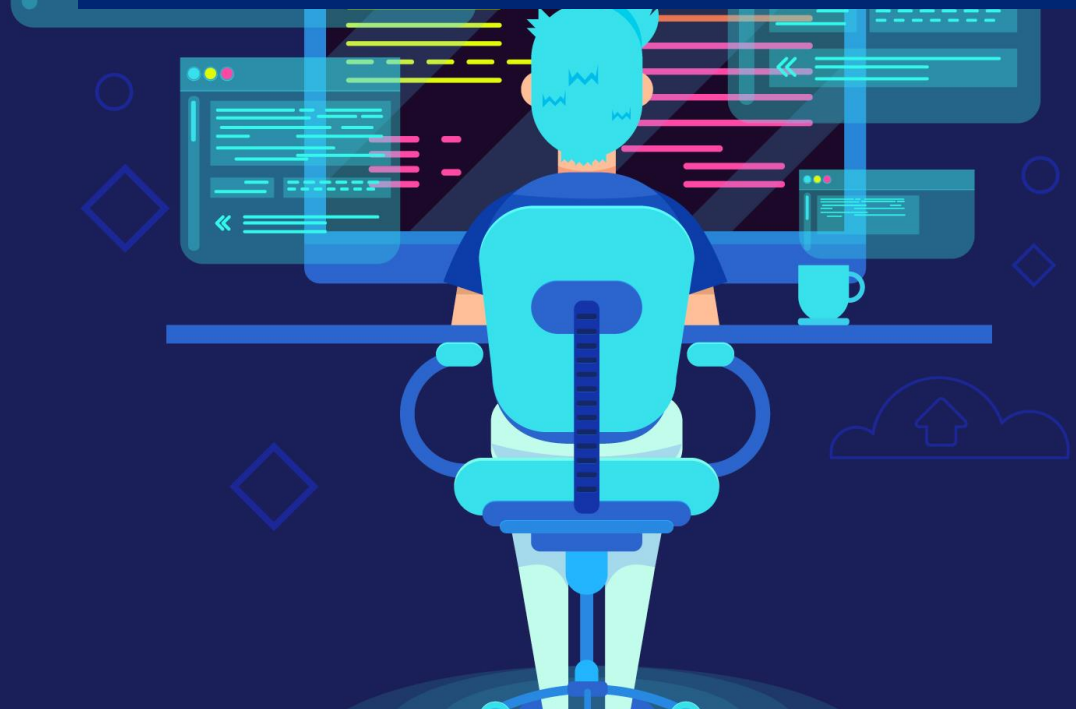
L3 Informatique - 2024-2025

Partie POO 2 – Héritage et polymorphisme



Paul-Antoine BISGAMBIGLIA – bisgambiglia_pa@univ-corse.fr
Marie-Laure NIVET – nivet_ml@univ-corse.fr
Evelyne VITTORI - vittori@univ-corse.fr

Università di Corsica Pasquale Paoli

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 2 – HERITAGE et POLYMORPHISME

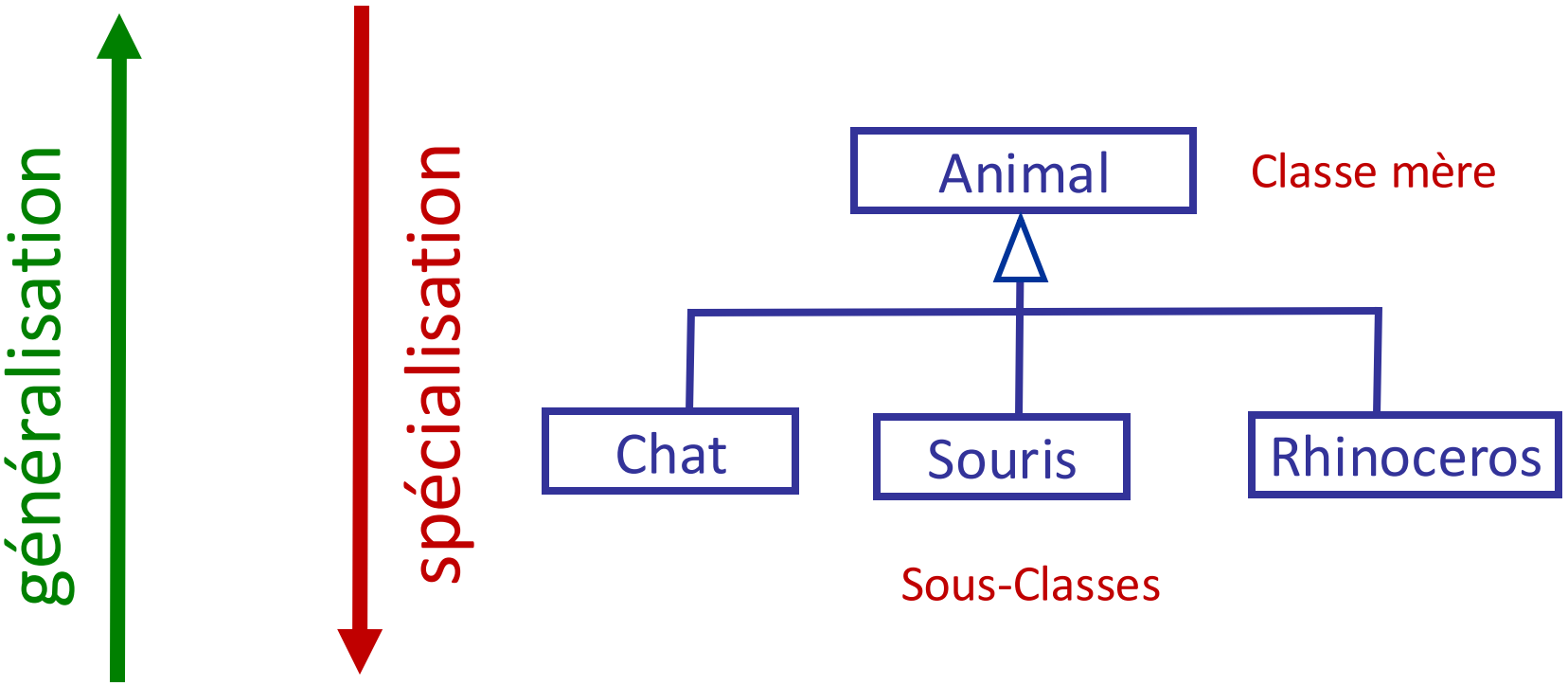
- Hiérarchie de Généralisation/Spécialisation
  - Relation EST-UN en UML
  - Sous-classes en Java
  - Démarche de construction

- Mécanisme d'Héritage
- Classes abstraites et polymorphisme
- Interfaces en java
- Comparaison d'objets
- Clonage

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 »

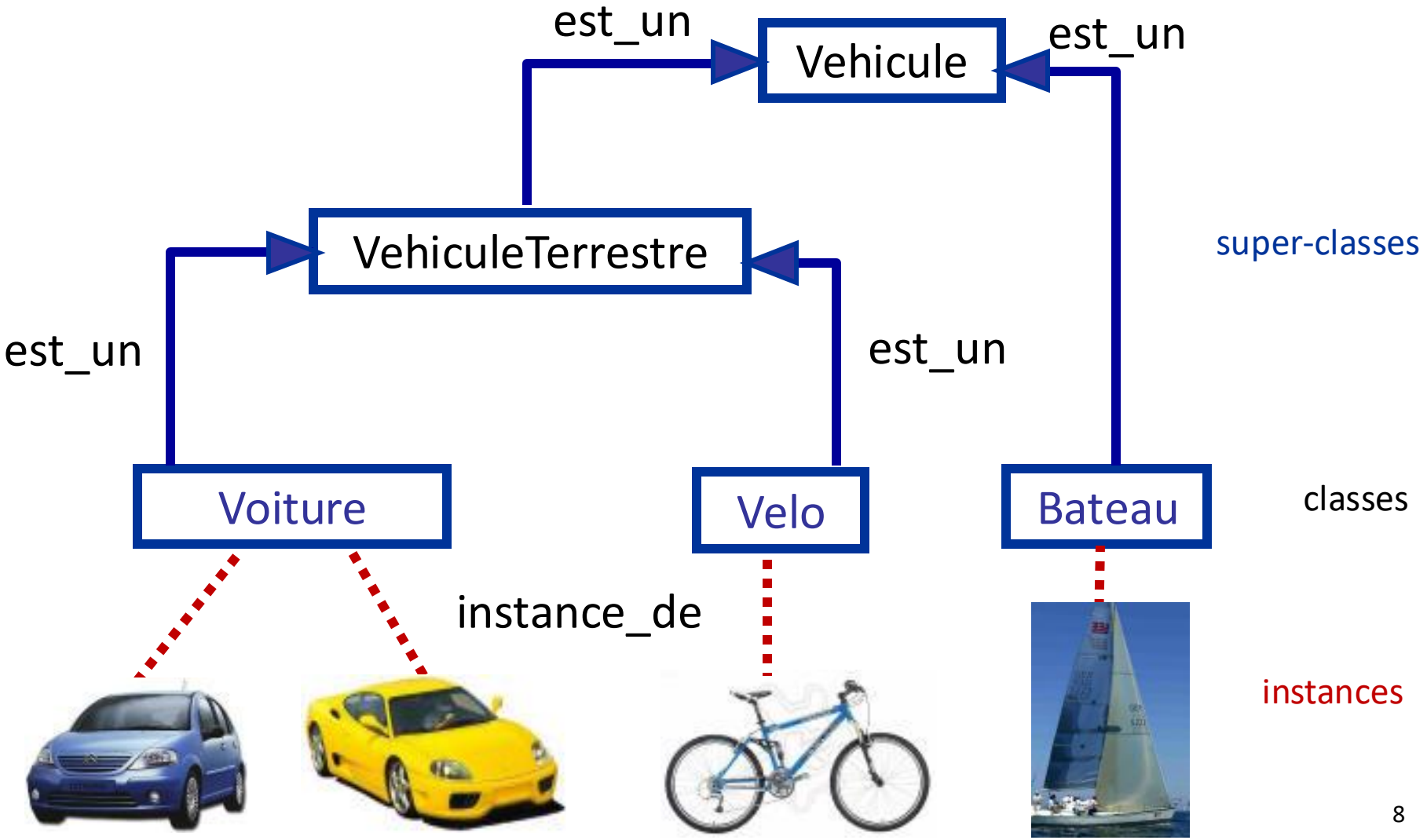
Pourquoi utiliser l'héritage ?

- 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

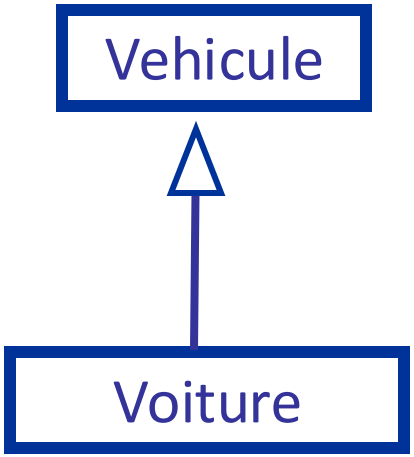
Hiérarchie de Généralisation -Spécialisation



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

Mécanisme d'Héritage

- Principe de l'héritage simple
- Héritage de propriétés
- Héritage et encapsulation
- Héritage et redéfinition de méthodes
- Héritage et constructeurs

De quoi hérite-t-on?

