Ecole Supérieure
d'Informatique et du Numérique
COLLEGE OF ENGINEERING & ARCHITECTURE

Chap 4: Héritage et polymorphisme

Java™

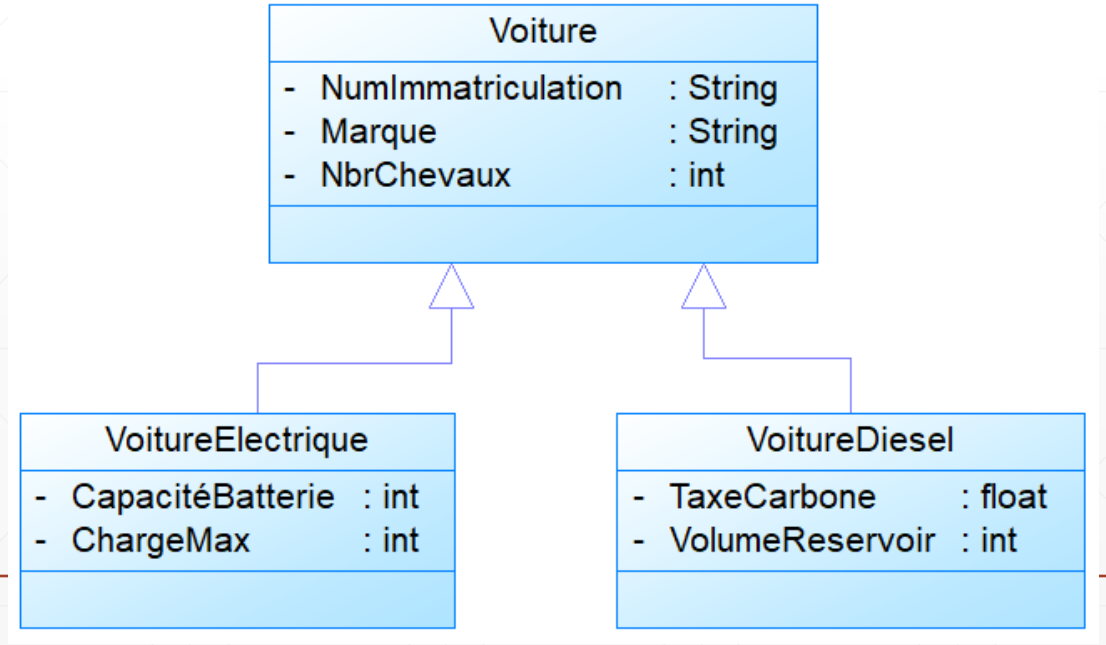
Sous-titre

Objectifs

- Les trois premiers chapitres que nous avons étudiés nous ont permis de découvrir les concepts de base et de s'initier dans la programmation en JAVA.
- Dans ce chapitre nous nous focaliserons :
  - Héritage
  - Polymorphisme
  - Redéfinition des méthodes, constructeurs
  - Conversion des objets

L’héritage

- L’héritage permet d’optimiser notre code par la réutilisation des classes existantes pour définir de nouvelles fonctionnalités
- **Exemple introductif**: la figure suivante définit une relation entre les classes voitures



