Programmation Orientée Objet

PROJET D'INFORMATIQUE

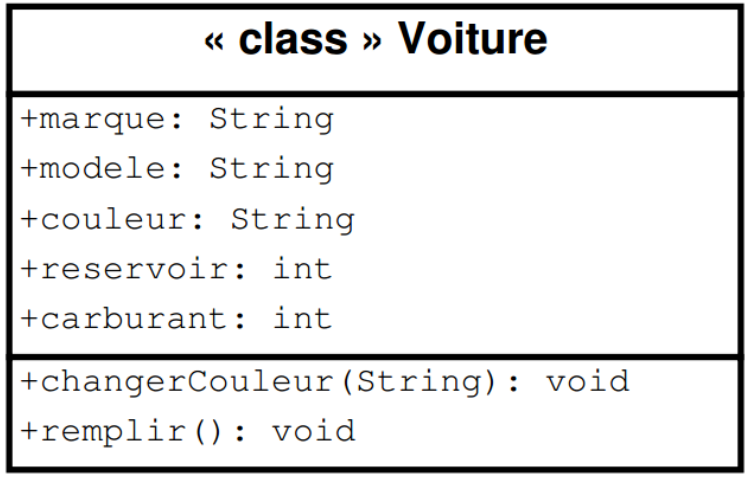
12/03/19

Sommaire

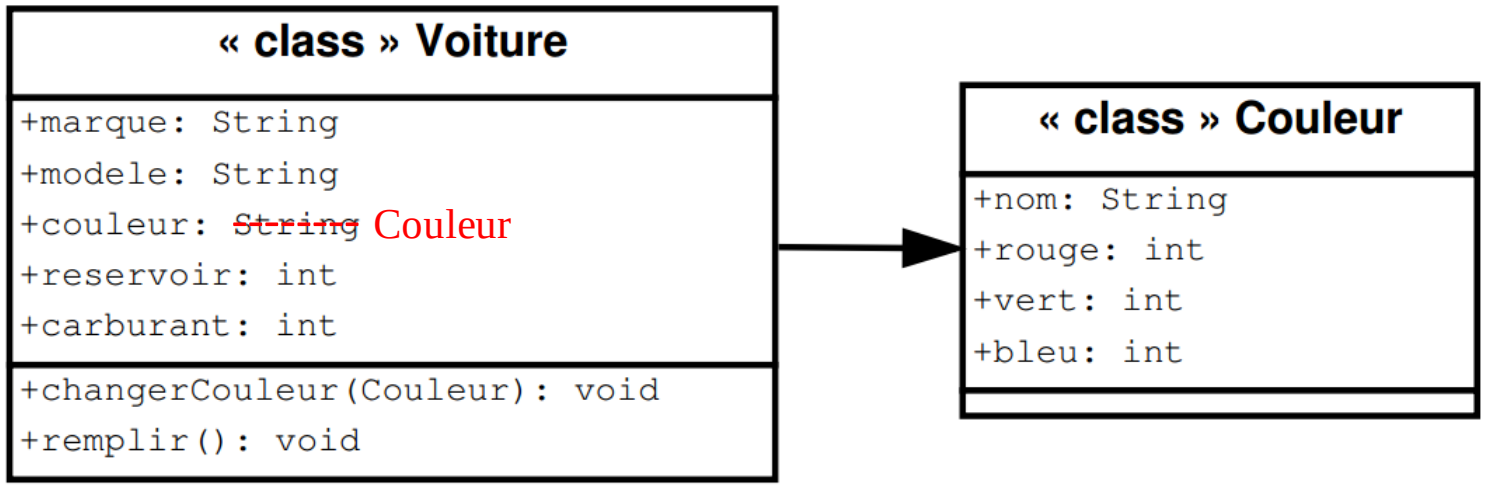
- Classe & Objet
- Polymorphisme
- Encapsulation
- Héritage
- Classe Abstraite
- Interface

Classe & Objet

- Classe != Objet
- Objet = Instance d'une classe
- Classe <> Type primitif



Classe & Objet



Encapsulation

Sans encapsulation :

```
class CompteBancaire{
    public int montantDisponible;

    public CompteBancaire(int montant){
        this.montantDisponible = montant;
    }
}
```

```
CompteBancaire compte = new CompteBancaire(1000);
compte.montantDisponible = -100;  // Pas possible avec encapsulation
compte.setMontant(-100);  // N'aura aucun effet
compte.setMontant(2000);  // Fonctionnera grace à l'encapsulation
```

Avec encapsulation :

```
class CompteBancaire{
    private int montantDisponible;

    public CompteBancaire(int montant){
        this.montantDisponible = montant;
    }

    public int getMontant(){
        return this.montant;
    }

    public setMontant(int montant){
        if(montant > 0){
            this.montantDisponible = montant;
        }
    }
}
```

