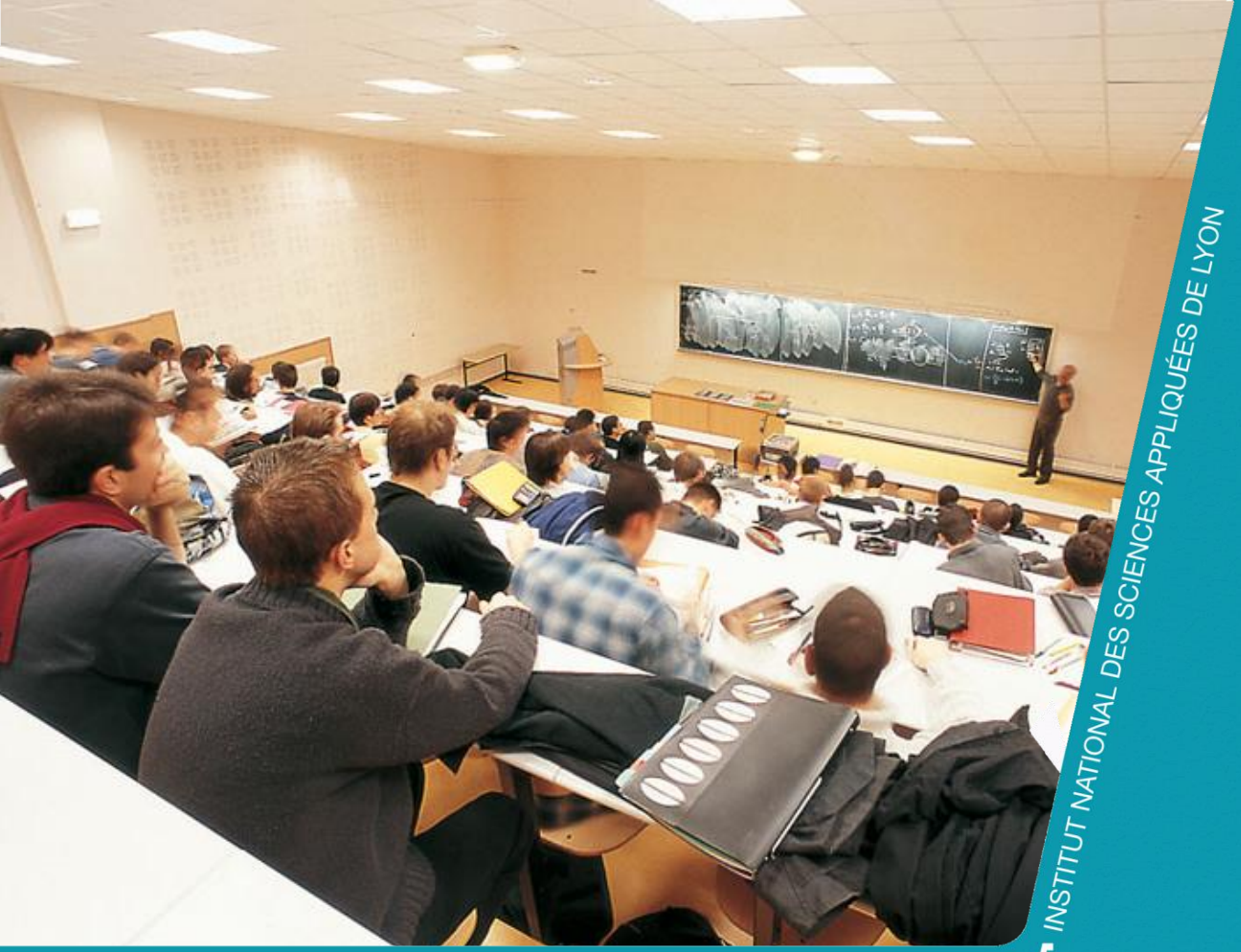
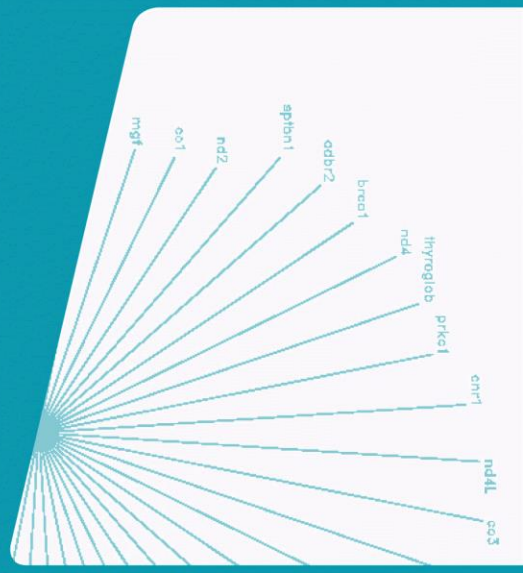


INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

FORMATION



Cours 3 – Héritage et polymorphisme

membre de
UNIVERSITÉ DE LYON

INSA LYON

Héritage

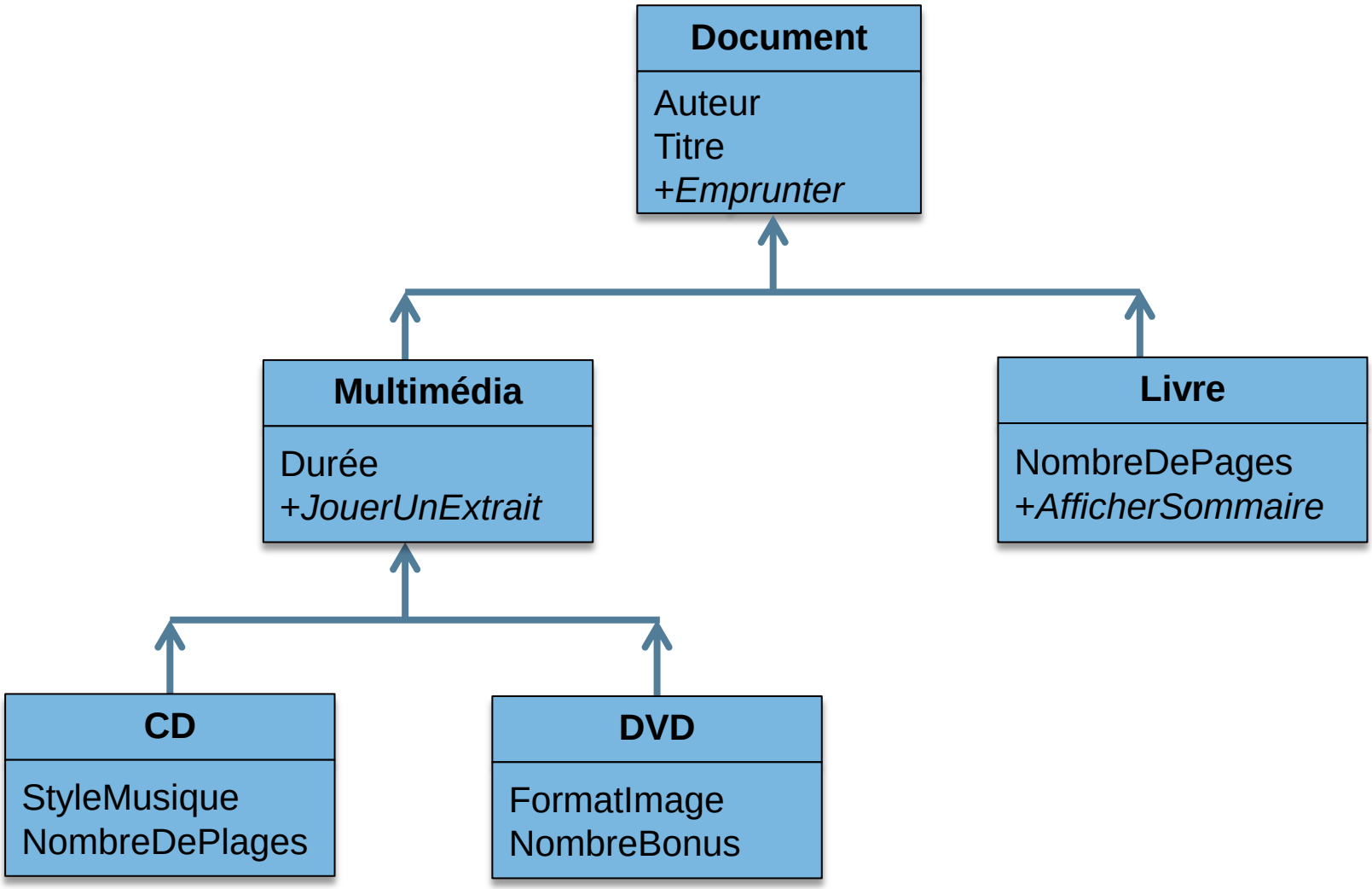
- **Définition**
  - **Héritage = principe permettant de créer une nouvelle classe à partir d'une classe existante**