L’héritage

- Exemple de la classe Voiture

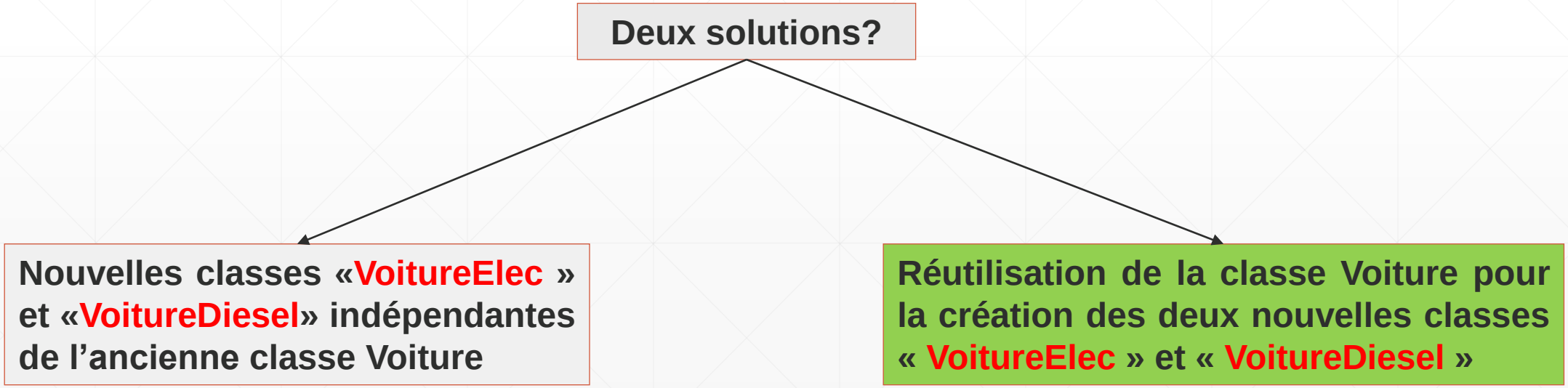
```
public class Voiture {
    public String numImmatriculation;
    public String marque;
    public int NbrChevaux;

    public Voiture() {
    }

    public String toString() {
        return "VoitureElectrique [numImmatriculation=" + numImmatriculation + ", marque=" +
        marque + ", NbrChevaux=«  + NbrChevaux + "]";
        }
}
```

L’héritage

- Nous souhaitons créer deux nouvelles classes pour créer des objets qui représentent les voitures électriques et les voitures Diesel



Héritage

- **Première solution: sans utilisation de l'héritage (Cas voitureElectrique)**

```
public class VoitureElectrique {
    public String numImmatriculation;
    public String marque;
    public int NbrChevaux;
    public int capacitebatterie;
    public int chargeMax;

    public VoitureElectrique() {
    }

    public String toString() {
        return "VoitureElectrique [numImmatriculation=" + numImmatriculation + ", marque=" + marque +
        ", NbrChevaux="
        + NbrChevaux + ", capacitebatterie=" + capacitebatterie + ", chargeMax=" + chargeMax + "]";
    }
}
```

Héritage

- La deuxième solution basée sur l'utilisation de l'héritage consiste en la création d'une classe fille « VoitureElectrique » qui étend (hérite) la classe Voiture.
- Pour définir cette relation d'héritage il faut utiliser le mot réservé extends

```
Public class ClasseDérivée extends ClasseBase{

}
```

On utilise aussi les termes suivants

- classe fille (dérivée): classe « **VoitureElectrique** », « **classeVoitureDiesel** »
- classe Mère (de base): classe « **Voiture** »
- Relation de **généralisation** est utilisée pour qualifier la relation entre la classe fille et la classe mère

Héritage

- Deuxième solution: utilisation de l'héritage pour la réutilisation de la partie déclarée dans la classe Voiture

```
public class VoitureElectrique extends Voiture {
    public int capacitebatterie;
    public int chargeMax;

    public VoitureElectrique() {
        super();
    }

    public String toString() {
        return "VoitureElectrique [capacitebatterie=" + capacitebatterie + ",
        chargeMax=" + chargeMax + ", numImmatriculation=" + numImmatriculation + ",
        marque=" + marque + ", NbrChevaux=" + NbrChevaux + "]";
    }
}
```

Héritage

- L’atout principal de l’héritage et la réutilisation des méthodes et des attributs déclarés dans la classe mère au niveau des classes filles.

- Les objets de la classe **VoitureElectrique** peuvent :
  - Utiliser les méthodes publiques et privées déclarées au niveau de la classe **VoitureElectrique**
  - Utiliser les méthodes et les attributs public déclarés dans la classe **Voiture**

Un objet d’une classe dérivée accède aux membres publics de sa classe de base

Visibilité depuis la classe dérivée

- Méthode publiques de la classe mère
  - visibilité généralisée
  - mot clé : **public**

- Méthodes privées de la classe mère
  - nonvisible dans les classes dérivées
  - mot clé : **private**

- Méthode protégée de la classe mère
  - Visibilité limité aux classes dérivées
  - mot clé : **private protected** !! Attention Obsolète

Redéfinition des méthodes et des attributs

- Surcharge de méthodes
  - Nouvelle définition des méthodes dans les classes dérivées
- concept de "is a", "est un"
  - héritage et surcharge de méthodes
- concept de "is like a", " est comme un"
  - héritage, surcharge et ajout de méthodes