Polymorphisme

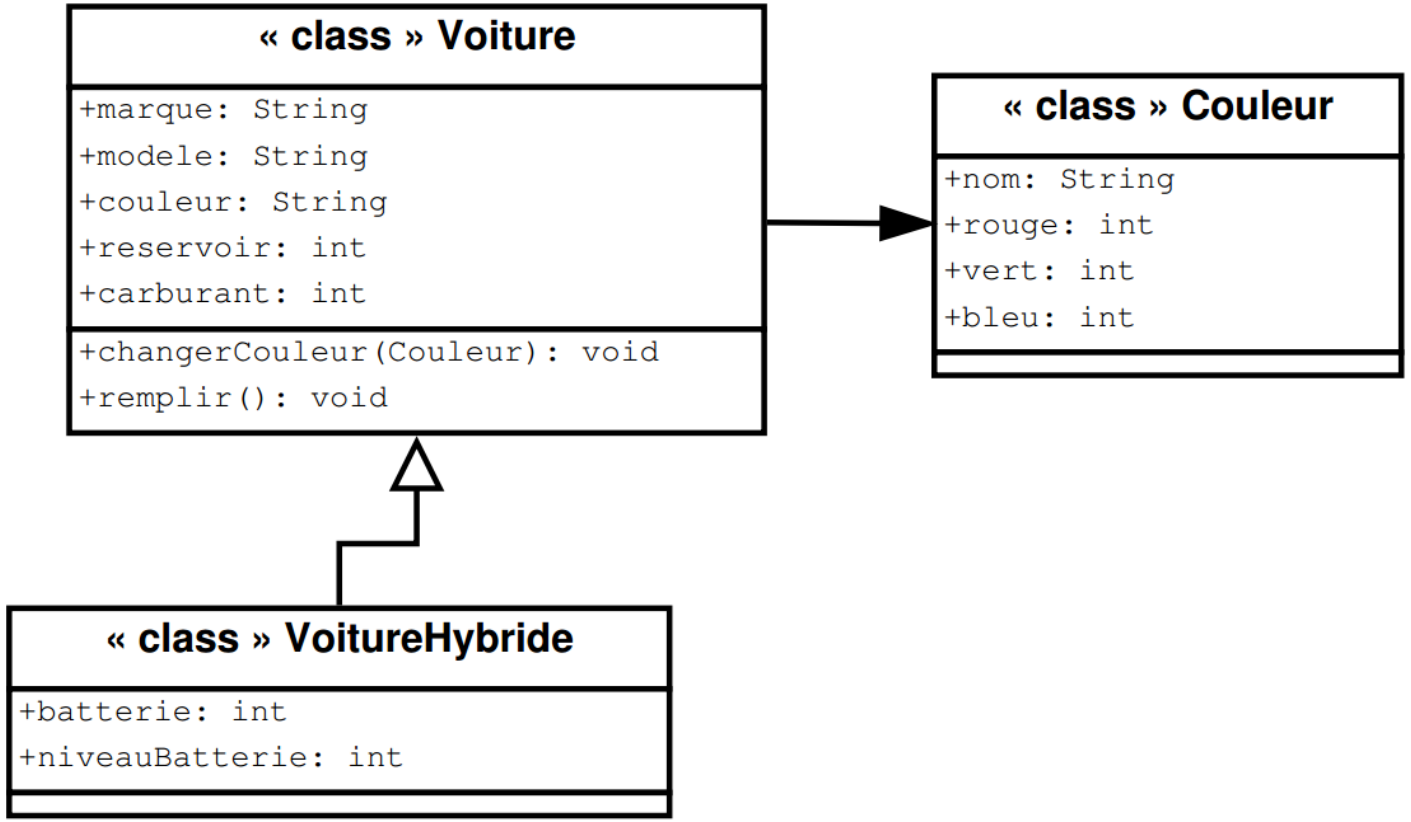
Trois types nous intéressent :

- Polymorphisme ad-hoc
- Polymorphisme d'héritage
- Polymorphisme paramétrique

Polymorphisme ad-hoc

```
class Voiture{
    ...
    public remplir(){
        ...
    }
    public remplir(int litre){
        ...
    }
    public remplir(Couleur c){
        if(this.couleur.equals(c)){
            ...
        }
    }
}
```

Polymorphisme d’héritage



Polymorphisme paramétrique

```java
public class Node<T> {

  //contenu du noeud
  private T data;
  private Node next;

  public Node(){
    this.data = null;
    this.next = null;
  }

  //Constructeur avec paramètre inconnu pour l'instant
  public Node(T val, Node next){
    this.data = val;
    this.next = next;
  }

  //Définit le contenu avec le paramètre
  public void setData(T val){
    this.data = val;
  }

  //Retourne la valeur déjà « castée » par la signature de la méthode !
  public T getData(){
    return this.data;
  }
}
```

```java
Node<String> noeud = new Node("Contenu", null);
String data = noeud.getData();
noeud.setData("Autre contenu");

Node<Integer> noeud2 = new Node(10, null);
Integer data = noeud2.getData();
noeud2.setData(20);
```