- **Propriétés de l'héritage**
  - **Une classe B qui hérite d'une classe A dispose implicitement de tous les attributs et méthodes définis dans la classe A**
  - **Il y a des attributs et/ou méthodes supplémentaires dans la classe B**
  - **La relation traduit : B est « une sorte de » A**

- **Possibilité de créer une hiérarchie complète d'objets héritant les uns des autres.**

- **Construction: partir d'un objet relativement simple pour aller à des objets plus complexes ou plus spécialisés.**

Exemple : Médiathèque (UML)



Exemple : Les véhicules

- **On veut représenter et traiter des véhicules**
  - **2 types de véhicules: Véhicule et Véhicule à moteur à explosion**
- **Informations :**
  - **Pour tous les véhicules : Marque, Année de fabrication**
  - **Véhicules à moteur à explosion: Marque, Année de fabrication, Cylindrée (cm3)**
- **Traitements:**
  - **Calculer l'âge (= année actuelle – année de fabrication)**
  - **Donner le type du véhicule (véhicule ou véhicule à moteur ?)**
  - **Afficher toutes informations du véhicule**
    - **Véhicule générique: Type, Marque, Année**
    - **Véhicules à moteur à explosion: Type, Marque, Année, Cylindrée**
  - **Véhicules à moteur à explosion: calculer une taxe**

Solution classique (sans héritage)

```java
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 info(){
      return (who() + " - marque " + marque
                  + " construit en " + annee);
   }

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

```java
public class VAMEX {

   protected String marque;   protected int annee;  protected 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 info(){
      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 float taxe(){
      return (float)(cylindree*0.1+50);
   }
}
```

Solution classique (sans héritage)

```
public class Vehicule {

   private String marque;      private int annee;

