
NFP121, Cnam/Paris

Cours 3-1

Cnam Paris

jean-michel Douin, douin au cnam point fr
version du 1 Octobre 2019

Notes de cours java : le langage : plusieurs classes, héritage, polymorphisme

Sommaire

- **Classe dérivée syntaxe**
- **Héritage**
- **Affectation**
- **Liaison dynamique**
 - **rappel**
- **Héritage d'interface**

Bibliographie utilisée

- The Java Handbook, Patrick Naughton. Osborne McGraw-Hill.1996.
<http://www.osborne.com>
- Thinking in Java, Bruce Eckel, <http://www.EckelObjects.com>
- <http://java.sun.com/docs/books/jls/>
- <http://java.sun.com/docs/books/tutorial.html>
- <http://hillside.net/patterns/>
- <http://www.eli.sdsu.edu/courses/spring98/cs635/notes/index.html>
- [GHJV95] DESIGN PATTERNS, catalogue de modèles de conception réutilisables, E.Gamma, R.Helm,R.Johnson et J.Vlissides. Thomson publishing.1995
- <http://www.enteract.com/~bradapp/docs/patterns-intro.html>

Concepts de l'orienté objet

- **Classe et objet (instance)**
- **Etat d'un objet et encapsulation**
- **Comportement d'un objet et méthodes**
- **Héritage**
- **polymorphisme**
 - ad'hoc
 - d'inclusion
 - paramétrique

Héritage et classification

- **définir une nouvelle classe en ajoutant de nouvelles fonctionnalités à une classe existante**
 - ajout de nouvelles fonctions
 - ajout de nouvelles données
 - redéfinition de certaines propriétés héritées
- **Classification en langage naturel**
- **les carrés sont des polygones réguliers**
- **les polygones réguliers sont des objets Java**
 - (en java `java.lang.Object` est la racine de toutes classe)

Polymorphisme, déjà vu

- **Polymorphisme ad'hoc**
 - surcharge
- **Polymorphisme d'inclusion**
 - est fondé sur la relation d'ordre partiel entre les types, relation induite par l'héritage. si le type B est inférieur selon cette relation au type A alors on peut passer un objet de type B à une méthode qui attend un paramètre de type A, le choix de la méthode est résolu dynamiquement en fonction du type de l'objet receveur
- **Polymorphisme paramétrique**
 - généricité
 - extrait de M Baudouin-Lafon. La Programmation Orientée Objet. ed. Armand Colin

Classe : Syntaxe

- **class** NomDeClasse **extends** NomDeLaSuperClasse{
 type variableDeClasse1;
 type variableDeClasse2;
 type variableDeClasseN;

 type nomDeMethodeDeClasse1(listeDeParametres) {

 }
 type nomDeMethodeDeClasse2(listeDeParametres) {

 }
 type nomDeMethodeDeClasseN(listeDeParametres) {

 }
• }
}

Héritage exemple

Les polygones réguliers sont des "objects java"

- class PolygoneRegulier extends java.lang.Object{}

Les carrés sont des polygones réguliers

- class Carre **extends** PolygoneRegulier{ ...}

Les carrés en couleur sont des carrés

- class CarreEnCouleur **extends** Carre {.....}

Héritage

- **Les instances des classes dérivées effectuent :**
 - **Le cumul des données d'instance,**
 - **Le "cumul" du comportement,**

- le comportement des instances issu de la classe dérivée dépend de :

**La surcharge des méthodes (la signature est différente)
et du masquage des méthodes (la signature est identique)**

Exemple la classe Carre

- **class Carre extends PolygoneRegulier{**

- *// pas de champ d'instance supplémentaire*

```
Carre( int longueur){  
    nombreDeCotes = 4;  
    longueurDuCote = longueur;  
}
```

// masquage de la méthode PolygoneRegulier.surface()

```
int surface(){  
    return longueurDuCote* longueurDuCote;  
}
```

```
String toString(){  
    return "<4,"+ longueurDuCote +">";  
}
```