Redéfinition des méthodes

- L'héritage permet de redéfinir les méthodes dans les classes dérivées en respectant la même définition de la classes mère.

```
//déclaration dans la classe mère voiture
public void afficher() {
        System.out.println("Je suis une voiture");
}
```

```
//déclaration dans la classe fille
public void afficher() {
        System.out.println("Je suis une voiture Electrique");
}
```

Activation des méthodes de la classe de base (surcharge)

- Ajout du comportement à la définition initiale et non la masquer (surcharger)
  - il n'est pas nécessaire de dupliquer un comportement. Appeler la méthode initiale dans le corps de la nouvelle et lui ajouter le comportement désiré suffit

- L'appel de la méthode initiale utilise le mot clé **super** qui transmet l'appel à la hiérarchie des classes

```
super.attribut
super.méthod(paramètres)
```

Activation des méthodes de la classe de base (surcharge)

- Exemple de la surcharge de la méthode afficher dans la classe dérivée « VoitureElectrique »,

```
public void afficher() {
    super.afficher();
    System.out.println("Je suis une voiture Electrique");
}
```

- Le super permet d'accéder aux méthodes déclarées dans la classe mère
- **Super.afficher()** permet d'appeler le traitement de la méthode afficher programmer dans la classe Voiture

Activation des méthodes de la classe de base (surcharge)

- Exemple de la méthode toString

```
// classe voitureElectrique
public String toString() {
    return super.toString() + " capacite = " + this.capacitebatterie
                            + " chargeMax " + this.chargeMax;
    }
```

➢ Le **super.toString()** fait appel à la méthode **toString** déclarée dans la classe de base (Voiture)

La surcharge des constructeurs

- La surcharge des constructeurs se fait par l'utilisation du mot réservé super sans faire appel au nom du constructeur

```
public class Voiture {
    public String numImmatriculation;
    public String marque;
    public int nbrChevaux;

    public Voiture(String numImmatriculation, String marque, int nbrChevaux) {
        this.numImmatriculation = numImmatriculation;
        this.marque = marque;
        this.nbrChevaux = nbrChevaux;
    }
}
```

La surcharge des constructeurs

- Exemple de constructeur

```
public class VoitureElectrique extends Voiture {
    public int capacitebatterie;
    public int chargeMax;

    public VoitureElectrique(String numImmatriculation, String marque, int nbrChevaux,
                                        int capacitebatterie,int chargeMax) {

        super(numImmatriculation, marque, nbrChevaux);
        this.capacitebatterie = capacitebatterie;
        this.chargeMax = chargeMax;
    }
}
```

Activation des méthodes de la classe de base (surcharge)

