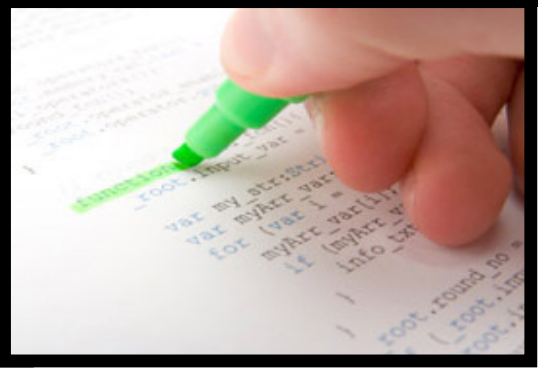
UNIVERSITE DE CORSE

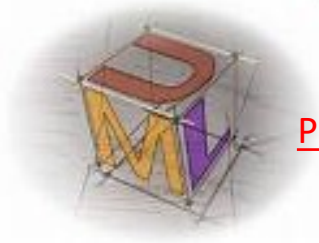
Licence SPI 2ème année
parcours informatique

UE Programmation Orientée Objet

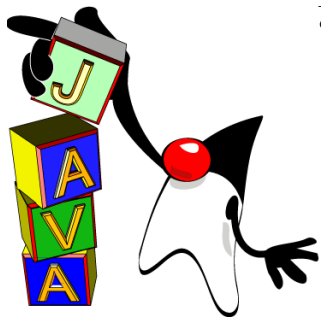
CH 4 – Héritage et Polymorphisme







Evelyne VITTORI
vittori_e@univ-corse.fr
Paul PINA-GHERARDI
PINA-GHERARDI_p@univ-corse.fr



Objectifs de ce chapitre

- Découvrir la notion de hiérarchie de classes
- Savoir mettre en œuvre le mécanisme d'héritage d'attributs et de méthodes
- Comprendre le polymorphisme et savoir l'utiliser

CH 4 – HERITAGE et POLYMORPHISME

4.1 – Hiérarchie de Généralisation/Spécialisation

- Relation EST-UN en UML
- Sous-classes en Java
- Démarche de construction

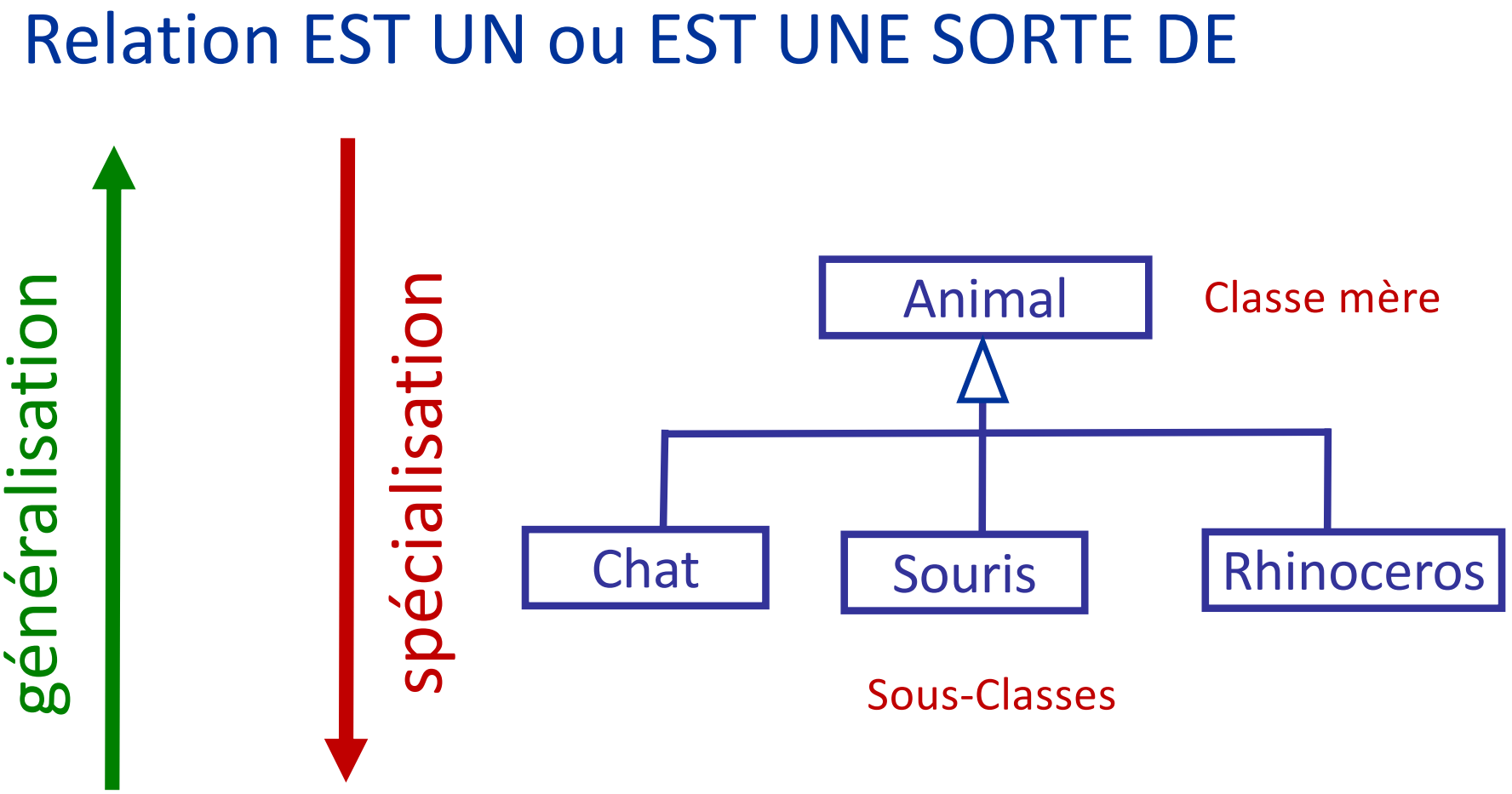
4.2 – Mécanisme d'Héritage

4.3 – Classes abstraites et polymorphisme

Hiérarchie de généralisation-spécialisation

Hiérarchie de Généralisation -Spécialisation

Relation EST UN ou EST UNE SORTE DE



Pourquoi utiliser l'héritage ?

- L'héritage permet de modéliser la relation « est un »
  - un Rectangle **est un** parallélépipède
  - un Cercle **est une** Ellipse
  - une Ellipse **est une** FormeGéométrique
    - Ils ont tous une position, une couleur,…
    - Ils peuvent tous être dessinés, déplacés,…
    - On peut calculer leur surface, leur périmètre
    - Ils ne sont pourtant pas semblables, $\pi * r^2 \neq L * l$

Attention: ne pas confondre avec « est une instance de »

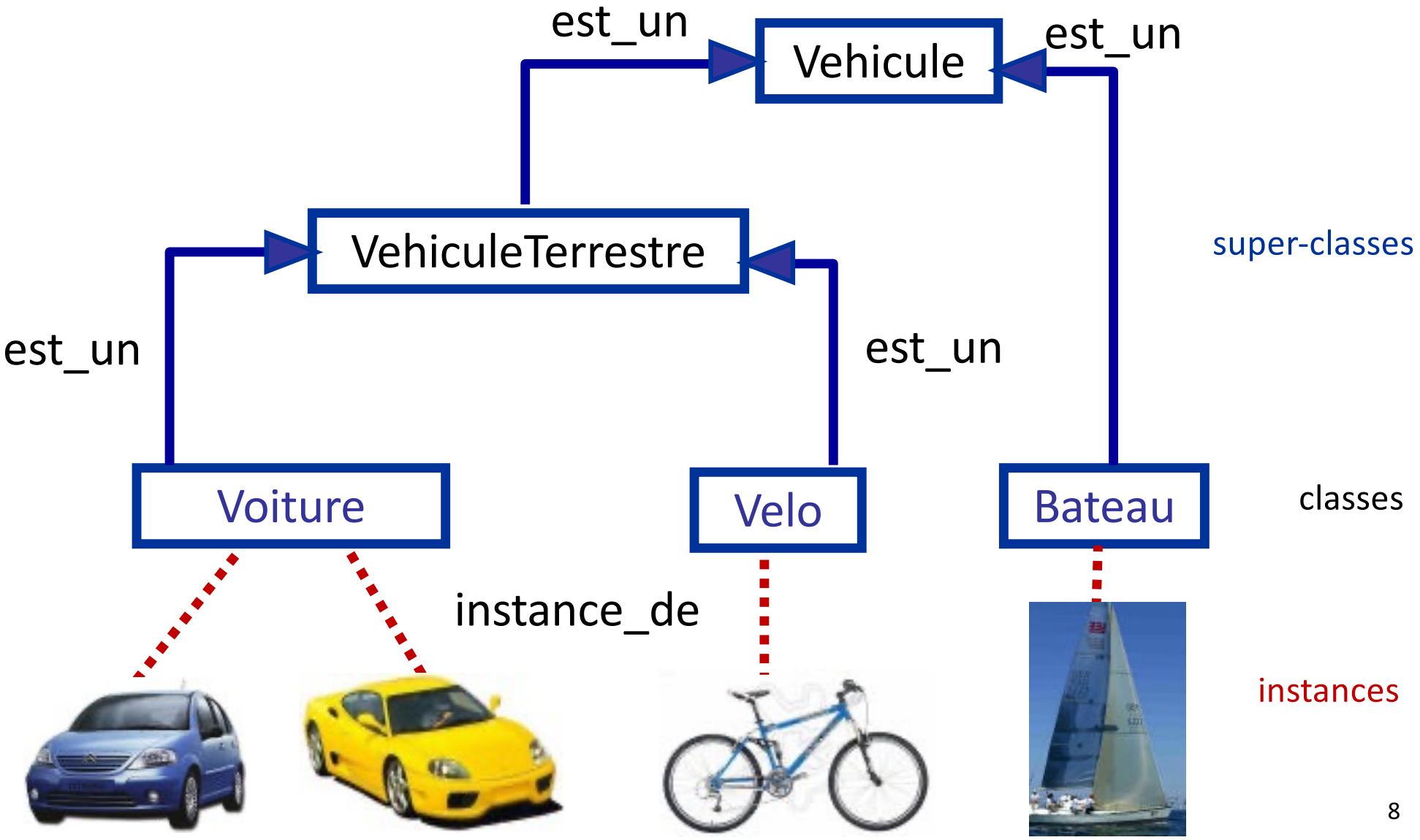
Pourquoi utiliser l'héritage ?

- Pour éviter la duplication de code: le code commun à plusieurs classes est placé dans la classe mère

  Attention: ne pas utiliser d'héritage sans signification juste pour regrouper du code

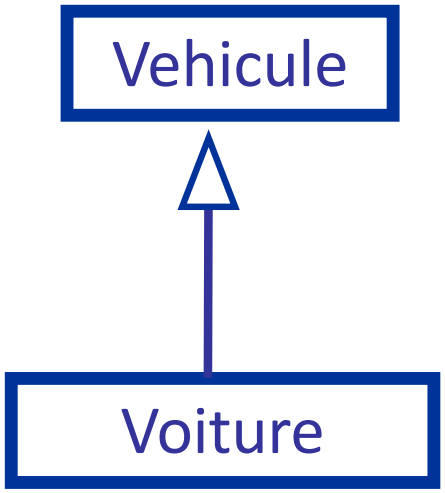
- Pour rendre le code plus évolutif
  - Ajout d'une classe fille sans modification de la classe mère
  - Polymorphisme

Hiérarchie de Généralisation -Spécialisation



Sous-Classes Java

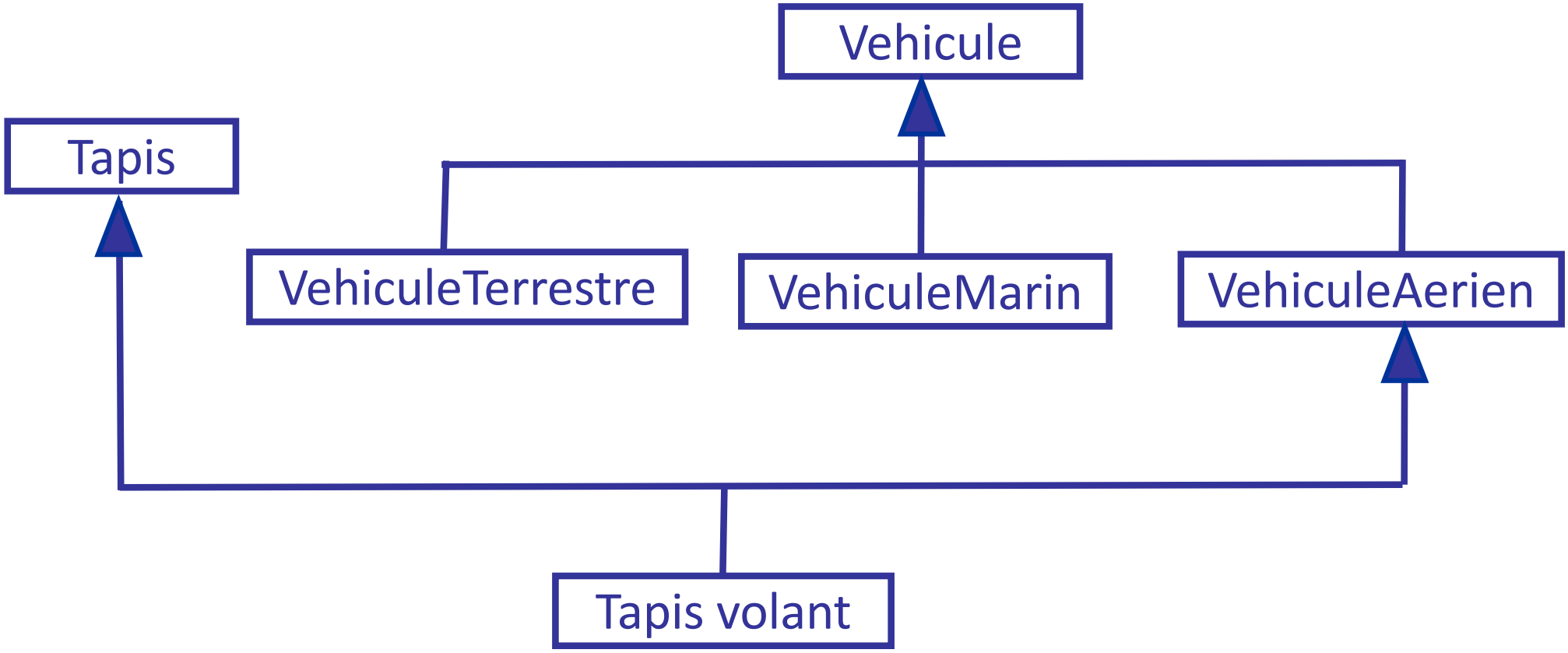
Déclaration d'une sous-classe ou classe dérivée en java




