BPOO - Sujet TD 8

Dut/Info-S2/M2103

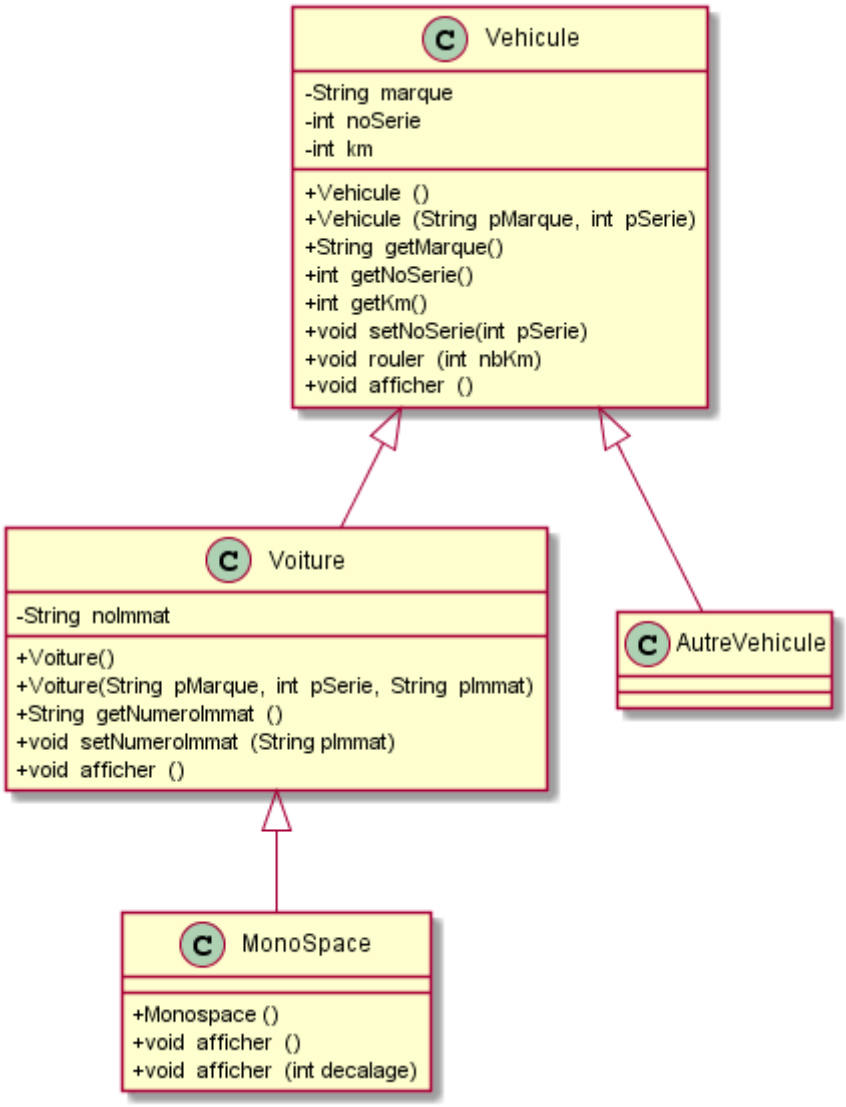
Table des matières

PreReq	Cours 4 : Principes de l'héritage et Héritage en Java. TPs sur héritage
ObjTD	**Comprendre l'héritage, l'héritage des méthodes - Classes abstraites - Polymorphisme - Interface.**
Durée	**1 séance** de 1.5h

1. Domaine d'étude

On va s'intéresser aux mécanismes d'héritage mis en oeuvre dans la hiérarchie de classes suivante.