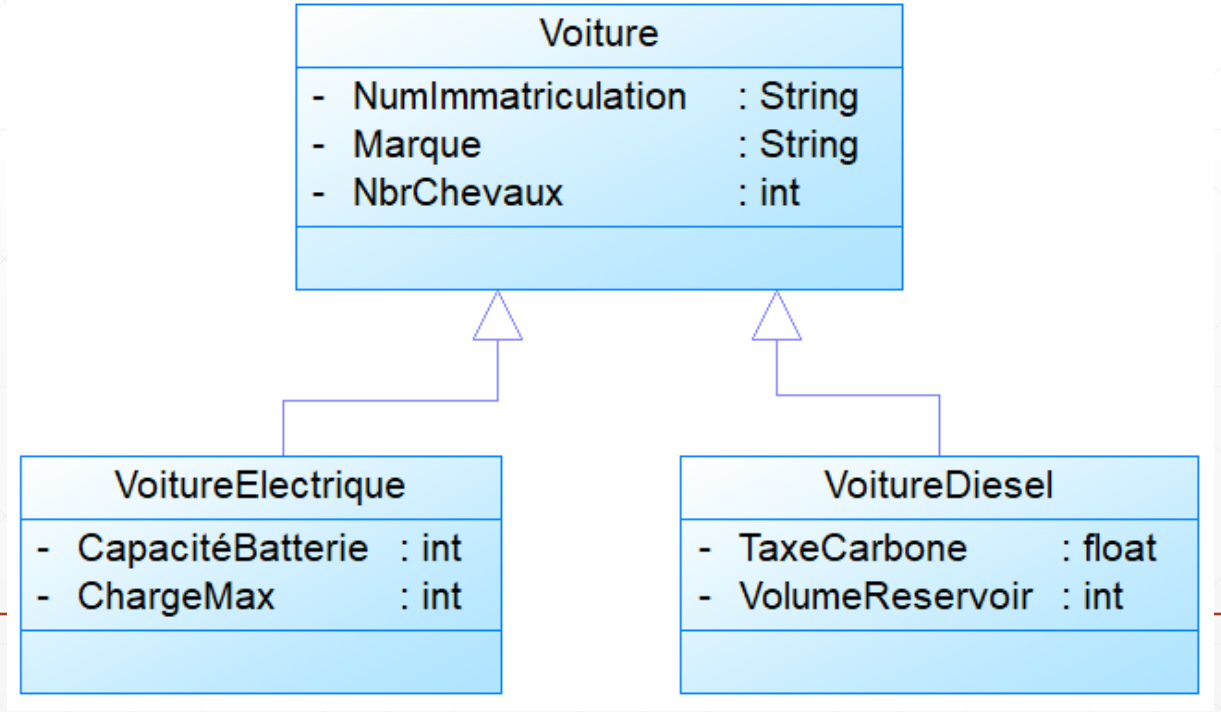
- On peut imposer la non-redéfinition d'une méthode au niveau des classes filles en utilisant **final**

```
// La classe mère / de base
public final void helloClient() {
        System.out.println("Hello je suis une superbe voiture");
}
```

```
// la classe fille
@Override
public void helloClient() {
        System.out.println("Hello je suis une superbe voiture Électrique");
}
```

Exercice d’application

- Donnez le code de la classe VoitureDiesel
- Donnez le code nécessaire pour la création d’une classe principale qui instancie et affiches les attributs d’un objet de type Voiture, VoitureDiesel, et VoitureElectrique,



Polymorphisme

- Le polymorphisme est un mécanisme qui facilite la modification du comportement des classes dérivée par rapport au comportement défini dans la classe mère.

```
On peut reprendre les traitements déclarés dans la classe mère et les
adaptés légèrement pour répondre aux spécificités des classes filles
```

Remarque : principe du ouvert/fermé

- **Classe ouverte à l'extension** : redéfinir des méthodes et réutilisée les traitements de base pour les adaptés aux besoins spécifiques des classes filles
- **Classe fermée à la modification** : redéfinition des méthodes pour implémentation de nouveau traitement

Comparaison des objets

- Deux objets sont égaux si tous leurs attributs sont égaux .
- La comparaison de deux objets passe par la spécification de la méthode **equals**.
- La comparaison des objets est basée sur la comparaison de leurs attributs

```
public boolean equals(VoitureElectrique ve) {
    return (this.capacitebatterie == ve.capacitebatterie &&
            this.chargeMax == ve.chargeMax);
}
```

Comparaison des objets

- Exemple de comparaison de deux objets

```
public static void main(String[] args) {
    VoitureElectrique ve1 = new VoitureElectrique("123 B 1", "Honda", 12, 5000, 500);
    VoitureElectrique ve2 = new VoitureElectrique("76 B 26", "Toyota", 8, 5000, 500);

    if(ve1.equals(ve2)) {
        System.out.println("Voiture 1 == voiture 2");
    }else {
        System.out.println("Voiture 1 != voiture 1");
    }
}
```

Conversion des objets

- La conversion des objets est possible en JAVA entre les objets qui sont liés par une relation des généralisation (Héritage)

```
VoitureElectrique ve1 = new VoitureElectrique("123 B 1", "Honda",
                        12, 5000, 500);
Voiture v1 = new Voiture("123 B 1", "Mercedes", 10);
v1 = ve1;
v1 = (Voiture)ve1;
ve1 = (VoitureElectrique)v1; // erreur
ve1 = v1; // erreur
```

- On peut affecter un objet d'une classe dérivée à un objet de la classe de base (Mère)
- L'inverse génère une erreur

Exercice d’application (Polymorphisme)

- Créez une méthode décrisToi() qui affiche les informations de chaque type de voiture.
- Créez des instances et ajouter ces objets dans un tableau
- Exécuter la méthode décris toi pour toutes les instances du tableau