M2103 – Bases de la Programmation Orientée Objets

Java – Cours 6

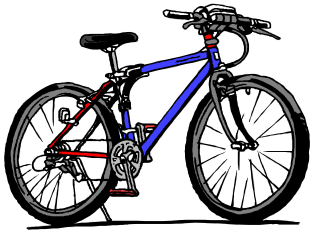
Héritage et Polymorphisme

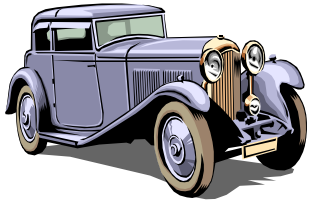
Plan du Cours

- Généralisation
- La visibilité ‘Protected’ dans les hiérarchies de classes
- Polymorphisme
- Liaison statique vs liaison dynamique
- Les classes java.lang.Object et java.lang.Class

Généralisation
Categorisation d'objets

- Des objets peuvent quelquefois partager certains attributs et méthodes communs mais pas d'autres
  - Vélos et voitures sont des véhicules, qui ont des roues et qui peuvent démarrer, tourner et s'arrêter
  - Les voitures ont des portes, des fenêtres et un réservoir, ce qui n'est pas le cas des vélos
  - Les vélos ont des pédales et des guidons, ce qui n'est pas le cas des voitures
  - Les voitures sont également motorisées…



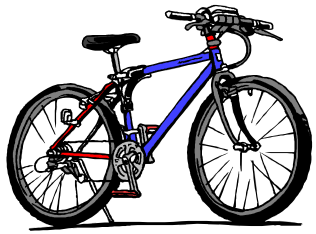


Généralisation

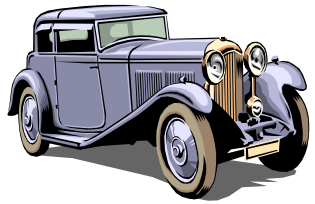
Traitement des membres communs

- Les membres communs doivent-ils être dupliqués dans des classes séparées ?

Vélo
roues **engrenages** guidon pédales
démarrer() **stop()** **tourner()** pédaler()



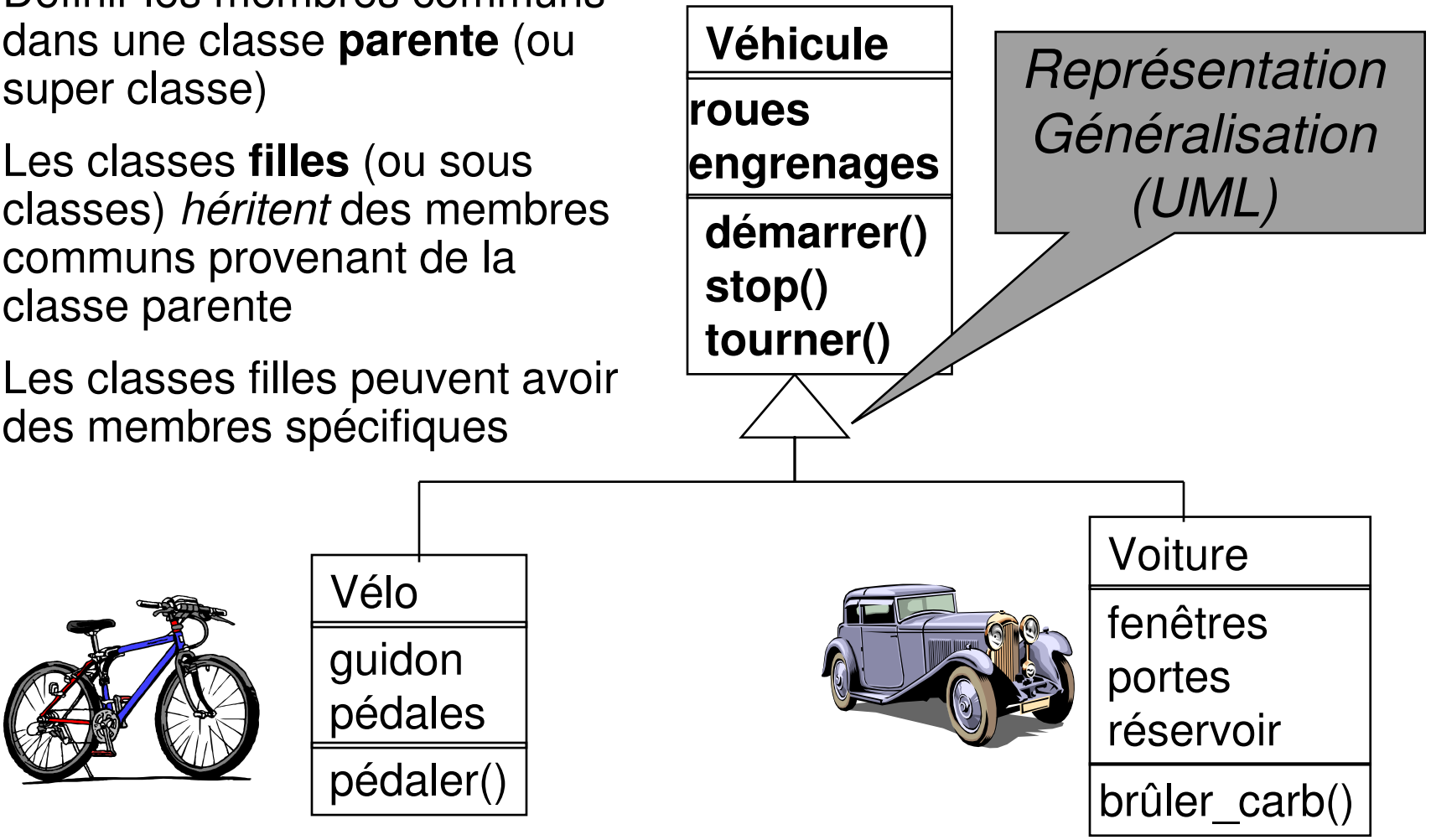
Voiture
roues **engrenages** fenêtres portes réservoir
démarrer() **stop()** **tourner()** brûler_carb()