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

- En Java, une classe ne peut avoir qu'un seul parent.
- Toutes les classes ont un « ancêtre » commun: OBJECT
  java.lang.Object

Notion d'Héritage multiple

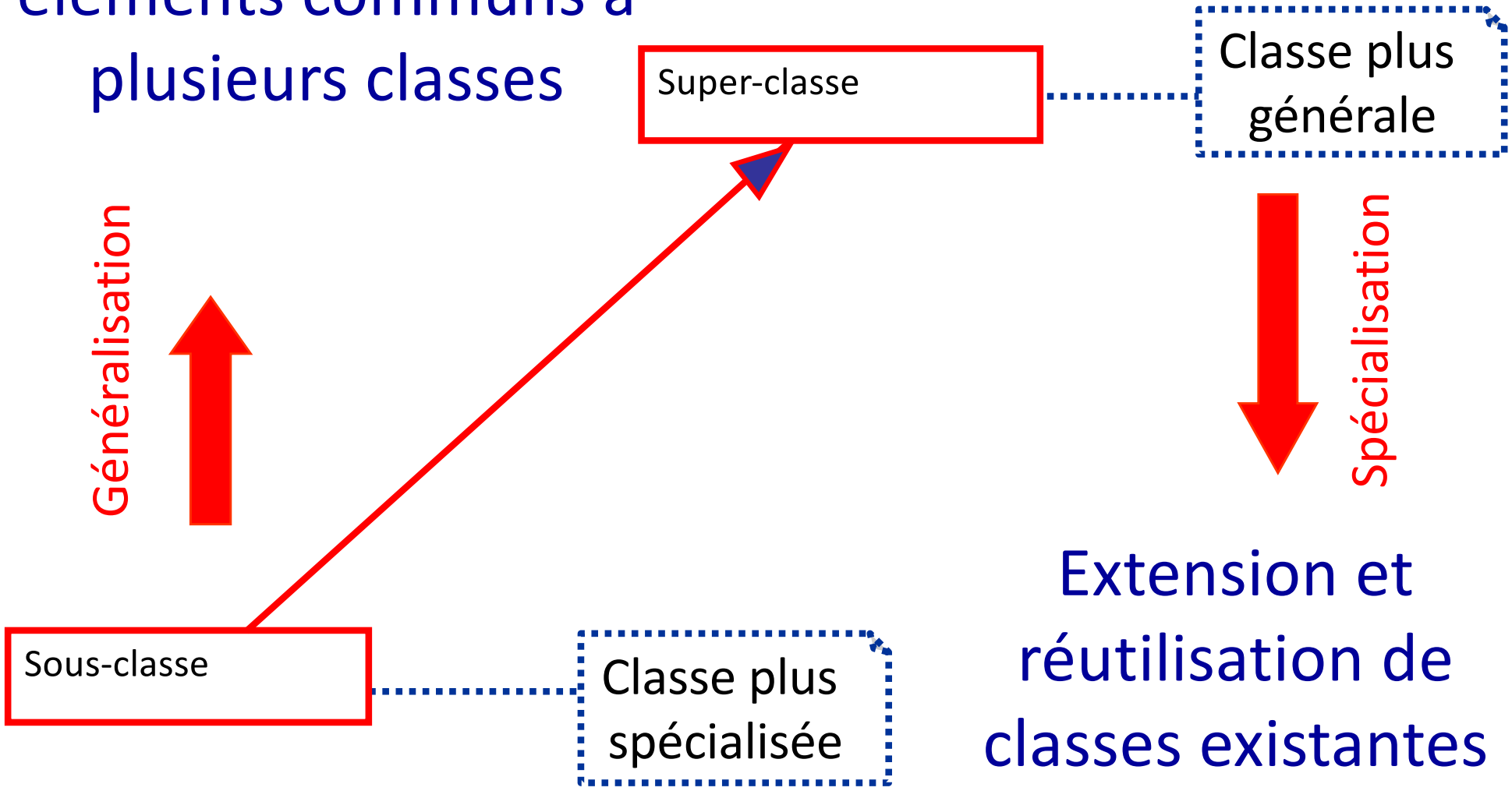


UML autorise la représentation de l'héritage multiple.

Java interdit l'héritage multiple de classes.

Démarche de construction

Factorisation des éléments communs à plusieurs classes



Extension et réutilisation de classes existantes

Mécanisme d'Héritage

- Principe de l'héritage simple
- Héritage de propriétés
- Héritage et encapsulation

Principe d'héritage

- L’héritage est une technique pour construire une classe à partir d'une ou plusieurs autres classes

- les classes enfants héritent des caractéristiques de leurs parents

De quoi hérite-t-on?

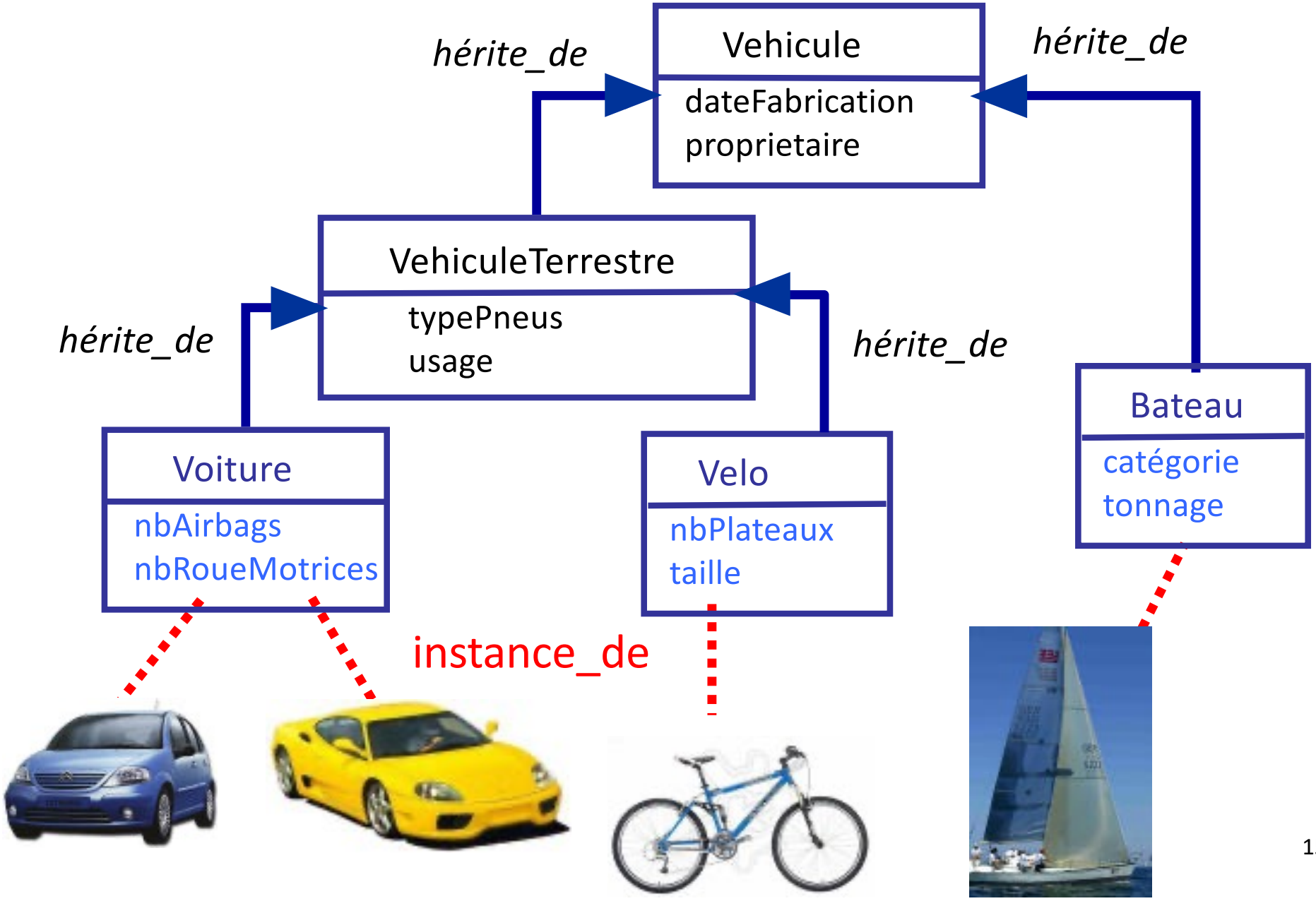
- Une sous classe hérite :

  des attributs (et associations) et des méthodes.

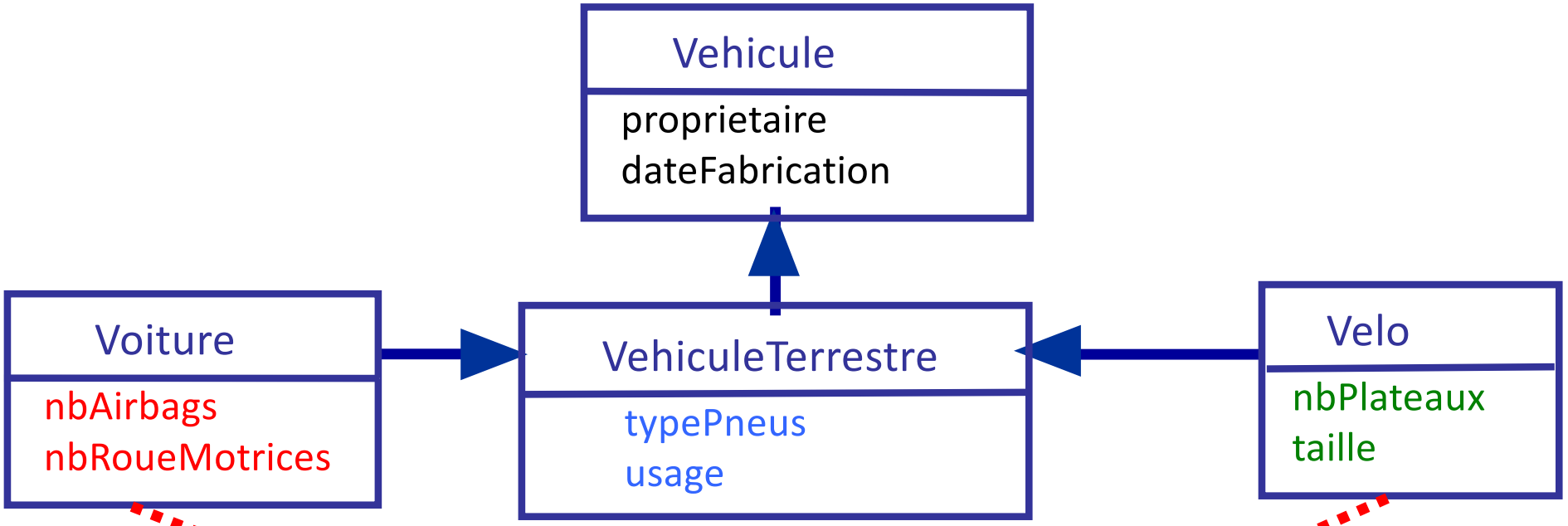
- Une sous classe peut:
  - **Ajouter** des attributs, méthodes et relations spécifiques.
  - **Redéfinir** les méthodes héritées.

Les attributs et méthodes communs sont visibles au niveau le plus haut de la hiérarchie.

Héritage de propriétés



Héritage de propriétés



titine: Voiture

datefabrication=22/04/2010
proprietaire= « toto »
typePneus= « normaux »
usage=« ville »
nbAirbags= 2
nbRoueMotrices=2

hector : Velo

datefabrication=10/02/2019
proprietaire= « titi »
typePneus= « tous temps »
usage=« tous terrains »
nbPlateaux= 3
taille=24

Héritage de propriétés en Java

```
class Vehicule {
        Date dateFabrication;
        String proprietaire;
}
```

```
class VehiculeTerrestre extends Vehicule {
        String typePneus;
        String usage;
```

```
class Velo extends VehiculeTerrestre {
        int nbPlateaux;
        int taille;
}
```

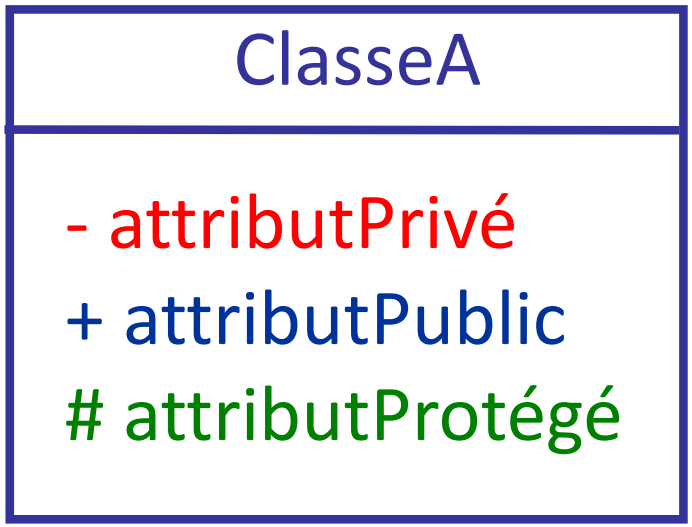
```
class Voiture extends VehiculeTerrestre {
        int nbAirbags;
        int nbRoueMotrices;
}
```

Héritage de propriétés en Java

```
public class testVehicule {
    public static void main(String[] args) {
        Voiture titine=new Voiture();
        Velo hector=new Velo();
        titine.proprietaire="toto";
        titine.usage="ville";
        titine.nbAirbags=2;
        titine.nbPlateaux=2;        // INTERDIT pour titine
        hector.proprietaire="titi";
        hector.usage="tous terrains";
        hector.nbPlateaux=3;
        hector.nbAirbags=2;         // INTERDIT pour hector
        }
}
```