Diagramme UML des classes mises en oeuvre (sans multiplicités)



Le code correspondant :

```
public class  Vehicule extends  Object   // Héritage implicite
{
   // Attributs
   private  String  marque;
   private  int  noSerie;
   private  int  km;

   // Constructeur
   public  Vehicule  () {
       this ("", 0);   // Appel  de Vehicule (String, int)
   }
   public  Vehicule  (String  pMarque,  int  pSerie) {
       super () ;  // Implicite : Appel de Object()
       this.marque  =  pMarque;
       this.noSerie  =  pSerie;
       this.km  =  0; // Implicite
   }

   // Méthodes
   public  String  getMarque()  {  return  this.marque;  }
   public  int  getNoSerie()  {  return  this.noSerie;  }
   public  int  getKm()  {  return  this.km;  }

   public  void  afficher  () {
       System.out.println ("Vehicule " + this.marque + " " + this.noSerie);
   }

   public  void  setNoSerie(int  pSerie)  {  this.noSerie  =  pSerie;  }
   public  void  rouler  (int  nbKm)  {  this.km  =  this.km  +  nbKm; }
} // Fin Classe Vehicule



public class  Voiture  extends  Vehicule
{
   // Attributs
   private  String  noImmat;

   // Constructeurs
   public  Voiture() {
       this  ("",  0,  ""); // Appel  de Voiture (String, int, String)
   }

   public  Voiture(String  pMarque,  int  pSerie,  String  pImmat) {
       super  (pMarque,  pSerie); // Appel  de Vehicule (String, int)
       this.noImmat  =  pImmat;
   }
```

```
    // Méthodes
    public String getNumeroImmat () { return this.noImmat; }
    public void setNumeroImmat (String pImmat) { this.noImmat = pImmat; }

    public void afficher () {
        System.out.println ("Immatriculation " + this.noImmat);
        super.afficher();
    }
} // Fin Classe Voiture

public class MonoSpace extends Voiture
{
    public Monospace () {
        super();
    }
    public void afficher () {
        super.afficher();
        System.out.println ("MonoSpace ");
    }
    public void afficher (int decalage) {
            int i;
            for (i=0; i<decalage; i++) System.out.print(" ");
            this.afficher();
    }
} // Fin Classe MonoSpace
```

2. Principe de l’héritage

Héritage :

- « Relation » établie entre classes d’objets.
- Relatif à la nature des objets.

Principe : mécanisme de transmission des **attributs et des méthodes** de la super-classe VERS sa sous-classe. Pas de transmission des constructeurs.

Exemple :

```
public class  Vehicule
{
   // ... Le code complet
}

public class  AutreVehicule  extends  Vehicule  {}

public class TestAutreVehicule
{
        public static void main(String[] args)
        {
                AutreVehicule av;

                av = new AutreVehicule (); // Constructeur par défaut créé
automatiquement

                av.setNoSerie(10);// setNoSerie() héritée
                             // => fait partie de l'interface des objets
AutreVehicule
                av.rouler (159); // Idem
                av.afficher ();
                System.out.println (av.getNoSerie());
                System.out.println (av.getKm());

        }
}
```

L'objet créé avec `av` contient bien tous les attributs déclarés dans la classe Vehicule dont hérite AutreVehicule.

Sur l'objet créé avec `av` on peut appliquer toutes les méthodes de la classe Vehicule dont hérite AutreVehicule.

---

3. Appel des constructeurs

3.1. Appel du constructeur de la super-classe

Dans une sous-classe :

- Constructeurs de la super-classe non hérités ⇒ problème : comment initialiser la partie des attributs hérités mais inaccessibles (private) ?
- Solution : les constructeurs de la classe dérivée invoqueront (exécuteront) un constructeur de la

classe de base.

Principes d'appels des constructeurs de la classe parente ou de la classe courante : **Obligatoirement la première ligne du code** :

- `super(param1, …);` ⇒ appel constructeur de la super-classe correspondant aux paramètres indiqués.
- `this(param1, …);` ⇒ appel autre constructeur de la classe courante correspondant aux paramètres indiqués.
- Autre instruction : ⇒ appel implicite `super()`. Revient à écrire explicitement : `super();`

3.2. Constructeur synthétisé

Rappel Règle :

1. Si aucun constructeur défini :
   - Il existe un constructeur par défaut créé par java (synthétisé).
   - Ce constructeur synthétisé ne fait rien de particulier mais **appelle** le constructeur par défaut de la super-classe (cf. ci-après).
2. Si au moins un constructeur est défini :
   - Cache le constructeur par défaut synthétisé.
   - Parfois ⇒ redéfinir le constructeur par défaut.

