INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Glossaire du jour :

- Héritage
- **extends**
- Redéfinition
- **protected**
- Polymorphisme
- **abstract**

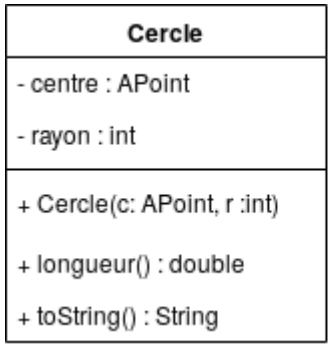
Quelques bases sur la programmation orientée objet

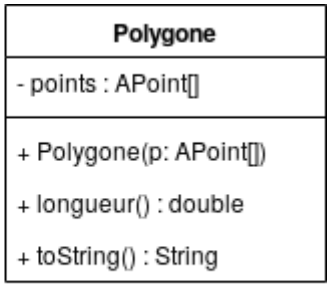
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Quel est le problème ?

Problème introductif ...

- Considérons deux concepts : un **cercle** et un **polygone**





- Problématique :
  - Comment stocker ces 2 entités différentes dans un tableau ?
  - En effet, **Cercle[]** ne peut contenir que des cercles et **Polygone[]** que des polygones.

Solution avec l'héritage

- Héritage = définition d'une classe par extension
  - Définir une classe **ancêtre** dont tous les **descendants** héritent de ses propiétés.
  - Héritage se fait par le mot clef « **extends** »

- L'ancêtre **regroupe toutes les propriétés communes** (méthodes+attributs) des descendants.

- Les descendants ont la possibilité de se spécialiser en ayant **chacun sa propre définition** de ces propriétés.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Avant d’aller plus loin quelques rappels

Notions de visibilité

Rappels de première année

- Un attribut et une méthode peuvent être :
  - **public** (accessible pour tout le monde)
  - **private** (accessible uniquement pour la classe courante)

- Exemple :

```
public class ClasseA{
    private int attribut1;
    public int attribut2;
    // ...

    public void methode1() {
        // ... ;
    }
    private void methode2() {
        // ... ;
    }
}
```

```
public class ClasseB{

    public void demo() {
        ClasseA monA = new ClasseA() ;
        monA.attribut2 ;
        monA.methode1() ;

        monA.attribut1 ;
        monA.methode2() ;
    }
}
```

Notions de visibilité

Rappels de première année

- Un attribut et une méthode peuvent être :
  - **public** (accessible pour tout le monde)
  - **private** (accessible uniquement pour la classe courante)

- Exemple :

```
public class ClasseA{
    private int attribut1;
    public int attribut2;
    // ...

    public void methode1() {
        // ... ;
    }
    private void methode2() {
        // ... ;
    }
}
```

```
public class ClasseB{

    public void demo() {
        ClasseA monA = new ClasseA() ;
        monA.attribut2 ;
        monA.methode1() ;

        monA.attribut1 ;
        monA.methode2() ;
    }
}
```

Ok car ils sont public

Notions de visibilité

Rappels de première année

- Un attribut et une méthode peuvent être :
  - **public** (accessible pour tout le monde)
  - **private** (accessible uniquement pour la classe courante)

- Exemple :

```
public class ClasseA{
    private int attribut1;
    public int attribut2;
    // ...

    public void methode1() {
        // ... ;
    }
    private void methode2() {
        // ... ;
    }
}
```

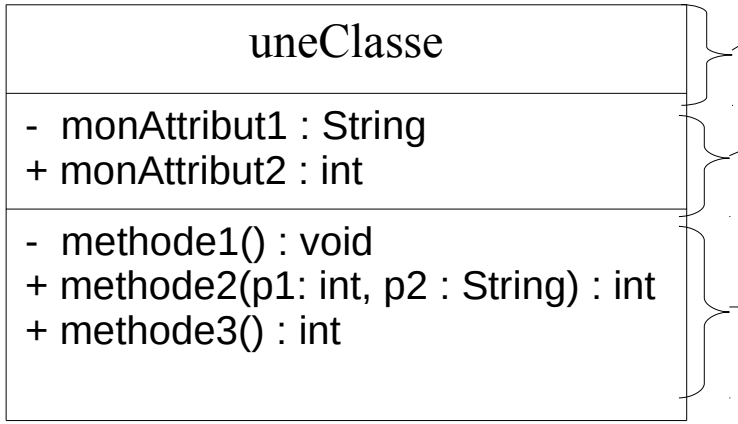
```
public class ClasseB{

    public void demo() {
        ClasseA monA = new ClasseA() ;
        monA.attribut2 ;
        monA.methode1() ;

        monA.attribut1 ;
        monA.methode2()
    }
}
```

Ok car ils sont public

Erreur car ils sont private

Représentation d'une classe

Rappels de première année



Zone 1 : Nom de la classe

Zone 2 : Liste des attributs.

Notation :

<visibilité> <nom> : <Type>

Zone 3 : Liste des méthodes.

Notation :

<visibilité> <nom> (<paramètres>) : <TypeRetour>

Notations pour la <visibilité>

+	pour public
-	pour private

Rappel : **surcharge** de méthodes

- Surcharge : une classe contenant plusieurs méthodes
  - avec le même nom et le même type de retour, mais …
  - avec nombre et/ou types des paramètres **différents**
- Les méthodes surchargées cohabitent.

```
public class ClasseA{
    private double monAttribut;
    // …

    public void maMethode(double p) {
        monAttribut = p ;
    }
    public void maMethode() {
        monAttribut = 0.0 ;
    }
}
```

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Héritage

Héritage : utilisation de **extends**

- Héritage est indiqué par le mot clef : **extends**

  ClasseB hérite de classeA s'écrit :

  public class classeB **extends** classeA

- Un descendant hérite des attributs et méthodes de l'ancêtre et peut en avoir des supplémentaires.

- Java : héritage simple uniquement *(arborescence)*
  - toute classe descend de **Object**
  - toute classe a **une** classe ancêtre *(sauf **Object**)*
  - une classe sans **extends** hérite de **Object**

Héritage : **redéfinition** de méthodes

- Dans une **classe héritée** :
  - **remplacement** de la définition d'une méthode
- Mise en œuvre :
  - Même **signature** (type de retour, nom et paramètres)

```
public class ClasseA {
        private double monAttribut;
        // ...

        public String toString() {
                return "Je suis une classe A";
        }
}
```

```
public class ClasseB extends ClasseA{
        // ...

        // Redéfinition de toString
        public String toString() {
                return "Je suis une classe B";
        }
}
```

Héritage : visibilité **protected**

- **protected** entre private et public
  (noté **#** en UML)

- Si un attribut ou une méthode est protected, il ou elle est **accessible dans la classe courante et chez ses descendants**

```
public class ClasseA {
        protected double monAttributA;
        // ...

        public void maMethodeA() {
                // ... ;
        }
}
```

```
public class ClasseB extends ClasseA{
        private double monAttributB;
        // ...

        public double maMethodeB() {
                return (monAttributA + monAttributB);
        }
}
```

MonAttributA est accessible dans *ClasseB*
car elle hérite de *ClasseA* et qu'il est *protected*