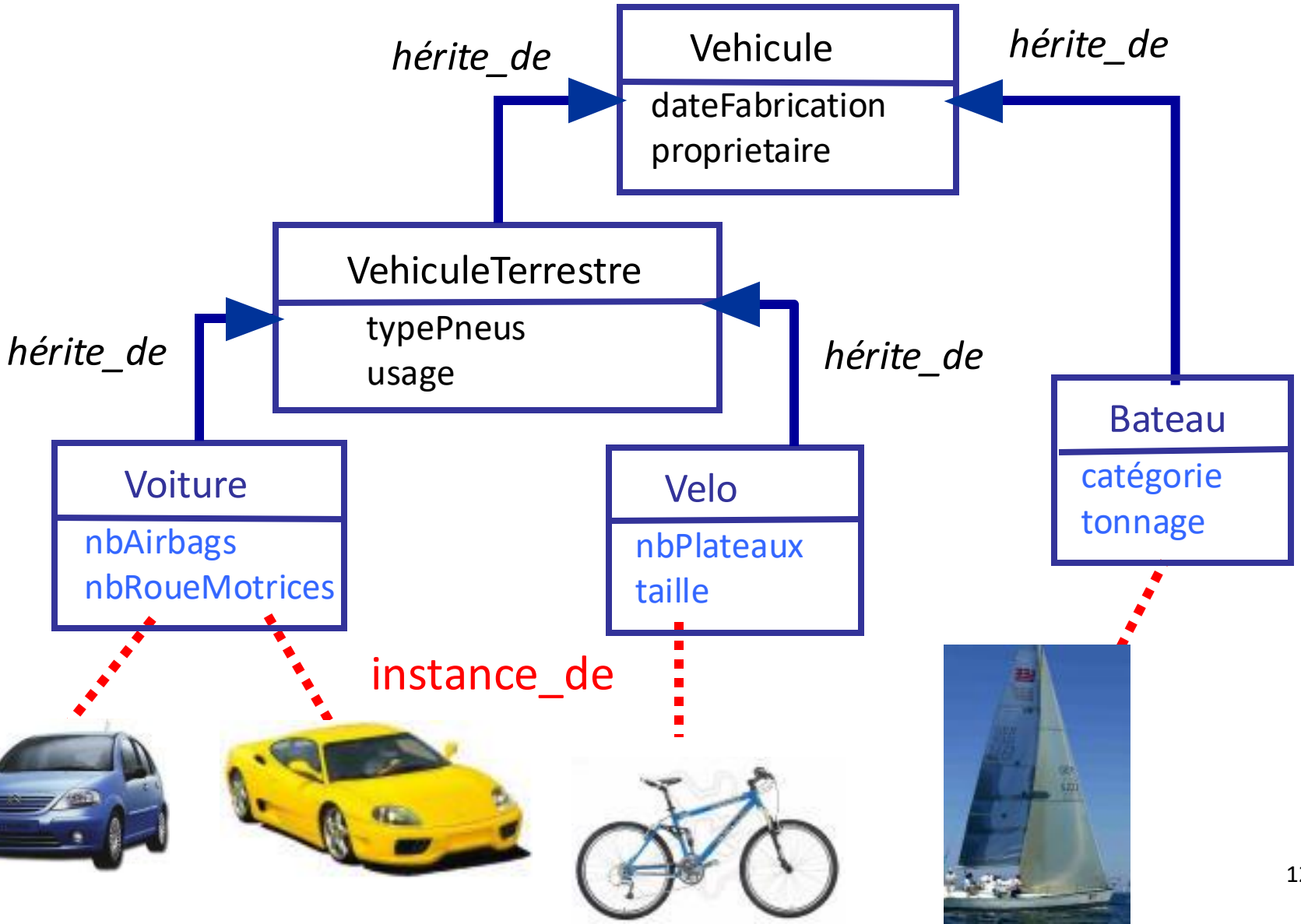
- Une sous classe hérite :

  des attributs, des opérations et des relations.

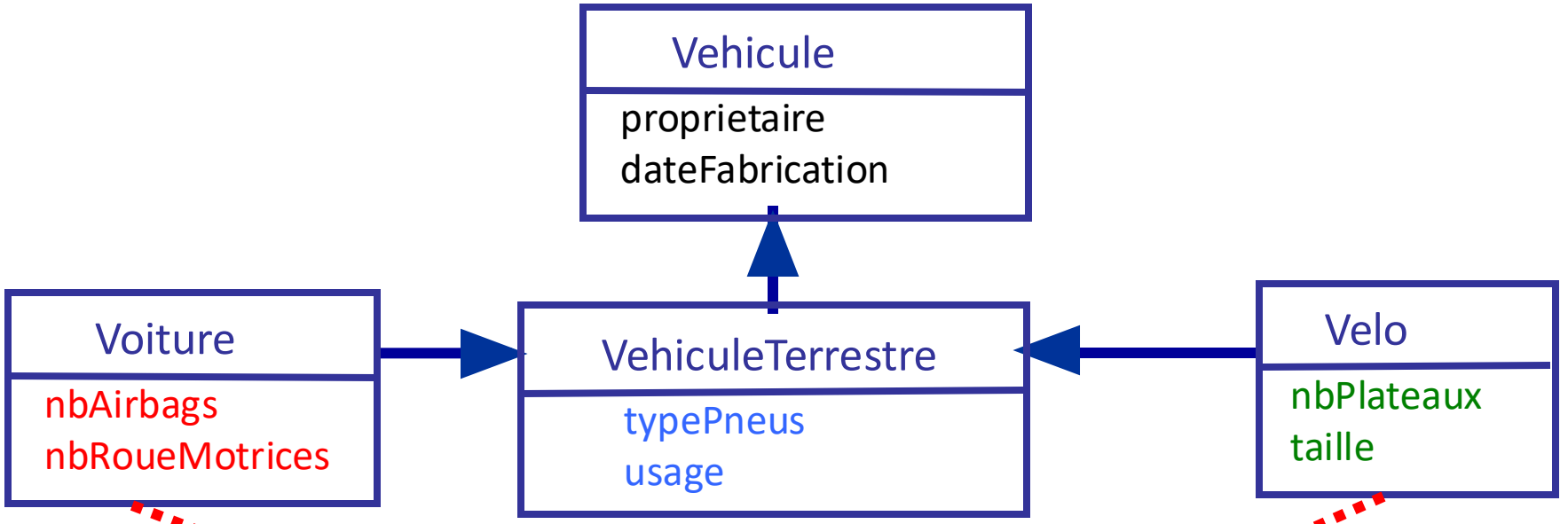
- Une sous classe peut:
  - **Ajouter** des attributs, opérations et relations spécifiques.
  - **Redéfinir** les opérations héritées.

Les attributs, relations et opérations communs sont visibles au niveau le plus haut de la hiérarchie.

Héritage de propriétés



Héritage de propriétés



titine: Voiture

datefabrication=22/04/2005
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;
}
```

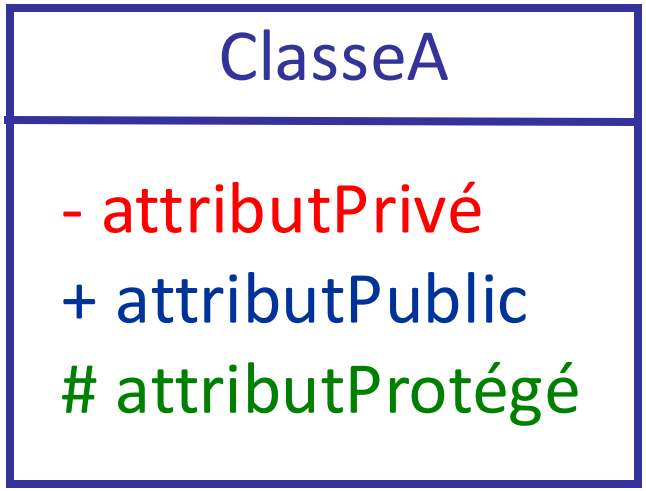
```
class Voiture extends VehiculeTerrestre {
        int nbAirbags;
        int nbRoueMotrices;
}
```

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

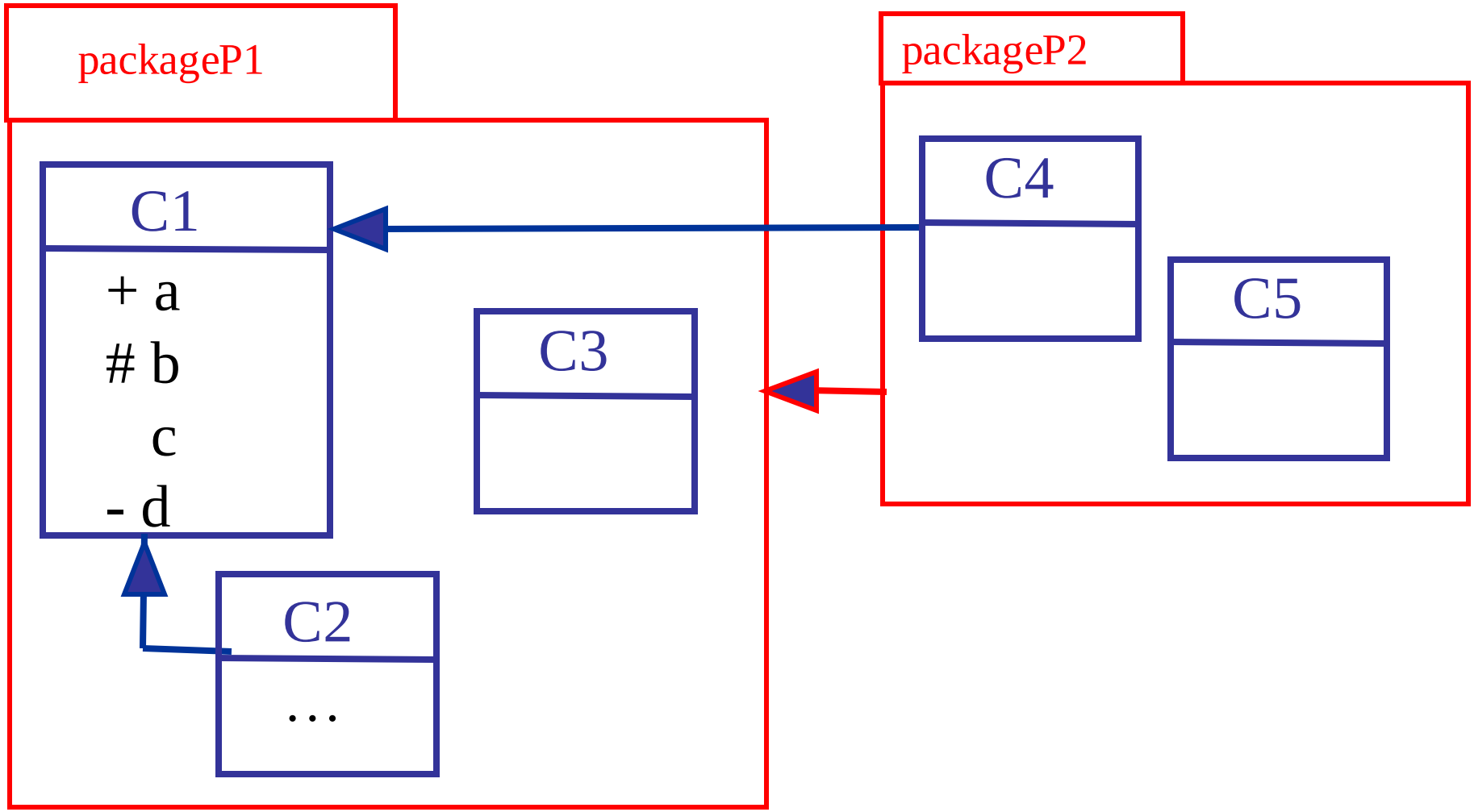
Héritage et Encapsulation

protected…

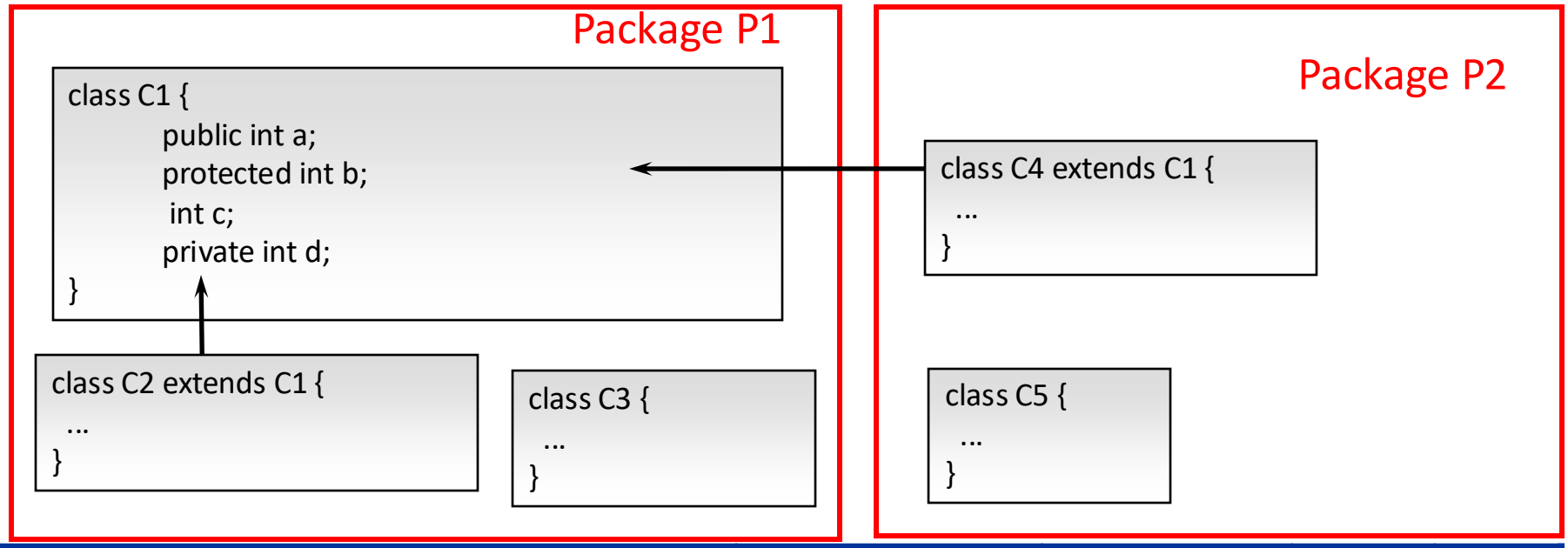
```
package vehicules;
class Vehicule {
        private String immat;
        protected String proprietaire;
        void immatriculer() { ….. // visibilité package  }
}
```

