

# Java

CM4-2 : Java, polymorphisme

Mickaël Martin Nevot

V1.2.0

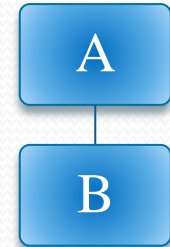


Cette œuvre de [Mickaël Martin Nevot](#) est mise à disposition selon les termes de la [licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Partage à l'Identique 3.0 non transposé](#).

# Java

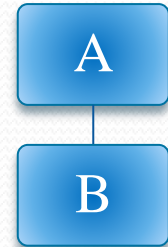
- I. Prés.
- II. POO
- III. Objet
- IV. Java
- V. Types
- VI. Héritage
- VII. Outils
- VIII. Exceptions
- IX. Polymorphisme
- X. Thread
- XI. Avancé

# Polymorphisme



- Surclassement (à la compilation) :
  - Vu comme un objet du type de la **référence**
  - Fonctionnalités restreintes à celles du type de la **référence**  
`A myObj = new B( ... );`
- Liaison dynamique (à l'exécution) :
  - Méthode de la **classe effective** de l'objet qui est exécuté  
`myObj.meth1( ... );`
- *Downcasting* :
  - Libère les fonctionnalités restreintes par le surclassement  
`((B) myObj).meth2( ... );`

# Polymorphisme



- Surclassement (à la compilation) :
  - Vu comme un objet du type de la **référence**
  - Fonctionnalités restreintes à celles du type de la **référence**

```
A myObj = new B( ... );
```

A : référence  
B : classe effective

- Liaison dynamique (à l'exécution) :
  - Méthode de la **classe effective** de l'objet qui est exécuté

```
myObj.meth1( ... );
```

meth1 (...) est une méthode de A, redéfinie par B :  
c'est celle de B qui est exécutée !

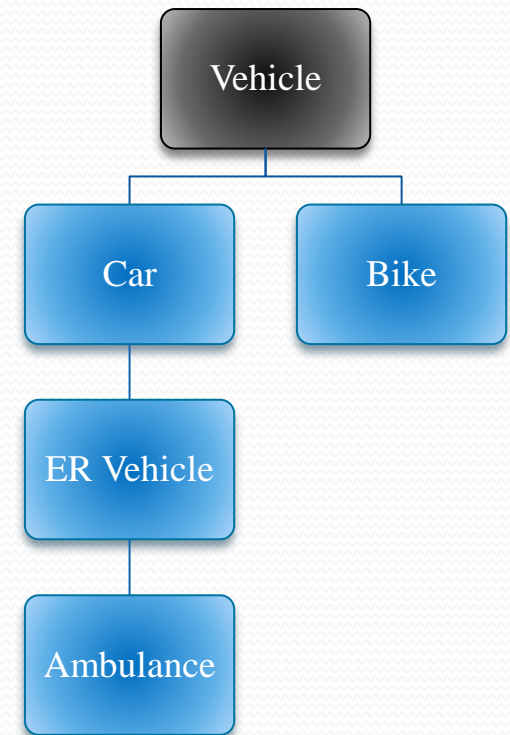
- *Downcasting* :
  - Libère les fonctionnalités restreintes par le surclassement

```
((B) myObj).meth2( ... );
```

Transtypage

# Polymorphisme

```
public class Vehicle {  
    ...  
    void move() { System.out.println("Avec deux ou quatre roues !"); }  
}  
...  
public class Bike extends Vehicle {  
    ...  
    void move() {System.out.println("Avec deux roues !"); }  
    void lean() {System.out.println("Je me penche !"); }  
}  
...  
public static void main(String[] args) {  
    Vehicle bike = new Bike( ... ); // Surclassement.  
    bike.move(); // Liaison dynamique.  
                // Affichage : Avec deux roues !  
    bike.lean(); // Erreur !  
                //(Vehicle n'a pas de méthode lean()).  
    ((Bike) bike).lean(); // Downcasting.  
                // Affichage : Je me penche !  
}
```