Exemple
Les véhicules

- On veut représenter 2 types de véhicules :
  - Véhicule
  - Véhicule à moteur à explosion (Vamex)
- Informations :
  - Tous les véhicules ont : marque, année de fabrication
  - Vamex : pareil + cylindrée (cm^3)
- Traitements:
  - Calculer l'âge
  - Afficher toutes les informations du véhicule
    - Véhicule générique : type, marque, année
    - Vamex : pareil + cylindrée
- Vamex a une particularité :
  - Une taxe est calculée en fonction de sa cylindrée

Exemple
Les véhicules (sans héritage)

```
public class Vehicule {

   private String marque; private int annee;

   public Vehicule( int uneAnnee,
                            String uneMarque){
       marque = uneMarque;
       annee = uneAnnee;
   }

   public String who(){
      return ("Je suis un véhicule");
   }

   public String toString(){
      return (who() + " - marque " + marque
              + " construit en " + annee);
   }

   public int age(){
     Calendar cal = Calendar.getInstance();
     return cal.get(Calendar.YEAR) - annee;
   }
}
```

```
public class Vamex {

   private String marque; private int annee ; private int cylindree;

   public Vamex( int uneAnnee, String uneMarque, int uneCylindree){
        marque = uneMarque;
        annee = uneAnnee;
        cylindree= uneCylindree;
    }

    public String who(){
       return ("Je suis un Vamex");
    }

    public String toString(){
       return (who() + " - marque " + marque +
        "construit en " + annee + " de
        cylindrée" + cylindree + " cm3");
    }

    public int age(){
      Calendar cal = Calendar.getInstance();
      return cal.get(Calendar.YEAR) - annee;
    }

    public double taxe(){
       return (cylindree*0.1+50);
    }
}
```

Exemple
Les véhicules (sans héritage)

```
public class Vehicule {

   private String marque; private int annee;

   public Vehicule( int uneAnnee,
                    String uneMarque){
      marque = uneMarque;
      annee = uneAnnee;
   }

   public String who(){
      return ("Je suis un véhicule");
   }

   public String toString(){
      return (who() + " - marque " + marque
              + " construit en " + annee);
   }

   public int age(){
      Calendar cal = Calendar.getInstance();
      return cal.get(Calendar.YEAR) - annee;
   }
}
```

Duplication

```
public class Vamex {

   private String marque; private int annee ; private int cylindree;

   public Vamex( int uneAnnee, String uneMarque, int uneCylindree){
      marque = uneMarque;
      annee = uneAnnee;
      cylindree= uneCylindree;
   }

   public String who(){
      return ("Je suis un Vamex");
   }

   public String toString(){
      return (who() + " - marque " + marque +
       "construit en " + annee + " de
       cylindrée" + cylindree + " cm3");
   }

   public int age(){
      Calendar cal = Calendar.getInstance();
      return cal.get(Calendar.YEAR) - annee;
   }

   public double taxe(){
      return (cylindree*0.1+50);
   }
}
```

Exemple
Les véhicules (sans héritage)

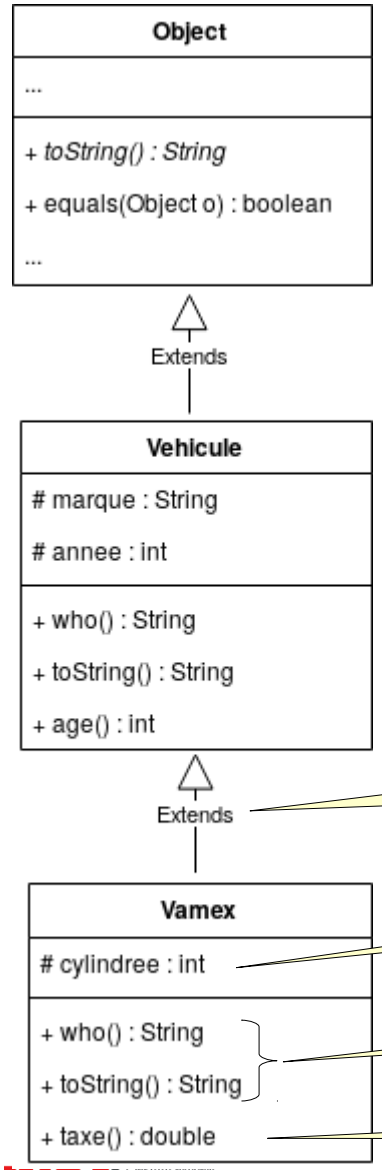
```
public class Vehicule {

   private String marque; private int annee;

   public Vehicule( int uneAnnee,
                    String uneMarque){
      marque = uneMarque;
      annee = uneAnnee;
   }

   public String who(){
      return ("Je suis un véhicule");
   }

   public S
      retur

   }

   public
      Calen
      retur
   }
}
```

```
public class Vamex {

   private String marque; private int annee ; private int cylindree;

   public Vamex( int uneAnnee, String uneMarque, int uneCylindree){
      marque = uneMarque;
      annee = uneAnnee;
      cylindree= uneCylindree;
   }

   public String who(){
      return ("Je suis un Vamex");
   }

   public double taxe(){
      return (cylindree*0.1+50);
   }
}
```

Duplication

- **Solution inefficace:**
  Attributs et méthodes dupliqués
- **Problème très pénalisant pour les applications réelles:**
  Si on veut gérer de nouveaux véhicules (motos, bateaux, avions, etc.), comment faire ?

Exemple
Les véhicules ... modélisation orientée objet



- Un Vamex est un véhicule particulier.
- Un Vehicule a une marque, une année de fabrication. Il peut afficher son type et ses informations et calculer son âge.
- Un Vamex a donc aussi une marque, une année de fabrication et peut aussi afficher son type et ses informations et calculer son âge ... **mais en plus** il a une cylindrée et peut calculer sa taxe

Vamex hérite de véhicule
il récupère les attributs et les méthodes de l'ancêtre

Attribut supplémentaire

Méthodes redéfinies

Nouvelle méthode

Exemple
Les véhicules

```
public class Vehicule {

   protected String marque;
   protected int annee;

   public Vehicule( int uneAnnee,
                    String uneMarque){
      marque = uneMarque;
      annee = uneAnnee;
   }

   public String who(){
      return ("Je suis un véhicule");
   }

   public String toString(){
      return (who() + " - marque " + marque
            + " construit en " + annee);
   }

   public int age(){
     Calendar cal = Calendar.getInstance();
     return cal.get(Calendar.YEAR) - annee;
   }
}
```

```
public class Vamex extends Vehicule  {

   protected int cylindree;

   public Vamex( int uneAnnee, String uneMarque,
                 int uneCylindree){
      annee = uneAnnee ;
      marque = uneMarque;
      cylindree= uneCylindree;
   }

   public String who(){
      return ("Je suis un Vamex");
   }

   public String toString(){
      return (who() + " - marque " + marque +
            "construit en " + annee + " de
            cylindrée" + cylindree + " cm3");
   }

   public double taxe(){
      return (cylindree*0.1+50);
   }
}
```