- **}**

super

- Appel d'une méthode de la super classe
- Appel du constructeur de la super classe

- `class Carre extends PolygoneRegulier{`

```
    Carre( int longueur){  
        super(4,longueur);  
    }
```

```
    int surface(){  
        return super.surface();  
    }
```

Création d'instances et affectation

- **Création d'instances et création**
 - Carre c1 = new Carre(100);
 - Carre c2 = new Carre(10);
 - PolygoneRegulier p1 = new PolygoneRegulier(4,100);
- **Affectation**
 - c1 = c2; *// synonymie, c2 est un autre nom pour c1*
- **Affectation polymorphe**
 - p1 = c1;
- **Affectation et changement de classe**
 - c1 = (Carre) p1; *// Hum, Hum ...*

Liaison dynamique

- **Sélection de la méthode en fonction de l'objet receveur**
- **type déclaré / type constaté à l'exécution**

// classe déclarée

- **PolygoneRegulier p1 = new PolygoneRegulier(5,100);**
- **Carre c1 = new Carre(100);**
- **int s = p1.surface();** *// la méthode surface() de PolygoneRegulier*
- **p1 = c1;** *// affectation polymorphe*
- **s = p1.surface();** *// la méthode surface() de Carre est sélectionnée*
- la recherche de la méthode s'effectue uniquement dans l'ensemble des méthodes masquées associé à la classe dérivée

- The **apparent type** of a variable is the type understood by the compiler from information available in declarations.
- The **actual type** of an Object is its real type -> the type it receives when it is created.

Ces notes de cours utilisent

- type *déclaré* pour **apparent type** et
- type *constaté* pour **actual type**

```
• class A{
•     void m(A a){ System.out.println(" m de A"); }
•     void n(A a){ System.out.println(" n de A"); }
• }

• public class B extends A{

•     public static void main(String args[]){
•         A a = new B();
•         B b = new B();

•         a.m(b);
•         a.n(b);
•     }

•     void m(A a){ System.out.println(" m de B"); }
•     void n(B b){ System.out.println(" n de B"); }
• }
```

- **Exécution de B.main : Quelle est la trace d'exécution ?**

•m de B
•n de A

En pratique : une explication

- **mécanisme de liaison dynamique en Java :**
 - **La liaison dynamique effectue la sélection d'une méthode en fonction du type constaté de l'objet receveur, la méthode doit appartenir à l'ensemble des méthodes masquées,**
 - **(la méthode est masquée dans l'une des sous-classes, si elle a exactement la même signature)**

Sémantique opérationnelle

- Sur l'exemple,
 - nous avons uniquement dans la classe B la méthode `m(A a)` masquée
- en conséquence :
- `A a = new B();` *// a est déclarée A, mais constatée B*
- `a.m` --> sélection de `((B)a).m(...)` car `m` est bien masquée
- `a.n` --> sélection de `((A)a.n(...))` car `n` n'est pas masquée dans B

Choix d'implémentation de Java : compromis vitesse d'exécution / sémantique ...

Et pourtant ...

- **Il était concevable ...**
 - **d'afficher**
 - **m de B**
 - **n de B**
 - **-> Recherche à l'exécution de la « meilleure » signature, pour chaque paramètre le « meilleur » type Coûteux, très coûteux ...**
 - **discussion**
 - **Ce n'est pas le comportement à l'exécution de Java**

- **Rappel**

- La méthode est masquée dans l'une des sous-classes, si elle a exactement la même signature
 - *Déjà vu à plusieurs reprises...*

- **Mais**

- Nous avons la covariance en Java :
 - <http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf>
 - Page 220, paragraphe 8.4.5
- Qu'est-ce que c'est ?

