

LIFI-Java 2004

Séance du Mercredi 22 sept.

Cours 3

Plan du cours

- Rappels cours précédents
- Variables et fonctions de classe
- Programmation Objet Avancée
 - notion d'héritage
 - polymorphisme

Variables de classe (1/6)

- Une classe `Vehicle` simple...

```
public class Vehicle {  
    private String name;  
    public Vehicle(String n) {  
        name = n;  
    }  
    public String toString() {  
        return name;  
    }  
};
```

```
Vehicle v=new Vehicle("Porsche");
```

- Comment rajouter une immatriculation?
 - un nombre unique par instance
 - attribué automatiquement

Variables de classe (2/6)

- On ajoute une variable `immat`

```
public class Vehicle {  
    private String name;  
    private int immat;  
    public Vehicle(String n,int i) {  
        name = n;  
        immat = i;  
    }  
    public String toString() {  
        return name+" "+immat;  
    }  
    public String getName() { ... }  
    public int getImmat() {  
        return immat;  
    }  
};
```

```
Vehicle v=new Vehicle("Porsche",0);  
  
Vehicle w=new Vehicle("Skoda",1);  
  
Vehicle x=new Vehicle("Alfa"
```



**Le programmeur doit
compter lui même!!!**

Variables de classe (3/6)

- On ajoute un compteur...

```
public class Vehicle {
    private String name;
    private int immat;
    public Vehicle(String n,int i) {
        name = n;
        immat = i;
    }
    public String toString() {
        return name+" "+immat;
    }
    public String getName() { ... }
    public int getImmat() { ... }
};
```

```
int nb=0;
Vehicle v=new Vehicle("Porsche",nb++);

Vehicle w=new Vehicle("Skoda",nb++);

Vehicle x=new Vehicle("Alfa", nb++);

Le client peut tricher!!!

nb = 0;
Vehicle x=new Vehicle("Alfa", nb++);
```

Variables de classe (4/6)

- Il faut protéger le compteur
 - le client ne peut le modifier
 - il ne doit même pas savoir qu’il existe!
- C’est la classe `vehicle` qui doit “compter” combien d’instances sont créés
 - information “interne” à la classe
 - ne dépend pas d’une instance en particulier

Variables de classe (5/6)

- On définit une variable pour la classe...

```
public class Vehicle {  
    private String name;  
    private int immat;  
    static private int nb=0;  
    public Vehicle(String n) {  
        name = n;  
        immat = nb++;  
    }  
    public String toString() { ... }  
    public String getName() { ... }  
    public int getImmat() { ... }  
};
```

```
Vehicle v=new Vehicle("Porsche");  
  
Vehicle w=new Vehicle("Skoda");  
  
Vehicle x=new Vehicle("Alfa");  
  
// affiche "Alfa 3" !  
System.out.println(x);
```

**Le client ne voit plus
le compteur !**

Variables de classe (6/6)

- En résumé:
 - on peut définir des *variables de classe*
 - on le fait avec le mot clé `static`
 - de même on définit des *fonctions de classe*
- On a déjà vu ça!
 - `out` est une variable de la classe `System`
 - `sqrtf()` est fonction de la classe `Math`
- Remarques diverses

POO avancée, pourquoi?

- L'*encapsulation*, ça suffit pas!
 - Factoriser du code
 - ex: classes `Car`, `Truck`
 - Permettre l'extension
 - dérivation *vs.* copier/coller
 - Manipuler des objets "semblables"
 - ex: tableau de `Vehicle`
- Il faut des *sous-classes*...

Sous-classes (1/3)

- se définit avec le mot clé `extends`
- *hérite* des variables et méthodes
- *étend* la classe avec des variables/fonctions

```
public class Vehicle {  
    public Vehicle(String n) { ... }  
    public String toString() { ... }  
    public String getName() { ... }  
    public int getImmat() { ... }  
};
```



```
public class Truck extends Vehicle {  
    private int size;  
    int getSize() { return size; }  
};
```

```
Truck t = new Truck();  
  
// t référence un Truck  
// -> on peut appeller getSize()  
System.out.println(t.getSize());  
  
// un Truck est *aussi* un Vehicle  
// -> on peut appeller getImmat()  
System.out.println(t.getImmat());
```

Sous-classes (2/3)

- On dit que
 - Vehicle est une *classe de base* de Truck
super classe
 - La class Truck *hérite* de Vehicle
dérive
 - Truck est une *sous-classe* de Vehicle
classe fille
classe dérivée

Sous-classes (3/3)

- Un `Truck` est aussi un `Vehicle` mais...



```
Vehicle v = new Truck();  
// v référence un Vehicle -> on peut appeler getImmat()  
System.out.println(v.getImmat());  
  
// même si le Vehicle référencé par v est en fait un Truck, on ne peut  
// appeler que les méthodes de Vehicle car c'est le type de v.  
System.out.println(t.getSize()); refusé à la compilation!!!
```

- Conséquences
 - sur les constructeurs -> mot clé `super`
 - sur les droits d'accès -> mot clé `protected`

Héritage et constructeurs (1/3)

- Pour construire (instancier) un `Truck`
 - il faut d'abord construire un `Vehicle`

```
public class Vehicle {  
    public Vehicle(String n) { ... }  
    public String toString() { ... }  
    public String getName() { ... }  
    public int getImmat() { ... }  
};
```

```
public class Truck extends Vehicle {  
    private int size;  
    int getSize() { return size; }  
};
```

~~Truck t = new Truck();~~

refusé à la compilation!!!