Exemple
Les véhicules

```
public class Vehicule {

   protected String marque;
   protected int annee;

   public V                 int uneAnnee,
   }

   public String who(){
      return ("Je suis un véhicule");
   }

   public String toString(){
      return (who() + " - marque " + marque
            + " construit en " + annee);
   }

   public int age(){
     Calendar cal = Calendar.getInstance();
     return cal.get(Calendar.YEAR) - annee;
   }
}
```

protected : Les classes filles auront accès à ces attributs

```
public class Vamex extends Vehicule  {

   protected int cylindree;

   public Vamex( int une                String uneMarque,

      annee = uneA
      marque = uneMarque;
      cylindree= uneCylindree;
   }

   public String who(){
      return ("Je suis un Vamex");
   }

   public String toString(){
      return (who() + " - marque " + marque +
            "construit en " + annee + " de
            cylindrée" + cylindree + " cm3");
   }

   public double taxe(){
      return (cylindree*0.      +50);
   }
}
```

Attribut supplémentaire

Redéfinitions de méthodes

Nouvelle méthode

Exemple
Les véhicules ... exemple d'utilisation

```
public class Principale {

   public static void main(String[] args) {

      Vamex b = new Vamex( 2009, "Peugeot", 1500);

      if ( ( b.taxe() > 0 ) & ( b.age() > 20 ))
            System.out.println( "Injuste !" );
   }
}
```

Exemple
Les véhicules ... exemple d'utilisation

```
public class Principale {

   public static void main(String[] args) {

      Vamex b = new Vamex( 2009, "Peugeot", 1500);

      if ( ( b.taxe() > 0 ) & ( b.age() > 20 ))
            System.out.println( "Injuste !" );
   }
}
```

Nouvelle méthode propre à Vamex

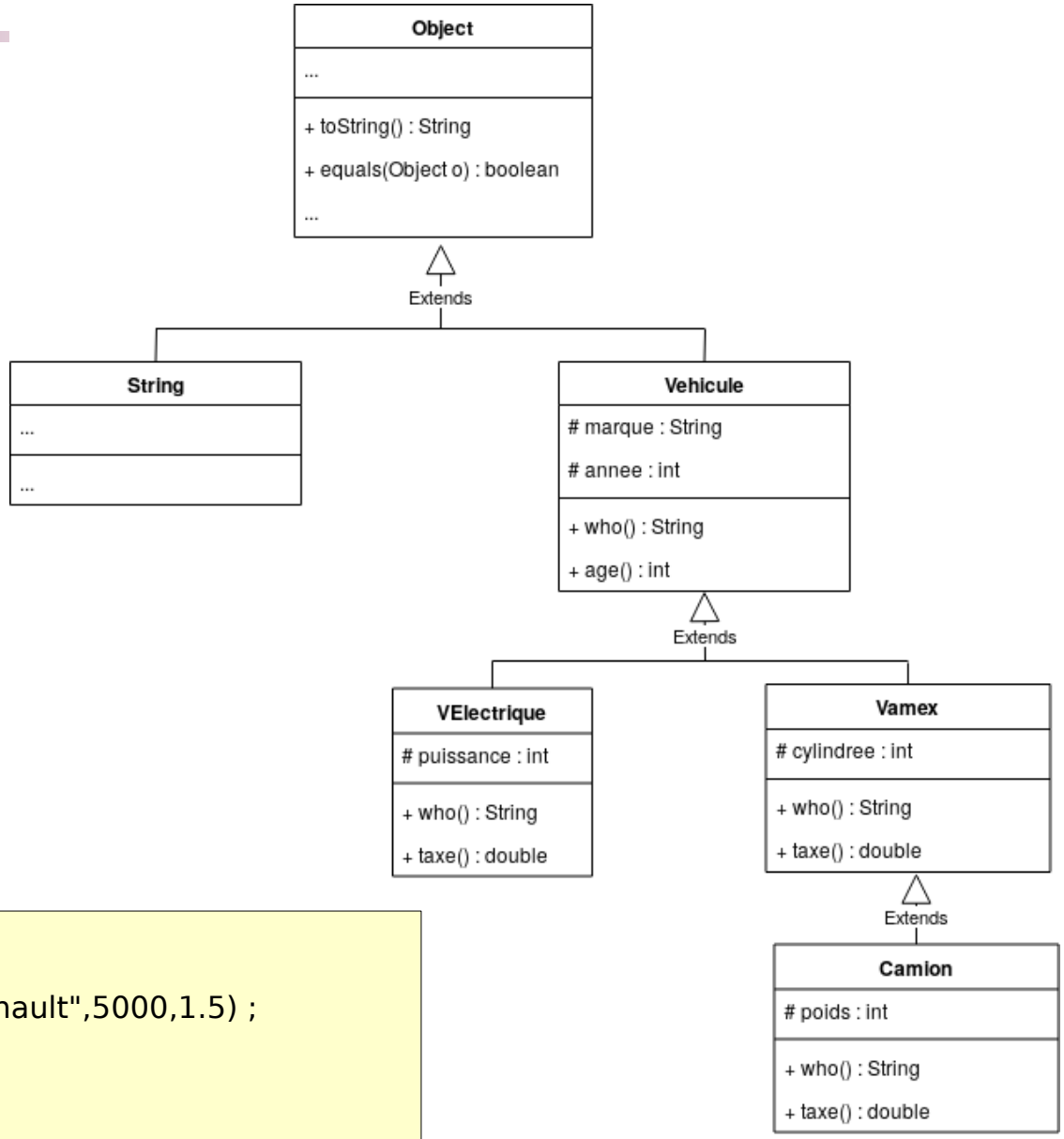
Méthode héritée de la classe ancêtre : Vehicule

