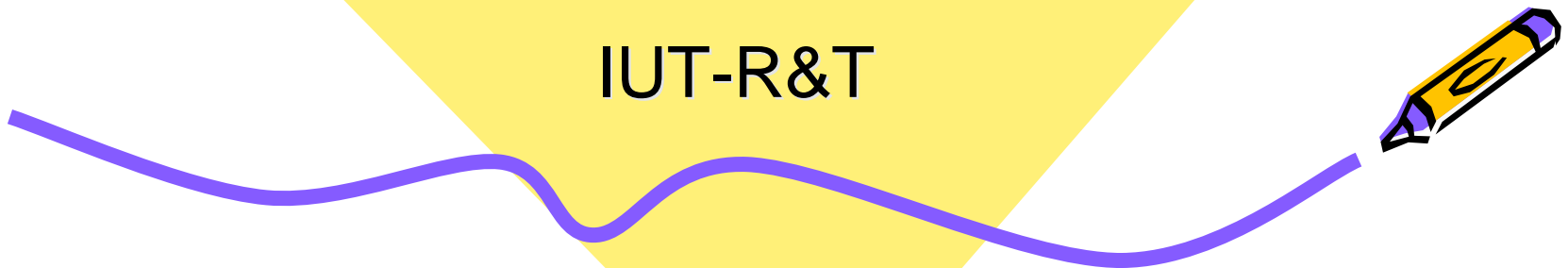


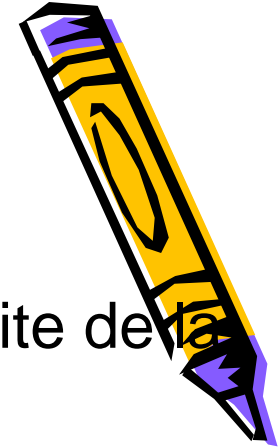


# Polymorphisme

Karima Boudaoud  
IUT-R&T



# Polymorphisme (1)



## ○ Question

- Reprenons l'exemple de la classe **Etudiant** qui hérite de la classe **Personne**. Soit une méthode **getNom()** de **Personne** qui est redéfinie dans **Etudiant**
- quelle méthode **getNom()** sera exécutée dans le code suivant, celle de **Personne** ou celle de **Etudiant**?

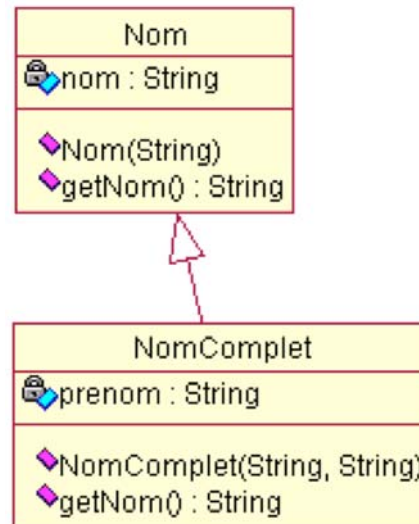
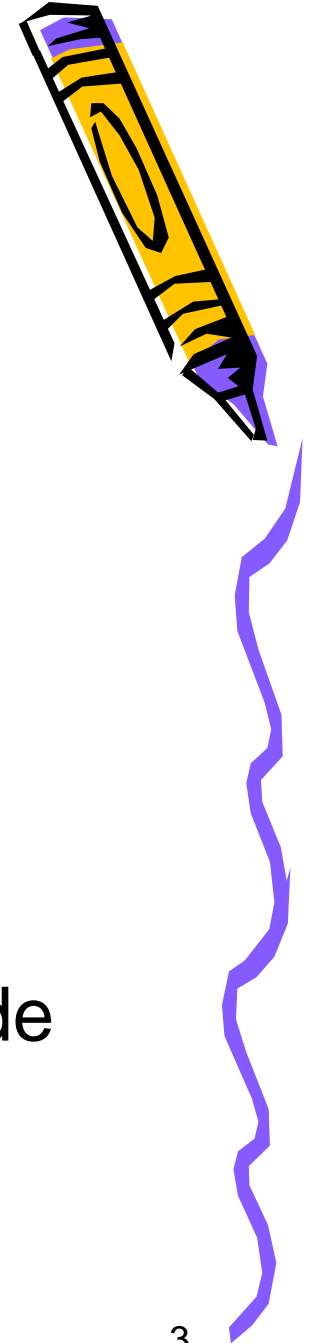
```
Personne a = new Etudiant(5); // a est un objet de la classe  
a.getNom();                    Etudiant mais il est  
déclaré                       de la classe  
Personne
```