```
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**

Héritage de méthodes

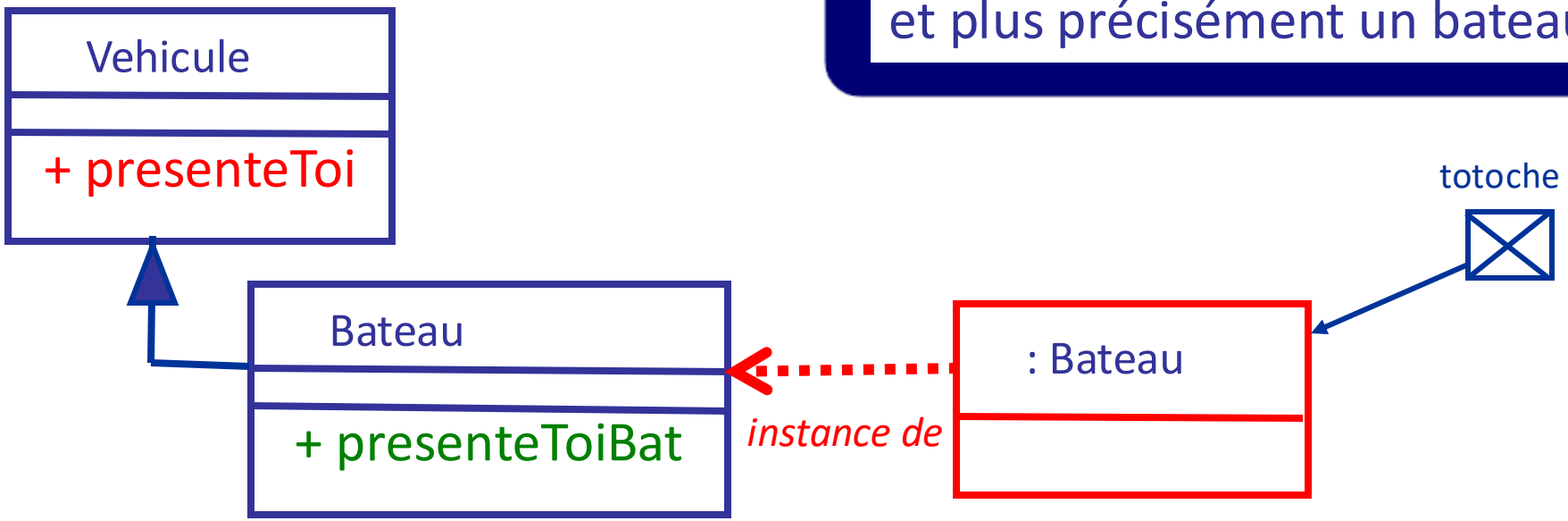
```
class Vehicule {
        ....
        public void presenteToi(){
                System.out.println("Je suis un vehicule");
        }
}
```

```
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



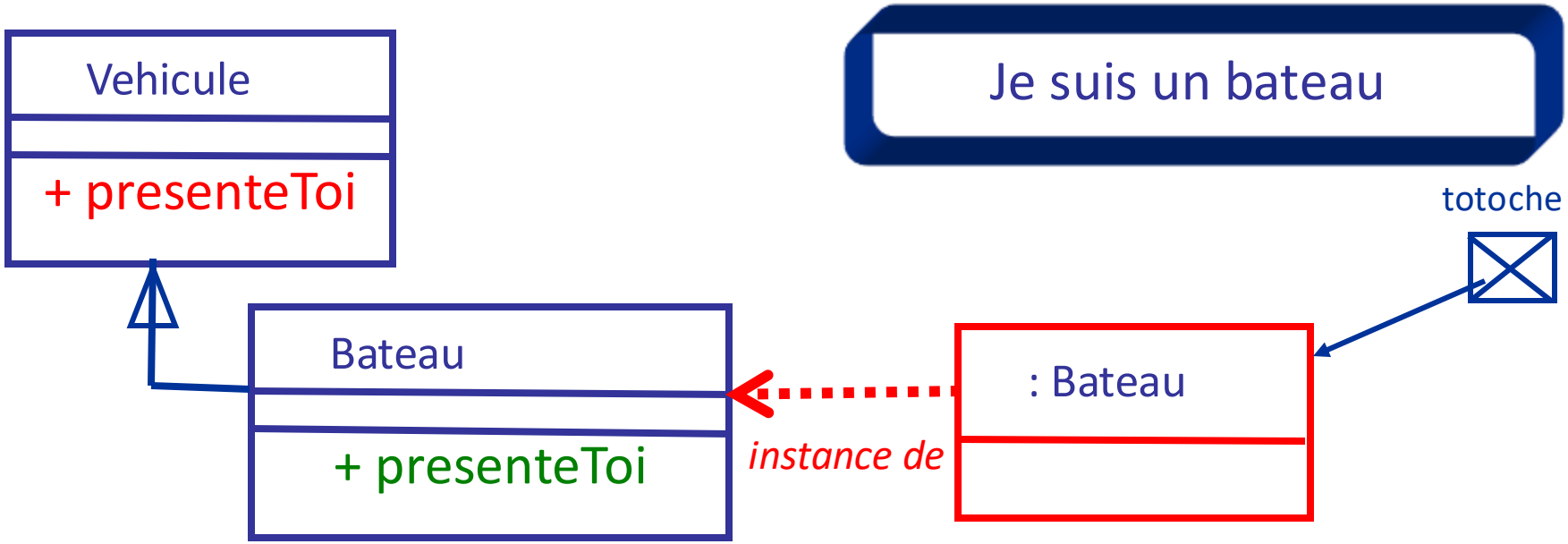
Rédéfinition de méthodes

```
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");
        }
}
```

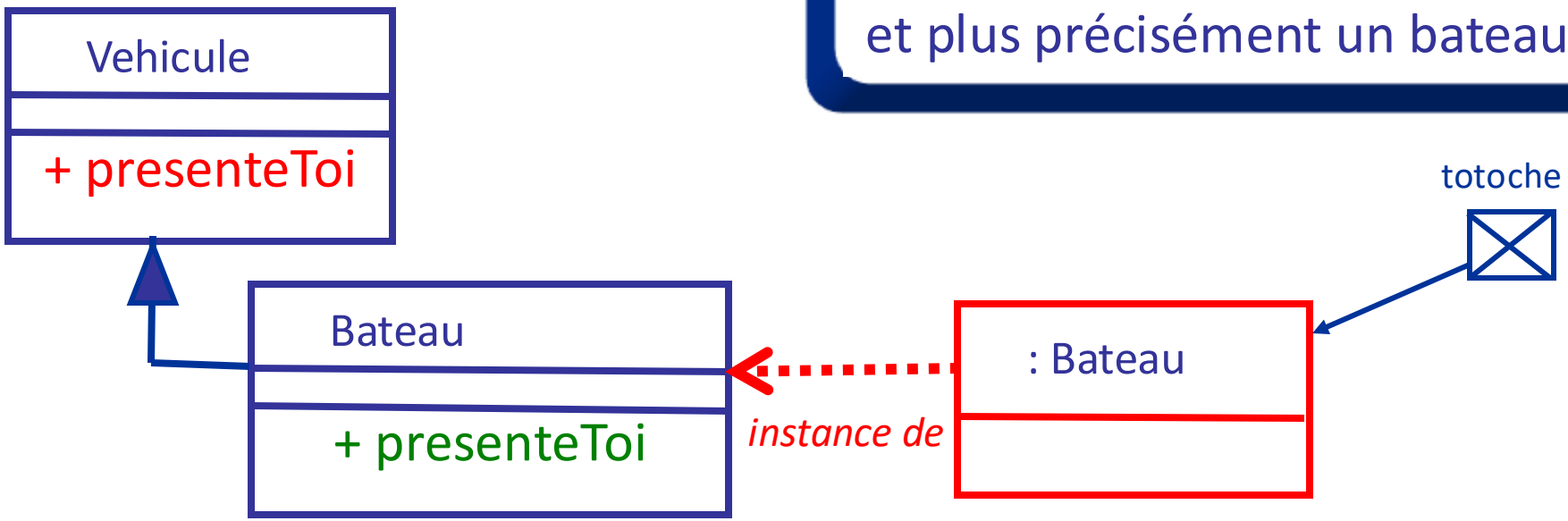
```
class Bateau extends Vehicule {
    ...
    public void presenteToi (){
      super.presenteToi();
      System.out.println("et plus précisément un bateau");
        }
}
```

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

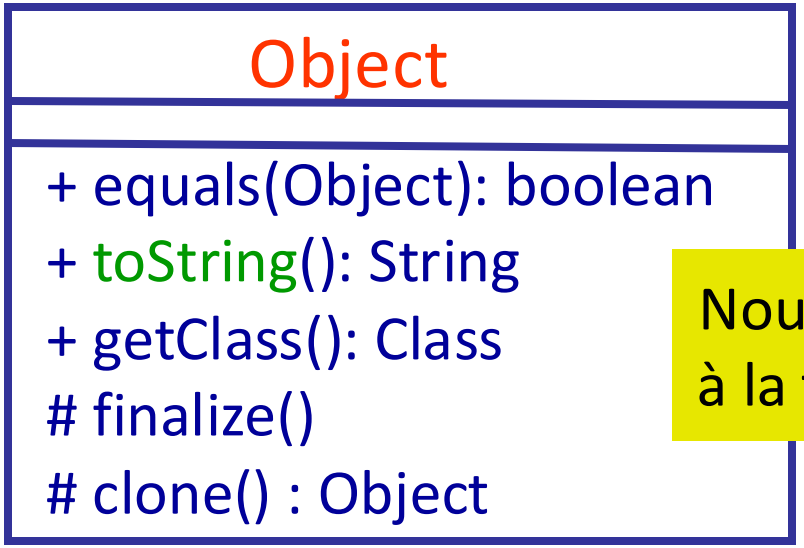
- Une **classe** déclarée **final** ne peut plus être dérivée

- 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

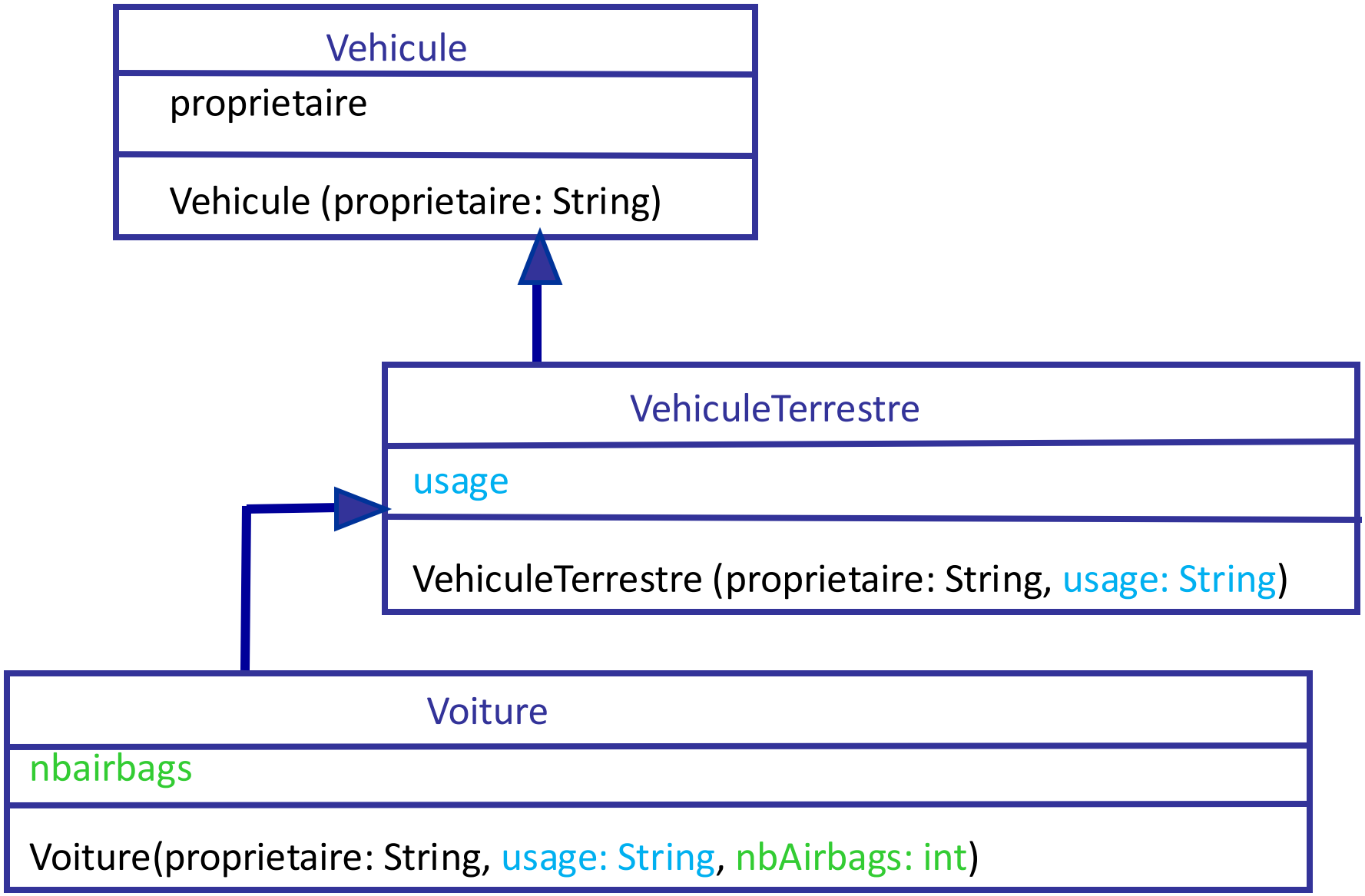
- Les constructeurs ne sont pas hérités.
- Un constructeur de la classe super-classe doit obligatoirement être appelé dans le constructeur de la classe fille