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Polymorphisme

membre de UNIVERSITÉ DE LYON

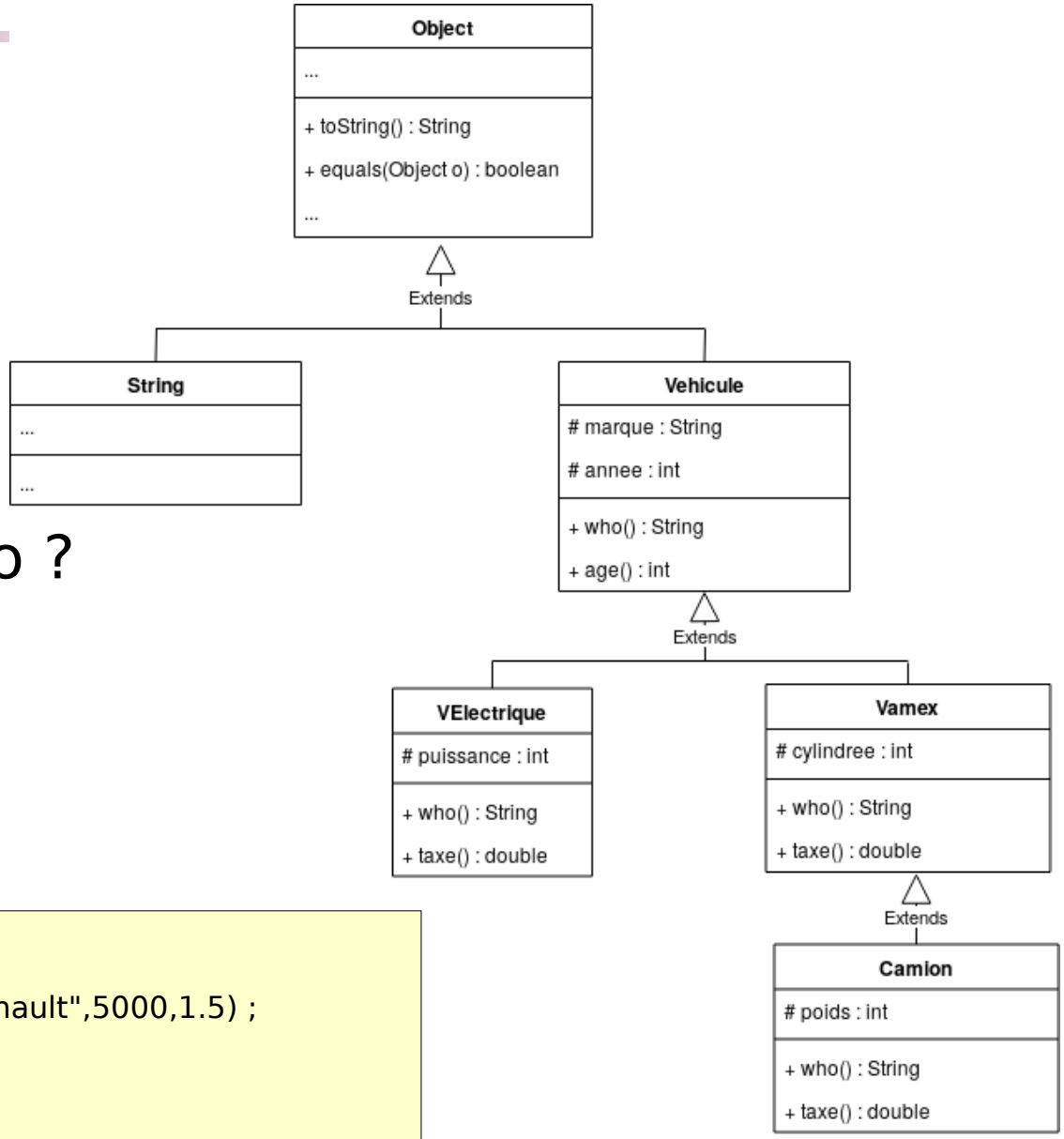
Polymorphisme : un objet a plusieurs types




```
public class UtilisationVehicules {
    public static void main(String[] a){
        Vamex v = new Camion(2017,"Renault",5000,1.5) ;
        Object o = v;
        String s = v;
    }
}
```


Exemple/Démo

Polymorphisme : un objet a plusieurs types



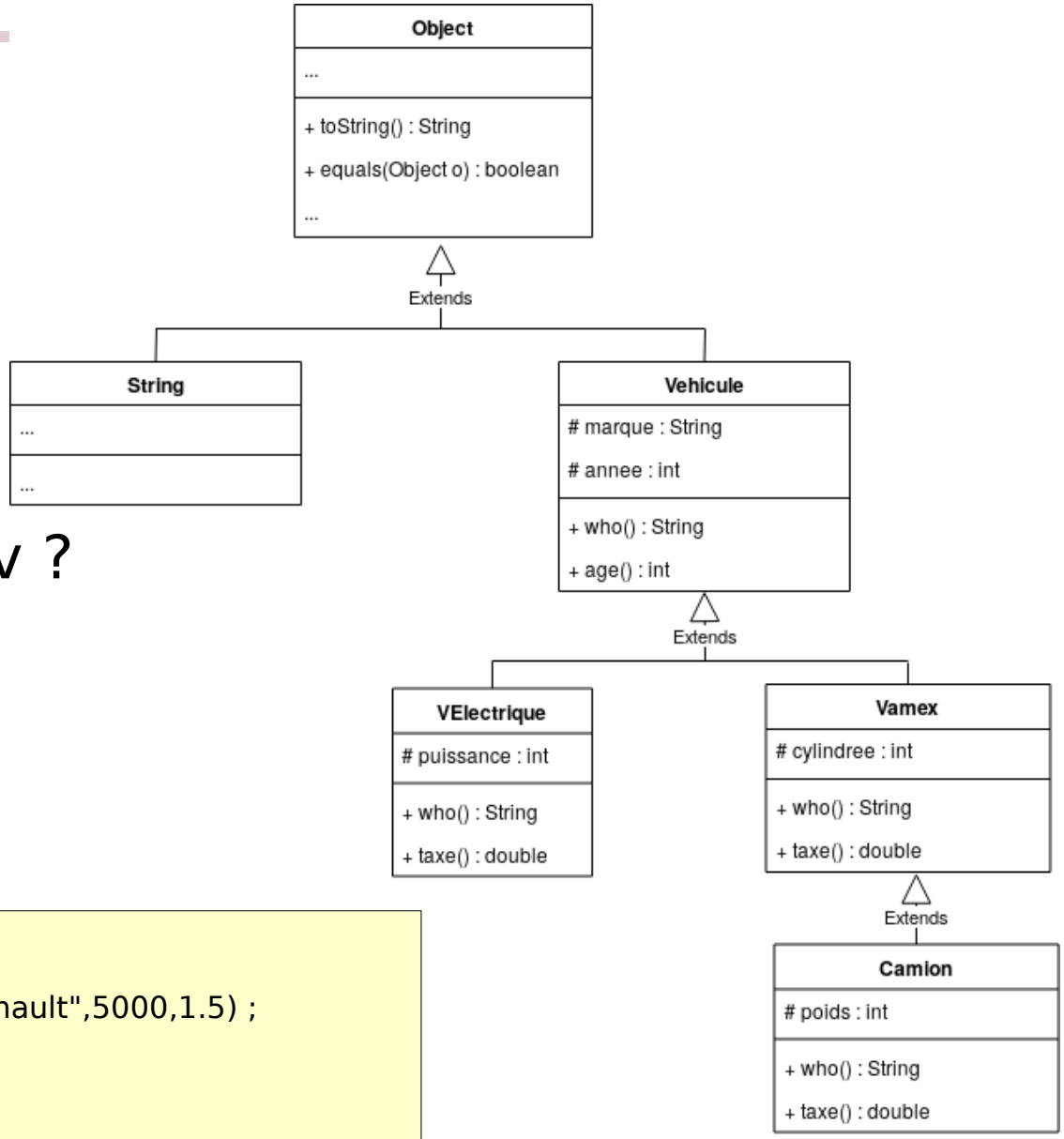
Quel est le type de o ?

- Object ?
- Vehicule ?
- Vamex ?
- Camion ?

```
public class UtilisationVehicules {
    public static void main(String[] a){
        Vamex v = new Camion(2017,"Renault",5000,1.5) ;
        Object o = v;
        String s = v;
    }
}
```


Exemple/Démo

Polymorphisme : un objet a plusieurs types



Quel est le type de v ?