Héritage

Sans héritage :

```
class Case{
    private type;
    ...
    public String description(){
        if     (this.type.equals("mur")) return "Case de type mur";
        else if(this.type.equals("caisse")) return "Case de type caisse";
        else if(this.type.equals("sol")) return "Case de type sol";
    }
}
```

```
Case[][] plateau = new Case[5][5];
plateau[0][0] = new Mur(..);
plateau[1][2] = new Caisse(..);
plateau[3][1] = new Sol(..);
```

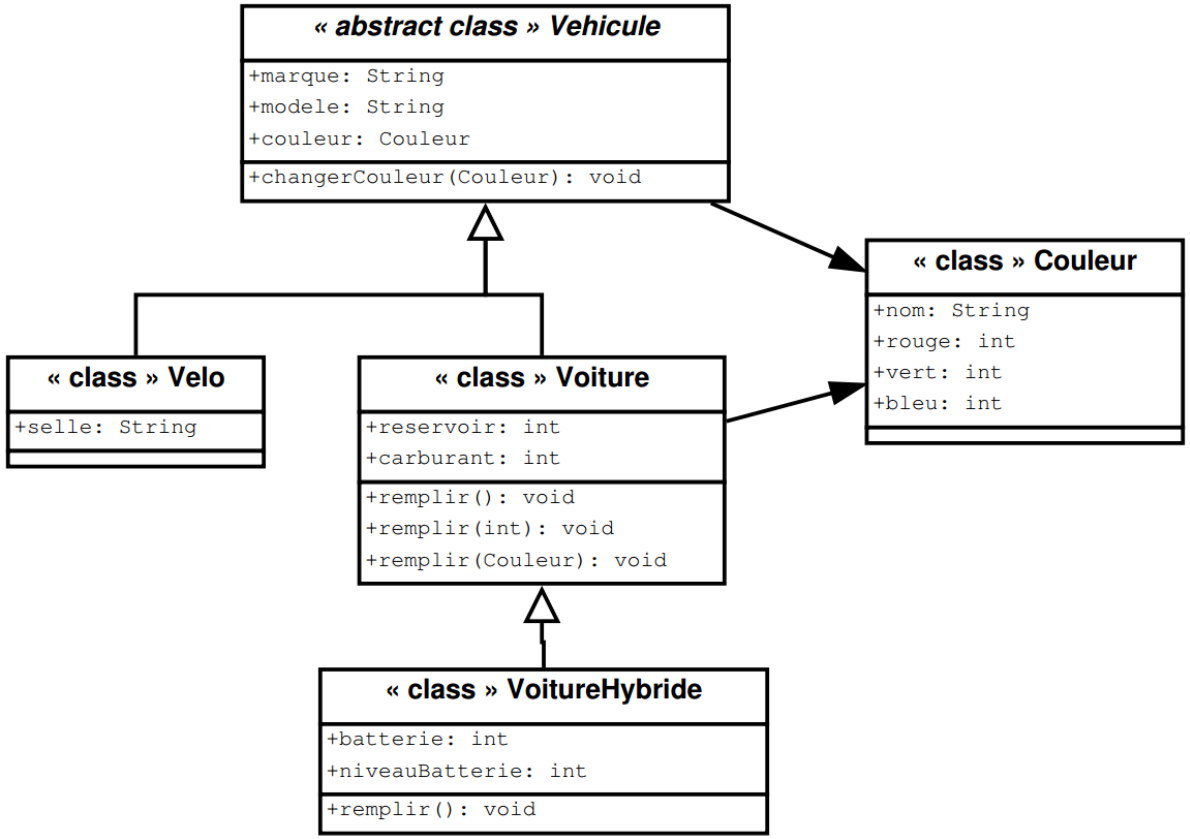
Avec héritage :

```
class Case{
    ...
    public String description(){
        return "Case qui ne sert à rien";
    }
}

class Mur extends Case{
    ...
    public String description(){
        return "Case de type mur";
    }
}

class Caisse extends Case{
    ...
    public String description(){
        return "Case de type caisse";
    }
}

class Sol extends Case{
    ...
    public String description(){
        return "Case de type sol";
    }
}
```