Généralisation

Mettre en oeuvre une hiérarchie de classes

- Définir les membres communs dans une classe **parente** (ou super classe)
- Les classes **filles** (ou sous classes) *héritent* des membres communs provenant de la classe parente
- Les classes filles peuvent avoir des membres spécifiques



Généralisation
en Java

```
class Véhicule {
    int roues;
    int engrenages;
    Véhicule (int roues,
                int engrenages) {
      this.roues = roues;
      this.engrenages =
    engrenages;
    }
    public void démarrer() {.. }
    public void stop() {.. }
    public void tourner() { ..}

   public String toString() {.. }

}
```

```
class Vélo extends Véhicule {
    String guidon;
    String pédales;
    Velo (int roues, int engrenages,
                String guidon, String pédales)
    {
       super(roues, engrenages);
       this.guidon = guidon;
       this.pédales = pédales;

    }
    public void pédaler () {.. }
    public String toString() {
        return (super.toString () + " "
        + guidon + " " + pédales);
    }

}
```

Généralisation
en Java

```
class Véhicule {
    int roues;
    int engrenages;
    Vehicule (int roues,
                int engrenages) {
      this.roues = roues;
      this.engrenages =
    engrenages;
    }
    public void démarrer() {.. }
    public void stop() {.. }
    public void tourner() { ..}

    public String toString() {.. }

}
```

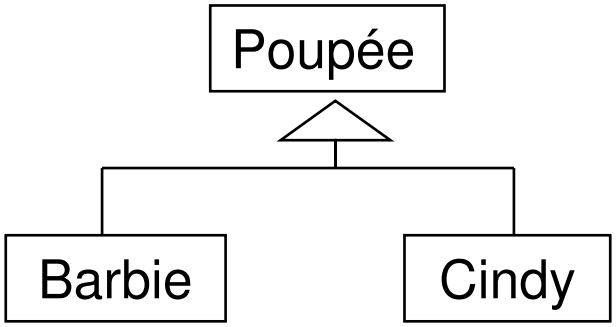
```
class Voiture extends Véhicule {
    int fenêtres;
    int portes;
    double reservoir;
    Voiture (int roues, int engrenages,
          int fenêtres, int portes) {
       super(roues, engrenages);
       this.fenêtres = fenêtres;
       this.portes = portes;
       réservoir = 0.0;
    }
    public void brûler_carb() { }

}
```

Instanciation & Transmission de messages

- Des classes parentes peuvent caractériser des entités réelles

```
Poupée poupée1 = new Poupée();
Barbie barbie1  = new Barbie();
```



- D'autres, comme `Véhicule`, représentent des concepts abstraits
  - Les messages sont transmis aux objets instanciés à partir des classes filles qui, comme `Voiture` et `Vélo`, caractérisent des objets réels

```
public static void main (String[ ] args) {

    Voiture maVoit = new Voiture (4, 5, 6, 4);
    Vélo tonVélo = new Vélo (2, 2, “haut“, “plates”);

    maVoit.tourner();
    System.out.println (" Vélo " + tonVélo);

}
```

Visibilité de membres dans une hiérarchie de classes

- Les classes parentes n'ont pas besoin d'accéder aux membres des classes filles
- Les classes filles ont elles besoin d'accéder aux membres des classes parentes
- Si les attributs des classes parentes sont déclarés privés, ils ne sont pas accessibles par les classes filles
- Si les attributs des classes parentes sont déclarés publics, d'autres classes peuvent y accéder alors qu'elles ne devraient pas
- Solution : la visibilité **protected** qui donne l'accès aux classes filles **seulement**

Visibilité dans une hiérarchie de classes : Protected

```
class Véhicule {
    protected int roues;
    protected int engrenages;
    Véhicule (int roues,
                int engrenages) {
      this.roues = roues;
      this.engrenages = engrenages;
    }
    public void démarrer() {.. }
    public void stop() {.. }
    public void tourner() { ..}

    public String toString() {.. }

}
```

```
class Vélo extends Véhicule {
    String guidon;
    String pédales;
    Velo (int roues, int engrenages,
                String guidon, String
    pédales) {
       super(roues, engrenages);
       this.guidon = guidon;
       this.pédales = pédales;

    }
    public void pédaler () {.. }
    public String toString() {
        return (super.toString () + " "
        + guidon + " " + pédales);
    }

}
```