# Classe abstraite

- Ne peut pas être instanciée (mais constructeur[s] possible[s])
- **Si une seule méthode est abstraite, la classe l'est aussi**
- Mot clef `abstract` :

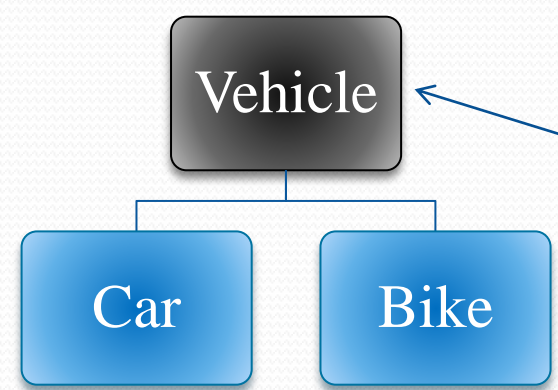
- **Classe :**

```
public abstract class MyClass { ... }
```

- **Méthode :**

```
public abstract void meth1( ... );
```

Pas de corps



La classe `Vehicle` est abstraite :  
il n'y a pas d'instance de `Vehicle` mais  
des instances de `Car` ou de `Bike`

# Interface

- Une interface **donne son type** aux classes l'implémentant

- Mot clef interface (pas abstract) :

```
public interface MyInterface { ... };
```

- Mot clef implements :

```
public class MyClass implements MyInterface1 { ... }  
public class MyClass1 implements MyInterface1, MyInterface2 ... { ... }  
public class MyClass2 extends MySuperClass implements MyInterface1 ... { ... }
```

- Les interfaces peuvent se dériver (mot clef extends)

```
public interface MyInterface2 extends MyInterface1 { ... };
```

- Méthode par défaut : ← 

```
public default void foo() {  
    System.out.println("Default implementation of foo()");  
}
```

# Classes interne/anonyme

- Classe locale ou **interne** :

```
public class MyClass {  
    ...  
    class MyLocalClass { ... }  
}
```

- Classe **anonyme** :

```
MyAnonymousClass myObj = new MyAnonymousClass() {  
    ...  
};
```

← Attention : il s'agit d'une instruction !

- *Bytecode* :

- Classe : `MyClass.class`
- Interne : `MyClass$MyLocalClass.class`
- Anonyme : `MyClass$1.class`



# Génériques

Classes typées  
à la compilation

- Polymorphisme paramétrique de type ←
- Comportement unique pour des types polymorphes
- **Un peu** comme les *templates* C++ :
  - Une seule copie du code : compilé une fois pour toutes
- Notation : `<Type1>`, `<Type2, Type3>`, etc.

```
MyClass<String> myObj = new MyClass<String>();  
  
public class MyList<B extends A, C>  
MyList<MyClass, Date> list = new MyList<MyClass, Date>();
```

# Génériques

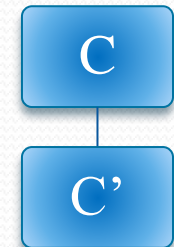
- *Wildcard* : ?

```
void myMeth1(List<? extends MyClass> a) {
    for(MyClass p : a) {
        myMeth12(p);
    }
}
```

Si C' hérite d'une classe C et G est un générique de paramètre C alors :  
il est **faux** de dire que G<C'> hérite de G<C>

- *Variance (limite de portée)* : &

```
final class MyClass<A extends Comparable<A> & Cloneable<A>,
    B extends Comparable<B> & Cloneable<B>>
    implements Comparable<MyClass<A, B>>,
        Cloneable<MyClass<A, B>> {
    ...
}
```



Il ne peut y avoir que des interfaces après le premier &

- *Méthode générique* :

```
public <T> T addTo(T e, Collection<T> c) {
    c.add(e);
    return e;
}
```

# Crédits

## Auteur

Mickaël Martin Nevot

[mmartin.nevot@gmail.com](mailto:mmartin.nevot@gmail.com)



Carte de visite électronique

## Relecteurs

Cours en ligne sur [www.mickael-martin-nevot.com](http://www.mickael-martin-nevot.com)

