```
public class Voiture
{
  private String modele;
  private Radio radio = null;
  // l'agregation est representee par le Radio radio

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

  public void installe( Radio radio ){
    if (!radio.getInstalle()){
      this.radio = radio;
      this.radio.setInstalle(true);
  }}

  public void desinstalle(){
    if (this.installe !=null){
      this.radio.setInstalle(false);
      this.radio=null;
  }}

  public Radio getRadio(){
     return radio; }
}
```

Exemple (5/6)

```
public class Personne
{
  private String nom;
  private Radio[] radios;  //association par le vecteur radios
  private Voiture[] voitures; //association par le vecteur voitures
  private int nbRadios=0;
  private int nbVoitures=0;

  public Personne( String nom ){
    this.nom = nom;
    voitures = new Voiture[4];
    radios = new Radio[10];}

  public void possede( Radio radio ){
    radios[nbRadios] = radio;
    nbRadios++;}

  public void achete( Voiture voiture ){
    voitures[nbVoitures] = voiture;
    nbVoitures++;}
}
```

Exemple (6/6)

```
public class Exemple{
  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() );
  }
}
```

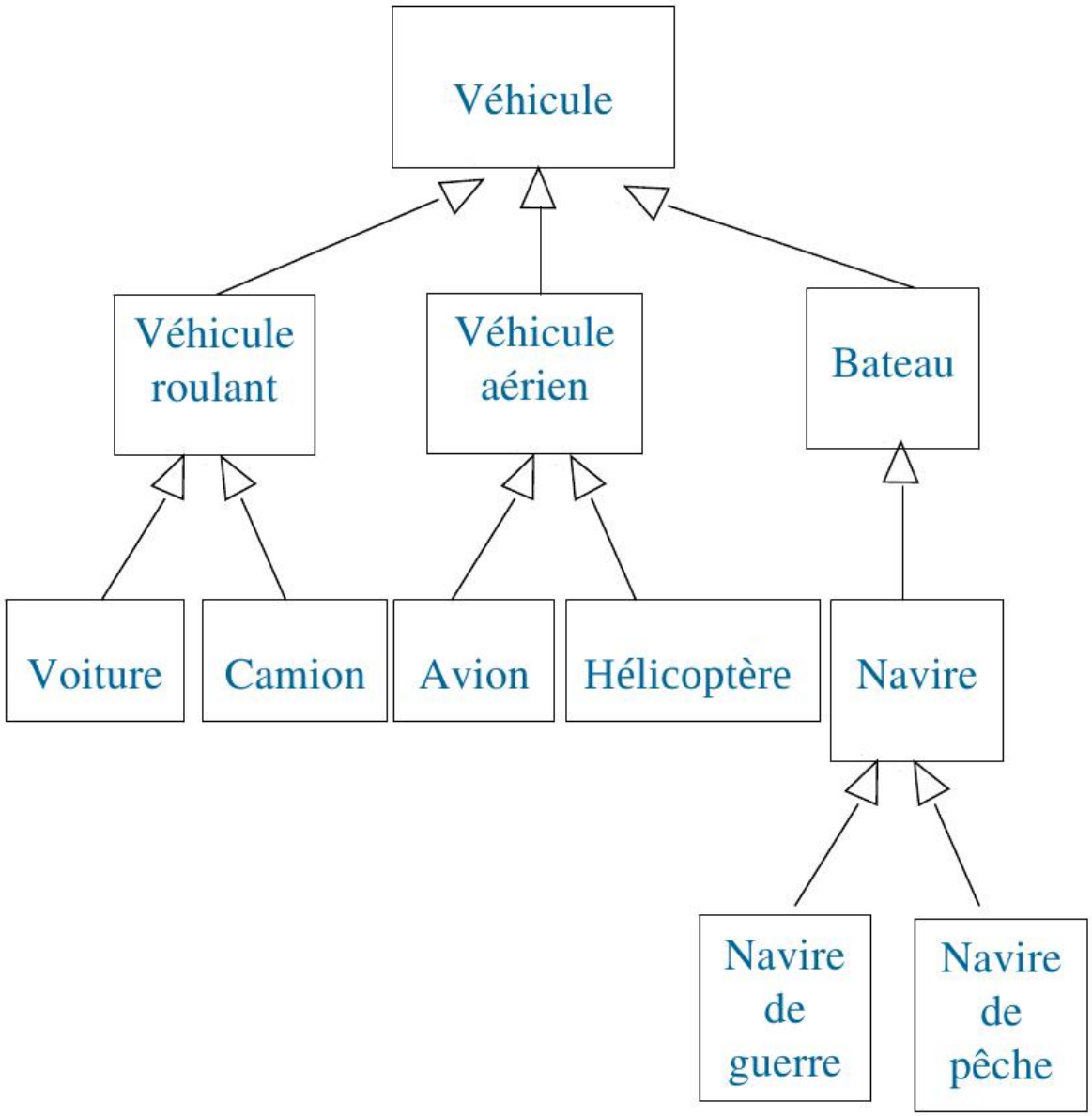
L'héritage

Le principe de l'héritage (1/3)

- Les hiérarchies de classes (classification) permettent de gérer la complexité en **ordonnant les objets au sein d'arborescence de classes d'abstraction croissante**.
- Les classes descendantes héritent des propriétés des classes ancêtres.
- **Exemple :**

  `Vertébrés` $\rightarrow$ `Mammifères` $\rightarrow$ `Hominidés` $\rightarrow$ `Hommes`