Redéfinition : le retour ... de fonction

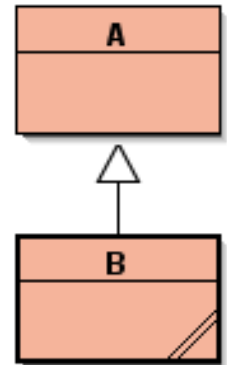
- **Hypothèses**

- Deux classes A et B, avec B extends A

- Une fonction `public A f(){return ...;}` définie dans A
 - Cette fonction est redéfinie dans B

Alors

- `A a = new B();`
 - `A a1 = a.f();` // redéfinition, la fonction f de la classe B est appelée



- **Affectation polymorphe**

- `A a1 = a.f();` // a1 pourrait recevoir n'importe quelle instance
// d'une classe fille de A

- ***Discussions***

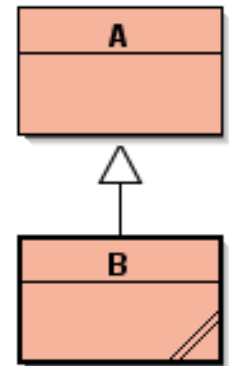
Le type de retour d'une méthode redéfinie

- `A a = new B();`
- `A a1 = a.f();` `// a1 peut recevoir n'importe quelle instance d'une`
 `// classe fille de A,`

• Covariance :

Le type de retour de la méthode redéfinie peut être un sous-type de celui de la classe mère

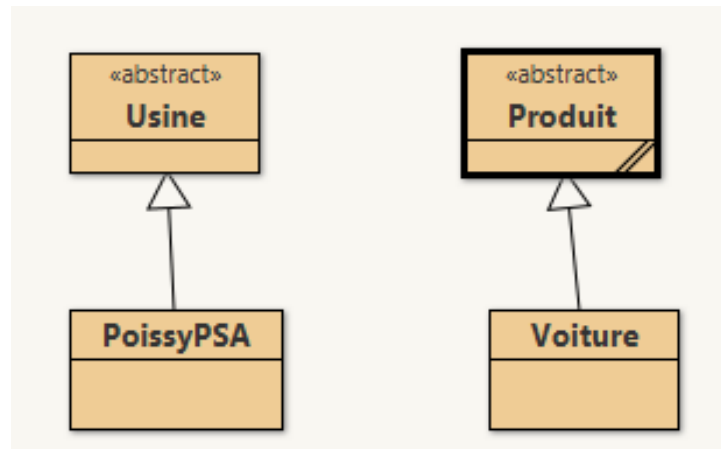
```
public class A{  
    A f(){ return new A();}  
}  
public class B extends A{  
    B f(){ return new B(); }  
}
```



Exemple une fabrique de produits

```
3 public abstract class Usine{  
4     public abstract Produit fabrique();  
5 }
```

```
3 public abstract class Produit{  
4  
5 }
```



```
3 public class PoissyPSA extends Usine{  
4  
5     public Voiture fabrique(){  
6         return new Voiture();  
7     }  
8 }
```

```
3 public class Voiture extends Produit{  
4  
5 }
```

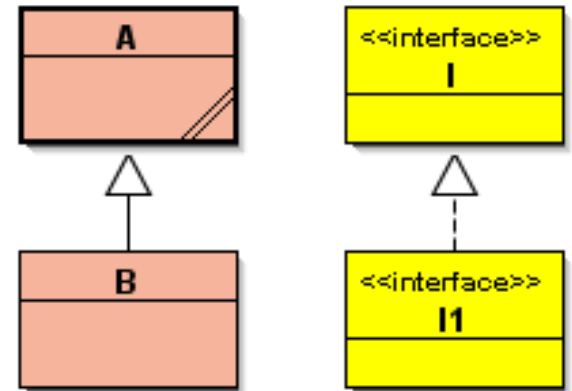
- Démonstration ...