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

   public String who(){
      retur
   }

   public S
      retur

   }

   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(){

   }

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

   public float taxe(){
      e*0.1+50);
   }
```

Duplication

- **Solution inefficace: Duplication d'attributs et réécriture de méthodes**
- **Problème très pénalisant pour les applications réelles:**
  → Implantation pour les motos, les bateaux, les avions, etc. ?

Réécriture de la même implantation

Solution orientée objet

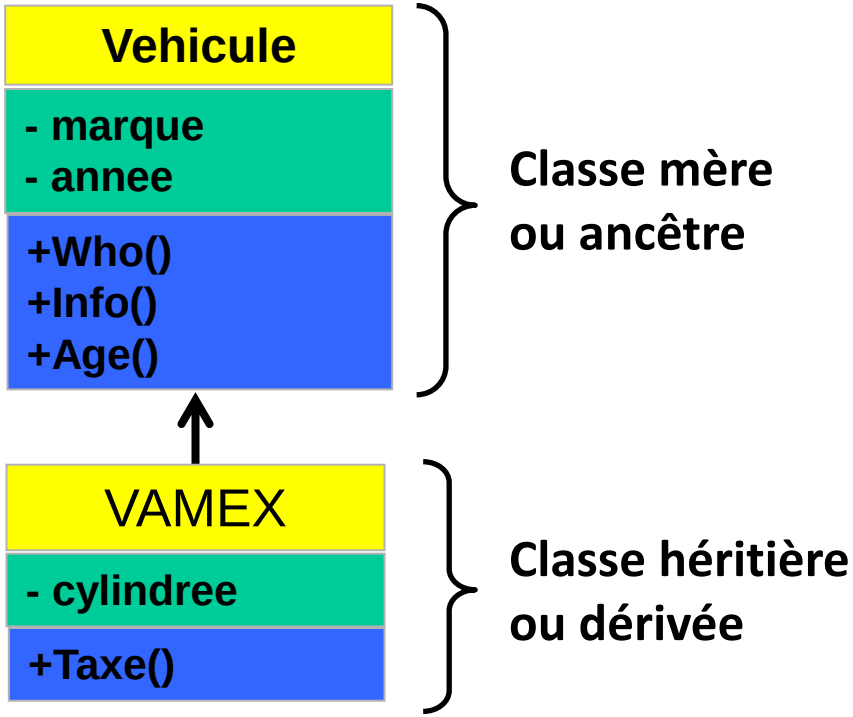
- **Réutilisation de ce qui est déjà réalisé :**
  - **Déclaration/implantation uniquement de ce qui est nouveau ou différent**
  - **Une méthode réalisant une fonctionnalité donnée doit garder le même nom:**
    - ***Si une méthode a la même fonctionnalité dans la classe mère et la classe dérivée, elle n'a pas besoin d'être réécrite.***
    - ***Si elle a une fonctionnalité différente, elle doit être re-déclarée et réécrite.***

      → On dit qu'elle redéfinit celle de son ancêtre.

- **Pour l'exemple des VAMEX à partir de Vehicule:**
  - **Inchangé: Marque, Année, Age()**
  - **Nouveau: Cylindrée, Taxe()**
  - **Modifié: Who() et Info() → à adapter selon le type**

Déclaration de la classe dérivée

- **Schéma UML :**



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

Déclaration de la classe dérivée

```java
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 info(){
     return (who() + " - marque " + marque
               + " construit en " + annee);
  }

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

```java
public class VAMEX extends Vehicule {

  protected 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 info(){
     return (who() + " - marque " + marque + " construit en "
          + annee + " de cylindrée " + cylindree + " cm3");
  }

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

Indique l'héritage

Attribut supplémentaire

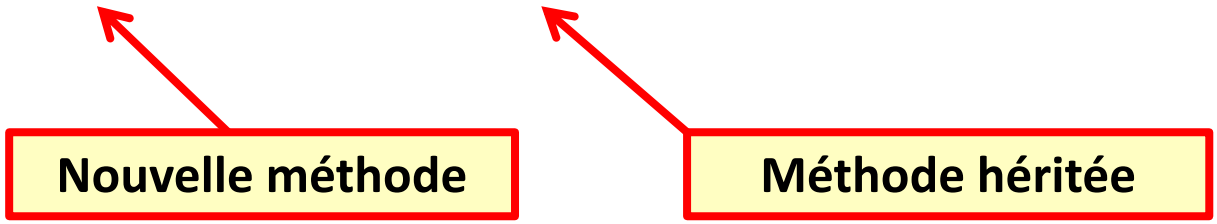
Redéfinition de méthodes

Accès aux attributs "protected" de la classe ancêtre

Nouvelle méthode

Utilisation de la classe dérivée

```
public class principal {

    public static void main(String[] args) {

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

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

    }
}
```



Compatibilités de types

- **Règle de compatibilité entre objets**
  - **Un type objet (une classe) est compatible avec lui-même, et tous ses ascendants directs ou non.**

- **Exemple 1**