- la méthode appelée ne dépend que du type réel (**Etudiant**) de l'objet **a** et pas du type déclaré (ici **Personne**)

➡ c'est la méthode de la classe **Etudiant** qui sera exécutée



# Polymorphisme (2)



- **NomComplet** peut être utilisé à la place de **Nom**



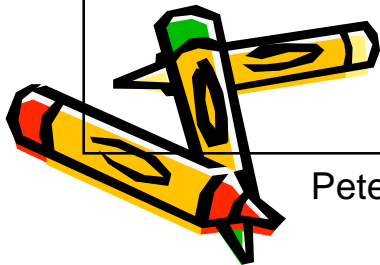
# Exemple (1)

```
Nom[] noms = new Nom[4];  
noms[0] = new NomCompleet("Cantonna",  
    "Eric");  
noms[1] = new Nom("Ronaldo");  
noms[2] = new NomCompleet("Overmars",  
    "Marc");  
...  
for (int i = 0; i < 4; i++) {  
    System.out.println(noms[i].getNom());  
}
```

Classe dérivée à la  
place de la classe

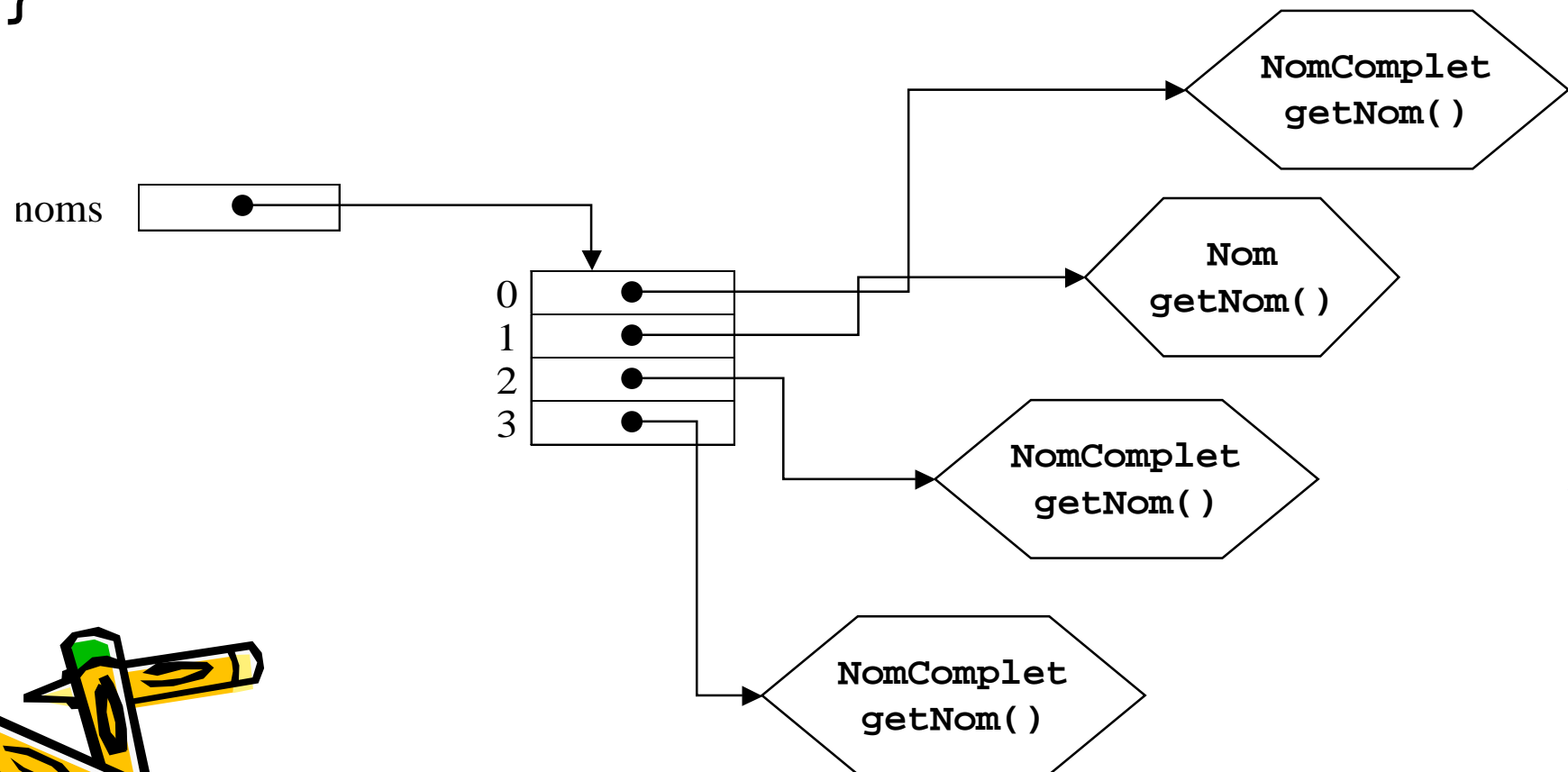
-> Eric Cantonna  
Ronaldo  
Marc Overmars