```
public class Test extends ParentTest{

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

Sauf en cas d'invocation implicite

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 (9)

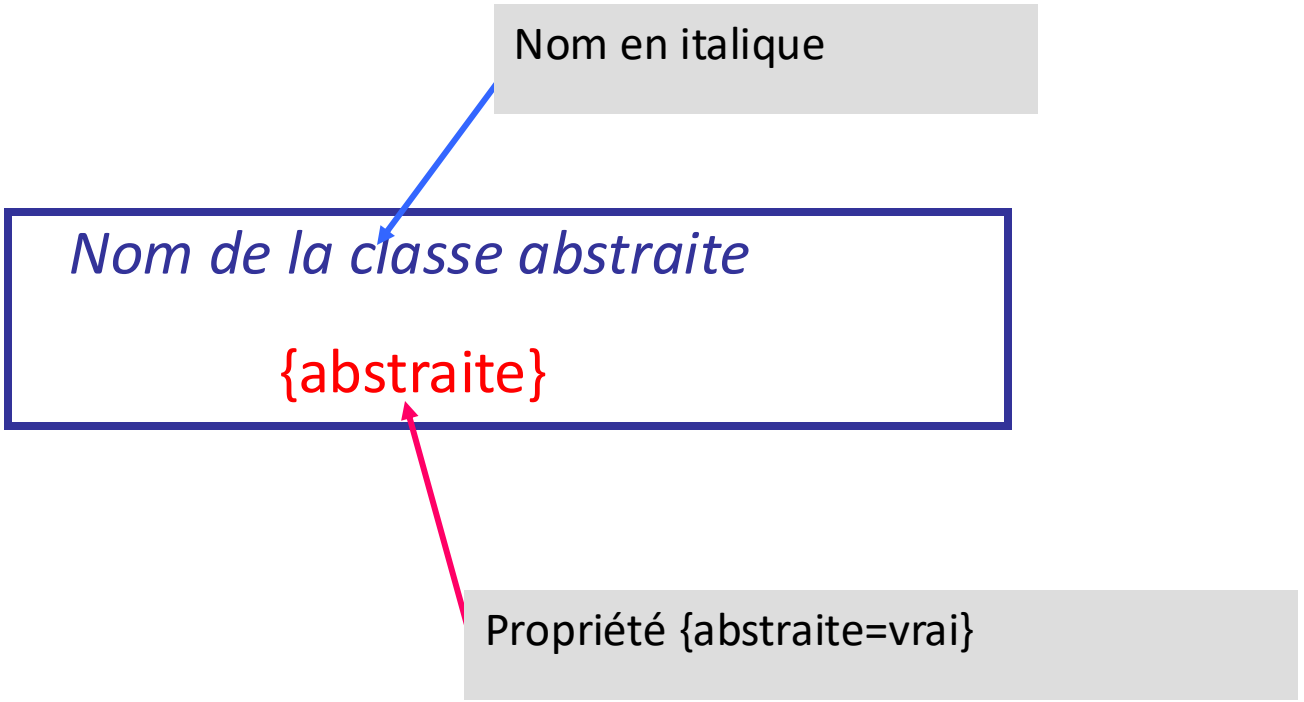
- Redéfinissez la méthode toString de la classe Personne afin qu'elle renvoie une chaine de la forme « nom, adresse, salaire »
- Définissez une classe Etudiant sous-classe de la classe Personne possédant un attribut de type String représentant son numéro étudiant.
- Redéfinissez la méthode toString de la classe Etudiant afin qu'elle renvoie une chaine de la forme « Etudiant N° numero, nom, adresse, salaire »
- Testez votre classe

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.

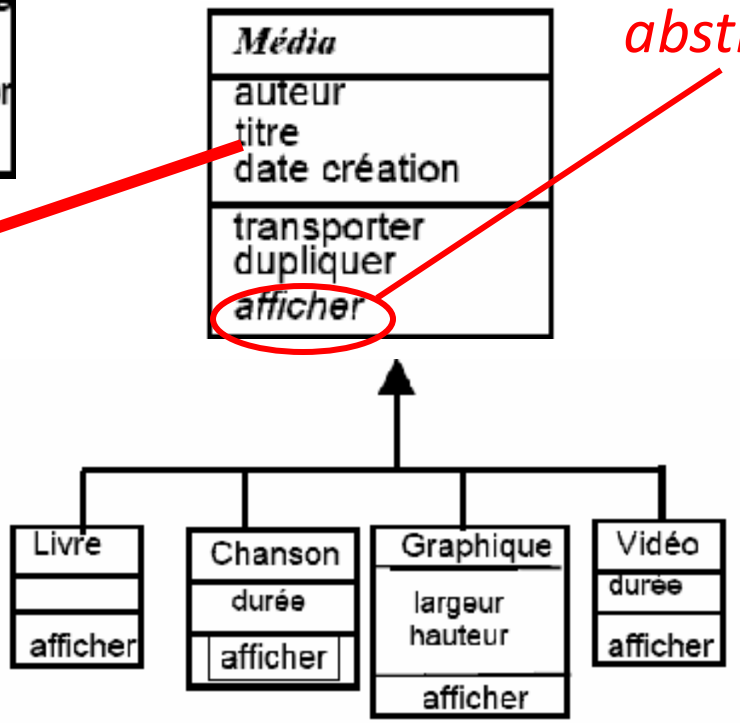


Classe abstraite

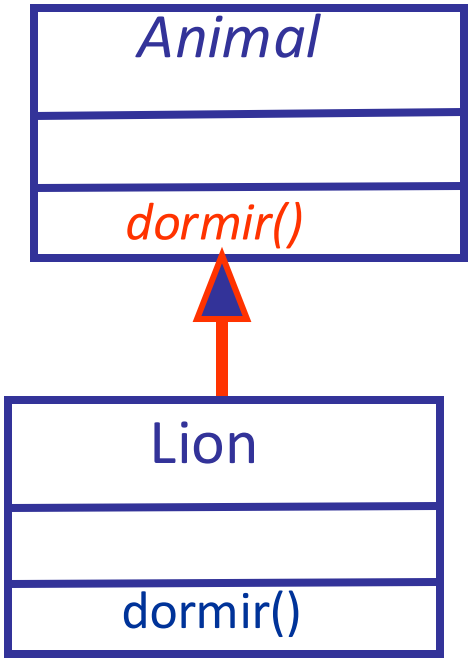
Un média peut être transporté, dupliqué, affiché.
Le transport et la duplication sont indépendantes du type du média (copie de fichiers).
Par contre, tout média peut être affiché et ce n'est pas la même chose pour l'audio, la vidéo, le graphisme, le texte.
Un média ne peut pas définir comment s'afficher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

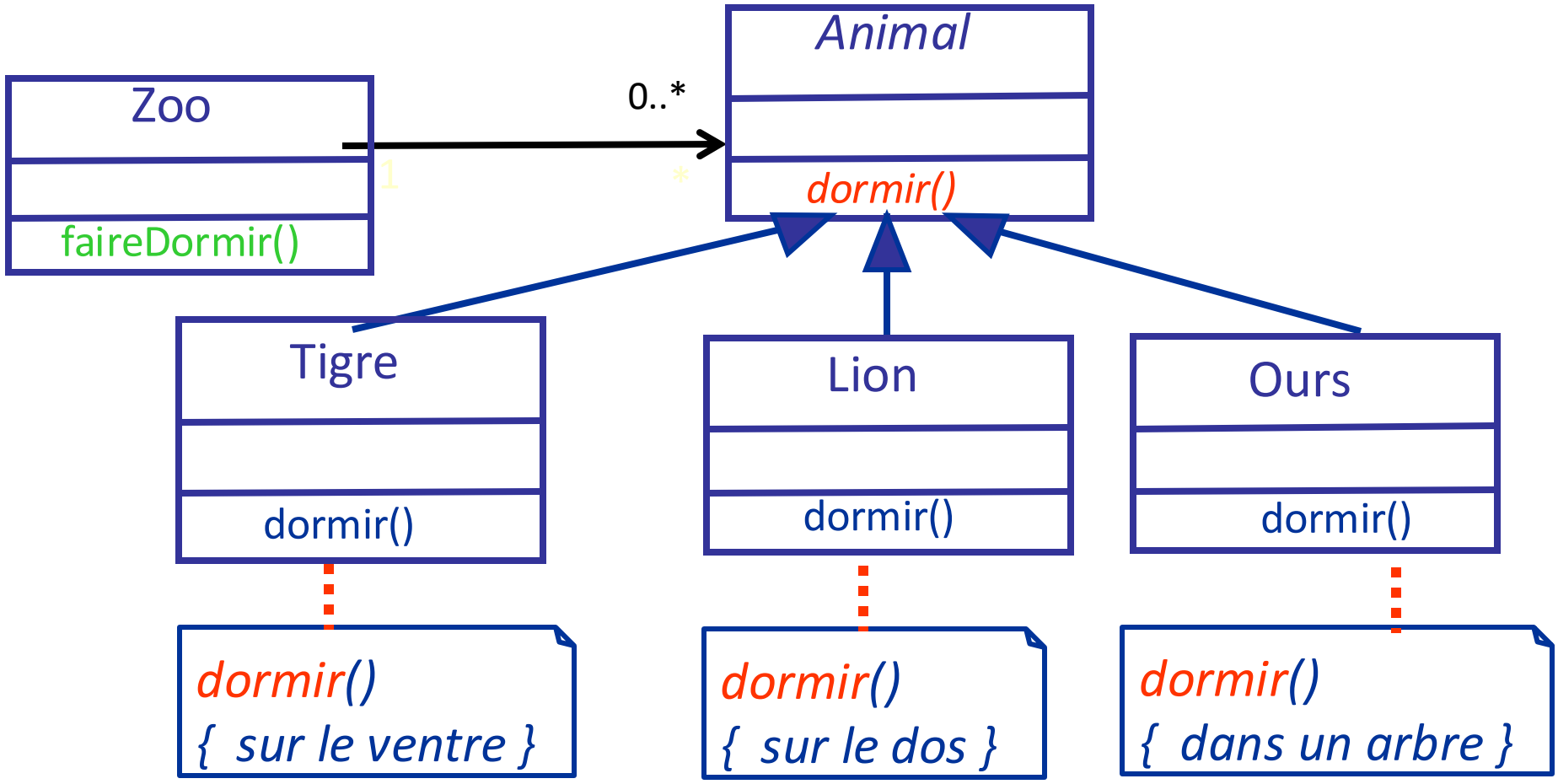



```
abstract class Animal
{
    abstract void dormir();
.....
}
```