Polymorphisme

- Méthodes avec le même nom, faisant la même chose mais de manières différentes :
  - **Surcharge** (Overloading)
    - Même classe, différents paramètres
    - Méthode utilisée est celle avec les paramètres correspondants
  - **Outrepassement** (Overriding)
    - Différentes classes dans la même hiérarchie
    - Méthode fille outrepasse la méthode parente
    - Méthodes statiques ou privées ne peuvent être outrepassées

Polymorphisme
Surcharge

```
class Véhicule {
        public void tourner ( ) {
             …
         }
         public void tourner (String direction) {
             …
         }
}
```

maVoit.tourner( );

maVoit.tourner (“gauche”);

La méthode **tourner** est surchargée dans la classe Vehicule – il existe 2 méthodes nommées *tourner*, avec des signatures uniques, dans la classe.

Polymorphisme
Outrepassement

```
class Véhicule {
      public void tourner (String dir)
       {
       System.out.println (" Véhicule
          tourne à " + dir);
       }
}
```

```
class Voiture extends Véhicule {
    public void tourner (String dir)
     {
      System.out.println ("Après utilisation du
         clignotant, voiture tourne à " + dir);
     }
}
class Vélo extends Vehicule {
     // Pas de méthodes déclarées.
}
```

Les méthodes ont la même ***signature*** *dans les 2 classes*

- `Voiture` et `Vélo` héritent de la méthode **tourner**() de la classe Véhicule
- Mais la méthode **tourner**() dans `Voiture` outrepasse la méthode **tourner**() de `Véhicule`
- `Vélo` n'a pas de méthode **tourner**() donc utilisera celle de `Véhicule`

Interdire l'outrepassement

- Normalement, les méthodes héritées peuvent être outrepassées
- Pour éviter l'outrepassement
  - Déclaration de la méthode en tant que **final**

Variables polymorphes

- Une variable peut être déclarée de type classe pour laquelle il existe des classes filles
- On peut affecter à cette variable un objet de n'importe lequel des types sous classes – ceci créant ***une variable polymorphe***
- Si un message est envoyé à la variable et la sous-classe a redéfini le comportement de la méthode, quelle est la méthode qui sera exécutée : la méthode originale ou celle de la sous-classe?

Exemple:

```
Véhicule maVoit = new Voiture(4,5,6,3);

maVoit.tourner("gauche");
```

Liaison Dynamique

- Java utilise la liaison ***tardive ou dynamique***
- Variables de référence objet peuvent caractériser :
  - Un objet de même type classe que la variable
  - Un objet de type sous-classe par rapport au type de la variable
  - null
- Au moment de la compilation
  - Le compilateur ne remonte jamais dans l'historique des instructions et ne peut donc savoir quel objet sera référencé par la variable à l'exécution
  - Toutefois, Java présente un mécanisme permettant de repousser la décision sur la méthode à choisir jusqu'à l'**exécution**.

Liaison Dynamique - Exemple

```
class A {
	void faire() {
		System.out.println("niveau a");
	}
}

class B extends A {
	void faire() {
		System.out.println("niveau b");
	}
}

class C extends B {}

class ExempleLiaisonDynamique {
	public static void main(String[] argv) {
		A a;

		a = new A();
		a.faire(); // instruction 1


		a = new B();
		a.faire(); // instruction 2


		a = new C();
		a.faire(); // instruction 3
	}
}
```

Liaison anticipée

- Si le compilateur connaît la méthode à exécuter, alors il peut choisir (liaison) cette méthode-ci dès la compilation – c'est la ***liaison anticipée*** ou ***statique***
- Java utilise la liaison statique pour :
  - Les méthodes déclarées avec **final**
  - Les méthodes privées (utilisées à l'intérieur d'autres méthodes de la même classe, en raison de la visibilité)
  - Les méthodes statiques sachant qu'elles n'ont pas besoin d'un objet particulier pour être appelées

La classe `Object`

- Toutes les classes Java sont des classes descendantes de la classe `java.lang.Object`
- Cette classe définit un ensemble de méthodes, utilisables par toutes les classes, parmi lesquelles :
  - public String **toString**() – retourne une représentation de type **String** de l'objet, utile pour l'affichage.
  - public boolean **equals**(Object other) – permet la comparaison d'objets
  - public Class **getClass**() – retourne un objet de type `Class` qui décrit le type de l'objet pour lequel la méthode est appelée.

La classe `Class`

- La classe `java.lang.Class` class est utilisée pour représenter de l'information à propos des types Classe utilisés dans un programme.
- La méthode la plus utile est :
  - public String **getName**() – retourne le nom de la classe caractérisée par l'objet.
- Exemple:

```
String bienvenue = "Salut";
System.out.println(bienvenue.getClass().getName() )
```

Trace :

```
java.lang.String
```