Les interfaces idem

```
public class A{  
    I i;  
    I f(){ return i;}  
}
```

```
public class B extends A{  
    I1 i1;  
    public I1 f(){return i1;}  
}
```

```
A a = new B();  
I i = a.f();
```



retour sur la classe incomplète dite abstraite

- Une classe partiellement définie, dans laquelle certaines méthodes sont laissées à la responsabilité des sous-classes
- pas de création d'instances possible,
- Affectation possible d'une référence de classe incomplète par une instance de classe dérivée
- la classe dérivée reste abstraite si toutes les implémentations ne sont pas effectuées

- exemple :

```
abstract class Figure {  
    ...  
}  
class Losange extends Figure {  
}
```

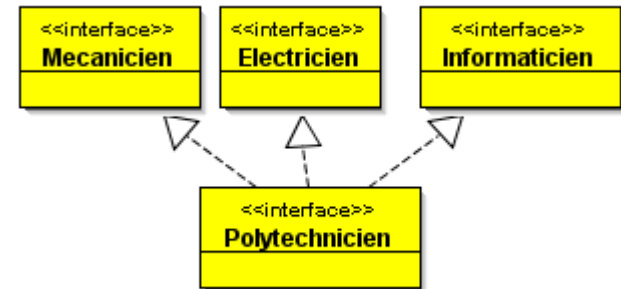
```
Figure f = new Losange(); ..
```


Interface et héritage

- *interface I extends I1,I2,I3 { }*
- `public interface Transformable extends Scalable, Rotateable, Reflectable{}`
- `public interface DrawingObject extends Drawable, Transformable{}`
- `public class Shape implements DrawingObject{....}`
- **interface comme marqueur**
- `public interface java.lang.Cloneable{ /** vide */ }`
 - --> **usage de clone possible, si la classe implémente cet interface**
- `public interface java.io.Serializable{ /** vide */ }`
 - --> **les instances pourront être sérialisées..**
 - test de la bonne implémentation par **instanceof**

Exemple : fred le polytechnicien

- Polytechnicien fred = ...
- Mecanicien m = fred;
- Electricien e = fred;
- Informaticien i = fred;
- fred sait tout faire ...
- **Démonstration, Discussions**



Annexes syntaxiques

Présentation optionnelle... *à la demande* ?

- **Les classes internes**
- **Les paquetages**

Classes internes et statiques (niveau 0)

```
package tp7.q3;
public class Exemple3{
    private static Object obj = new Object();

    public static class InterneEtStatique{
        public void methode(){
            Object o = Exemple3.obj;
            new Exemple3().methode();
            o = new Exemple3.InterneEtStatique();
            o = new Exemple3();
        }
    }
    public void methode(){
        InterneEtStatique is = new InterneEtStatique();
        is.methode();
    }
}
```

Exemple3\$InterneEtStatique.class
Exemple3.class

Interface internes -> niveau 0

```
package tp7.q3;
```

```
public class Exemple4{
```

```
    private static Object obj = new Object();
```

```
    public interface Exemple4I{
```

```
        public void methode();
```

```
    }
```

```
}
```

```
Exemple4$Exemple4I.class
```

```
Exemple4.class
```

Classes internes et membres

```
package tp7.q3;  
public class Exemple5{  
    private Object obj = new Object();
```

```
    public class InterneEtMembre{  
        public void methode(){  
            obj = null;  
            Exemple5.this.obj = null;  
        }  
    }
```

```
    public void methode(){  
        this.new InterneEtMembre();  
    }  
}
```

```
Exemple5$InterneEtMembre.class  
Exemple5.class
```

Classes internes et membres !

```
package tp7.q3;

public class Exemple5{
    private Object obj = new Object();
    public class InterneEtMembre{
        private Object obj = new Object();
        public class InterneEtMembre2{
            private Object obj = new Object();
            public void methode(){
                Object o = this.obj;
                o = Exemple5.this.obj;
                o = InterneEtMembre.this.obj;
            }
        }
    }
    public void methode(){
        this.new InterneEtMembre();
        this.new InterneEtMembre().new InterneEtMembre2();
        Exemple5 e = new Exemple5();
        Exemple5.InterneEtMembre e1 = e.new InterneEtMembre();
    }
}
```