- 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");
   }
}
```

Collection polymorphe

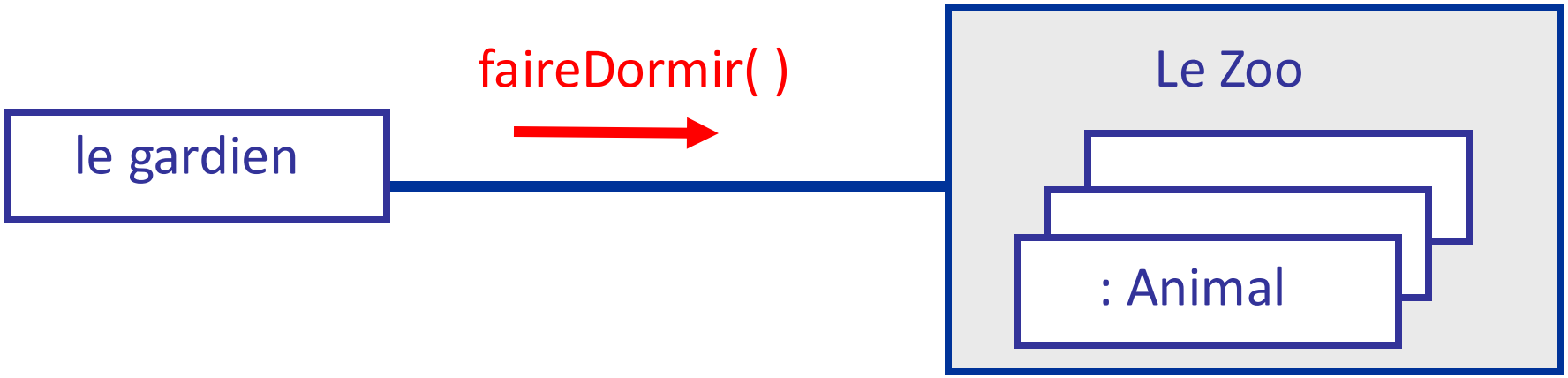
```
public class Zoo {
// Liste des animaux du zoo
private  ArrayList<Animal> animaux
                          =new ArrayList<Animal>() ;

// Constructeur de la classe Zoo
public Zoo (){
        // Instantiation des animaux
       Animal unLion = new Lion("Prince");
       Animal unTigre = new Tigre("Share Khan");
       Animal unOurs = new Ours("Petit Jean");
       // ajout de chaque animal dans la liste
       animaux.add(unLion);
       animaux.add(unTigre);
       animaux.add(unOurs);
}
```

Le polymorphisme

Collection polymorphe *Zoo*

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



Collection polymorphe

```
public class Zoo { //suite
// implémentation de la méthode
//faireDormir() de la classe Zoo

    public void faireDormir(){
        for (Animal e:animaux )
            e.dormir();
    }
}
```

Envoi de message à une Collection polymorphe

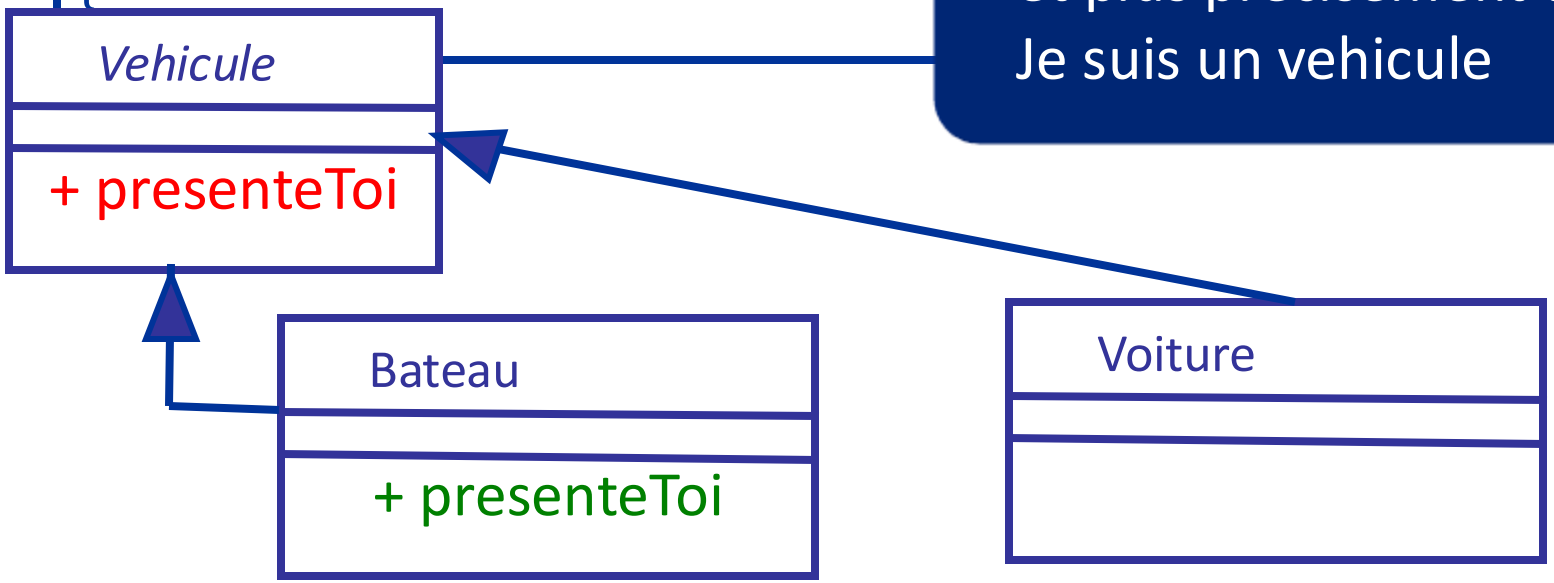
```
public class Gardien {
    public static void main (String arg[]){
    // Création du zoo
       Zoo leZoo = new Zoo();
      // Ordre de dormir à tous les animaux
       leZoo.faireDormir();
}
}
```

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

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
```

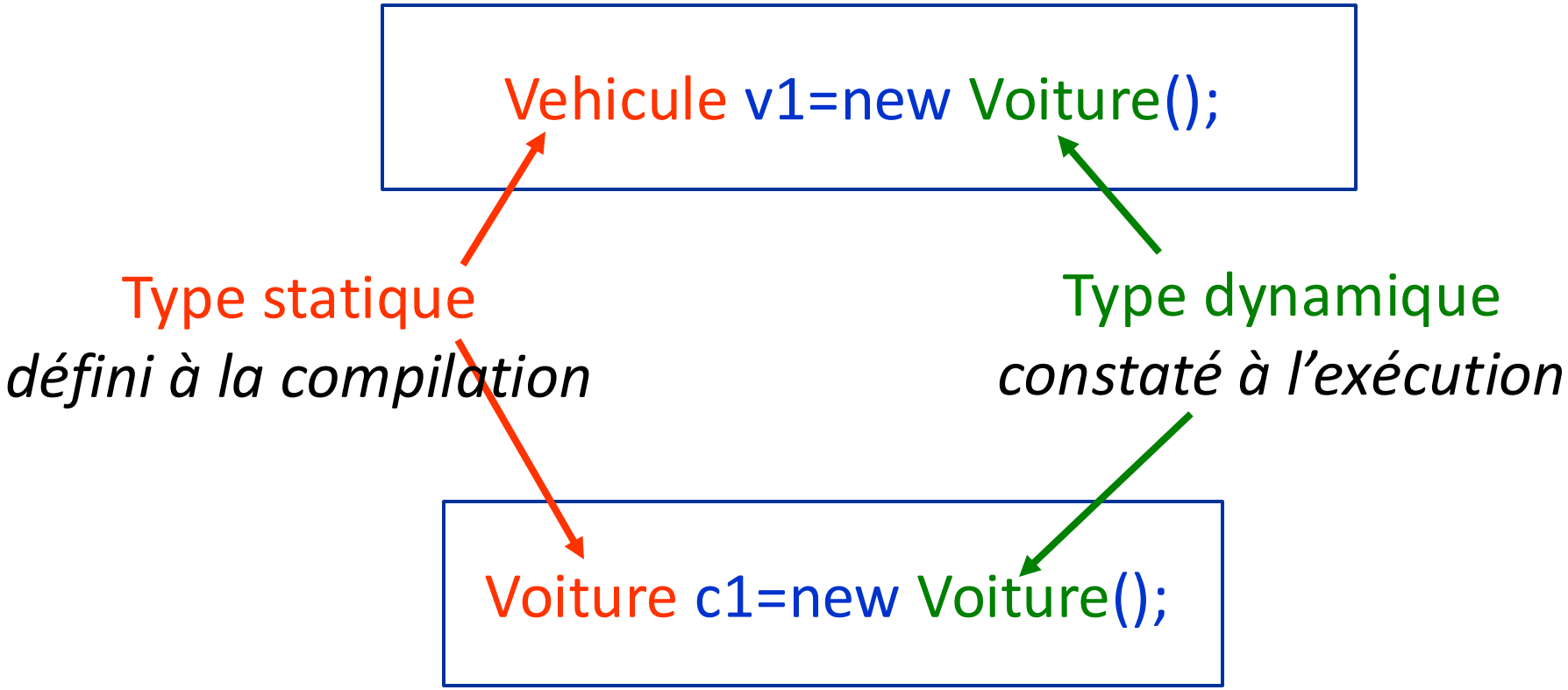
TPCours - Exercice (10)

- Transformer la classe Personne en classe abstraite et ajoutez lui une méthode abstraite
  - void sePresenter() .
- Définissez une classe Enseignant sous-classe de la classe Personne possédant un attribut de type int représentant son nombre d'heures.
  - La méthode sePresenter de la classe Enseignant doit afficher un message « Nom est un enseignant et il effectue nbHeures de cours »
- Ajoutez également une méthode sePresenter dans la classe Etudiant : elle doit afficher un message de la forme « Nom est un étudiant et son numero est … »
- Testez vos classes

TPCours - Exercice (11)

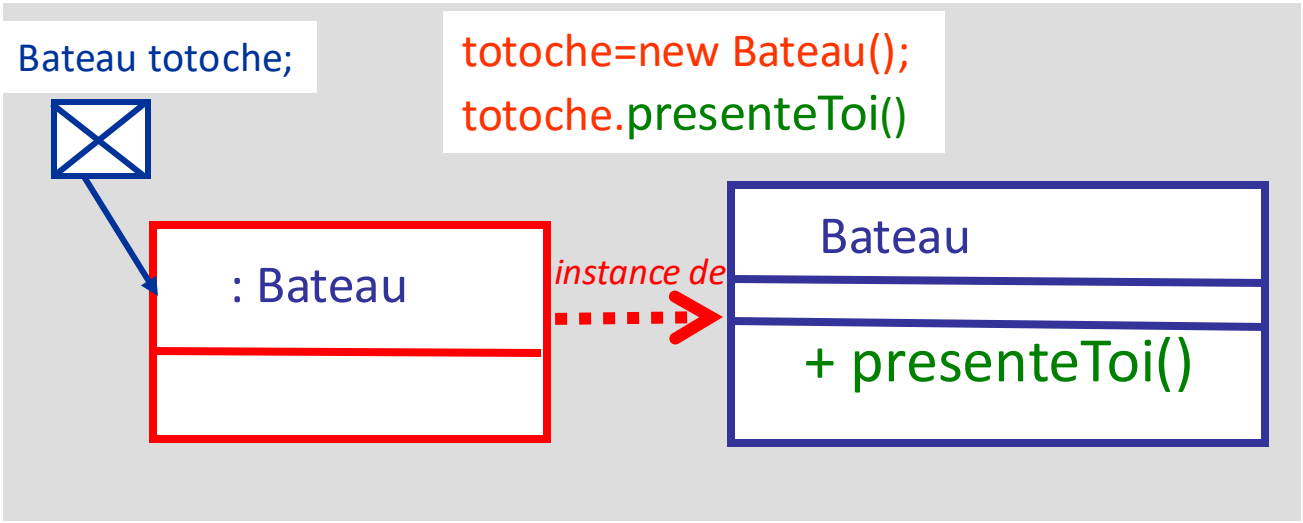
- Définissez une classe Université ayant comme attributs:
  - titre: String
  - membres: un ArrayList<Personne> de personnes pouvant être des étudiants et des enseignants
- Définissez les méthodes suivantes:
  - ajouterMembre(Personne p) qui ajoute une personne à mambres
  - afficherMembres() qui affiche la liste des membres de l'université
- Testez votre classe Université

Typage Statique et Dynamique



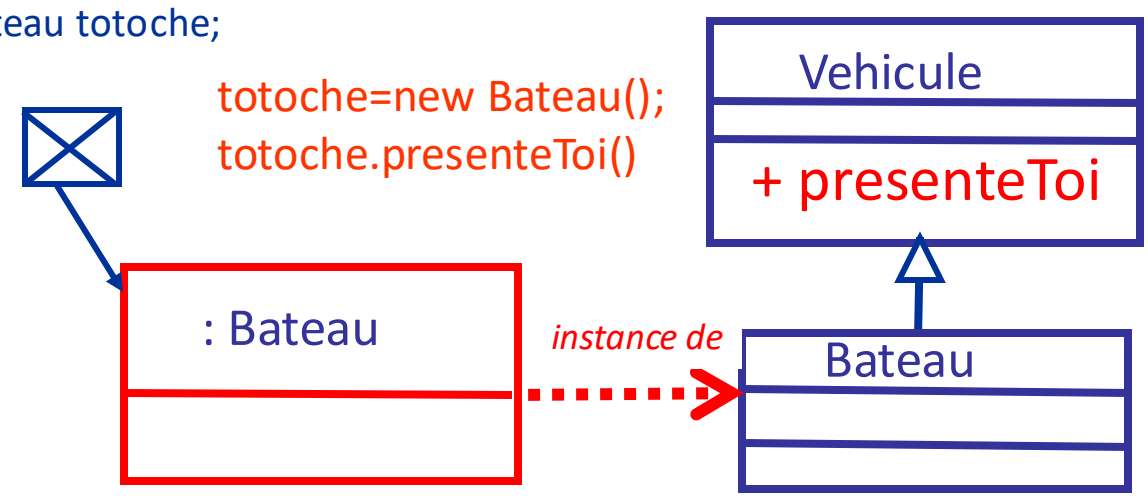
Le type dynamique détermine le point de départ de la recherche de la méthode à exécuter dans la hiérarchie d'héritage