Héritage et Encapsulation

Niveaux de visibilité en UML



Modificateurs en Java

- **private** : accès réduit, seulement depuis la classe
- **public** : accès libre depuis partout
- **protected** : accès depuis la classe, les classes filles et les classes du package
- **package (ou rien)** : accès depuis la classe et les classes du package

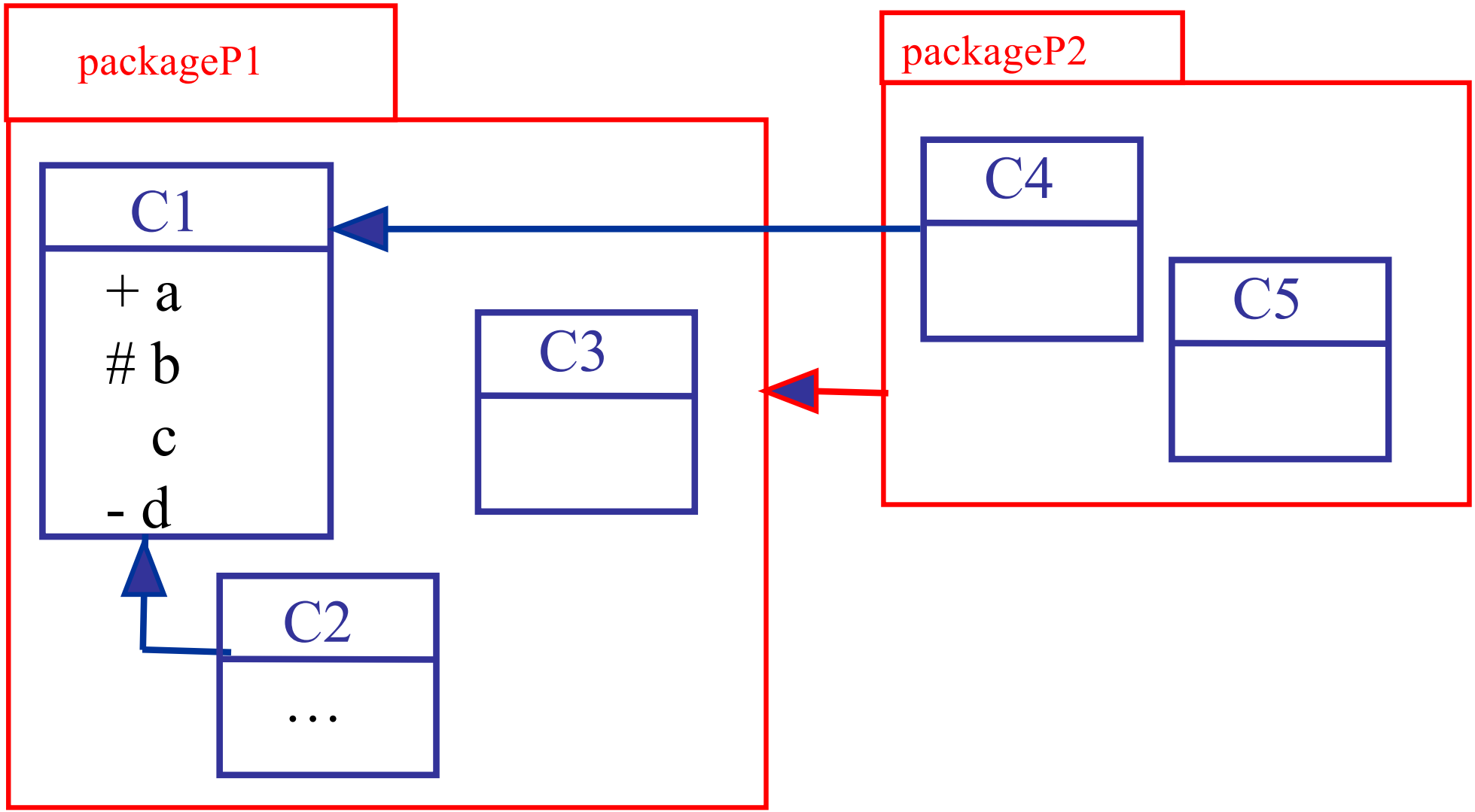
Héritage et Encapsulation

protected…

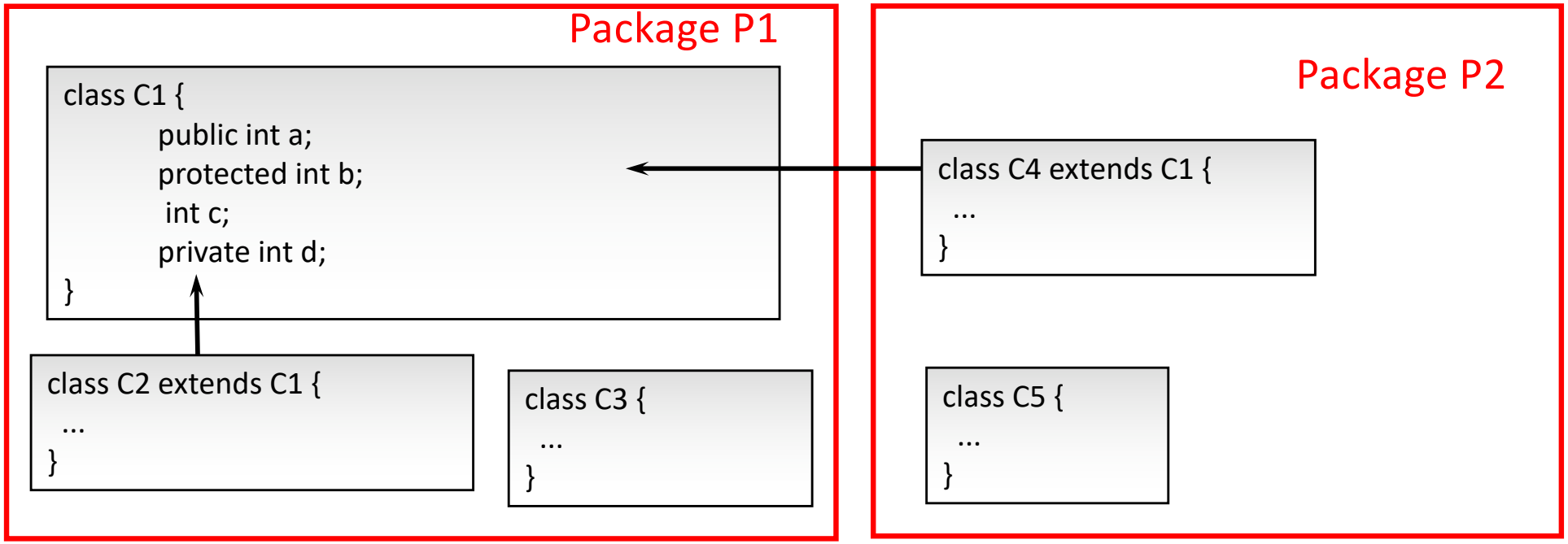
```
package vehicules;
class Vehicule {
        private String immat;
        protected String proprietaire;
        void immatriculer() { ….. // visibilité package  }
}
```

```
package vehicules;
class Atelier{
        void fabrication(){
                Vehicule a = new Vehicule();
                a.proprietaire = “titi”;
                a.immatriculer();
                a.immat=“1234 HY 2A”; //INTERDIT
        }
}
```

Héritage et Encapsulation



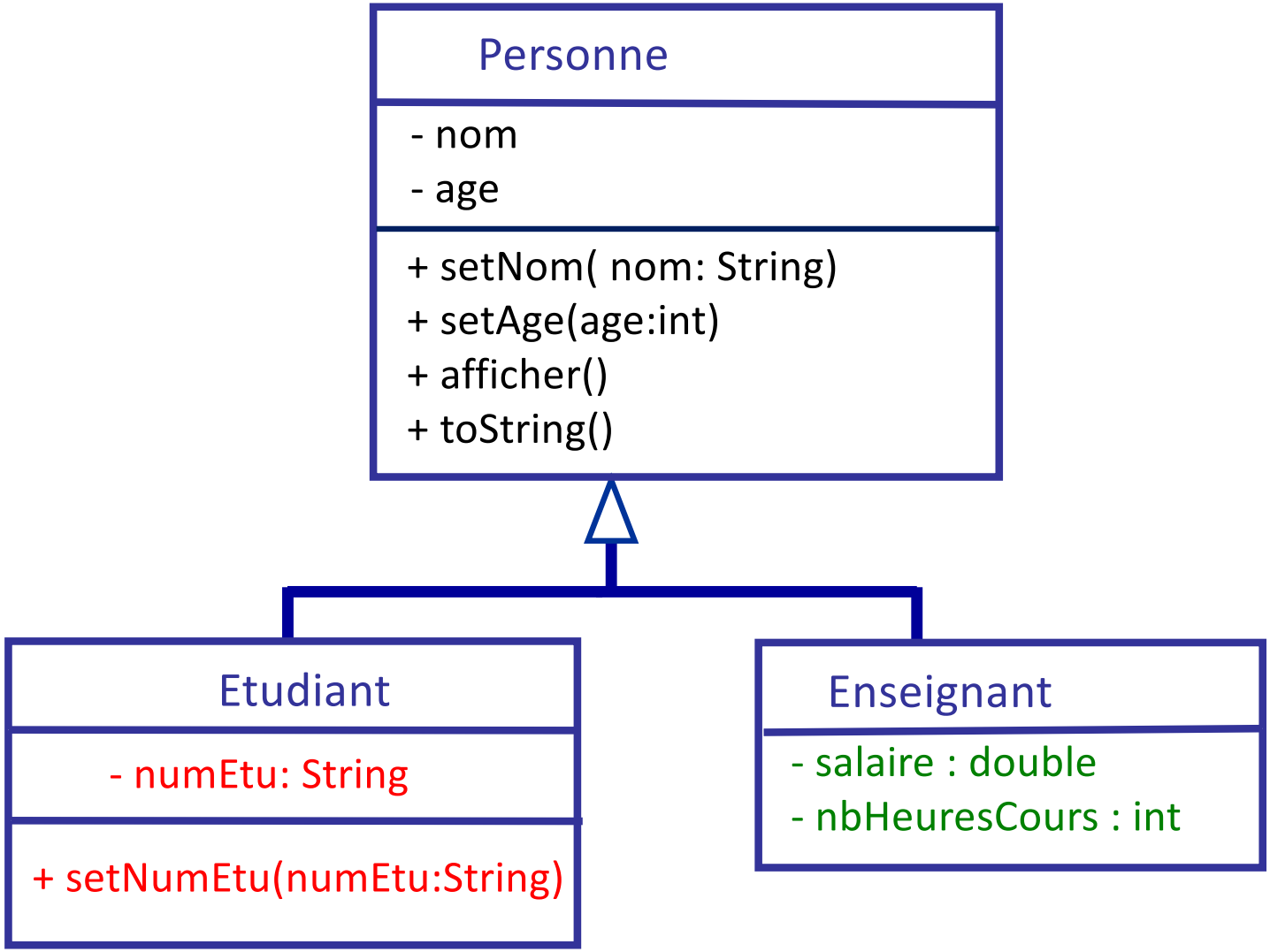
Héritage et Encapsulation



	a	b	c	d
Accessible par C2	oui	oui	oui	non
Accessible par C3	oui	oui	oui	non
Accessible par C4	oui	oui	non	non
Accessible par C5	oui	non	non	non

TPCours - Exercice 4.1

On considère la hiérarchie de classes, suivante :



TPCours - Exercice 4.1

Code de la classe Personne

```
public class Personne {
    private String nom;
    private int age;
    public Personne(String nom, int age) {
        this.nom=nom;
        this.age=age;}
    public Personne() {
        this("", 0);}
    public  void setNom(String nom) {
        this.nom=nom;}
    public  void setAge(int age) {
        this.age=age;}
    public void afficher() {
        System.out.println("Nom : "+this.nom + "\nAge : " +
        this.age );}
    public String toString() {
        return this.nom + " (" + this.age + " ans)";
    }
```

code à récupérer sur l'ENT

TPCours - Exercice 4.1

- Définissez le code d'une classe Etudiant (avec un attribut de type String numEtu) et d'une classe Enseignant (attributs salaire et nbheuresCours) sans définir de constructeur ni ajouter de méthode autre que setNumEtu dans Etudiant.
- Créez une classe TestPersonne avec

La classe Personne ne doit pas être modifiée

un main réalisant les actions suivantes:

- Créer une Personne pers de nom Marie et d'age 20 ans et l'afficher
- Créer un Etudiant etu (vide) et l'afficher
- Lui affecter le nom Pierre, l'age 21

et le numEtu « 20203456 »

- Afficher à nouveau etu

Vous devez obtenir l'affichage suivant :

```
Nom : Marie
Age : 20
Nom :
Age : 0
Nom : Pierre
Age : 21
```

Héritage et redéfinition de méthodes

Héritage de méthodes

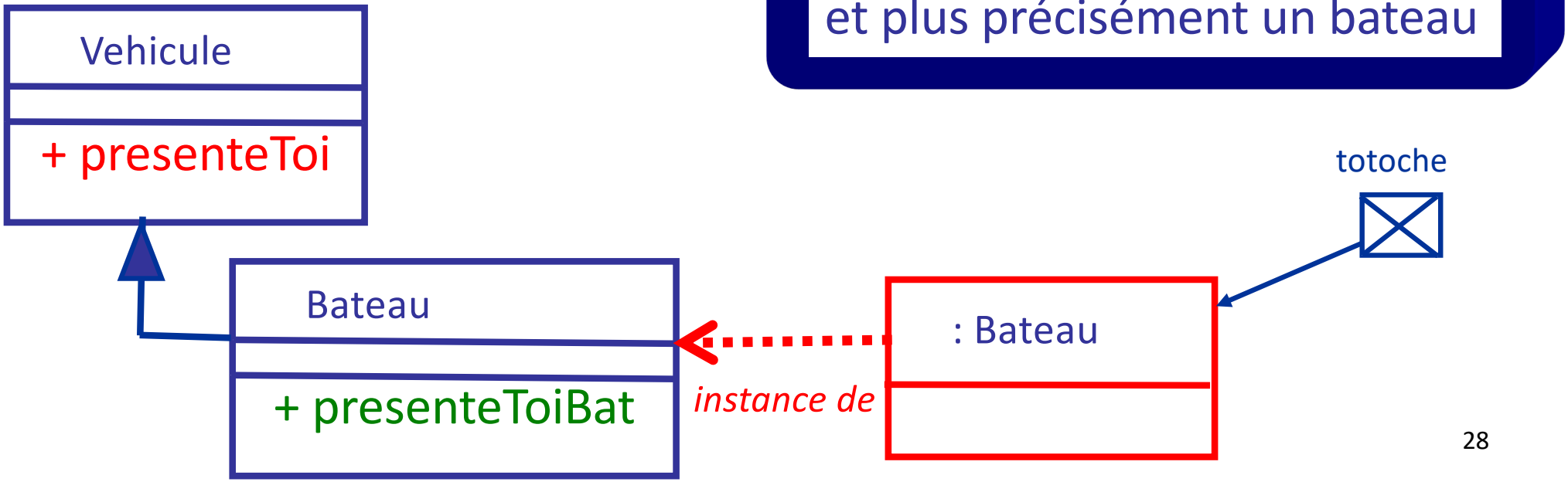
```
class Vehicule {
        ....
        public void presenteToi(){
                System.out.println("Je suis un vehicule");
        }
}
```

```
class Bateau extends Vehicule {
    ...
    public void presenteToiBat(){
        System.out.println("et plus précisément un bateau");
    }
}
```