Quelle méthode ?



# Exemple (2)

```
for (int i = 0; i < 4; i++) {  
    System.out.println(noms[i].getNom());  
}
```

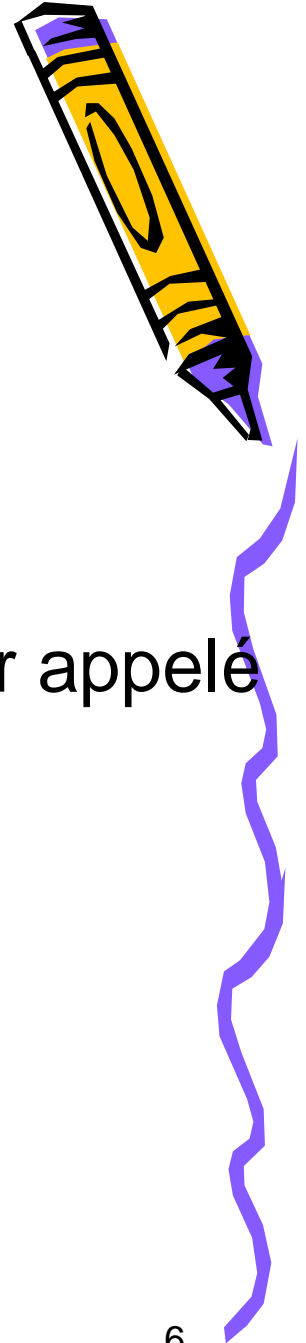


Peter Sander

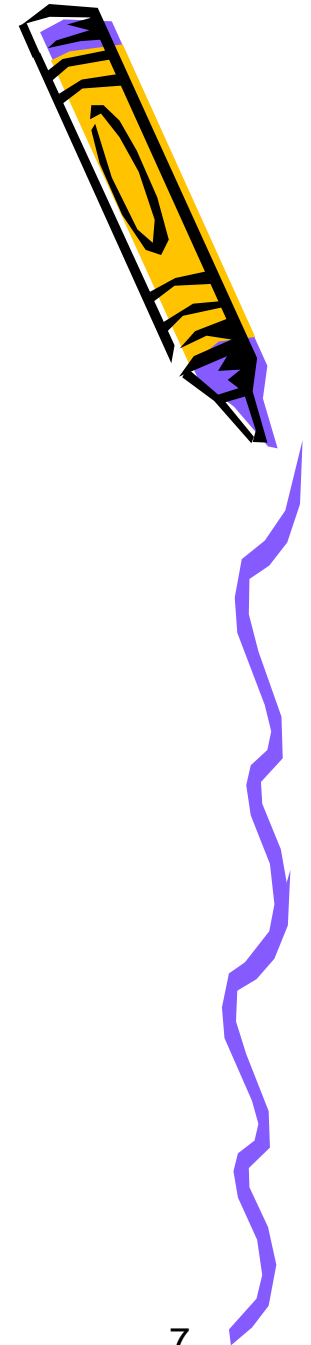
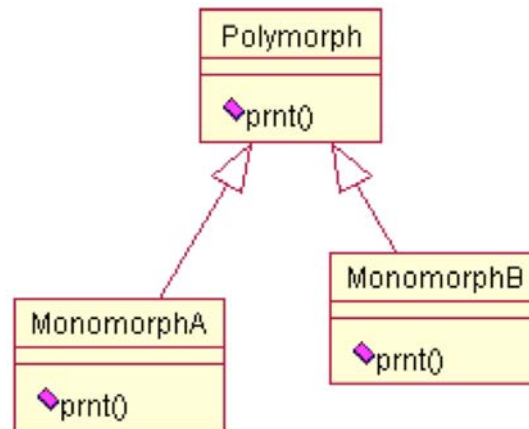
ESSI-Université de Nice Sophia  
Antipolis