Typage Statique et Dynamique



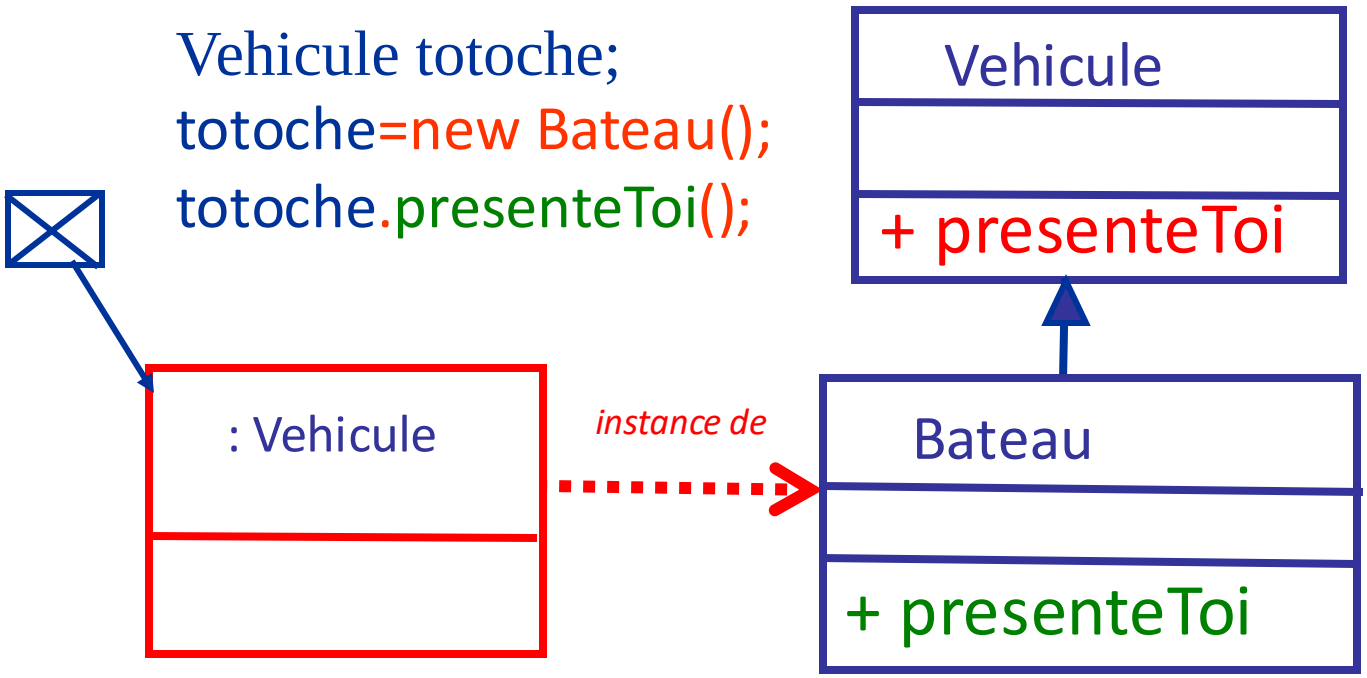
Pas d'héritage

Héritage mais
Pas de rédéfinition



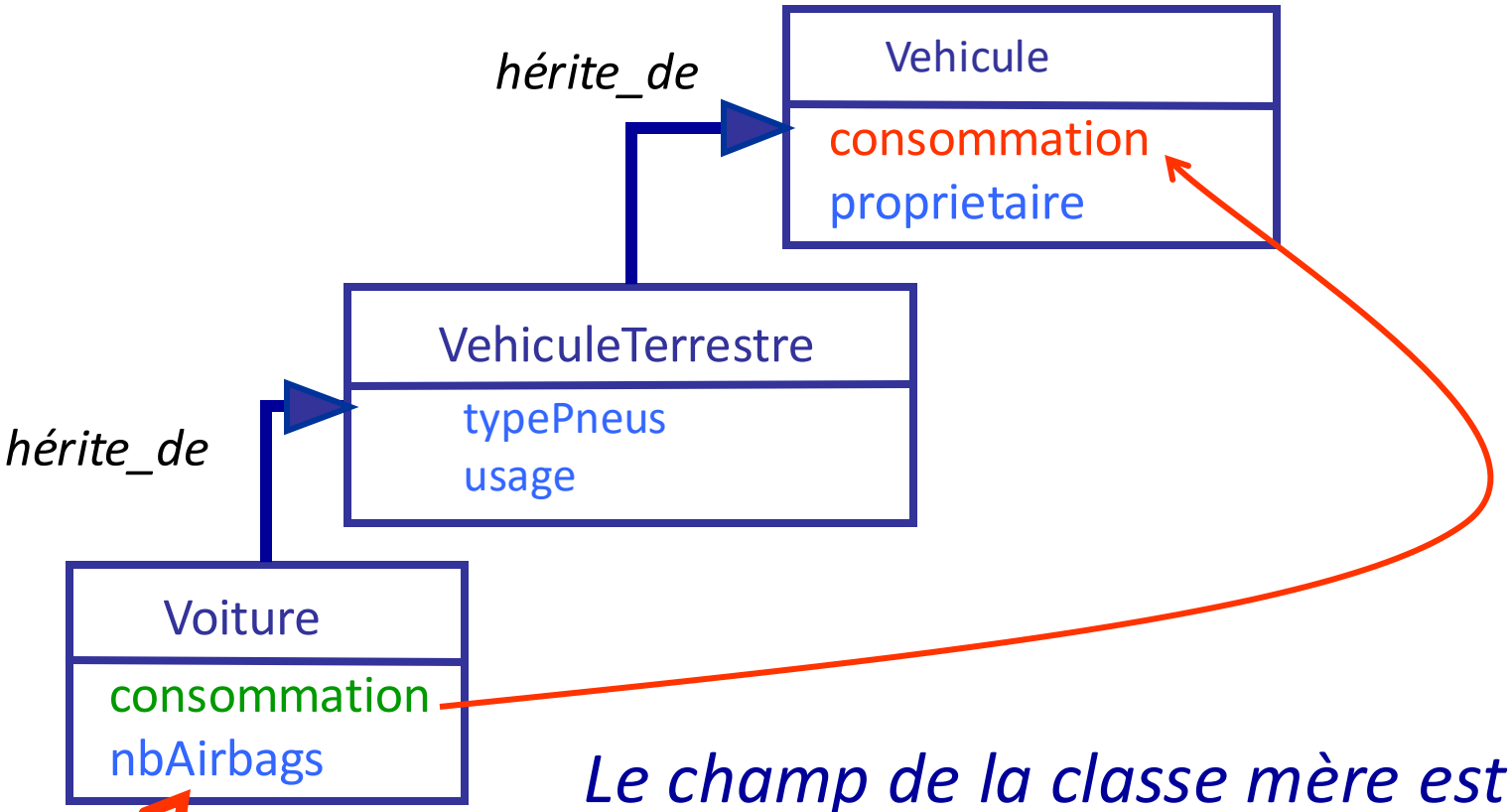
Typage Statique et Dynamique

Héritage + Rédéfinition : « dynamic binding »



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

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

Les interfaces en Java

Interfaces

- Déclare un ensemble de méthodes (sans implémentation) permettant de définir un comportement
- Peut intégrer des déclarations de constantes
- Peut être vu comme une classe ou toutes les méthodes sont abstraites : une classe purement abstraite…

Notion de « protocol » en Swift

Définition

- Toutes les déclarations de méthodes faîtes dans une interface sont par défaut abstraites et publiques les modificateurs `abstract public` sont optionnels
- Les déclarations de variables sont implicitement des constantes c'est-à-dire `public static final`

```
[modificateur] interface NomInterface{
    //Déclaration éventuelles de constantes
    type NOM_CONSTANTE;
    type nomMéthode(types paramètres);
}
```

Pourquoi les interfaces ?

- Permet dans une certaine mesure de « remplacer » l'héritage multiple
- Permet de se limiter à la définition d'un comportement
- Permet de définir un protocole de communication devant être suivi par les classes implémentant l'interface
- Ultérieurement les objets issus des classes implémentant l'interface peuvent être manipulés comme des instances de l'interface

Polymorphisme…

Utilisation

- On ne peut pas instancier une interface
  - Pas de new…
- Une classe peut implémenter une ou plusieurs interfaces (*implements*) et en même temps hériter d'une seule classe (*extends*)
- Une interface peut hériter d'une ou plusieurs interfaces (*extends*)

De même qu'une classe abstraite…

Implémenter une interface

- On utilise le mot clé `implements`

```
[modificateur] class NomClasse implements NomInterface{
    …
}
```

- La classe `NomClasse` doit alors implémenter toutes les méthodes de l'interface, sinon elle doit être déclarée `abstract`

Exemple d'utilisation

```
interface Insurable{
      void setRisk(String level);
      String getRisk();
}
public class Car extends Vehicle implements Insurable{
      private String risk = "Tier";
      private int seatNumber;
      …
      public void setRisk(String theRisk){
            …
            risk=theRisk;
      }
      public String getRisk(){return risk;}
      public void drive(…){
      …}
}
public class House implements Insurable{
      private String risk;
      private int roomNumber;
      …
      public void setRisk(String theRisk){
      …
      risk=theRisk;
      }
      public String getRisk(){return risk;}
      …
}
```

Tableau polymorphe…

```
public class InsuranceFirm{
  private Insurable[] InsuredProperties;
  …
  public void insure(Insurable i){
            …
            i.setRisk("…");
  }
  public double compensate(Insurable i,int damage){
            …
            i.getRisk();}
            …
            return …;
  }
  …
}
```

InsuranceFirm n'aura pas à être modifiée si l'on ajoute une nouvelle classe implémentant Insurable

Héritage d'interfaces

- Une interface peut hériter de plusieurs autres interfaces

```
interface NomInterfaceFille extends NomInterfaceMere1
                                    NomInterfaceMere2
                                    …
                                    NomInterfaceMereN{
    … //déclaration des méthodes propres à l'interface
      //fille
}
```

Exemple d'héritage

```
interface Monstre{
     void menace();
}
interface MonstreDangereux
          extends Monstre{
     void detruire();
}
interface Mortel {
     void tuer();
}
class Dragon
          implements MonstreDangereux{
     public void menace() {…}
     public void detruire() {…}
}
interface Vampire
          extends MonstreDangereux,
                  Mortel{
     void BoitSang();
}
```

```
class SpectacleHorreur{
     static void u(Monstre b){
          b.menace();}
     static void v(MonstreDangereux d){
          d.menace();
          d.detruire();
     }
```

Code extrait de ThinkInJava…

```
public static void main(String[] args){
     Dragon toto = new Dragon();
     SpectacleHorreur.u(toto);
     SpectacleHorreur.v(toto);
}
```

Grouper des constantes

- L'interface est un excellent moyen de grouper des constantes puisque par défaut les variables déclarées dans une interface sont `public`, `static` et `final`

```
public interface Mois {
       int    JANVIER = 1, FEVRIER = 2,
              MARS = 3,AVRIL = 4, MAI = 5,
              JUIN = 6, JUILLET = 7, AOUT = 8,
              SEPTEMBRE = 9, OCTOBRE = 10,
              NOVEMBRE = 11, DECEMBRE = 12;
}
```

Utilisation :
Mois.JANVIER

Initialiser les champs dans une interface

- Les champs déclarés dans une interface sont des constantes mais ils peuvent être initialisés avec des expressions non constantes (expressions)

```
import java.util.*;
public interface RandVals {
       int rint = (int)(Math.random()*10);
       long rlong = (long)(Math.random()*10);
       float rfloat = (float)(Math.random()*10);
       double rdouble = Math.random()*10;
}
```

Exemple de code

```
import java.util.*;
public interface RandVals {
       int rint = (int)(Math.random()*10);
       long rlong = (long)(Math.random()*10);
       float rfloat = (float)(Math.random()*10);
       double rdouble = Math.random()*10;
}
```

```
public class TestRandVals {
       public static void main(String[] args) {
              System.out.println(RandVals.rint);
              System.out.println(RandVals.rlong);
              System.out.println(RandVals.rfloat);
              System.out.println(RandVals.rdouble);
       }
}
```

Comparaison d’objets

Méthode equals

Rédéfinition des méthodes de la classe Object

- String toString()

  Renvoie une String décrivant l'objet. Par défaut, renvoie le type et l'adresse de stockage

- boolean equals(Object o)

  Compare les valeurs des champs mais compare uniquement les références des attributs de type objet.