Héritage de méthodes

```
public class testVehicule {
    public static void main(String[] args) {
            Bateau totoche=new Bateau();
            totoche.presenteToi();
            totoche.presenteToiBat();
            }
}
```

Je suis un vehicule
et plus précisément un bateau



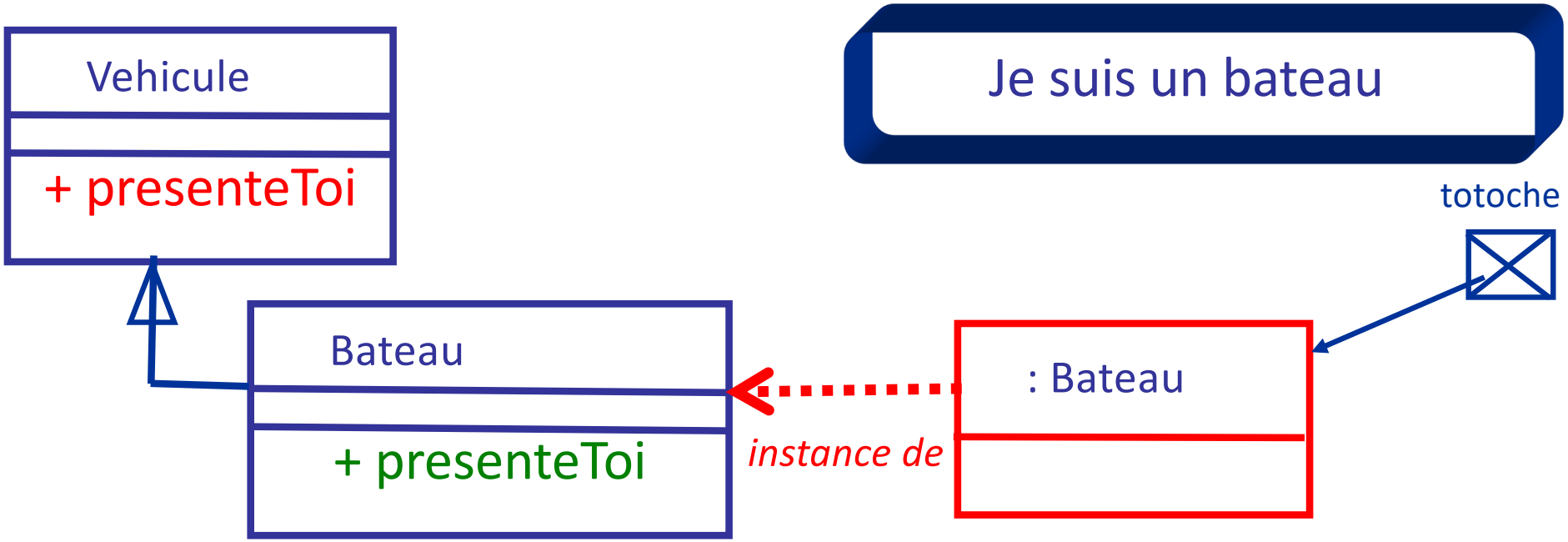
Rédéfinition de méthodes

```
class Vehicule {
        ....
        public void presenteToi(){
                System.out.println("Je suis un vehicule");
        }
}
```

```
class Bateau extends Vehicule {
    ...
    public void presenteToi (){
      System.out.println("Je suis un bateau");
    }
}
```

Rédéfinition de méthodes

```
public class testVehicule {
    public static void main(String[] args) {
            Bateau totoche=new Bateau();
            totoche.presenteToi();
            }
}
```



Rédéfinition de méthodes

Une méthode peut redéfinir une méthode d'une classe parent pour éventuellement la compléter.

```
public class Test extends ParentTest{

        public void uneMethode(){
            //autres actions
            super.uneMethode();
            //autres actions
    . . .
        }...}
```

Invocation de la méthode de ParentTest

Instructions Complémentaires

Rédéfinition de méthodes

```
class Vehicule {
        ....
        public void presenteToi(){
                System.out.println("Je suis un vehicule");
        }
}
```

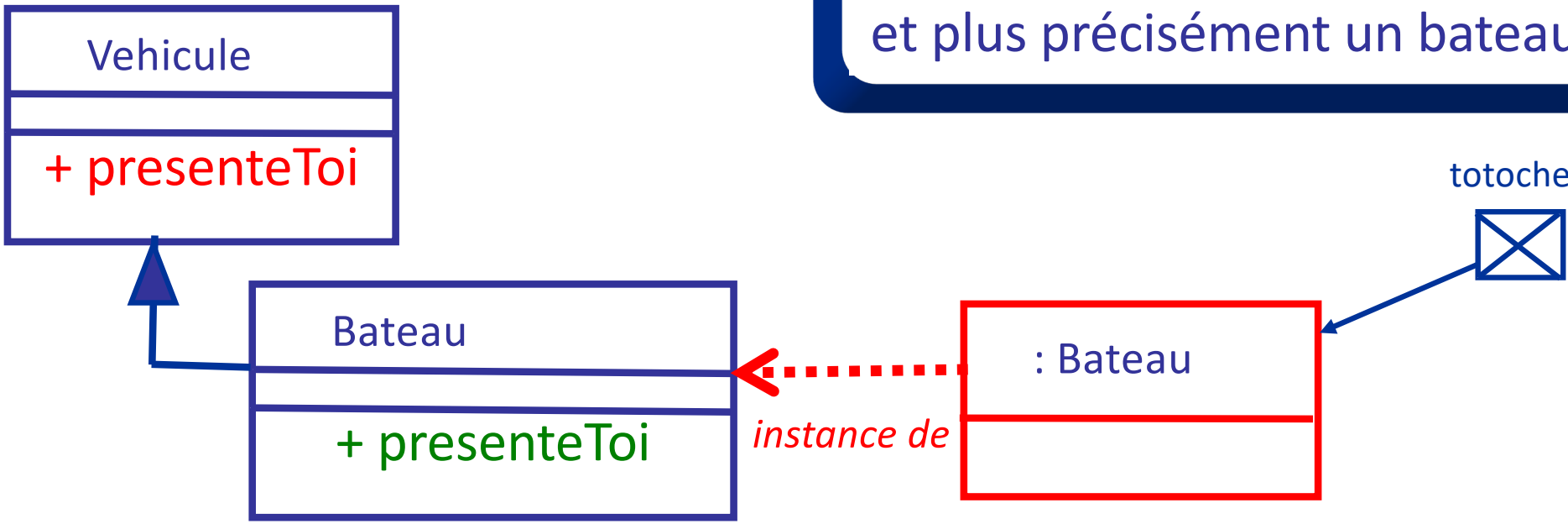
```
class Bateau extends Vehicule {
    ...
    public void presenteToi (){
      super.presenteToi();
      System.out.println("et plus précisément un bateau");
        }
}
```

Référence à l'objet parent

Rédéfinition de méthodes

```
public class testVehicule {
    public static void main(String[] args) {
            Bateau totoche=new Bateau();
            totoche.presenteToi();
            }
}
```

Je suis un vehicule
et plus précisément un bateau



Rédéfinition de méthodes

```
public class Vehicule{
        String immatriculation;
        String proprietaire;
        public String caracteristiques(){
          return immatriculation+proprietaire;
        }…
}
```

```
public class Voiture extends Vehicule{
            int nbairbags;
            int nbRoueMotrices;
            public String caracteristiques(){
              return super.caracteristiques()+
                        nbairbags + nbRoueMotrices;
            }…}
```

Rédéfinition de méthodes

Redéfinition

- Des méthodes différentes ont exactement la même signature
- Le choix de la méthode appelée dépend du type réel ou constaté (type dynamique) de l'objet (déterminé à l'exécution)

≠ Surcharge (ou Surdéfinition)

- Deux méthodes ont le même nom et le même type de retour mais des signatures différentes

Exemple : les constructeurs

- Le choix de la méthode appelée dépend des paramètres d'appel (déterminé à la compilation)

Héritage de méthodes

Modificateur final

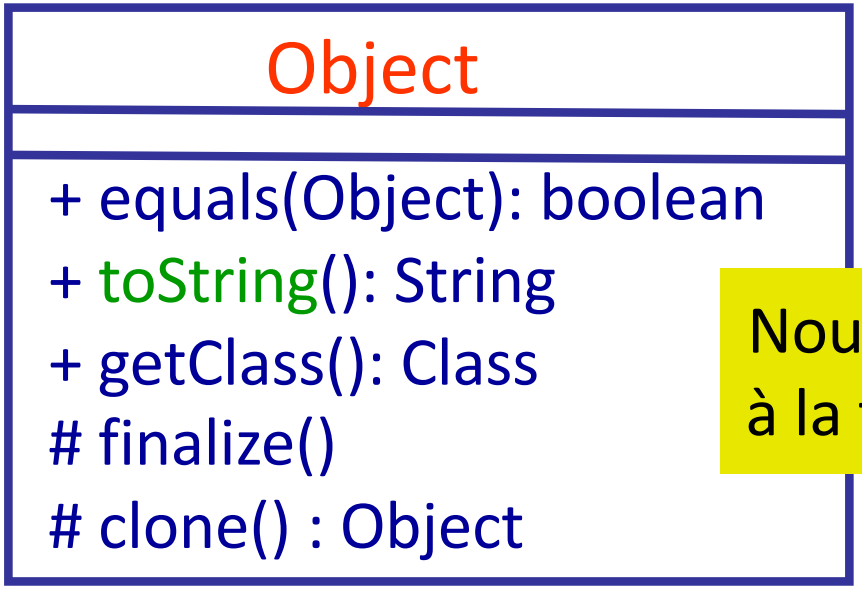
- Une **classe** déclarée **final** ne peut plus être dérivée

- Une **méthode** déclarée **final** ne peut plus être redéfinie dans une sous-classe

Méthodes de la classe Object

Object : « le père de nos pères »

- Racine de la hiérarchie d'héritage en Java
- Permet de définir des collections génériques d'objets
- Contient plusieurs méthodes héritées et pouvant être redéfinies



Nous y reviendrons à la fin du chapitre!

Héritage et constructeurs

Héritage et constructeurs

- Les constructeurs ne sont pas hérités.
- Un constructeur de la classe super-classe doit obligatoirement être appelé dans le constructeur de la classe fille

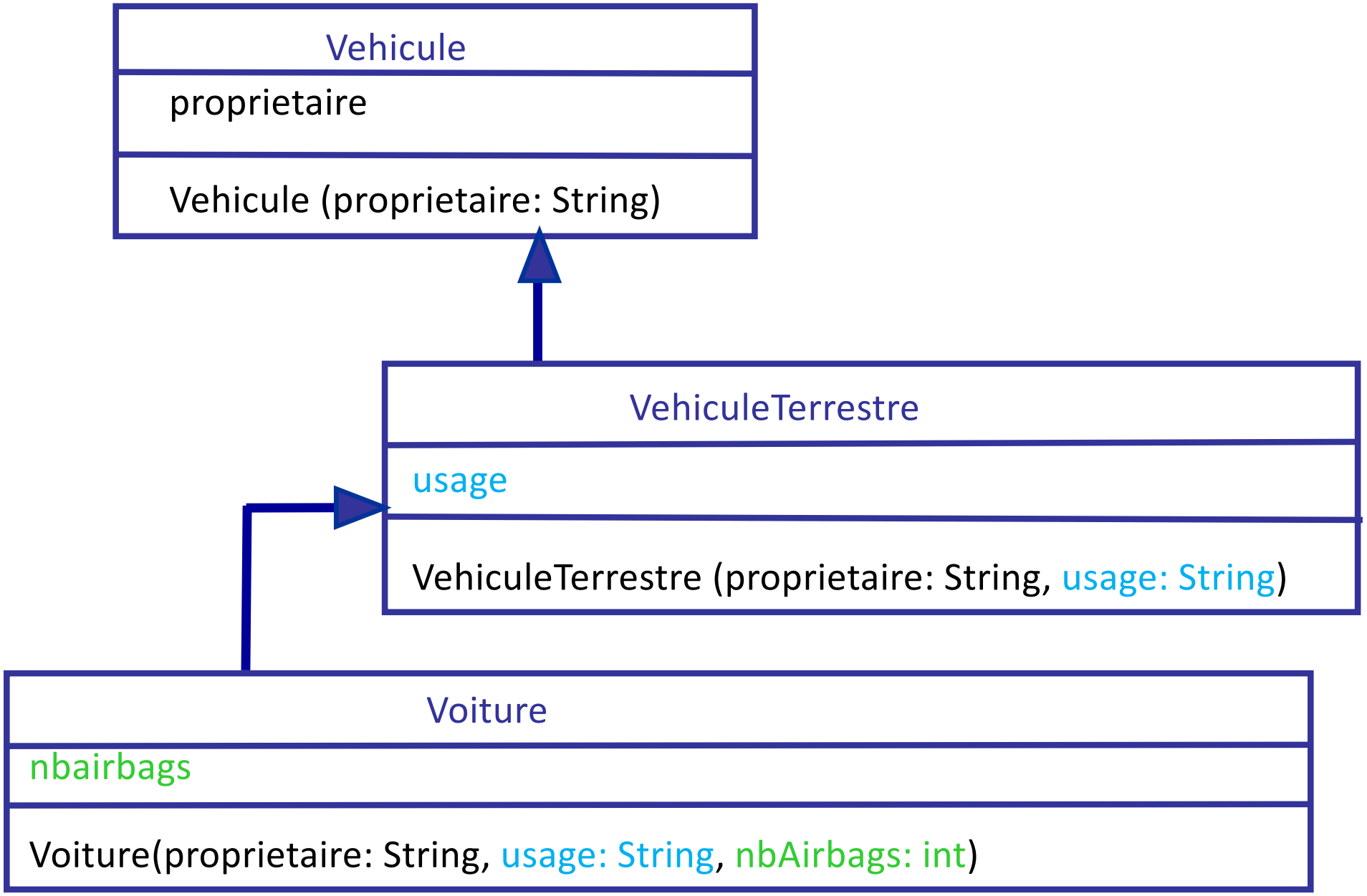
```
public class Test extends ParentTest{

    public Test(…){
        super(…);
            // Première instruction obligatoire
        ..
    }
…
```