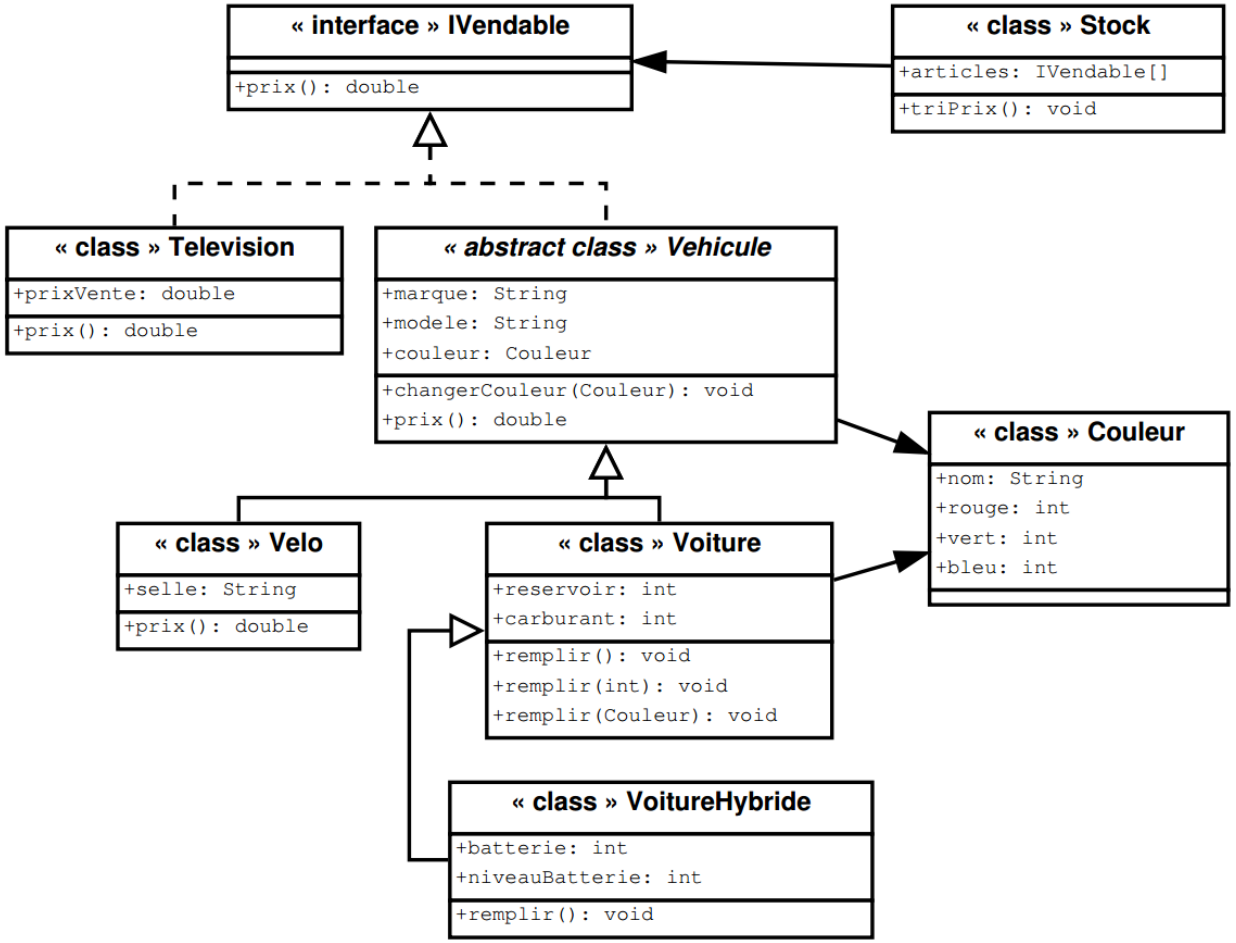
Classe Abstraite

```
abstract class Forme {
    abstract float aire() ;
}

class Carre extends Forme {
    float cote;
    float aire() {
        return cote * cote;
    }
}

class Cercle extends Forme {
    float rayon;
    float aire() {
        return Math.PI*rayon*rayon;
    }
}
```

Classe Abstraite



Interface




```
IVendable[]    stock    = {new Television("LG", 300),
                           new Vehicule("Ford",..),
                           new Vehicule("Audi",..),
                           new Television("Samsung", 300)};

for(int i; i < stock.length; i++){
    IVendable objetVendable = stock[i];
    double prix = objectVendable.prix();
    System.out.println(prix);
}
```

Synthèse : Classe Abstraite VS Interface

Classe Abstraite	Interface
Pas de création d'objet possible	Pas de création d'objet possible
Implémentation de méthodes autorisée, mais pas obligatoire	**Pas d'implémentation de méthode**, juste une liste de signature
Utile pour rassembler des implémentations communes à plusieurs classes	Utile pour spécifier des propriétés communes à plusieurs classes, mais dont le comportement peut différer.
Une classe ne peut hériter que d'une seule classe (abstraite)	Une classe peut implémenter plusieurs interfaces