---

4. Appel des méthodes héritées

Définition : La **redéfinition** de méthode est le fait de redéfinir le corps d'une méthode héritée dans une sous-classe.

Règles : une définition de méthode est une redéfinition de méthode ssi

- Elle est héritée.
- Elle porte le même identificateur.
- Elle comporte les mêmes paramètres (en nombre, type et ordre).
- Elle a le même type de retour.

Une méthode redéfinie **cache** la méthode héritée à tous les clients.

Une méthode héritée peut :

- Définir un comportement complètement nouveau ⇒ écrire le code.
- Définir un comportement réutilisant le comportement hérité.* Compléter le comportement hérité :

Accès au comportement hérité : appel (invocation) du corps de la méthode héritée : * `super . idMethode (…);` * Interprété comme « exécuter le code de la méthode idMethode() telle qu'elle est définie dans la superclasse ».

Remarque : utiliser `this.methode(…);` ou `super.methode(…);` ?

- A priori `this.methode(…)` (majorité des cas)
- Sauf cas particulier : `super.methode (…)` lorsque
  - methode(…) redéfinie dans classe courante
  - **ET on veut le comportement de la classe parente** et pas la méthode locale

But d'utiliser `this.methode(…)` : ne pas "cacher" les redéfinitions ultérieures car +super.methode (…)+ les bloquerait …