- Object clone()

  Crée une copie de l'objet mais copie uniquement les références des attributs de type objet (*clonage de surface)*

La méthode equals()

- La méthode equals est héritée de la classe Object
- public boolean equals(Object obj)
  - renvoie vrai si obj a la même valeur que l'instance courante (this). C'est-à-dire si obj et this référencent le même objet

Comparaison des références et non comparaison des objets

- Si ce comportement par défaut ne vous convient pas vous devez redéfinir la méthode equals

Exemple de code : equals()

- Comportement par défaut

```
class Value {
        int i;
}
public class EqualsMethod2 {
        public static void main(String[] args) {
                Value v1 = new Value();
                Value v2 = new Value();
                v1.i = v2.i = 100;
                System.out.println(v1.equals(v2));
        }
}
```

Le test rend FAUX !!

Exemple de code : methode equals()

```
class Value {
        int i;
        public boolean equals(Object o) {
                boolean result = false;
                if ((o != null) && (o instanceof Value))
                        result =(i == ((Value)o).i);
                return result;
        }
}

public class EqualsMethod2 {
        public static void main(String[] args) {
                Value v1 = new Value();
                Value v2 = new Value();
                v1.i = v2.i = 100;
                System.out.println(v1.equals(v2));
        }
}
```

Le test rend VRAI !!

Méthode equals: exercice

```
class Truc {
        public int i ;
        public Truc(int a) { i = a ; }
        public Truc(Truc t) { i = t.i ; }
        public boolean equals( Truc t ) {return (t.i==i)
        public static void main(String args[]) {
                Truc y = new Truc(1);
                Truc z = y;
                Truc w = new Truc(y);
                if (z==y) System.out.println ( " 1 ") ;
                if (w==y) System.out.println ( " 2 ") ;
                if (z.equals(y)) System.out.println ( " 3 ") ;
                if (w.equals(y)) System.out.println ( " 4 ") ;
}
}
```

Déroulez ce code à la main et donnez sa sortie écran

1
3
4

S'assurer du type effectif d'une instance : l'opérateur instanceof

- Savoir si une variable fait référence à un objet d'une Classe donnée
- Opérateur booléen
- Syntaxe d'appel

```
référence instanceof NomClasse; // retourne un booléen
```

- Utilisation
  - Tester la nature d'un objet avant de décider ce qu'il convient d'en faire

A utiliser avec parcimonie!!
Contraire au polymorphisme

L'opérateur instanceof

- Exemples d'utilisation

```
Ellipse e = new Ellipse(2.0, 3.0) ;
Circle c = new Circle(4.0) ;
System.out.println(e instanceof Circle) ;
System.out.println(e instanceof Ellipse) ;
System.out.println(c instanceof Circle) ;
System.out.println(c instanceof Ellipse) ;
System.out.println(e instanceof Object) ;
```

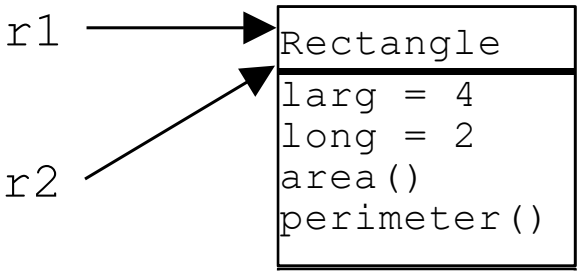
false
true
true
true
true

```
Salarie[] equipe ={…};
for (…)
    if (equipe[i] instanceof Chef)
     System.out.println(
        ((Chef)équipe[i]).getSubordonnés().length);
```

Clonage d’objets

Cloner les objets

- Il faut copier l'objet et pas seulement la référence…




```
Rectangle r1 = new Rectangle(2,4);
Rectangle r2 = r1;
if (r1==r2) …
```

- Il n'y a pas copie, duplication, il n'y a toujours qu'un seul objet
- Si on modifie r2, r1 l'est également

Cloner des objets

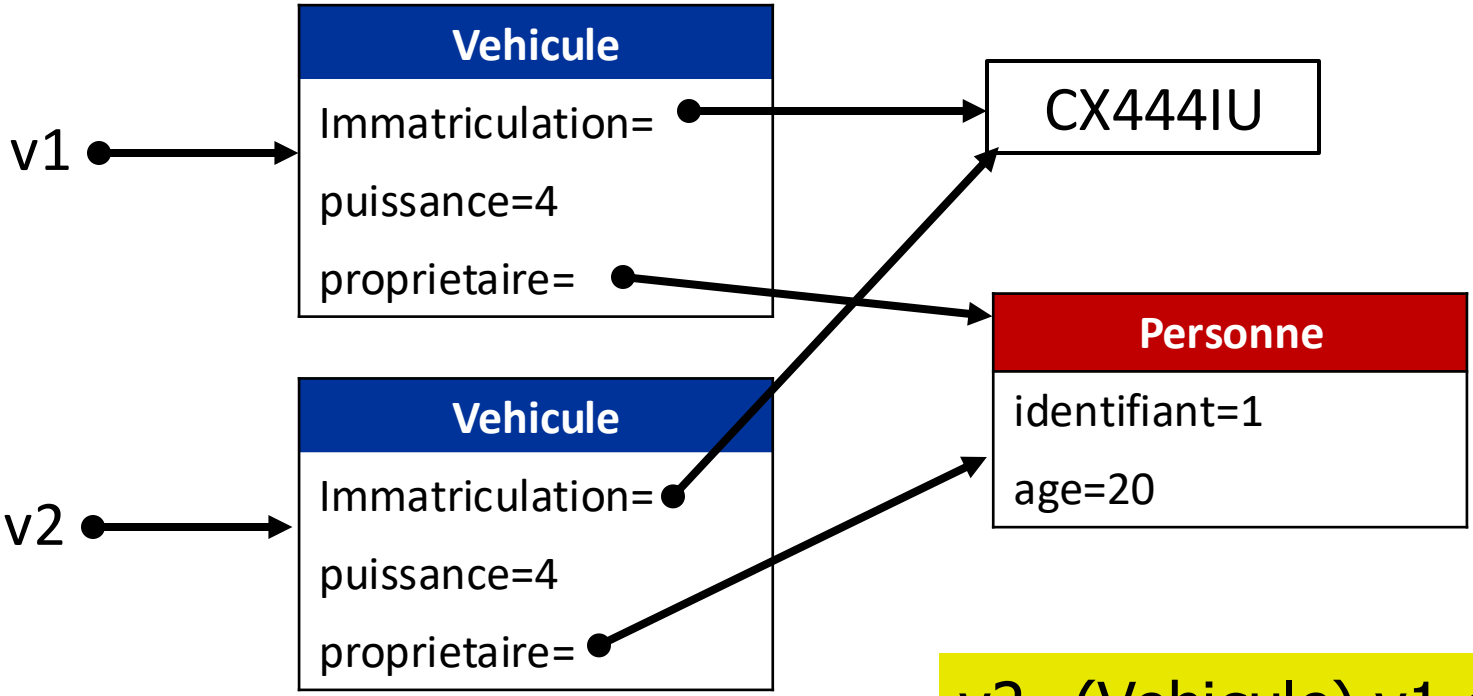
- Il peut être nécessaire de construire une véritable copie de l'objet on parle alors de clone
  - Conserver l'état initial d'un objet avant modification
  - Éviter de fournir à des objets extérieurs une référence à un objet sensible

Cloner des objets

- Les champs de type primitif sont toujours dupliqués
- Pour les champs de type référence (non immutables) on distingue deux types de clonage:
  - Surface ou superficiel, ou *shallow copy* : seules les références sont dupliquées
  - Profondeur ou *deep copy* : les objets référencés sont également clonés

Clonage de surface, shallow copy

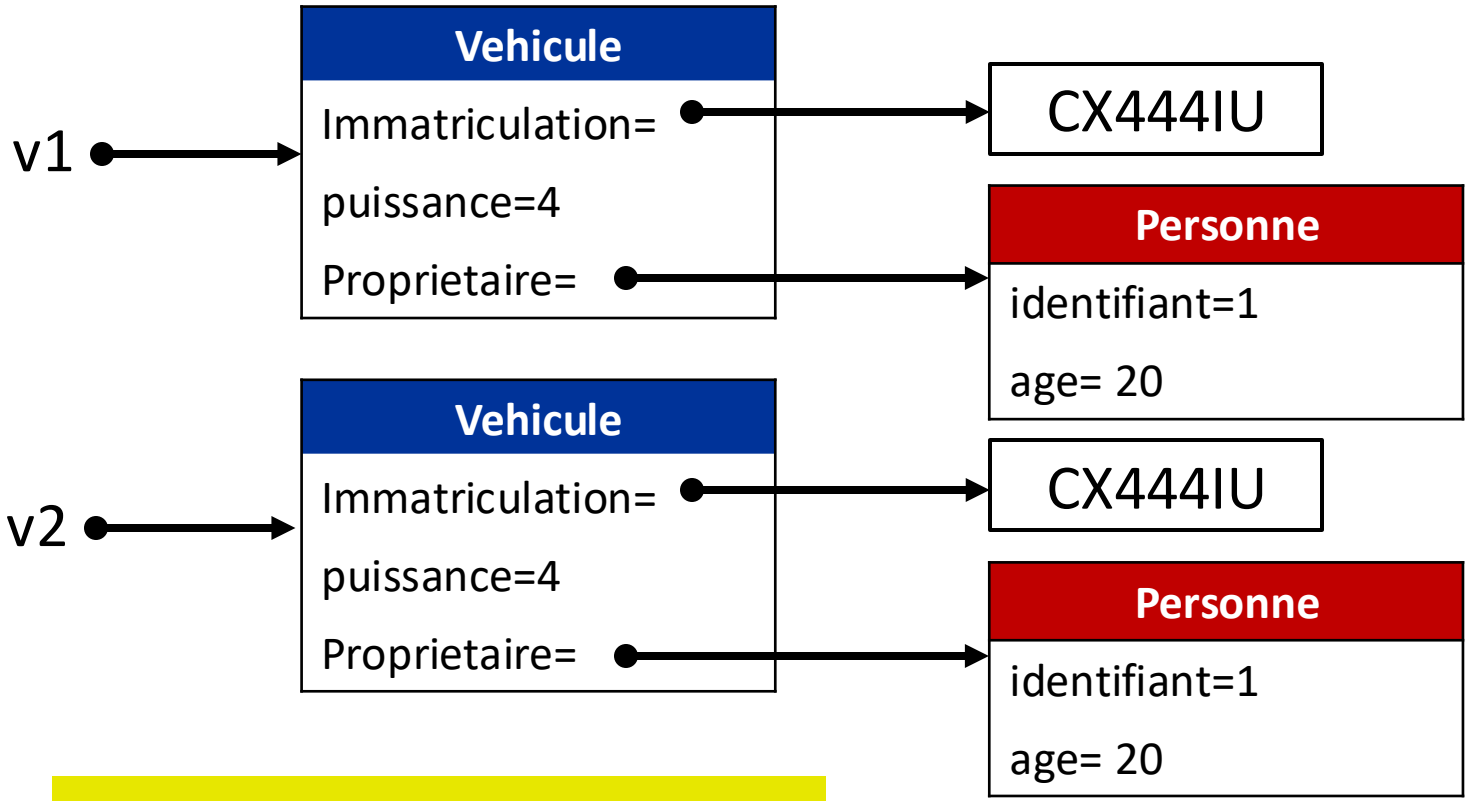
- Les références aux objets sont simplement copiées



v2=(Vehicule) v1.clone()