Sauf en cas d'invocation implicite :

//1er cas : invocation implicite du constructeur par défaut

//2éme cas : il y a un constructeur sans paramètre dans la classe mère

Héritage et constructeurs



Héritage et constructeurs

```
class Vehicule {
        String proprietaire;
        public Vehicule(String proprietaire){
                this.proprietaire=proprietaire;
        }
}
```

```
class VehiculeTerrestre extends Vehicule {
        String usage;
        public VehiculeTerrestre(String proprietaire, String usage){
                super(proprietaire);
                this.usage=usage;
        }
}
```

```
class Voiture extends VehiculeTerrestre {
        int nbAirbags;
        public Voiture(String proprietaire, String usage, int nbAirbags){
                super(proprietaire,usage);
                this.nbAirbags=nbAirbags;
 }
```

Héritage et constructeurs

Pseudo-variable super

- Référence aux membres de la super-classe:

  Invocation d'une méthode de la super-classe en cas de redéfinition

  super.methode()

- Invocation du constructeur de la super-classe

  super() ou super(…)

En résumé

Que contient le code d'une classe fille?

- Code (variables et/ou méthodes) spécifique au comportement de la classe fille
- Redéfinition de certaines méthodes dont le comportement n'est plus approprié
- Spécificité des constructeurs d'une classe fille

TPCours - Exercice 4.2

- Complétez vos classes Etudiant et Enseignant de l'exercice 4.1 afin que le programme de test ci-dessous provoque l'affichage présenté dans la diapositive suivante

```
public class TestPersonne {
    public static void main(String[] args) {
    Personne pers=new Personne("Marie",20);
    Etudiant etu=new Etudiant("Jean", 21,"20203433");
    Enseignant ens= new Enseignant("Pierre",54,3000,250);
    System.out.println(pers);
    System.out.println(etu);
    System.out.println(ens);
    pers.afficher();
    etu.afficher();
    ens.afficher();
}
```

code à récupérer sur l'ENT

TPCours - Exercice 4.2

- Affichage que vous devez obtenir :

```
Marie (20 ans)
Etudiant numero 20203433 Jean (21 ans)
Enseignant Pierre (54 ans) 3000.0 euros
Nom : Marie
Age : 20
*******Etudiant*******
Numero étudiant : 20203433
Nom : Jean
Age : 21
*******Enseignant******
Nom : Pierre
Age : 54
Salaire : 3000.0
Nombre d'heures de cours : 250
```

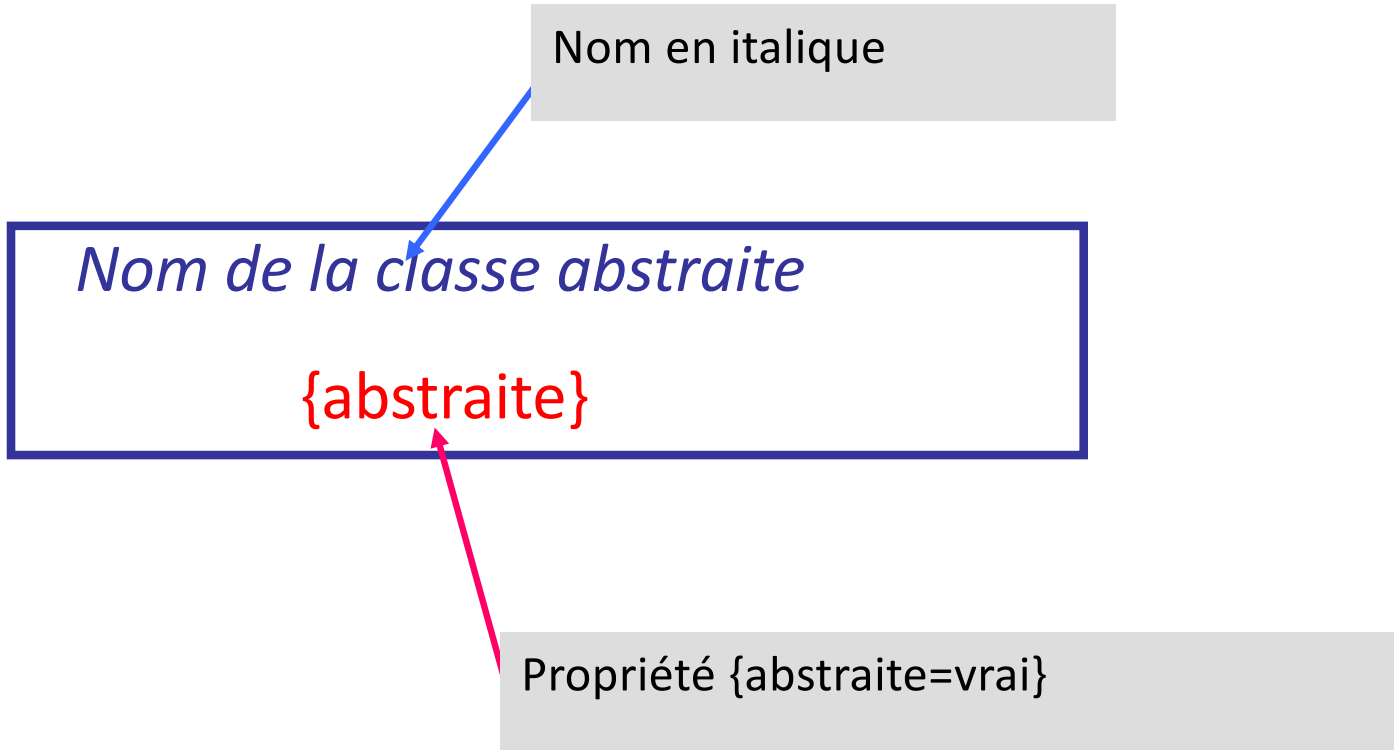
La classe Personne ne doit pas être modifiée

Classes abstraites et polymorphisme

- Notion de méthode et classe abstraites
- Polymorphisme
- Typage statique et dynamique

Classe abstraite

- Une classe abstraite est une classe qui n'a pas d'instances directes.

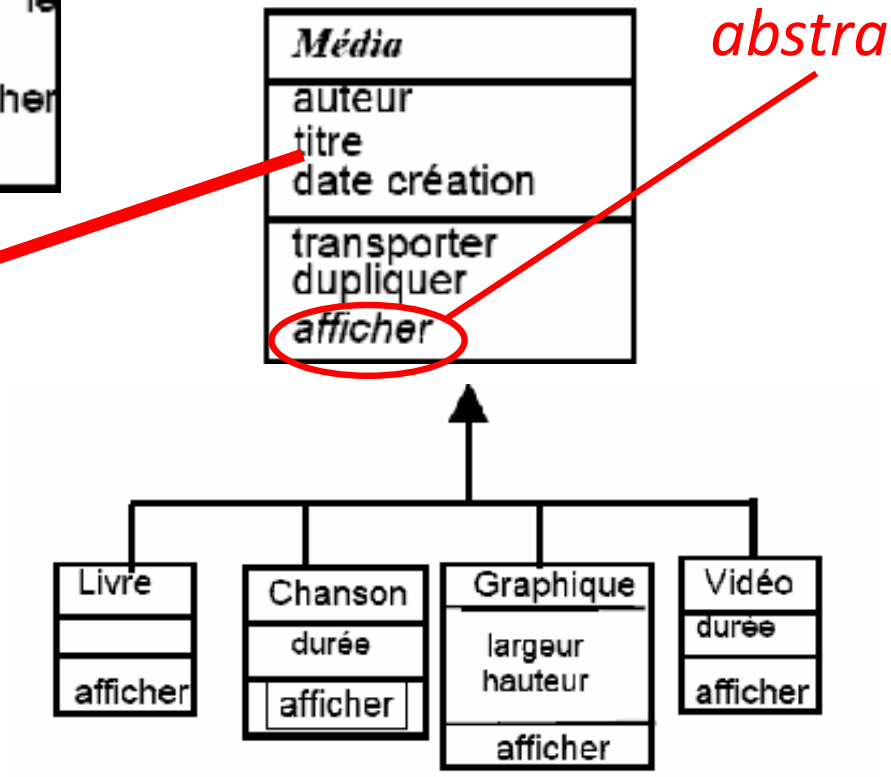


Classe abstraite

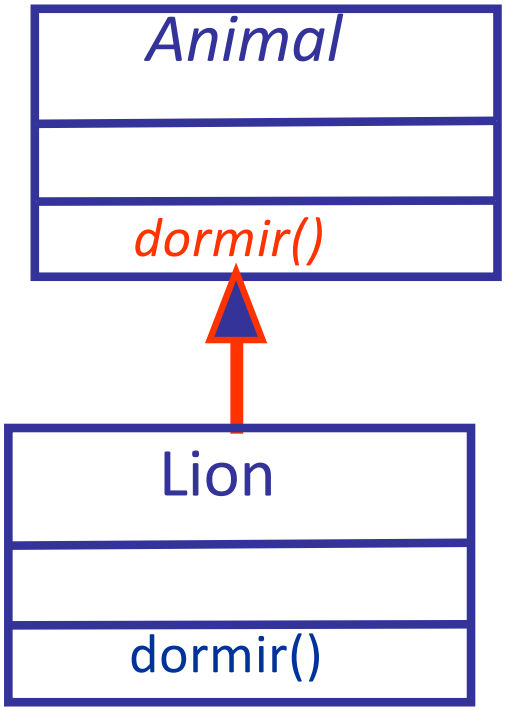
Un média peut être transporté, dupliqué, affiché.
Le transport et la duplication sont indépendantes du type du média (copie de fichiers).
Par contre, tout média peut être affiché et ce n'est pas la même chose pour l'audio, la vidéo, le graphisme, le texte.
Un média ne peut pas définir comment s'afficher tant qu'il ne sait pas ce qu'il est.

Il n'y a pas d'instance de la classe média .
Un média existe en tant que livre, chanson, graphique, vidéo.

Opération abstraite



Classe et méthodes abstraites en Java

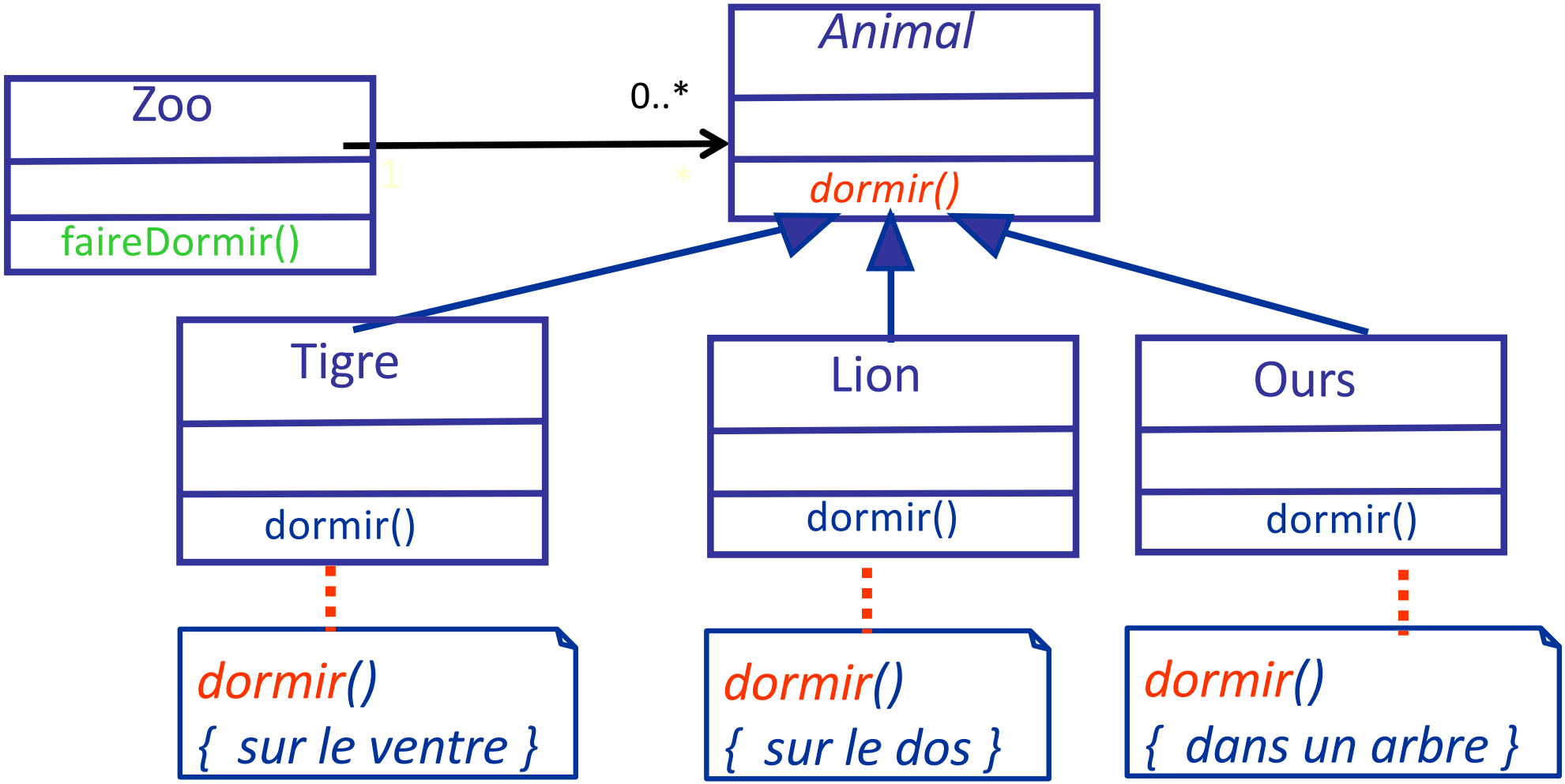



```
abstract class Animal
{
    abstract void dormir();
.....
}
```

- Une méthode est abstraite dans une classe si elle n'est pas implémentée dans la classe mais dans une de ses filles.
- Seules les classes abstraites peuvent posséder des méthodes abstraites.