# Polymorphisme

- même code d'invocation de `getNom()`
    - ✓ toujours sur un objet déclaré de type `Nom`
- ```
System.out.println(noms[i].getNom());
```
- appliqué aux objets de types différents...
    - ✓ on a un effet différent selon le constructeur appelé
      - ☐ `new NomCompleet(...)`
      - ☐ ou `new Nom(...)`



# Autre exemple (1)

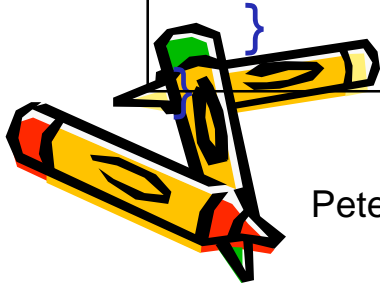
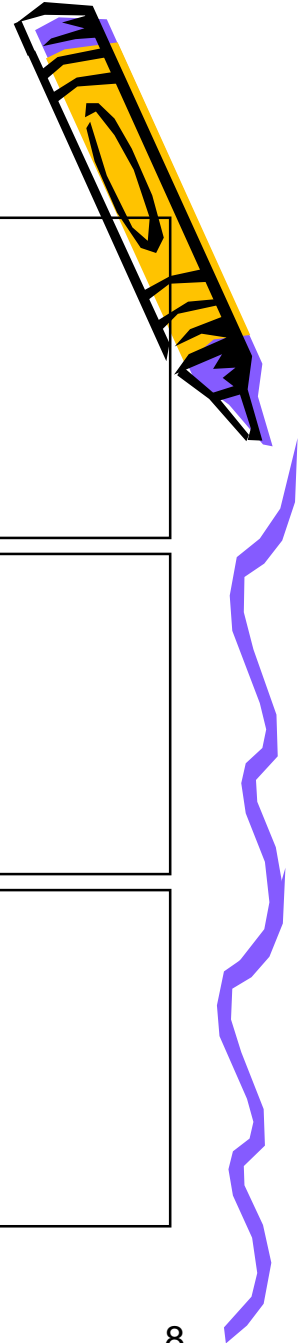


# Autre exemple (2)

```
public class Polymorph {  
    void prnt() {  
        System.out.println("poly");  
    }  
}
```

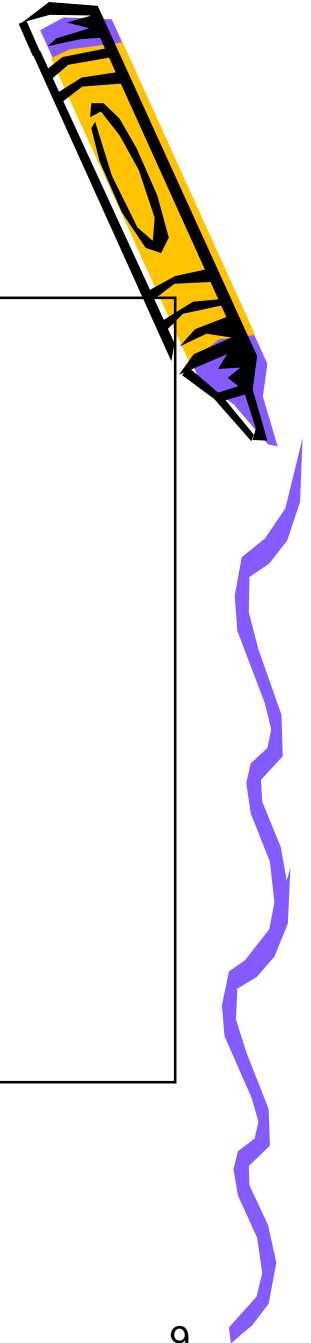
```
public class MonomorphA extends Polymorph {  
    void prnt() {  
        System.out.println("type A");  
    }  
}
```

```
public class MonomorphB extends Polymorph {  
    void prnt() {  
        System.out.println("type B");  
    }  
}
```