Le principe de l'héritage : Exemple (2/3)

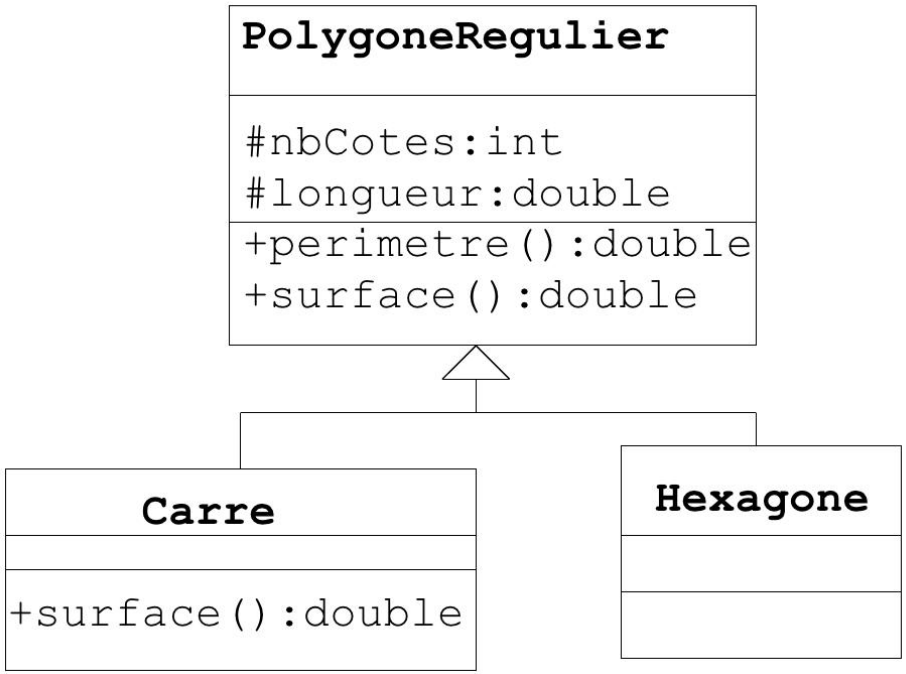


Le principe de l'héritage (3/3)

- Une classe ancêtre d'une classe descendante est appelée **super-classe**.
- La classe decendante **possède obligatoirement tous les attributs et méthodes de ses super-classes**.
- 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

Héritage et diagramme de classe

Il est possible de représenter l'héritage par un diagramme de classes.



Héritage (traduction en Java) (1/2)

```
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;}
}
```

Héritage (traduction en Java) (2/2)

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

public class Exemple{
  public static void main(String[] args){
    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 la classe Carre
    //puis celle de la classe Hexagone sont invoquées
  }
}
```

La protection lors de l'héritage

- 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

Les 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

Exemple (1/4)

Nous revenons vers la classe `Radio`. On souhaite créer un nouveau modèle de radio d'écouter la Radio et de lire des CDs.

1. Concevoir une nouvelle classe `Radio`.
2. Concevoir une nouvelle classe `RadioCD`.
3. 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

Exemple (2/4)

```
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]);
  }

  public void ecouter(double frequence){
    System.out.println("Ecoute de la station : " + frequence);
  }
}
```

Exemple (3/4)

```
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");}
}
```

Exemple (4/4)

```
public class Exemple{
  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);
  }
}
```

La surcharge et la 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**.

  Exemple :

```
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, lors de l'héritage, **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**.

La surcharge et la redéfinition (exemple)

```
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) ;
  }
}
```

La surcharge et la redéfinition (exemple)

```
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) ;
  }
}
```

Affichage du programme :

```
Aucune valeur définie
valeur flottante égale à 3.14159
```

La surcharge et la redéfinition (ambiguité)

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

La notion de Sous-classement

Il est possible d'affecter un objet de classe mère à un objet de classe fille.

Supposons que la classe `Hexagone` (classe fille) hérite de la classe `PolygoneRegulier` (classe mère).

```
PolygoneRegulier poly = new PolygoneRegulier(5,2.5);
Hexagone hexa = new Hexagone(4.67);
poly = hexa;
```

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 `PolygoneRegulier`.

Pour la même raison, et puisque tout est objet, il est tout aussi légal d'écrire :

```
Object o = new Object();
o = hexa;
```

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 (Exemple)

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, car le type de `poly` (`PolygoneRegulier`) ne possède pas de méthode `dessiner()`

Sous-classement et sous-typage (Exemple)

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

Le transtypage (cast) - Exemple

Le cast consiste à convertir une valeur d'un type dans un autre type.

```
hexa = poly;
```

Cette affectation est illégale : types incompatibles

```
poly = hexa;
```

Pour rendre possible l'appel à la méthode :

```
poly.dessiner();
```

Il faut vérifier si `poly` contient une référence sur une instance d'`Hexagone` :

```
if ( poly instanceof Hexagone ){
    Hexagone h = (Hexagone)poly; // conversion de type
    h.dessiner();}


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 ou une méthode d'une sous-classe peut avoir le même nom que celle de sa super-classe
- Pour utiliser la variable d'instance ou la méthode de la super-classe, on utilise le mot clé `super`
- **Exemple :**

  Appel de la méthode `surface()` de la classe mère `PolygoneRegulier` d'un `Hexagone h` :

```
System.out.println("La surface de h est " + super.surface());
```

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;
  }
  ...
}
```