Clonage en profondeur

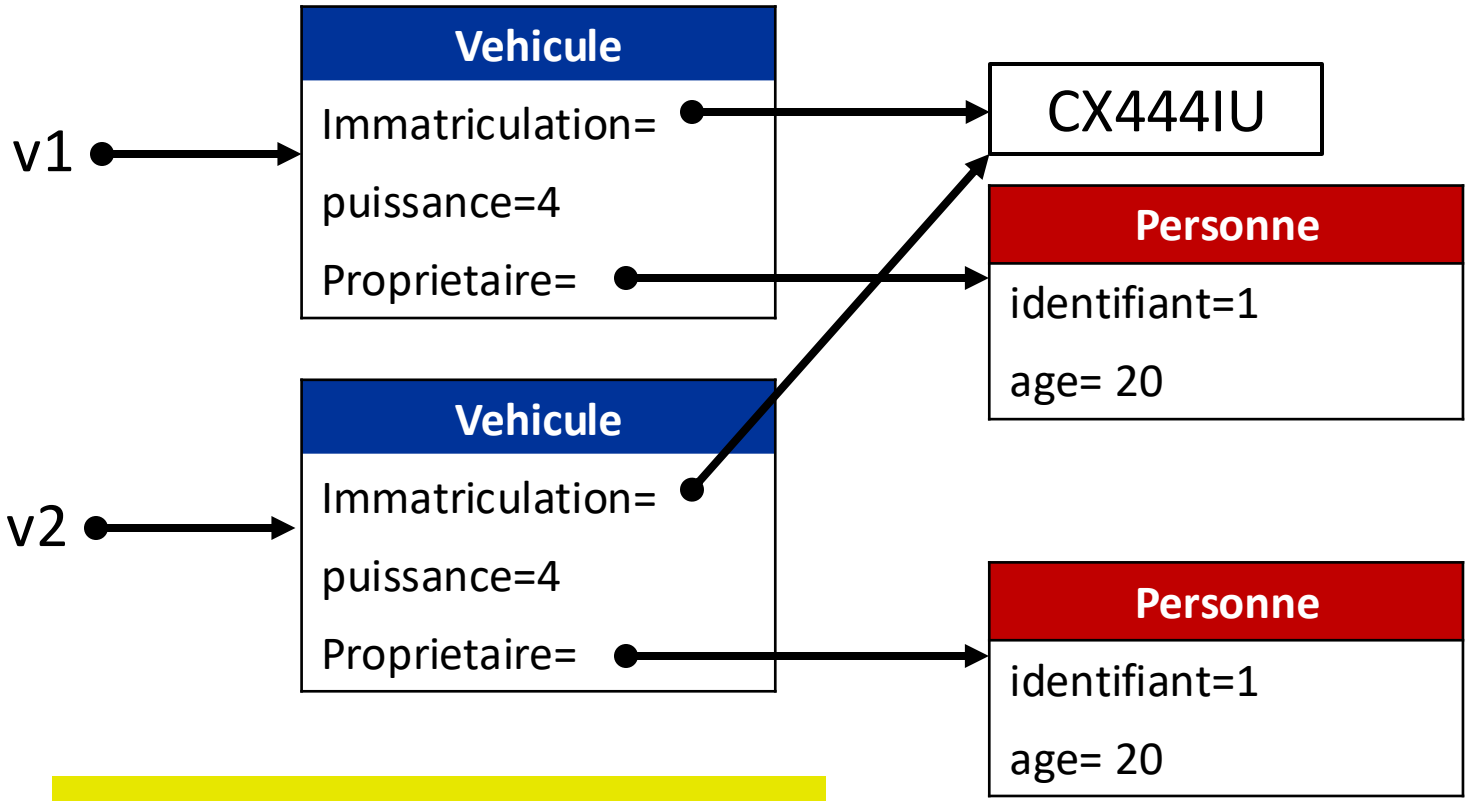
- Les objets référencés sont également clonés



v2=(Vehicule) v1.clone()

Cas particulier des String

- Comme les String en java sont immutables, ils n'ont pas à être clonés lors d'un clonage en profondeur



v2=(Vehicule) v1.clone()

Copier les objets, la méthode clone()

- Cette méthode héritée de la classe `Object` permet de faire une véritable copie de l'objet courant
- Cette méthode est `protected`, une routine extérieure à la classe ne peut l'appeler directement, il nous faut donc créer une sous-classe et la redéfinir
- Par défaut elle effectue un clonage de surface

```
protected Object clone() throws CloneNotSupportedException
```

Copier les objets, l'interface Cloneable

- Un objet ne peut être cloné que s'il est instance d'une classe implémentant l'interface `Cloneable`
- Dans le cas contraire la méthode clone de `Object` lève une `CloneNotSupportedException`
- Cette interface est vide, elle a seulement un rôle de marqueur pour indiquer si un objet peut être cloné ou non

```
…
if (obj instanceOf Cloneable)…
```

Comportement de clonage

- Vous avez trois possibilités
  - Décider que le clonage par défaut (clonage de surface) est approprié.
  - Adapter le clonage par défaut en appelant la méthode clone sur les instances rattachées à votre objet (clonage de profondeur)
  - Abandonner la possibilité de clonage, ne pas l'autoriser, donc ne pas implémenter l'interface `Cloneable`

Permettre le clonage

- Votre classe doit alors
  - Implémenter l'interface `Cloneable`
  - Redéfinir la méthode `clone` suivant le comportement souhaité (surface, profondeur)
  - Rendre `public` la méthode `clone()` (pour pouvoir l'invoquer depuis n'importe quelle classe)
  - propager ou gérer les `CloneNotSupportedException` si la copie profonde rencontre un objet non clonable

Méthode clone (clonage de surface)

```
import java.util.*;
class Vehicule implements Cloneable {
    String immatriculation;
    int puissance;
    Personne propriétaire
    …
    public Object clone() {
         Vehicule v = null;
         try {
                  v = (Vehicule) super.clone();
         } catch (CloneNotSupportedException e) {
                  System.out.println("Mon objet n'est pas clonable");
         }
         return v;
    }
}
```

Clonage de surface

Simple invocation de la méthode clone de la classe mère

Exemple de code: clonage de surface

```
import java.util.*;
class MyObject implements Cloneable {
   int i;
   MyObject(int ii) { i = ii; }
   public Object clone() {
        Object o = null;
        try {
                o = super.clone();
        } catch (CloneNotSupportedException e) {
                System.out.println( "Mon objet n'est pas clonable");
        }
        return o;
   }
   public String toString() {
        return Integer.toString(i);
   }
}
```

Copie locale

```
public class LocalCopy {
        static MyObject g(MyObject v) {
                v.i++;
                return v;
        }
        static MyObject f(MyObject v) {
                v = (MyObject)v.clone();
                v.i++;
                return v;
        }
        public static void main(String[] args) {
                MyObject a = new MyObject(11);
                MyObject b = g(a);
                if(a == b) System.out.println("a == b");
                else System.out.println("a != b");
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                MyObject c = new MyObject(47);
                MyObject d = f(c);
                if(c == d) System.out.println("c == d");
                else System.out.println("c != d");
                System.out.println("c = " + c);
                System.out.println("d = " + d);
        }
}
```

Exercice

Déroulez ce code à la main et affichez le résultat

```
a == b
a = 12
b = 12
c != d
c = 47
d =48
```

Méthode clone: clonage profond

```
import java.util.*;
class Vehicule implements Cloneable {
    String immatriculation;
    int puissance;
    Personne proprietaire
    …
    public Object clone() {
         Vehicule v = null;
         try {
                  v = (Vehicule) super.clone();
                  v.proprietaire= (Personne) proprietaire.clone();
         } catch (CloneNotSupportedException e) {
                  System.out.println("MyObject can't clone");
         }
         return v;
    }
}
```

Création d'un nouvel objet
Et clonage de ses attributs objets non immutables

La classe Personne doit aussi être clonable

Exemple de code de clonage profond

```
public class Stack implements Cloneable {
        private Vector items; // code for Stack's methods and
                              // constructor not shown protected
        public Object clone() {
                try {
                        // clone the stack
                        Stack s = (Stack)super.clone();
                        // clone the vector
                        s.items = (Vector)items.clone();
                        return s; // return the clone
                } catch (CloneNotSupportedException e) {
                // this shouldn't happen because Stack is
                // Cloneable
                        throw new InternalError();
                }
        }
}
```

Le cast est obligatoire car la méthode clone renvoie un Object

Création d'un nouvel objet Et clonage de ses attributs

Remarque sur le clonage

- Pendant l'opération de clonage, la méthode `clone()` n'appelle aucun constructeur
- `clone()` crée une copie de l'objet plus rapidement que ne le ferait un appel à new et à un constructeur
- La méthode `clone()` retourne toujours un `Object`. Vous devez donc forcer son type de retour

Cast obligatoire lors de l'invocation

Refuser le Clonage

- Une classe permet le clonage
- Vous dérivez cette classe, et vous ne voulez pas que votre classe dérivée accepte le clonage
- Redéfinissez la méthode clone et faites lui lever une **`CloneNotSupportedException`**

```
class Subclass extends SuperClass
{
   protected Object clone () throws CloneNotSupportedException
   {
      throw new CloneNotSupportedException ();
   }
}
```

Exercice Clonage

```
public class Personne implements Cloneable{
    String nom;
    int age;
    Voiture maVoiture;
    int[] salaires;
    public Personne(String nom, int age, Voiture maVoiture) {
        this.nom=nom;
        this.age=age;
        this.maVoiture=maVoiture;
        salaires= new int[12];
        salaires[0]=1500;
   }
   public Object clone() {
     Object o = null;
     try {
       o = super.clone();
     }catch (CloneNotSupportedException e) {
       System.out.println("clonage impossible");
     }
     return o;
    }
}
```

Clonage de surface

```
public class Voiture{
        String immat;
        public Voiture(String immat) {
            this.immat=immat;}
}
```

Clonage de surface

Exercice

```
Class TestClone {
public static void main(String args[]) {
  Voiture v=new Voiture("CX555XR");
  Personne p=new Personne("Toto",20,v);
  Personne pclone=(Personne) p.clone();
  //AFFICHAGE de p et pclone (nom-age-immatriculation-salaire janvier)
  //MODIFICATIONS DE P: changement nom, immatriculation, salaire janvier
  p.nom="Titi"; p.age=21;
  p.maVoiture.immat="CH666TY";
  p.salaires[0]=2000;
  //AFFICHAGE de p et pclone (nom-age-immatriculation-salaire janvier)
  //MODIFICATIONS DE P: nouvelle voiture, réinitialisation du tableau salaires
  Voiture nouvVoiture=new Voiture("CH777TY");
  pclone.maVoiture=nouvVoiture;
  p.salaires=new int[12];
  p.salaires[0]=2500;
 //AFFICHAGE de p et pclone (nom-age-immatriculation-salaire janvier)
}
}
```

AVANT MODIFICATIONS
Objet p = Toto -20 - CX555XR- 1500
Objet pclone = Toto - 20 - CX555XR- 1500
APRES MODIFICATIONS N°1
Objet p = **Titi – 21** - **CH666TY- 2000**
Objet pclone = **Toto – 20** - **CH666TY- 2000**
APRES MODIFICATIONS N°2
Objet p = **Titi – 21 - CH666TY- 2500**
Objet pclone = **Toto – 20 - CH777TY- 2000**

Affichage écran?

Exercice (suite)

Clonage profond

```
public class Personne implements Cloneable{
    String nom;
    int age;
    Voiture maVoiture;
    int[] salaires;//salaires de chaque mois
    public Personne(String nom, int age, Voiture maVoiture) {
        this.nom=nom;
        this.age=age;
        this.maVoiture=maVoiture;
        salaires= new int[12];
        salaires[0]=1500;}
   public Object clone() {
   try {
      Personne s = (Personne)super.clone();
      s.salaires=(int [])salaires.clone();
      s.maVoiture = (Voiture)maVoiture.clone();
      return s;
    } catch (Clone
       // ne devrait
     throw new Int
}
}
}
```

```
public class Voiture implements Cloneable{
        String immat;
        public Voiture(String immat) {
            this.immat=immat;}
 public Object clone() {
     Object o = null;
     try { o = super.clone();
     }catch (CloneNotSupportedException e) {
       System.out.println("clonage impossible");}
     return o;}
}
```

Clonage profond

Exercice (suite)

```
AVANT MODIFICATIONS
Objet p = Toto-20-CX555XR- 1500
Objet pclone = Toto-20-CX555XR- 1500
APRES MODIFICATIONS N°1
Objet p = Titi-21-CH666TY- 2000
Objet pclone = Toto-20-CX555XR- 1500
APRES MODIFICATIONS N°2
Objet p = Titi - 21 - CH666TY - 2500
Objet pclone = Toto - 20 - CH777TY - 1500
```

```
Class TestClone {
public static void main(String args[]) {
  Voiture v=new Voiture("CX555XR");
  Personne p=new Personne("Toto",20,v);
  Personne pclone=(Personne) p.clone();
  //AFFICHAGE de p et pclone (nom-age-immatriculation-salaire janvier)
  //MODIFICATIONS DE P: changement nom, immatriculation, salaire janvier
  p.nom="Titi"; p.age=21;
  p.maVoiture.immat="CH666TY";
  p.salaires[0]=2000;
  //AFFICHAGE de p et pclone (nom-age-immatriculation-salaire janvier)
  //MODIFICATIONS DE P: nouvelle voiture, réinitialisation du tableau salaires
  Voiture nouvVoiture=new Voiture("CH777TY");
  pclone.maVoiture=nouvVoiture;
  p.salaires=new int[12];
  p.salaires[0]=2500;
 //AFFICHAGE de p et pclone (nom-age-immatriculation-salaire janvier)
}
}
```

Affichage écran?