- Object ?
- Vehicule ?
- Vamex ?
- Camion ?

```
public class UtilisationVehicules {
    public static void main(String[] a){
        Vamex v = new Camion(2017,"Renault",5000,1.5) ;
        Object o = v;
        String s = v;
    }
}
```


Exemple/Démo

Type statique vs type dynamique

```
public class UtilisationVehicules {
    public static void main(String[] a){
        Vamex v = new Camion(2017,"Renault",5000,1.5);
        Object o = v;
        String s = v;
    }
}
```

- Distinction :
  - Type **statique** : type de **déclaration** d'une variable
  - Type **dynamique** : type d'**instanciation** de l'objet (type réel, à l'exécution)
- Exemple:
  - Type statique de v : Vamex
  - Type dynamique v : Camion
  - Type statique de o : Object
  - Type dynamique o : Camion
  - v et o sont interprétables en Camion, Vamex, Vehicule et Object

Type statique vs type dynamique

La méthode la plus spécifique est toujours celle appelée

```
Vamex v1 ;                                  // Type statique de v1 : Vamex
Vamex v2;                                   // Type statique de v2 : Vamex

v1 = new Vamex (2016,"Saab",2000);          // Type dynamique de v1 : Vamex
v2 = new Camion (2017,"Renault",5000,1.5); // Type dynamique de v2 : Camion

System.out.println(v1.who()) ;
System.out.println(v2.who()) ;
```

Type statique vs type dynamique

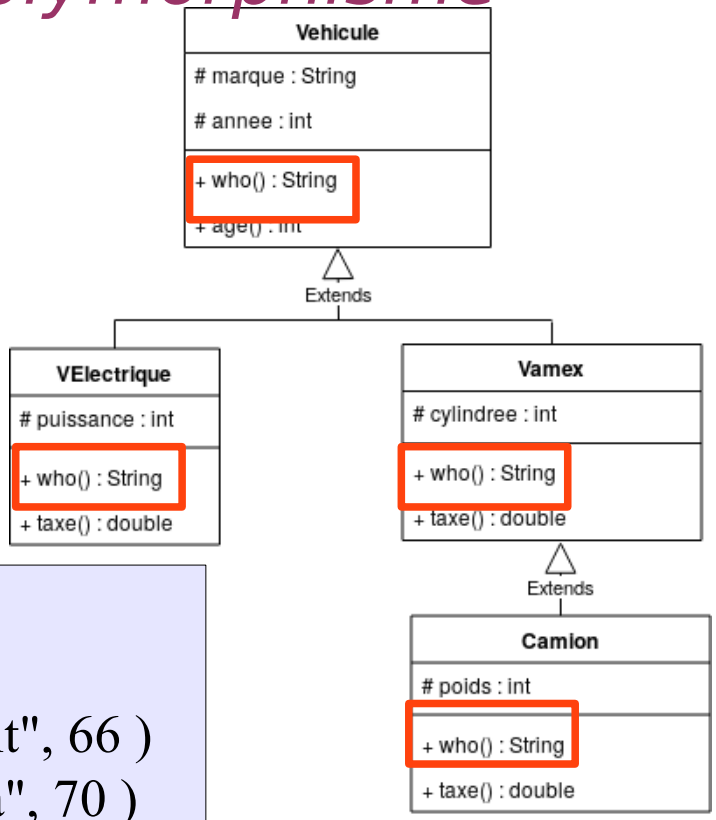
La méthode la plus spécifique est toujours celle appelée

```
Vamex v1 ;                          // Type statique de v1 : Vamex
Vamex v2;                           // Type statique de v2 : Vamex

v1 = new Vamex (2016,"Saab",2000);      // Type dynamique de v1 : Vamex
v2 = new Camion (2017,"Renault",5000,1.5); // Type dynamique de v2 : Camion

System.out.println(v1.who()) ;      Je suis un Vamex
System.out.println(v2.who()) ;      Je suis un Camion
```

Remarque :

Si redéfinition, la méthode la plus spécialisée est appelée.
Elle est choisie par rapport au type dynamique (type réel).

… c'est ce qu'on appelle le *polymorphisme*

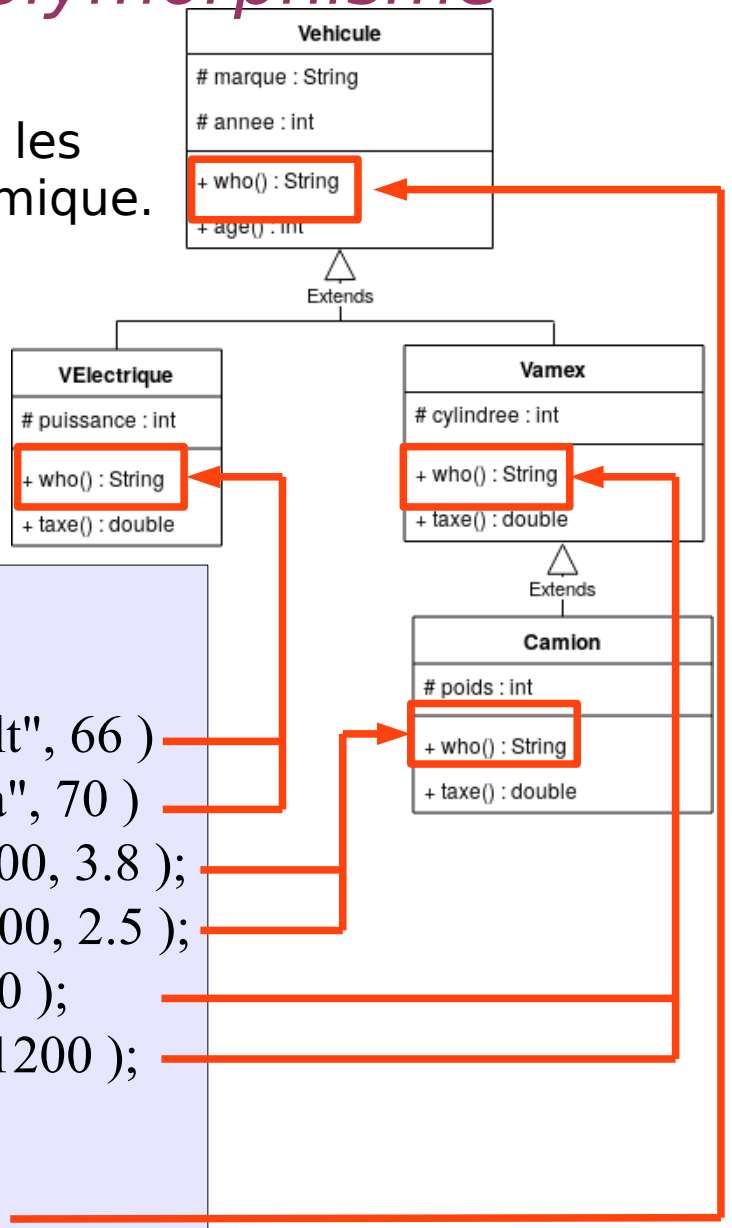



```
Vehicule[] myTab = new Vehicule[50];

myTab[0] = new VElectrique(  2013, "Renault", 66 )
myTab[1] = new VElectrique(  2012, "Toyota", 70 )
myTab[2] = new Camion(  1997, "MAN", 5500, 3.8 );
myTab[3] = new Camion( 2009, "Scania", 4200, 2.5 );
myTab[4] = new Vamex(  2005, "Ford",  1500 );
myTab[5] = new Vamex(  2005, "Peugeot",  1200 );

for(int i=0; i<6; i++)
        System.out.println( myTab[i].who());
```

… c'est ce qu'on appelle le *polymorphisme*

- On peut appeler la méthode **who()** sur tous les véhicules sans se soucier de leur type dynamique.
- La méthode s'adapte automatiquement au type dynamique.