Pour construire un `Truck`, il faut
construire un `Vehicle`

or

un `Vehicle` a besoin d'un nom pour
se construire

Héritage et constructeurs (2/3)

- Pour construire (instancier) un `Truck`
 - il faut d'abord construire un `Vehicle`
 - appel à un constructeur de la classe mère

```
public class Vehicle {  
    public Vehicle(String n) { ... }  
};
```



```
public class Truck extends Vehicle {  
    private int size;  
    public Truck(String n) {  
        super(n);  
    }  
};
```

```
Truck t = new Truck("Renault");
```

OK!

Héritage et constructeurs (3/3)

- Pour construire (instancier) un `Truck`
 - il faut d'abord construire un `Vehicle`
 - ensuite on construit les variables de `Truck`

```
public class Vehicle {  
    public Vehicle(String n) { ... }  
};
```

```
public class Truck extends Vehicle {  
    private int size;  
    public Truck(String n,int s) {  
        super(n);  
        size(s);  
    }  
};
```

```
Truck t = new Truck("Renault");
```

Héritage et droits d'accès (1/2)

- Rappel:

- une classe accède à tout ses membres
- le reste du monde n'accède que les `public`

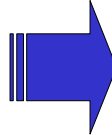
- Une classe fille hérite de tout (`variable+fonctions`)
- Mais elle ne peut accéder que:
 - les membres `public` et `protected`

Héritage et droits d'accès (2/2)

```
public class Point {  
    private double x,  
    public Point(double x,double y) {  
        this.x = x; this.y = y;  
    }  
};
```

```
public class Polar extends Point {  
    private double r,th;  
    public Polar(double x,double y) {  
        super(x,y);  
        update();  
    }  
    protected update() {  
        r = Math.sqrt(x*x+y*y);  
        th = Math.atan(y/x);  
    }  
};
```

x, y private !!!



```
public class Point {  
    private double x,y;  
    public Point(double x,double y) {  
        this.x = x; this.y = y;  
    }  
    public double getX() { return x; }  
    public double getY() { return y; }  
};
```

```
public class Polar extends Point {  
    private double r,th;  
    public Polar(double x,double y) {  
        super(x,y);  
        update();  
    }  
    protected update() {  
        r = Math.sqrt(getX()*2+getY()*2);  
        th = Math.atan(getY()/getX());  
    }  
};
```

Surcharge (1/3)

- On sait étendre avec de nouveaux membres
- Comment étendre une fonction héritée?

```
public class Point {  
    public Point(double x,double y){...}  
    public double getX() { ... }  
    public double getY() { ... }  
    public void set(double x,double y) {  
        this.x = x;this.y = double.y;  
    }  
};
```



```
public class Polar extends Point {  
    public Polar(double x,double y){...}  
    protected update() { ... }  
    public double getR() { return r; }  
    public double getTheta() { return th; }  
};
```

```
Polar p = new Polar(3,4);  
  
System.out.println(p.getR()); // 5 -> OK!  
  
p.set(1,0);  
  
System.out.println(p.getR()); // 5 -> !?!
```

**update() n'est pas
appelée!!!**

Surcharge (2/3)

- On *surcharge* la fonction
 - on peut appeler la version héritée
 - (mais c'est pas obligatoire!)

```
public class Point {  
    // ...  
    public void set(double x,double y) {  
        this.x = x;this.y = double.y;  
    }  
};
```

```
public class Polar extends Point {  
    // ...  
    public set(double x,double y) {  
        super.set(x,y);  
        update();  
    }  
};
```

```
Polar p = new Polar(3,4);  
  
System.out.println(p.getR()); // 5 -> OK!  
  
p.set(1,0);  
  
System.out.println(p.getR()); // 1 -> OK!
```

**update() est
appelée!!!**

Surcharge (3/3)

- Méthodes *virtuelles*
 - à l'exécution, Java connaît le vrai type
 - il appelle la fonction surchargée adéquate

```
public class Point {  
    // ...  
    public void set(double x,double y) {  
        this.x = x;this.y = double.y;  
    }  
};
```



```
public class Polar extends Point {  
    // ...  
    public set(double x,double y) {  
        super(x,y);  
        update();  
    }  
};
```

```
Polar p = new Polar(3,4);  
  
System.out.println(p.getR()); // 5 -> OK!  
Vehicle v = p;  
v.set(1,0);  
  
System.out.println(p.getR()); // 1 -> OK!
```