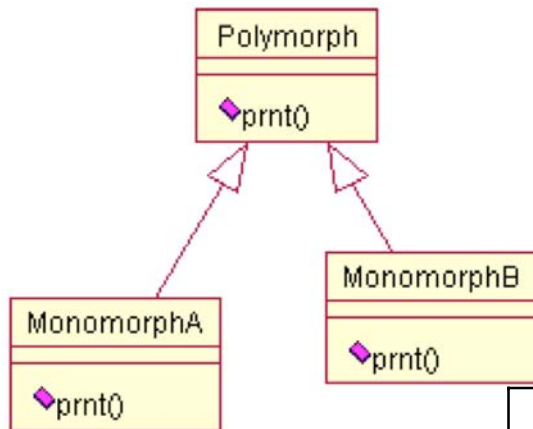
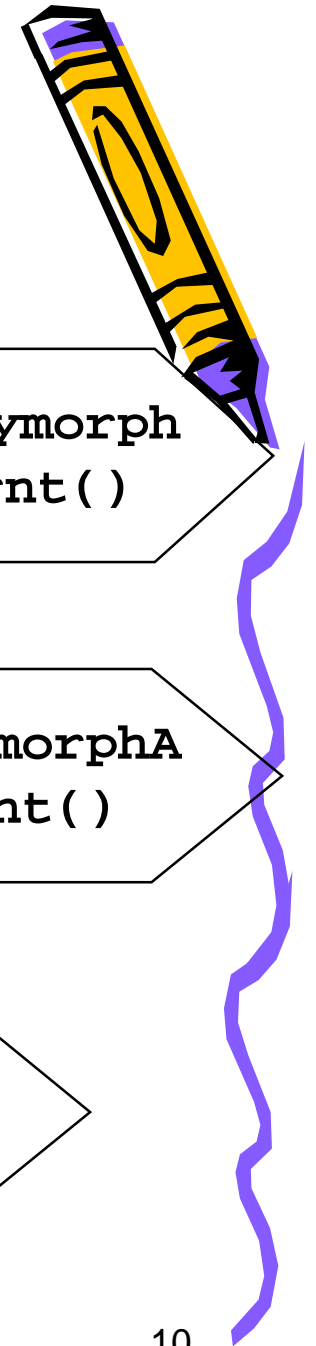
# Autre exemple (3)



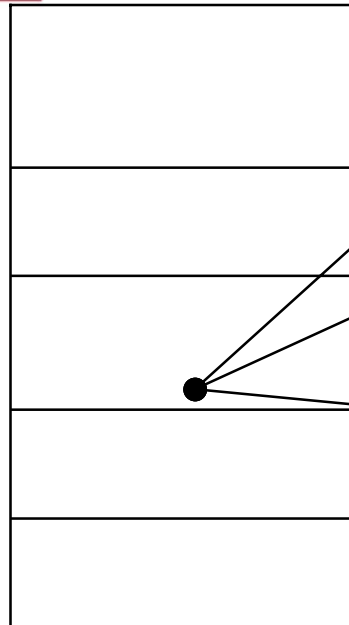
```
...  
Polymorph pm = new Polymorph();  
pm.prnt();    // Poly à la place de Poly  
pm = new MonomorphA();  
pm.prnt();    // MonoA à la place de Poly  
pm = new MonomorphB();  
pm.prnt();    // MonoB à la place de Poly  
-> poly  
    type A  
    type B
```



# Polymorphisme (1)



Polymorph  
pm



Polymorph  
print ( )

MonomorphA  
print ( )

MonomorphB  
print ( )



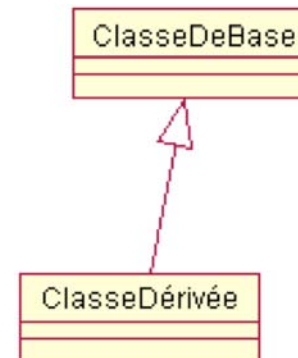
# Polymorphisme (2)