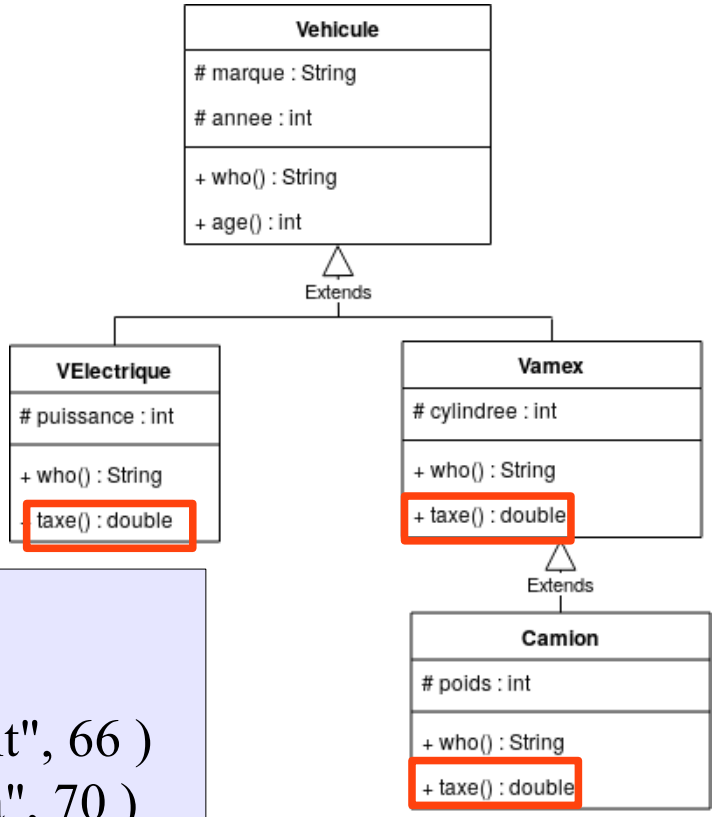
  -> Unification des traitements




```
Vehicule[] myTab = new Vehicule[50];

myTab[0] = new VElectrique(  2013, "Renault", 66 )
myTab[1] = new VElectrique(  2012, "Toyota", 70 )
myTab[2] = new Camion(  1997, "MAN", 5500, 3.8 );
myTab[3] = new Camion( 2009, "Scania", 4200, 2.5 );
myTab[4] = new Vamex(  2005, "Ford",  1500 );
myTab[5] = new Vamex(  2005, "Peugeot",  1200 );

for(int i=0; i<6; i++)
        System.out.println( myTab[i].who());
```

Petit problème ...

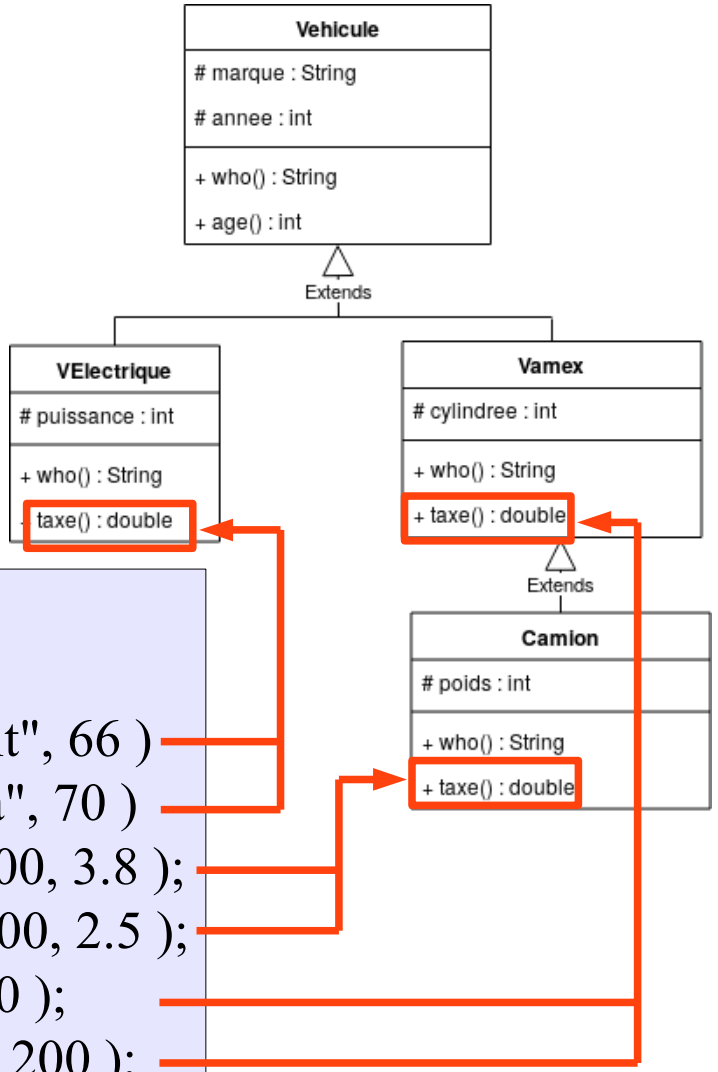
- Ça marche aussi
  avec **taxe()** ?




```
Vehicule[] myTab = new Vehicule[50];

myTab[0] = new VElectrique(  2013, "Renault", 66 )
myTab[1] = new VElectrique(  2012, "Toyota", 70 )
myTab[2] = new Camion(  1997, "MAN", 5500, 3.8 );
myTab[3] = new Camion( 2009, "Scania", 4200, 2.5 );
myTab[4] = new Vamex(  2005, "Ford",  1500 );
myTab[5] = new Vamex(  2005, "Peugeot",  1200 );

for(int i=0; i<6; i++)
        System.out.println( myTab[i].taxe());
```

Petit problème ...

- Ça marche aussi
  avec **taxe()** ?