Exemple: la class `Object`

- D'où vient la méthode `getClass()`?
 - Toute classe dérive de la classe `Object`!
 - classe spéciale de Java
 - héritage "automatique"
 - la méthode `toString()`
- Pourquoi une classe de base
 - Chaque instance est un `Object`!
 - Permet d'avoir des ensembles hétérogènes

Exemple: tableaux hétérogènes

- Définir les classes
 - Point.java
 - Polar.java
- Faire un programme
 - Main.java
 - crée un tableau de Point
 - le remplit avec
 - des Point
 - des Polar
 - L'affiche

```
public class Point {  
    public Point(double x,double y);  
    public double getX();  
    public double getY();  
    public void set(double x,double y);  
    public String toString();  
};
```



```
public class Polar extends Point {  
    public Polar(double x,double y);  
    public void set(double x,double y);  
    public String toString();  
    public double getR();  
    public double getTheta();  
};
```

Polymorphisme (1/3)

- Comment n'afficher que les `Polar`?
- Il faut tester si une instance de `Point` est en fait une instance de `Polar`
 - en utilisant `getClass()` -> **pas propre**
 - en utilisant `instanceof` -> **propre!**

```
Vector v = new Vector()
// ... remplissage de v...
for (int i=0;i<v.size();++i) {
    if (tab.elementAt(i) instanceof Polar) {
        System.out.println(tab.elementAt(i));
    }
}
```

Polymorphisme (2/3)

- Comment accéder aux méthodes de la classe `Polar` quand on a une référence de type `Vehicle` sur une instance de `Polar`?
- En utilisant un *cast*!

```
Point m = new Polar(3,4);  
m.set(1,0);
```

```
System.out.println(m.getR());
```

refusé à la compilation!!!

Une référence à un `Vehicle` ne peut pas accéder à une méthode de `Polar` même si

elle référence en fait un `Polar`

```
Point m = new Polar(3,4);  
m.set(1,0);
```

```
Polar p = (Polar) m;  
System.out.println(p.getR());
```


Polymorphisme (3/3)

- Que se passe-t-il si le *cast* est invalide?
 - génère une erreur
- Il faut faire d'abord un test avec `instanceof`

```
Point m = new Point(3,4);  
  
Polar p = (Polar) m;  
System.out.println(p.getR());
```

erreur à l'exécution!!!

**p ne référence pas une
instance de type Polar donc
on ne peut pas faire de cast.**

```
Point m = new Polar(3,4);  
  
if (m instanceof Polar) {  
    Polar p = (Polar) m;  
    System.out.println(p.getR());  
}
```

Avancé (1/2)

- Comment interdire la surcharge?

```
public class Vehicle {
    private int immat;
    static private int nb=0;
    public Vehicle() { immat = nb++; }
    public int getImmat() {
        return immat;
    }
};
```



```
public class Stolen extends Vehicle {
    private int fakeimmat;
    public Stolen(int i) {
        fakeimmat = i;
    }
    public int getImmat() {
        return fakeimmat;
    }
};
```

```
Vehicle v=new Vehicle();
// v.getImmat() vaut 0 -> OK
Vehicle w=new Vehicle();
// v.getImmat() vaut 1 -> OK
```

```
Vehicle x=new Stolen(0);
// He! x.getImmat() renvoie un faux
// numéro déjà utilisé au lieu de
// renvoyer le numéro 2!!!
```

Avancé (2/2)

- Comment interdire la surcharge?
- En utilisant le mot clé `final`!

```
public class Vehicle {  
    private int immat;  
    static private int nb=0;  
    public Vehicle() { immat = nb++; }  
    final public int getImmat() {  
        return immat;  
    }  
};
```



```
public class Stolen extends Vehicle {  
    private int fakeimmat;  
    public Stolen(int i) {  
        fakeimmat = i;  
    }  
    public int getImmat() {  
        return fakeimmat;  
    }  
    getImmat() final!!!  
};
```

```
Vehicle v=new Vehicle();  
// v.getImmat() vaut 0 -> OK  
Vehicle w=new Vehicle();  
// v.getImmat() vaut 1 -> OK
```

```
Vehicle x=new Stolen(0);  
// He! x.getImmat() renvoie un faux  
// numéro déjà utilisé au lieu de  
// renvoyer le numéro 2!!!
```

Bilan

- On connaît les bases de la POO
 - Principes qu'on retrouve ailleurs
 - Avec des variations (ex: pas de `final` en C++)
- Pour les séances à venir:
 - se familiariser avec la syntaxe Java
 - apprendre à concevoir “objet”
 - maîtriser les subtilités
- Ensuite: encore des concepts à voir
 - Interface et implémentation...
 - Classes abstraites...
 - Exceptions...