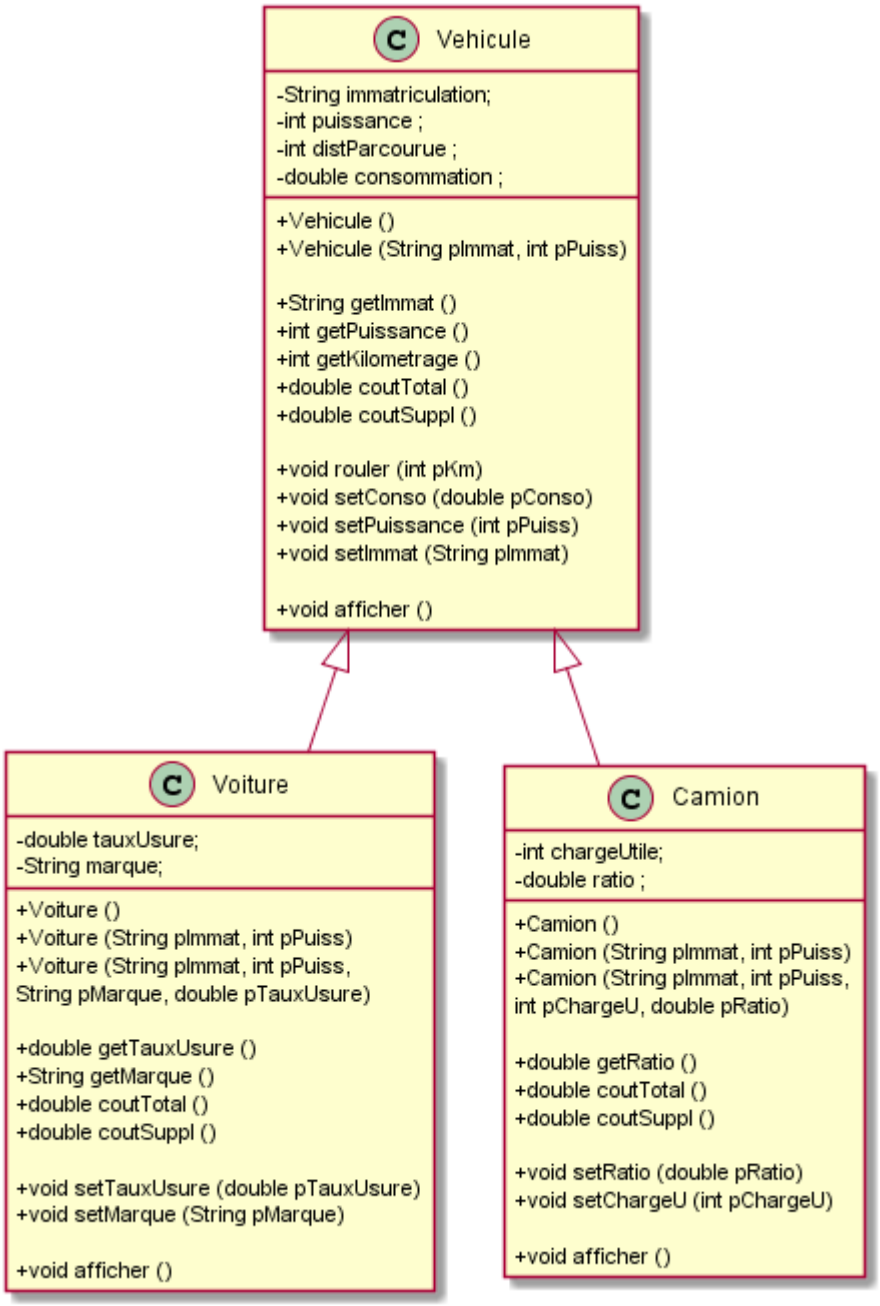
---

5. Classes abstraites

On va s'intéresser aux mécanismes d'héritage mis en oeuvre dans la hiérarchie de classes suivante.

Le code de la classe Vehicule vous est donné.

Diagramme UML des classes mises en oeuvre (sans multiplicités)



5.1. Méthodes coutTotal() et coutSuppl()

Les spécialistes donnent les spécification suivantes :

1. Classe Vehicule
   a. Le coût total d'un Vehicule est de 15% de la distance parcourue fois la consommation.
2. Classe Voiture
   a. Le coût total d'une Voiture est l'addition de :
      - coût total d'un véhicule,
      - un forfait de coût de 0.85 fois la distance parcourue,
      - le coût supplémentaire particulier aux voitures.
   b. Le coût supplémentaire particulier aux voitures est la distance parcourue multipliée par le taux d'usure (en pourcentage).

3. Classe Camion
   a. Le coût total d'un Camion est l'addition de :
      - coût total d'un véhicule,
      - un forfait de 1.35 fois la distance parcourue,
      - le coût supplémentaire particulier aux camions.
   b. Le coût supplémentaire particulier aux camions est la distance parcourue multipliée par le ratio (en pourcentage) et la charge utile.

On souhaite faire fonctionner le code suivant :

```
Vehicule tabv[] = new Vehicule [4];

tabv[0] = new Voiture ("2222 VV 22", 5);
tabv[1] = new Camion ("2222 CC 22", 12);
tabv[2] = new Voiture ("3333 VV 33", 6, "Opel", 10);
tabv[3] = new Camion ("3333 CC 33", 14, 1000, 2);

for (int i=0 ; i<4 ; i++) {
        tabv[i].afficher();
        System.out.println (tabv[i].coutTotal());
        System.out.println (tabv[i].coutSuppl());
}
```

Questions :

1. Écrire les méthodes coutTotal() et coutSuppl() dans les 3 classes.
2. Nous n'avons pas de définition pour le coût supplémentaire particulier aux véhicules. Il n'existe pas. Doit-on déclarer la méthode coutSuppl() dans la classe Vehicule ? Comment l'écrire ?

5.2. Déclaration d'une classes abstraite

On peut constater plusieurs éléments :

1. Point de vue client (garagiste, …) : un objet Vehicule n'existe pas ⇒ seules existent vraiment une voiture ou un camion.
2. Point de vue client (garagiste, …) : on parle des véhicules en général pour signifier un ensemble de voitures et camions (le parc du garage, …).
3. La classe Vehicule :
   a. Point de vue fonctionnel : généralise des attributs et méthodes communs à tous les véhicules

(voitures et camions).

b. Point de vue ensembliste : permet de **désigner** "indifféremment" des voitures et des camions.

Question :

1. Déclarer la classe Vehicule abstraite.
2. Modifier la déclaration de coutSuppl() dans Vehicule en conséquence.

---

6. Polymorphisme et interface

Voir support de cours.

Dernière mise à jour 2015-04-19 16:41:56 CEST