```
VAMEX  unVAMEX1, unVAMEX2;
Vehicule  unVehicule;

unVAMEX1= new VAMEX( 2005, "Ford",  1200 );

unVAMEX2 = unVAMEX1;    // OK même type

unVehicule = unVAMEX1;    // OK type ascendant

unVAMEX2 = unVehicule;    // FAUX type descendant
```

Compatibilités de types

- **Exemple 2**

```
VAMEX unVAMEX1, unVAMEX2;
Vehicule  unVehicule1, unVehicule2;

unVehicule1 = new Vehicule( 2009, "Nakamura" );     // OK même type

unVAMEX1 = new VAMEX(  2005, "Ford", 1500 );        // OK même type

unVehicule2 = new VAMEX(  2010, "BMW", 1750 );      // OK type ascendant

unVAMEX2 = new Vehicule( 2007, "Scott");            // FAUX type descendant
```

Compatibilités de types

- **Intérêt de la compatibilité:**

  Si l'on ne connaît pas à l'avance le type d'un objet : on lui donne le type le plus général (ici Véhicule) et on choisit seulement à l'exécution le type utilisé parmi les type dérivés.

- **Exemple:**

```
public static void main(String[] args) {

        int choix = Integer.parseInt(args[0]);
        Vehicule A;

        switch(choix){
           case 1:  A = new Vehicule( 2009, "Nakamura" ); break;
           case 2 : A = new VAMEX(  2005, "Ford" , 1500 ); break;
        }
}
```

Polymorphisme

- **La méthode appliquée à un objet est déterminée à l'exécution par le type de l'objet pointé**
- **Exemple:**

```
Vehicule unVehicule1, unVehicule2;
VAMEX unVAMEX;

unVehicule1 = new Vehicule( 2009, "Nakamura" );
System.out.println( unVehicule1.info() );
```

→ **Je suis un véhicule - marque Nakamura construit en 2009**

```
unVAMEX = new VAMEX(  2005, "Ford", 1500 );
System.out.println( unVAMEX.info() );
```

→ **Je suis un VAMEX - marque Ford construit en 2005 de cylindrée 1500 cm3**

```
unVehicule2 = new VAMEX(  2010, "BMW", 1750 );
System.out.println( unVehicule2.info() );
```

→ **Je suis un VAMEX - marque BMW construit en 2010 de cylindrée 1750 cm3**

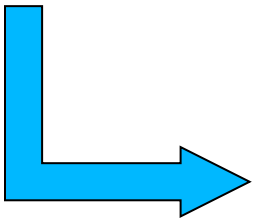
Polymorphisme

- **Intérêt du polymorphisme :**
  - **Une méthode réalisant une fonctionnalité donnée a le même nom et s'adapte en fonction du type de l'objet à traiter → Unification des traitements**

- **Exemple:**

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

myTab[0] = new VAMEX(  2005, "Ford",  1500 );
myTab[1] = new Vehicule(  2005, "Nakamura" );
myTab[2] = new VAMEX(  2012, "BMW",  1750 );
myTab[3] = new Vehicule(  2010, "Scott" );
myTab[4] = new Vehicule(  1997, "Marin" );

for(int i=0; i<5; i++) System.out.println( myTab[i].info() );
```

Adaptation au type de l'objet: la programmation sans héritage impliquerait un test et 2 procédures



```
Je suis un VAMEX - marque Ford construit en 2005 de cylindrée 1500 cm3
Je suis un véhicule - marque Nakamura construit en 2005
Je suis un VAMEX - marque BMW construit en 2012 de cylindrée 1750 cm3
Je suis un véhicule - marque Scott construit en 2010
Je suis un véhicule - marque Marin construit en 1997
Process exited with exit code 0.
```

Appel à la méthode ancêtre : super()

- **Intérêt**
  - **Pouvoir appeler une méthode de la classe ancêtre**
  - **Dans le corps de la méthode de la classe dérivée**
    - **Traiter les nouveaux attributs (spécialisation)**
    - **Appeler la méthode de la classe ancêtre pour traiter les attributs hérités**