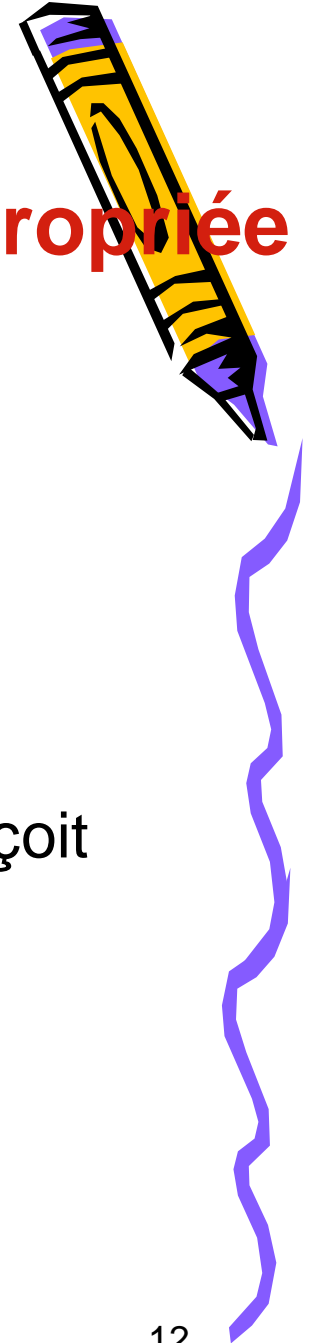
## ○ Upcasting

- faire passer une classe dérivée pour sa super-classe
- sans risque
  - toute méthode de la super-classe est aussi une méthode de la classe dérivée (par définition)

*Upcast*



# Polymorphisme (3)



## ○ Comment déterminer la méthode appropriée ?

- pendant la compilation

- ✓ le compilateur se base sur le type *déclaré*

- ☐ exemple

- `Polymorph pm;`

- pendant l'exécution

- ✓ la JVM se base sur le type *réel* de l'objet qui reçoit l'invocation

- ☐ exemple

- `pm = new MonomorphA();`

- ✓ c'est le "*late binding* » (liaison retardée)

- ☐ "*dynamic binding*" (liaison dynamique)



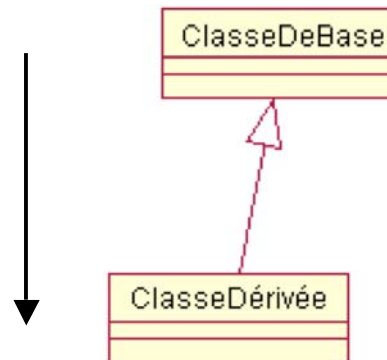
# Polymorphisme (4)



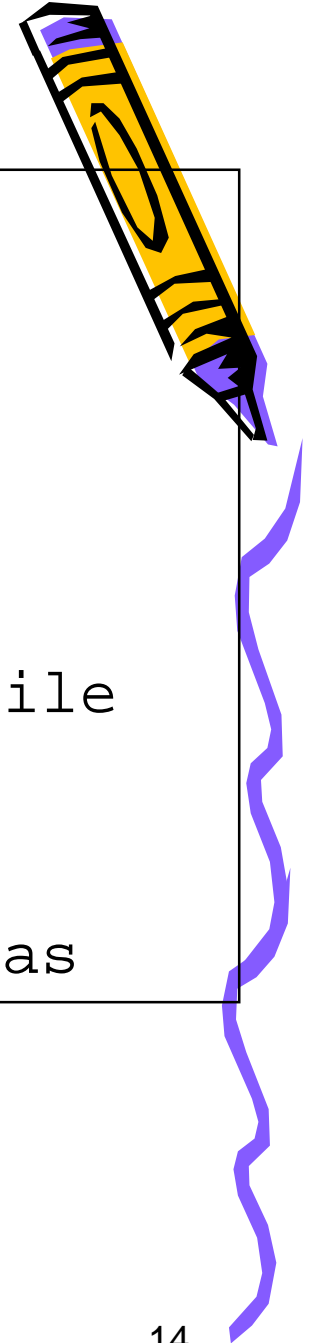
## ○Downcasting

- faire passer une super-classe pour une classe dérivée
- pas sans risque
  - ✓ la classe dérivée étend la super-classe
  - ✓ peut avoir des méthodes que la super-classe n'a pas

*Downcast*



# Polymorphisme (5)



```
public class Velo extends Vehicule {  
    ...  
    public void pedaler(...) {...}  
}  
...
```



```
Velo unVelo = new Vehicule(); // ne compile  
pas
```



```
Velo unVelo = (Velo) new Vehicule();  
unVelo.pedaler(); // méthode n'existe pas
```

- **Downcast** provoque **ClassCastException**  
à l'exécution