Polymorphisme

Collection polymorphe *Zoo*



Classe abstraite en Java

```
public abstract class Animal {
  protected String nom;
  protected int age;
  // constructeur de la classe Animal
  protected Animal (String nom) {
      this.nom = nom; }
  // implémentation de la méthode affichetonNom()
  protected void afficheTonNom() {
  System.out.println(getClass().getName() + ":" + nom);
  }
  // déclaration de la méthode abstraite dormir()
  abstract public void dormir();
}
```

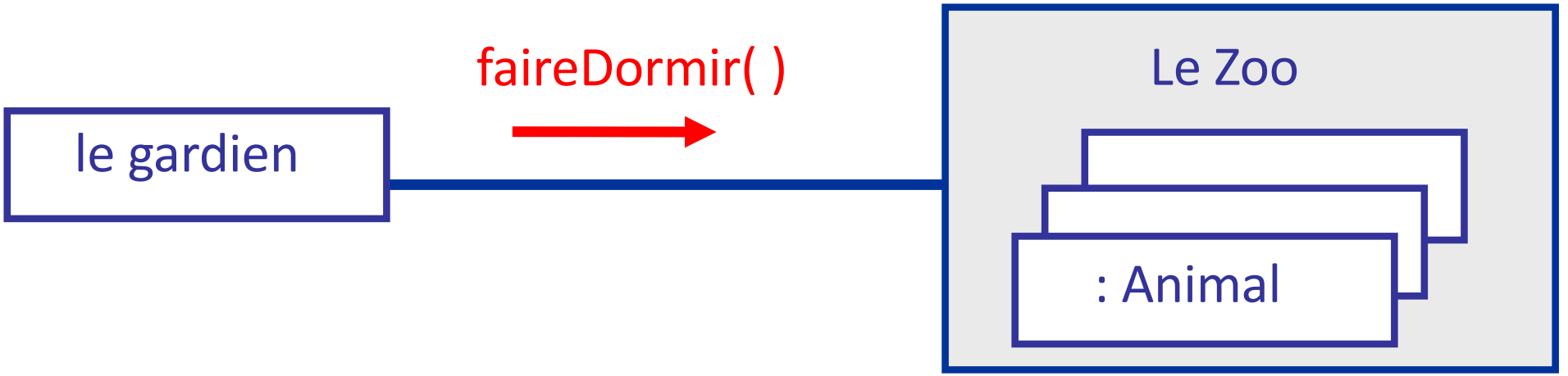
Réalisation de méthode abstraite

```
public class Lion extends Animal {
   // Constructeur de la classe Lion
   public Lion (String nom){
       super(nom);
   }
   //implémentation de la méthode dormir()
   public void dormir() {
       System.out.println (nom + "dort sur
        le dos");
   }
}
```

Le polymorphisme

Collection polymorphe *Zoo*

Un message ***dormir*** est envoyé à chaque animal, qui particularisera cette fonction.



Collection polymorphe

```
public class Zoo {
// Liste des animaux du zoo
private  ArrayList<Animal> animaux ;
public Zoo (){
     animaux= =new ArrayList<Animal>() ;
}
public void ajouterAnimal(Animal a){
     animaux.add(a);
public void faireDormir(){
          for (Animal e:animaux )
                    e.dormir();}
}
```

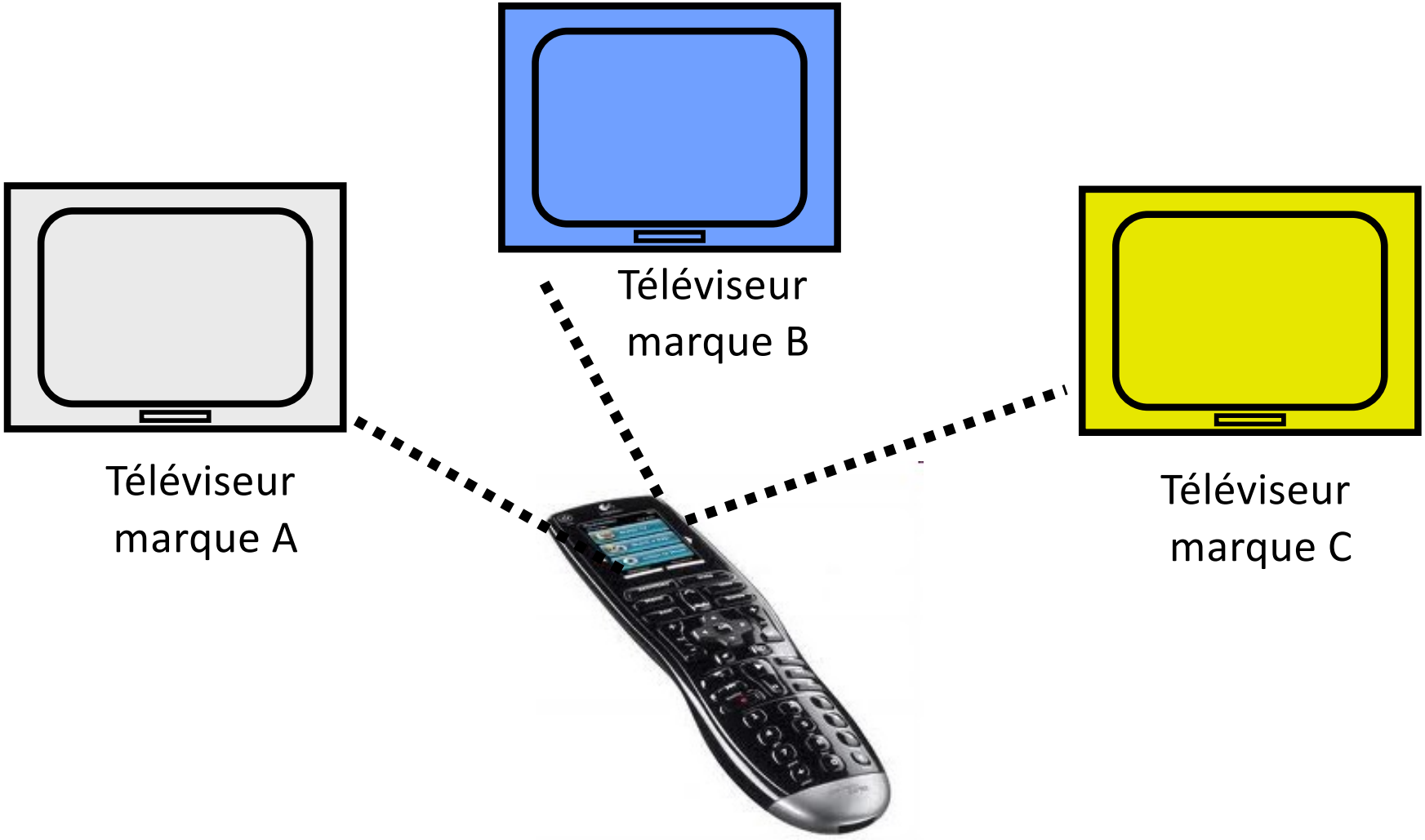
Envoi de message à une Collection polymorphe

```
public class Test {
    public static void main (String arg[]){
    // Création du zoo
    Zoo leZoo = new Zoo();
   //Ajout d'animaux
   leZoo.ajouter(new Lion("Prince");
   leZoo.ajouter(new Tigre("Share khan");
   leZoo.ajouter(new Ours("Petit Jean");
   // Ordre de dormir à tous les animaux
   leZoo.faireDormir();
}
}
```

Prince dort sur le ventre
Share Khan dort sur le dos
Petit Jean dort dans un arbre

Polymorphisme

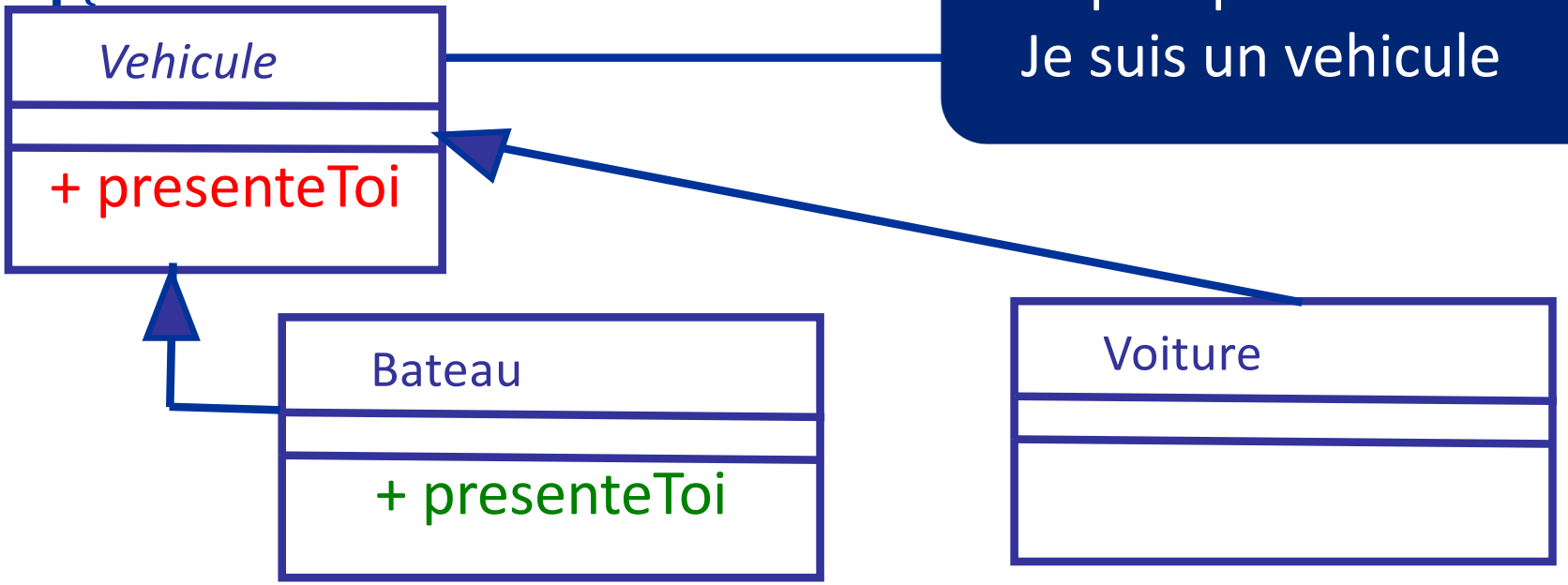
Possibilité de cacher plusieurs implémentations derrière la même interface



Rédéfinition et polymorphisme

```
public class testVehicule {
    public static void main(String[] args) {
            Vehicule totoche=new Bateau();
            totoche.presenteToi();
            Vehicule titine=new Voiture();
            titine.presenteToi();
            }
}
```

```
Je suis un vehicule
et plus précisément un bateau
Je suis un vehicule
```



Rédéfinition et polymorphisme

- Une référence vers une classe C peut contenir des instances de C ou des classes dérivées de C.

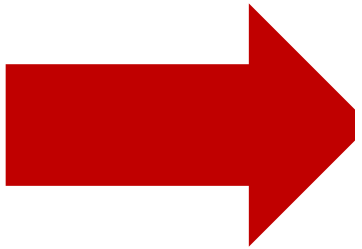
```
public class testVehicule {
   public static void main(String[] args) {
      Vehicule[] garage=new Vehicule[2];

      garage[0]=new Voiture();
      garage[1]=new Bateau();
      for (int i=0; i<garage.length; i++)
            garage[i].presenteToi();
       }
}
```

```
Je suis un vehicule
Je suis un vehicule
et plus précisément un bateau
```

TPCours - Exercice 4.3

- Les salaires des enseignants d'une université sont calculés de manière spécifique en fonction de leur catégorie :
  - les titulaires qui ont un salaire fixe
  - les vacataires ont un salaire qui dépend du nombre d'heures de cours qu'ils assurent sachant qu'une heure de cours est rémunérée 40 euros.
- On souhaite définir une classe abstraite Enseignant comportant une méthode abstraite *salaire()* et utiliser le polymorphisme pour manipuler une liste d'enseignants dans une classe Université.