Appel à la méthode ancêtre : super()

```java
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 info(){
     return (who() + " - marque " + marque
                + " construit en " + annee);
  }

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

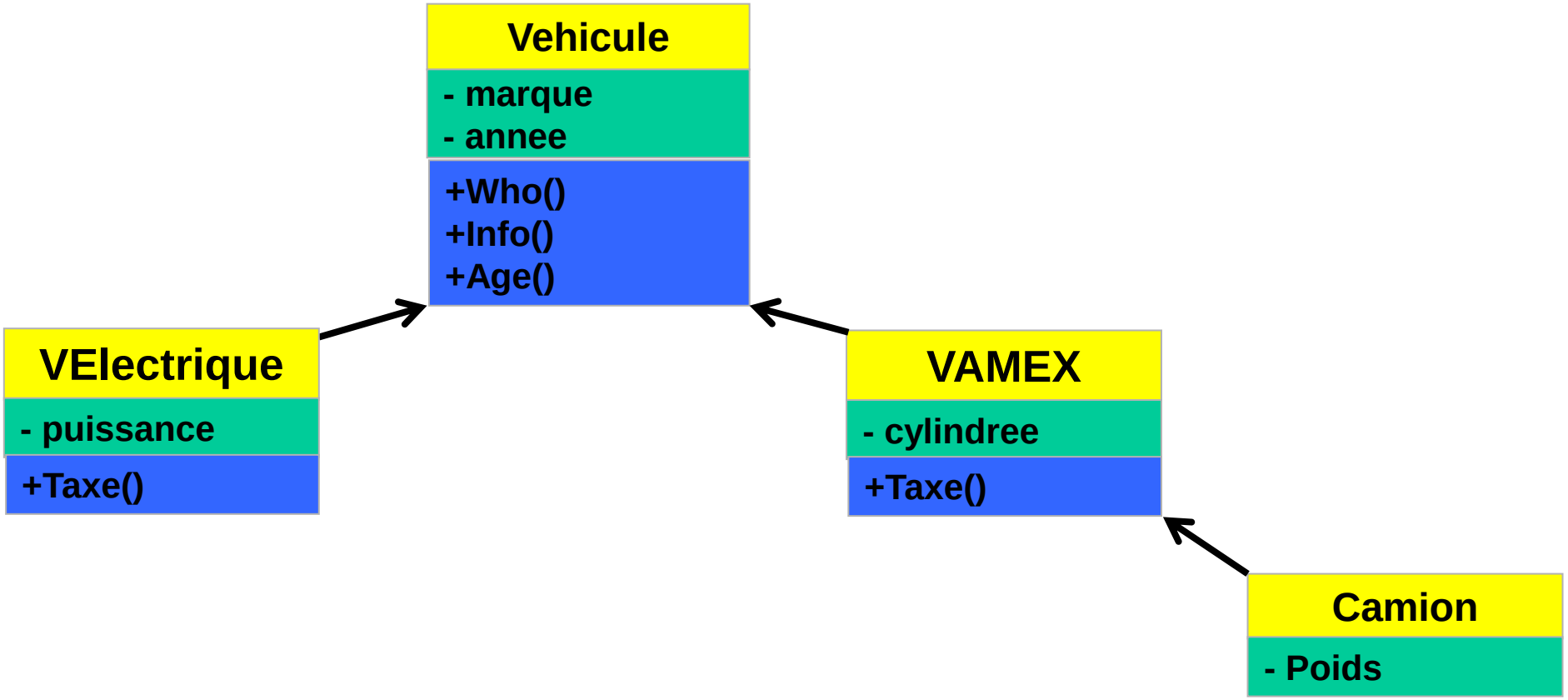
```java
public class VAMEX extends Vehicule{

  protected int cylindree;

  public VAMEX( int uneAnnee, String uneMarque, int uneCylindree){
     super( uneAnnee, uneMarque );
     cylindree= uneCylindree;
  }