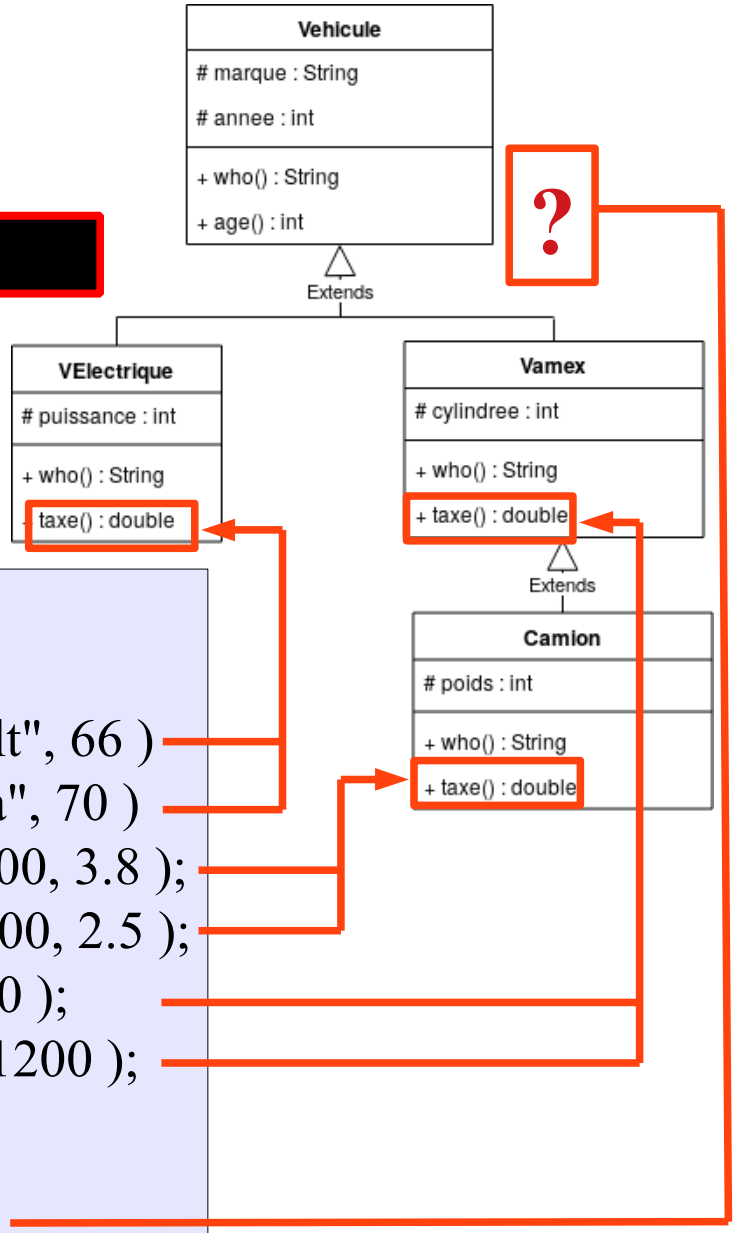

```
Vehicule[] myTab = new Vehicule[50];

myTab[0] = new VElectrique(  2013, "Renault", 66 )
myTab[1] = new VElectrique(  2012, "Toyota", 70 )
myTab[2] = new Camion(  1997, "MAN", 5500, 3.8 );
myTab[3] = new Camion( 2009, "Scania", 4200, 2.5 );
myTab[4] = new Vamex(  2005, "Ford",  1500 );
myTab[5] = new Vamex(  2005, "Peugeot",  1200 );

for(int i=0; i<6; i++)
        System.out.println( myTab[i].taxe());
```

Petit problème …

- NON

Compilation → Error: cannot find method taxe()

La méthode **taxe()** n'existe pas pour le type Vehicule




```
Vehicule[] myTab = new Vehicule[50];

myTab[0] = new VElectrique(  2013, "Renault", 66 )
myTab[1] = new VElectrique(  2012, "Toyota", 70 )
myTab[2] = new Camion(  1997, "MAN", 5500, 3.8 );
myTab[3] = new Camion( 2009, "Scania", 4200, 2.5 );
myTab[4] = new Vamex(  2005, "Ford",  1500 );
myTab[5] = new Vamex(  2005, "Peugeot",  1200 );

for(int i=0; i<6; i++)
        System.out.println( myTab[i].taxe());
```

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Classes abstraites
en Java

membre de UNIVERSITÉ DE LYON

Classes et méthodes abstraites

- Pour résoudre ce problème de compilation, on ajoute la méthode *taxe* dans la classe Vehicule.
- Comme on ne peut pas produire son implémentation, on la définit comme *abstraite*.
- Comme elle possède une méthode abstraite, la classe devient abstraite.

```
public abstract class Vehicule {

   protected String marque;
   protected int annee;

   public Vehicule( int uneAnnee,
                    String uneMarque){
            marque = uneMarque;
            annee = uneAnnee;
   }

   public String who(){
      return ("Je suis un véhicule");
   }

   (...)

   public abstract double taxe();
}
```

La classe est abstraite : elle ne produira jamais d'objets (erreur à la compilation).

```
Vehicule A = new Vehicule(2005, ''Nakamura'')
```

Error: Vehicule is abstract; cannot be instantiated

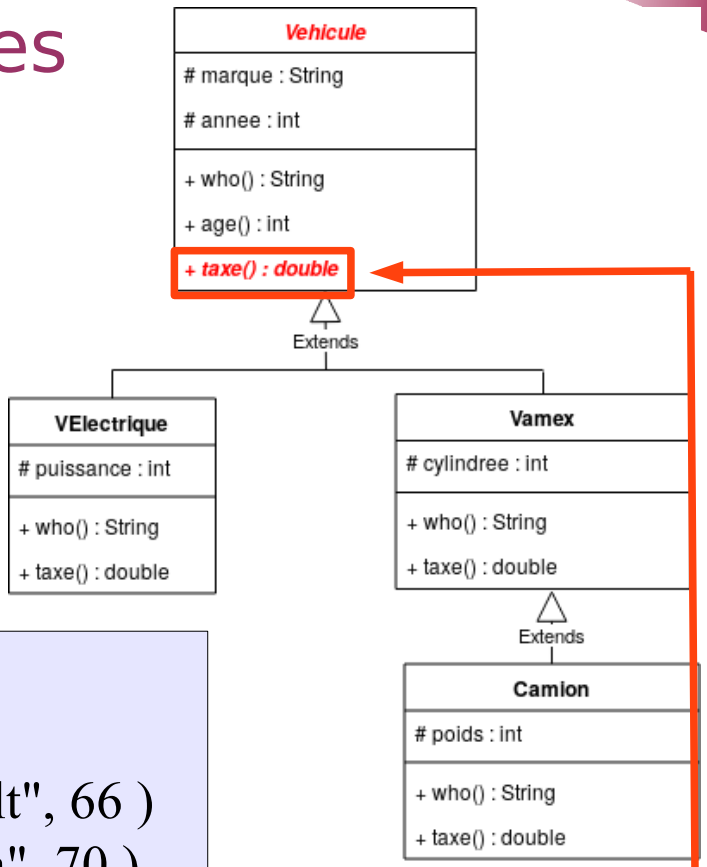
Méthode abstraite:
- Aucune implémentation (pas de corps)
- Impose l'implémentation dans les classes dérivées

Classes et méthodes abstraites

- Ça marche ?

  Maintenant, oui !