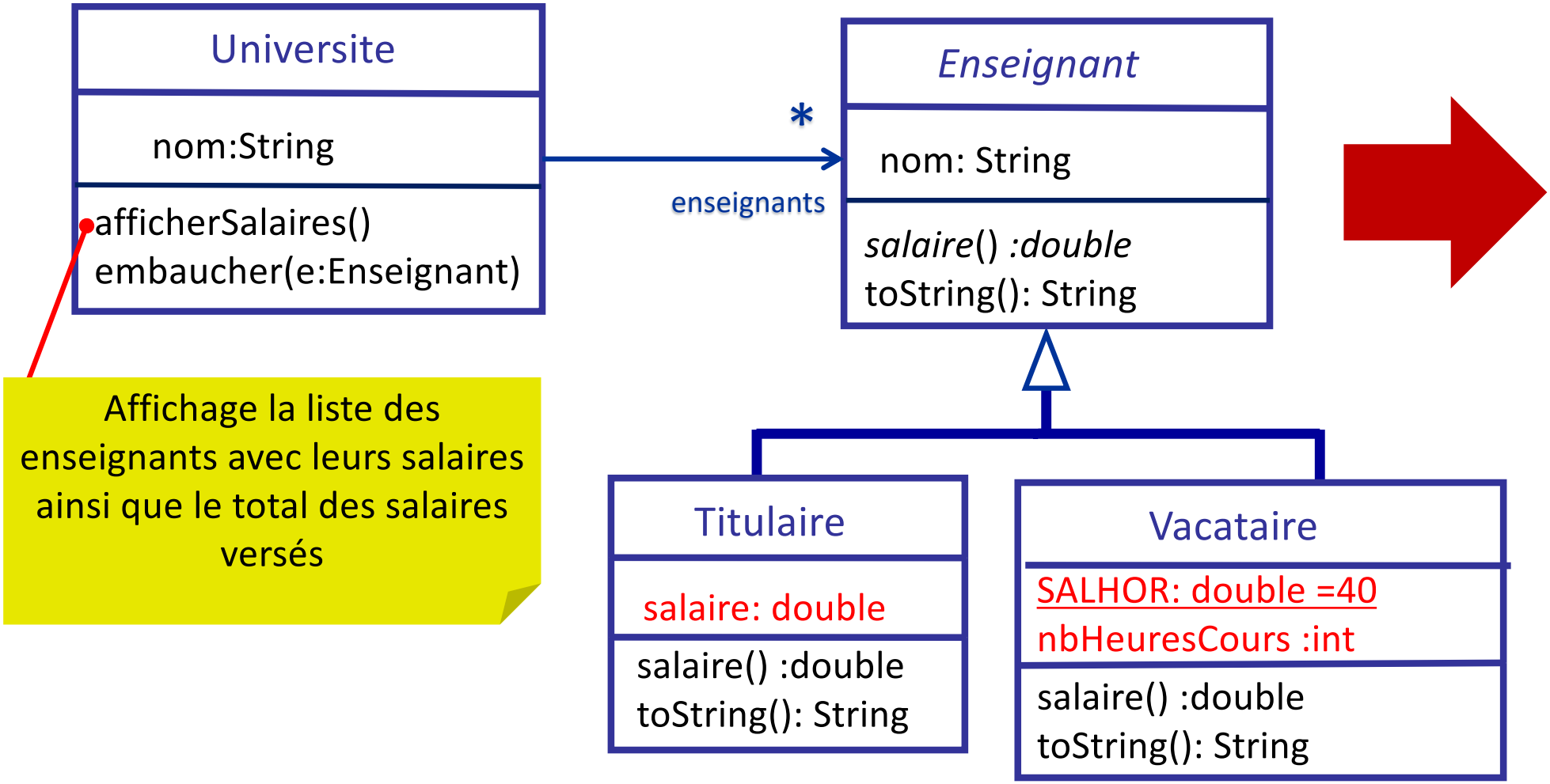
Exercice indépendant des exercices précédents



TPCours - Exercice 4.3

Exercice indépendant des exercices précédents

1 - Implémentez le diagramme de classe suivant:



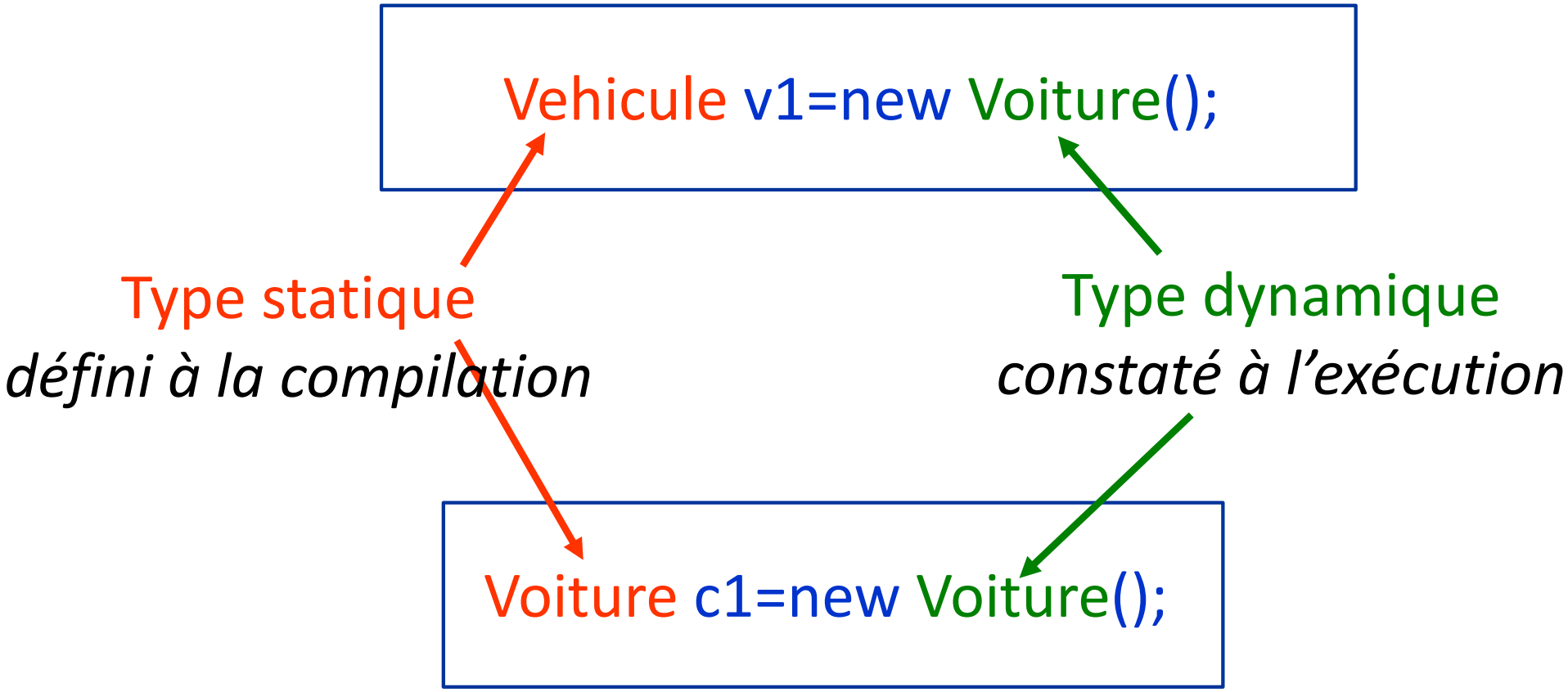
TPCours - Exercice 4.3

- 2 - Définissez un programme de test dont l'exécution du programme doit donner l'affichage suivant :

```
LISTE DES ENSEIGNANTS DE l'UNIVERSITE Pascal Paoli
Effectif: 4 enseignants
Pierre (titulaire)  : 1500.0 euros
Laurent (titulaire)  : 2500.0 euros
Michel (vacataire 15 heures) : 600.0 euros
Marie (vacataire 60 heures) : 2400.0 euros
Total des salaires = 7000.0 euros
```

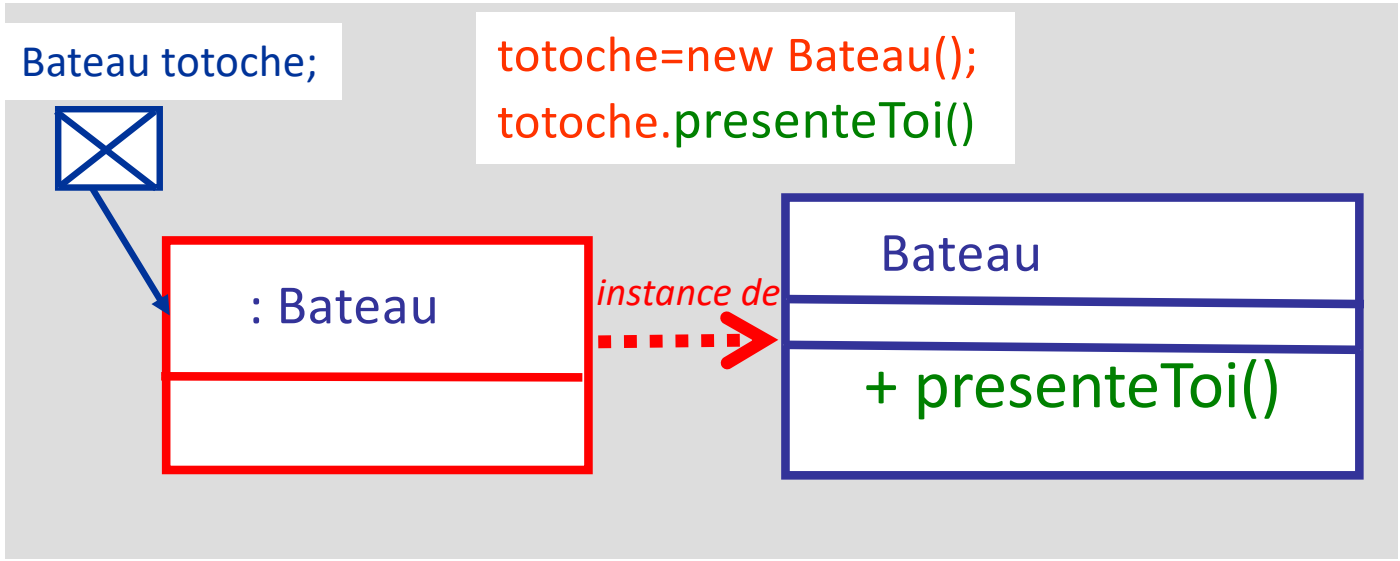
Types statiques et dynamiques

Typage Statique et Dynamique



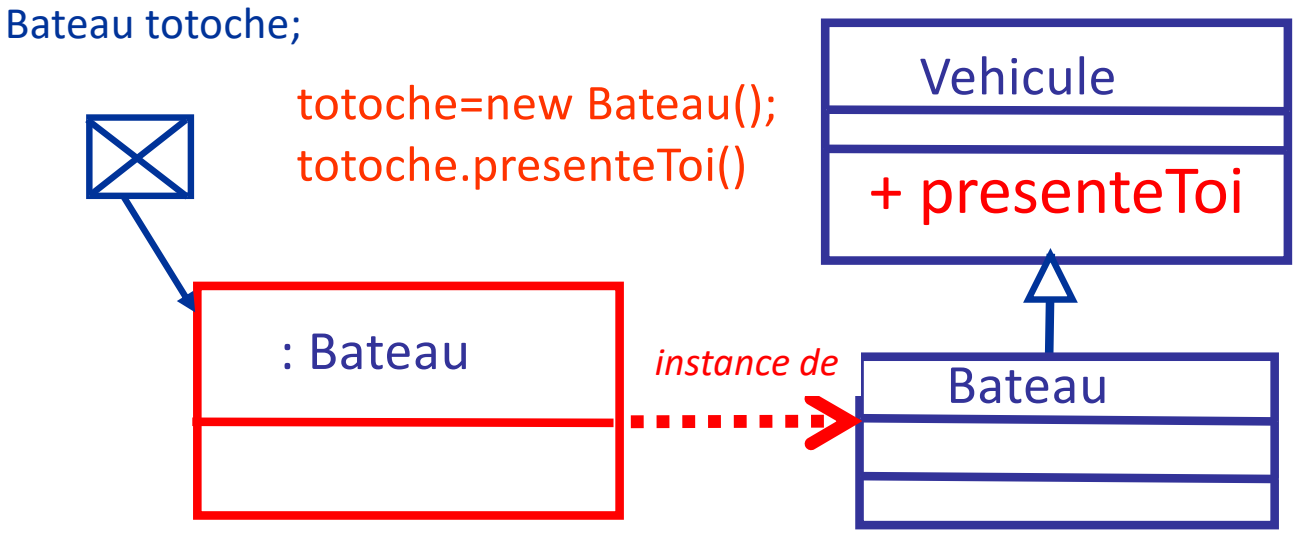
Le type dynamique détermine le point de départ de la recherche de la méthode à exécuter dans la hiérarchie d'héritage

Typage Statique et Dynamique



Pas d'héritage

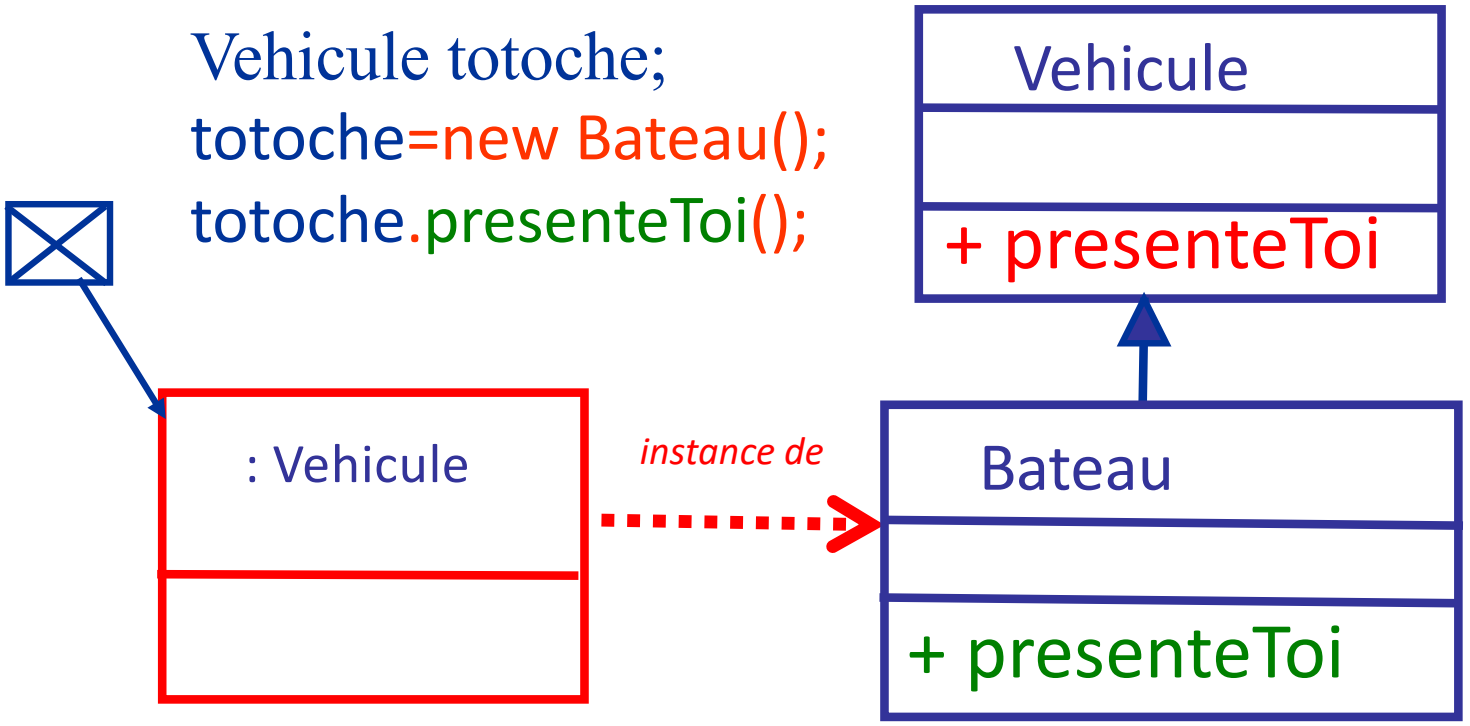
Héritage mais
Pas de rédéfinition



Typage Statique et Dynamique

Héritage + Rédéfinition : « dynamic binding »