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

  public String info(){
     return (super.info()+ " de cylindrée " + cylindree + " cm3");
  }

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

Appel de la méthode ancêtre

Classes et méthodes abstraites

- **On enrichit la hiérarchie:**
  - **En plus des Véhicule et VAMEX, on veut traiter les véhicules électriques et les camions**
- **Informations supplémentaires:**
  - **Véhicules Electriques : Puissance maximum délivrée (kW)**
  - **Camions : Poids à vide (tonnes)**
- **Traitements:**
  - **On impose un calcul de taxe à tous les véhicules pour lesquels cela est possible**

Classes et méthodes abstraites



Classes et méthodes abstraites

```
public class Velectrique extends Vehicule{

   protected int puissance;

   public Velectrique(  int uneAnnee, String uneMarque,
                        int unePuissance ){
      super( uneAnnee, uneMarque );
      puissance = unePuissance;
   }

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

   public String info(){
      return ( super.info() + " de puissance " + puissance + " kW");
   }

   public float taxe(){
      return (float)( puissance*12.23 );
   }
}
```

```
public class Camion extends VAMEX{

   protected double poids;

   public Camion(  int uneAnnee, String uneMarque,
                   int uneCylindree, double unPoids ){
      super( uneAnnee, uneMarque, uneCylindree );
      poids= unPoids;
   }

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

   public String info(){
      return ( super.info() + " pesant " + poids + " tonnes");
   }

   public float taxe(){
      return (float)( super.taxe() + poids*115 );
   }
}
```

Classes et méthodes abstraites

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

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

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

Compilation → Error: cannot find method taxe()

- **Problème avec l'héritage de la méthode taxe()**
  - **Le compilateur demande une implémentation de la méthode "taxe()" pour Vehicule et toutes les classes dérivées**
  - **Or, la méthode "taxe()" n'est ni déclarée ni implémentée dans la classe Vehicule .**

Classes et méthodes abstraites

```
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 String info(){
      return (who() + " - marque " + marque
       + " contruit en " + annee);
   }

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

   public abstract float taxe();
}
```

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

```
Vehicule  A = Vehicule( 2005, "Nakamura" )
```

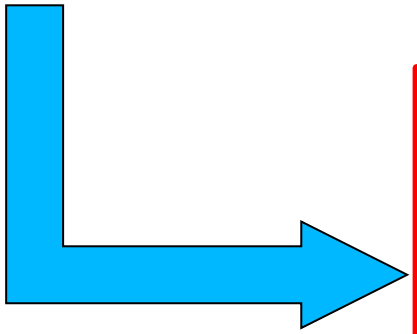
Error: Vehicule is abstract; cannot be instantiated

Méthode abstraite:

- **<u>Aucune</u> implémentation (pas de corps)**
- **Impose l'implémentation <u>dans les classes dérivées</u>**

Classes et méthodes abstraites

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

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

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



```
200.0
757.5
807.18
856.1
1037.0
170.0
Process exited with exit code 0.
```

POO : bilan

- **En POO un problème est modélisé par un ensemble de classes et objets**
- **Lorsque différents types d'objets ont assez de caractéristiques et de traitements communs, on crée une classe de base (classe mère) et des classes dérivées**
- **Lorsqu'une classe dérive d'une classe ancêtre:**
  - **La classe dérivée hérite de l'ensemble des attributs et méthodes de la classe ancêtre**
  - **Les nouveaux attributs/méthodes de la classe dérivée sont spécifiques à celle-ci.**
  - **La classe dérivée peut créer et modifier une méthode ayant le même nom que l'une des méthodes dont elle hérite**

POO : bilan

- **Particularités pour les méthodes:**
  - **Une méthode d'une classe dérivée peut faire appel à la méthode de la classe ancêtre en utilisant le mot-clé "super"**
  - **Une classe ancêtre peut être déclarée "abstract" et comprendre des méthodes "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**