- Et cela empêche la création d'objets de la classe Véhicule ce qui correspond à la logique de notre modélisation.




```
Vehicule[] myTab = new Vehicule[50];

myTab[0] = new VElectrique(  2013, "Renault", 66 )
myTab[1] = new VElectrique(  2012, "Toyota", 70 )
myTab[2] = new Camion(  1997, "MAN", 5500, 3.8 );
myTab[3] = new Camion( 2009, "Scania", 4200, 2.5 );
myTab[4] = new Vamex(  2005, "Ford",  1500 );
myTab[5] = new Vamex(  2005, "Peugeot",  1200 );

for(int i=0; i<6; i++)
        System.out.println( myTab[i].taxe());
```

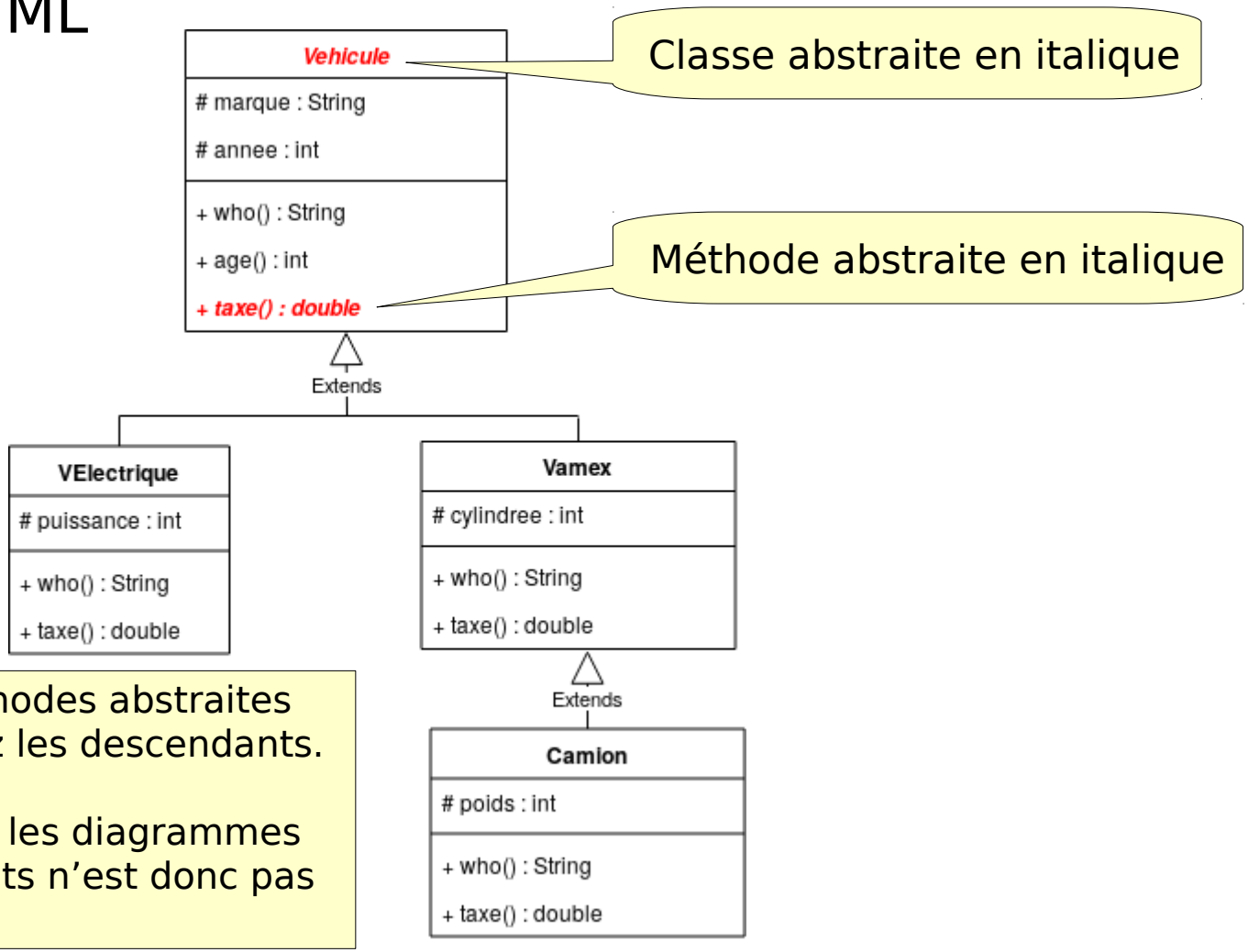
Bilan sur les classes et méthodes abstraites

- Si une classe contient des méthodes "abstract", elle est alors une classe "abstract"
  - On ne peut pas créer d'objets à partir d'une classe abstraite.
  - Une méthode abstraite n'a pas d'implémentation
  - L'implémentation d'une méthode abstraite est réalisée dans les classes dérivées.

- Finalement, une classe abstraite permet de définir dans une classe de base des fonctionnalités communes à toutes ses descendantes, tout en leur imposant de redéfinir certaines méthodes.

Bilan sur les classes et méthodes abstraites

- Syntaxe UML



Par nature, les méthodes abstraites sont redéfinies chez les descendants.

Leur présence dans les diagrammes UML des descendants n'est donc pas obligatoire.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Bilan sur l'héritage et le polymorphisme

membre de UNIVERSITÉ DE LYON

Bilan sur l'héritage et polymorphisme

- **Héritage : classe mère (ou ancêtre) et classes filles**

  L'héritage permet de mettre en commun des caractéristiques et des traitements entre différents types d'objets.

- **extends**

  classeB ***extends*** *classeA* signifie que *classeB* hérite des attributs/méthodes de *classeA* mais peut en avoir en plus. *classeB* est une spécialisation de *classeA*.

- 3 niveaux de visibilité **public / private / protected**

  public : accès possible pour tout le monde.
  private : accès restreint à la classe courante.
  protected : accès pour la classe courante et ses descendants.

Bilan sur l'héritage et polymorphisme

- **Polymorphisme : un objet a plusieurs types**

  Type statique à la déclaration.
  Type dynamique à l'exécution.
  La méthode la plus spécifique est toujours appelée.

- **abstract**

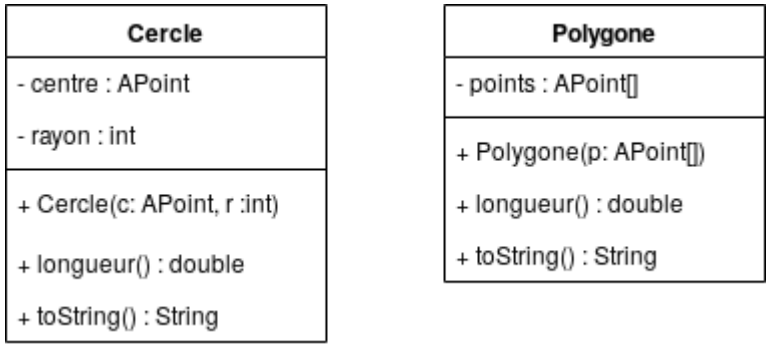
  Une classe est abstraite si elle possède une méthode abstraite.
  Une classe abstraite ne peut pas être instanciée.
  Une méthode abstraite est vide.
  Son implémentation est faite dans les classes dérivées.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Retour sur notre problème

Proposition de solution au problème initial

- Considérons deux concepts : un **cercle** et un **polygone**



- Création d'une classe ancêtre **Courbe**
  - Un tableau de Courbe peut stocker ces 2 entités différentes.
  - Dorénavant, les attributs et les méthodes similaires sont déclarées chez l'ancêtre.

Proposition de solution au problème initial