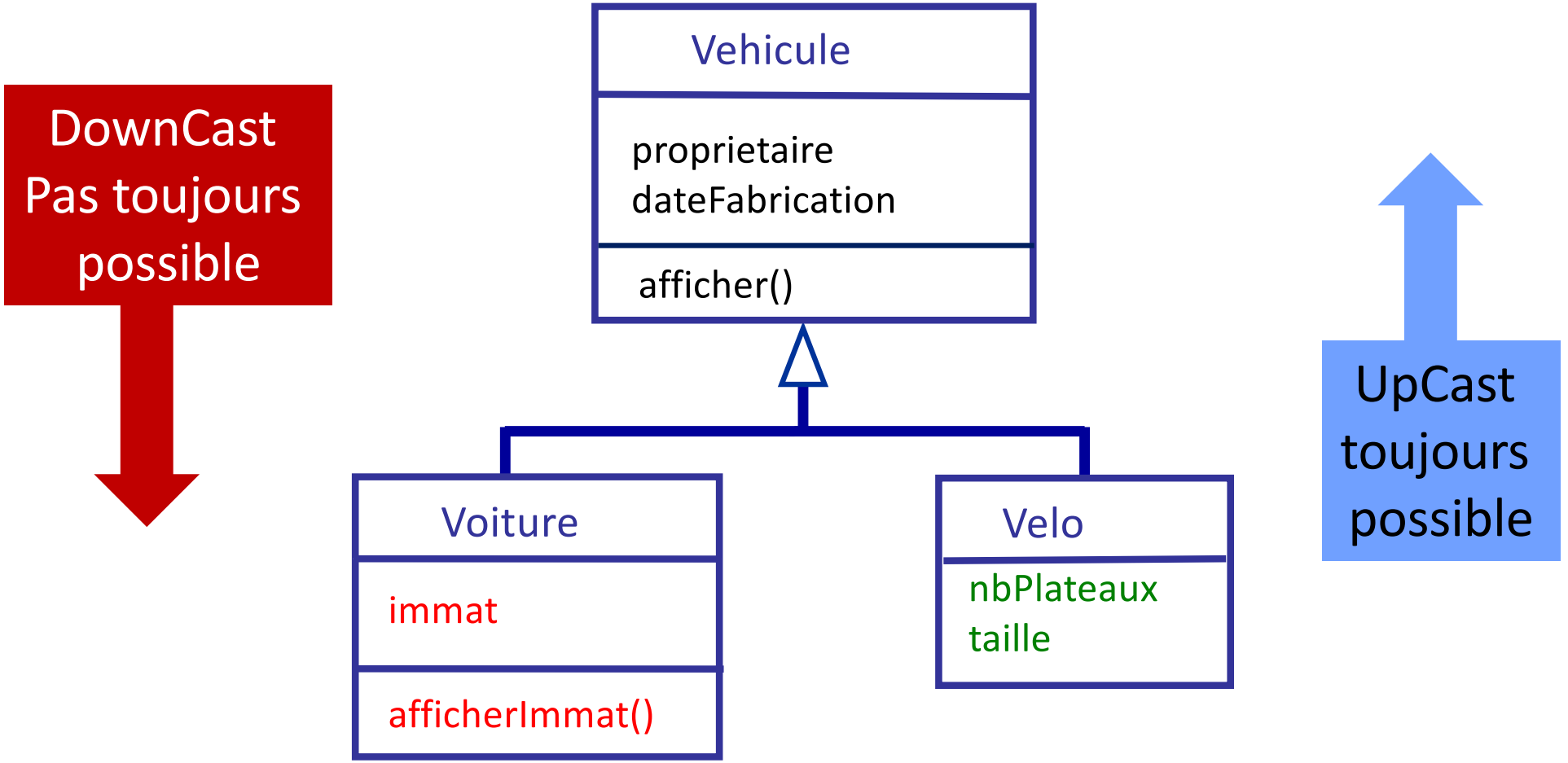
```
Vehicule totoche;
totoche=new Bateau();
totoche.presenteToi();
```



C'est le type dynamique qui détermine la méthode à exécuter

Conversion de type et héritage

Principe de conversion de types



Tous les véhicules ne sont pas des voitures!
Tous les véhicules ne sont pas des vélos!

Toutes les voitures sont des véhicules
Tous les vélos sont des véhicules

Upcasts et downcasts

```
Vehicule v1=new Vehicule();
Voiture v2=new Voiture();
v1=v2;  //OK: upcast toujours possible
v2=v1; //Erreur: downcast dangereux identifié
```

Type mismatch cannot convert from Vehicule to Voiture

```
//Solution: ajouter un cast explicite
v2=(Voiture) v1;
```

OK à la compilation mais Erreur à l'exécution

Problems Javadoc Declaration Console

<terminated> TestPersonne (7) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_144.jdk/Contents/Home/bin/java (24 nov. 2020 à 19:58:30 – 19:58:33)

Exception in thread "main" java.lang.ClassCastException: Vehicule cannot be cast to Voiture

Vérifications à la compilation

Le type statique est utilisé à la compilation

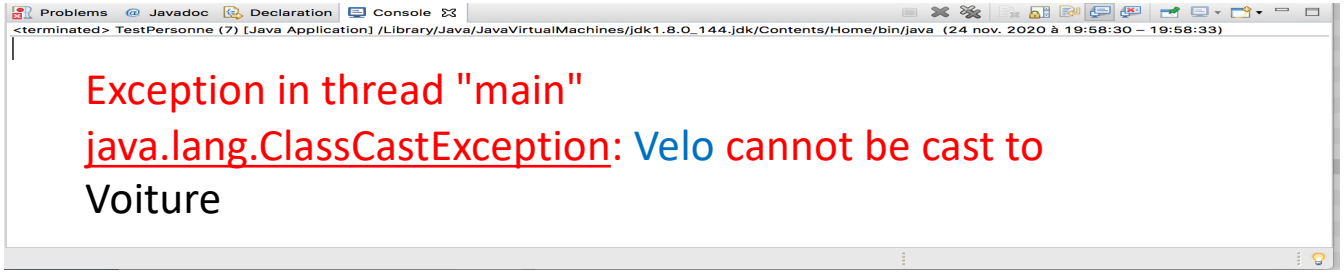
```
Voiture v2=new Voiture();
Vehicule v3=new Voiture();
v2.afficher();  //OK: afficher est hérité de Vehicule
v3.afficher(); //OK: afficher est dans  Vehicule
v2.afficherImmat(); //OK: afficherImmat est dans Voiture
v3.afficherImmat();//Erreur: afficherImmat n'est pas dans  Vehicule
//Solution: ajouter un cast explicite
((Voiture) v3).afficherImmat() ;
```

Erreurs d'exécution

Le type statique est utilisé à la compilation

```
Voiture v2=new Voiture();
Vehicule v4=new Velo();
v2.afficherImmat(); //OK: afficherImmat est dans Voiture
v4.afficherImmat();//Erreur: afficherImmat n'est pas dans  Vehicule
//Solution: ajouter un cast explicite
((Voiture) v4).afficherImmat() ;
```

OK à la compilation mais Erreur à l'exécution



Upcasts et downcasts (en résumé)

- Upcast= toujours possible (cast implicite)
  - On peut toujours convertir une variable de type statique Voiture ou Velo en Vehicule
- Downcast= jugé dangereux par le compilateur (cast explicite necessaire)
  - Avec un cast explicite, on peut forcer la conversion d'une variable de type statique Vehicule en Voiture ou Velo
  - Mais attention si le type dynamique ne correspond pas : Erreur à l'exécution

TPCours - Exercice 4.4

- Identifiez les erreurs de compilation dans le programme suivant, expliquez-les en ajoutant des commentaires et corrigez les si cela est possible:

```
public class TestPersonne4_4 {
//Classes Personne, Etudiant, Enseignant de l'exercice 4.2
    public static void main(String[] args) {
    Personne p1=new Personne("Marie",20);
    Personne p2=new Etudiant("Machin",26,"20202134");
    Personne p3=new Enseignant("Toto",34,2000,192);
    Etudiant p4=new Etudiant("Jean", 21,"20203433");
    Enseignant p5= new Enseignant("Pierre",54,3000,250);
    p1=p4;
    p1=p5;
    p4=p2;
    p5=p1;
    p5= p4;
    ((Etudiant)p2).setNumEtu("20205784");
    p4.setNumEtu("20205785");
    p5.setNumEtu("20205786");}}
```

code à récupérer sur l'ENT

TPCours - Exercice 4.5

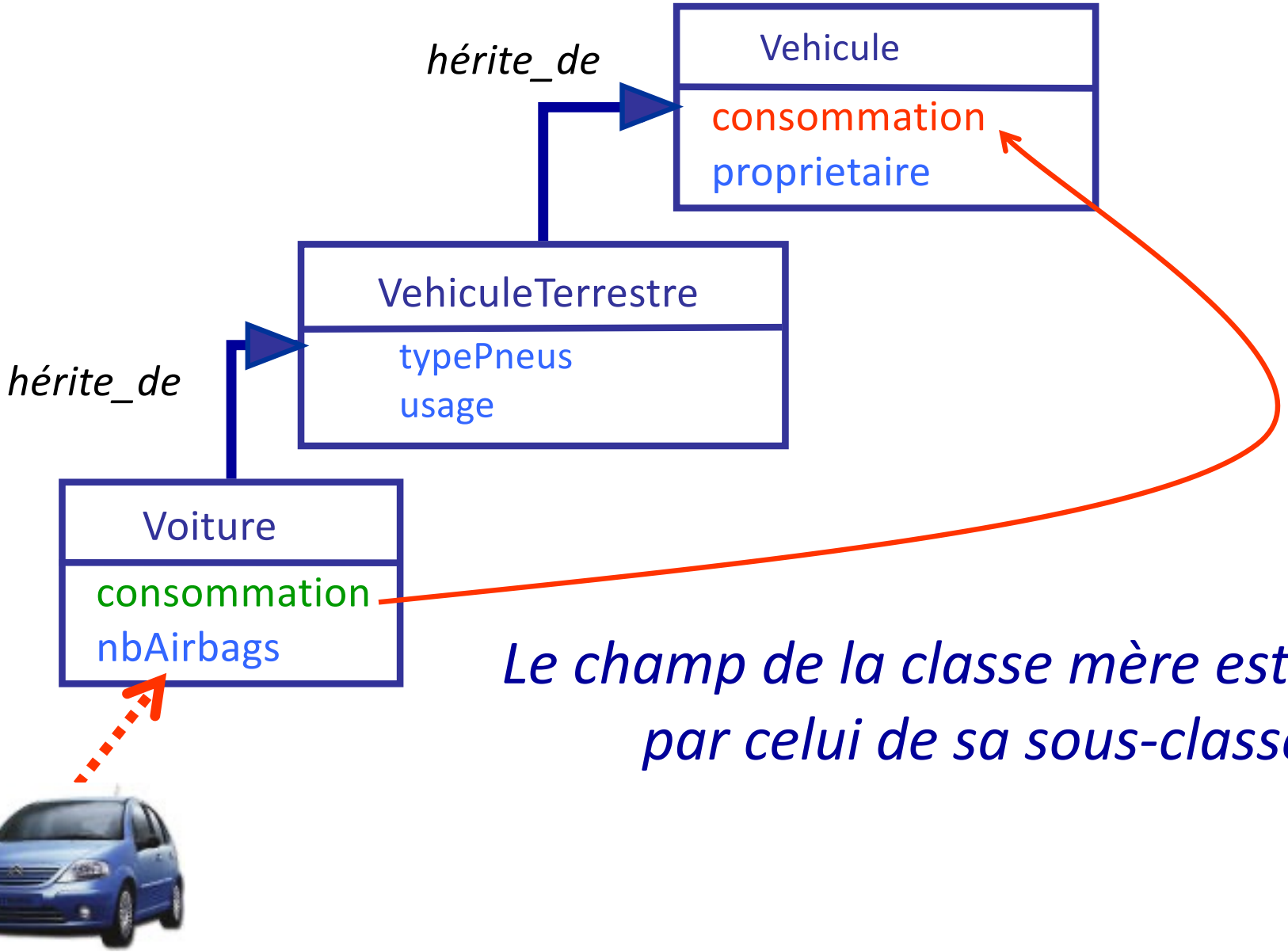
- Identifiez les erreurs d'exécution dans le programme suivant et expliquez-les en ajoutant des commentaires :

```
public class TestPersonne4_5 {
    public static void main(String[] args) {
    Personne p1=new Personne("Marie",20);
    Personne p2=new Etudiant("Machin",26,"20202134");
    Personne p3=new Enseignant("Toto",34,2000,192);
    Etudiant p4=new Etudiant("Jean", 21,"20203433");
    Enseignant p5= new Enseignant("Pierre",54,3000,250);
    p4=(Etudiant) p2;
    p5=(Enseignant) p1;
    ((Etudiant) p2).setNumEtu("20205784");
    p4.setNumEtu("20205785");
    ((Etudiant) p3).setNumEtu("20205786");
    }
```

code à récupérer sur l'ENT

Héritage et masquage de champs

Héritage + masquage de champs



Le champ de la classe mère est caché par celui de sa sous-classe

Héritage + masquage de champs

```
class Vehicule {
        int puissance;
}
class Voiture extends Vehicule{
        String puissance;
}
class TestVehicule{
        public static void main(String args[]){
                Vehicule v=new Voiture();
                v.puissance="4CV"; // ERREUR
                v.puissance=4; //Vehicule
                Voiture c=new Voiture();
                c.puissance=5; // ERREUR
                c.puissance="5CV"; //Voiture
        }
}
```

C'est le type statique (déclaré) qui détermine le champ à considérer