Exemple5\$InterneEtMembre\$InterneEtMembre2.class

Classes internes et anonymes

```
import java.awt.Button;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class Exemple6{
    public void methode(){
        Button b = new Button("b");
        b.addActionListener( new ActionListener(){
            public void actionPerformed(ActionEvent ae){
                }
        });
    }
}
```

Exemple6\$1.class

Exemple6.class

Classes internes et locales

```
import java.util.Iterator;
public class Exemple7{
    public Iterator methode() {

        class Locale implements Iterator{
            public boolean hasNext(){ return true;}
            public Object next(){return null;}
            public void remove(){}
        }
        return new Locale();
    }
}
```

Exemple7\$1\$Locale.class

Exemple7.class

Une petite dernière, enfin !

```
import java.util.Iterator;
public class Exemple8{
    public Iterator methode() {

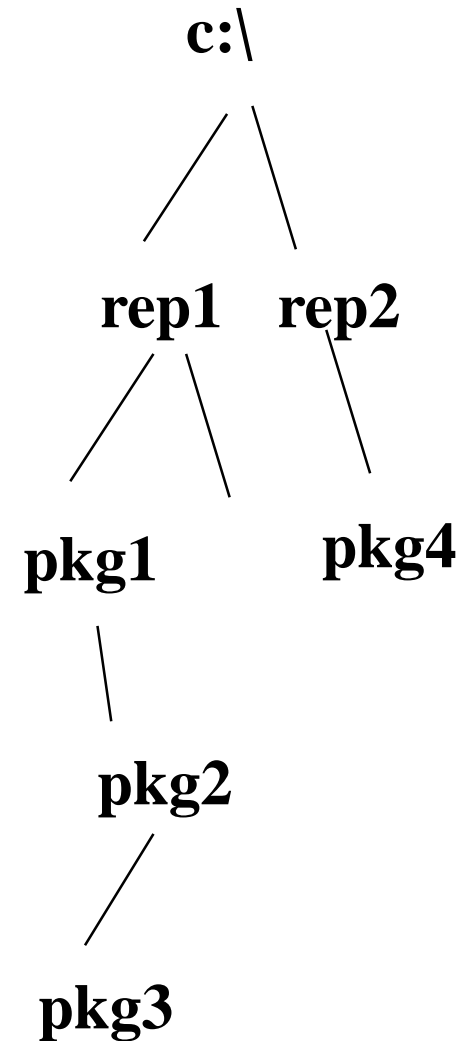
        return new Iterator() {
            public boolean hasNext() { return true;}
            public Object next(){return null;}
            public void remove(){}
        }
    }
}
```

Exemple8\$1.class

Exemple8.class

Package, bis

- **Fonction**
 - Unité logique par famille de classes
 - découpage hiérarchique des paquetages
 - (ils doivent être importés explicitement sauf java.lang)
- **Buts**
 - espace de noms
 - restriction visibilité
- **Instructions**
 - *package pkg1[.pkg2[.pkg3];*
 - les noms sont en minuscules
 - c'est la première instruction du source java
 - *import pkg1[.pkg2[.pkg3].(nomdeclasse/*);*
 - liés aux options de la commande de compilation
 - dos> javac -classpath .;c:\rep1;c:\rep2



Exemple

```
package tp7.q1;
```

```
public class Exemple{
```

```
    public void m(){
```

```
        System.out.println(" methode m");
```

```
    }
```

```
}
```

```
DOS> javac -classpath . tp7/q1/Exemple.java
```

Exemple

```
package tp7.q1;
```

```
public class Exemple{
```

```
    public void m(){
```

```
        System.out.println(" methode m");
```

```
    }
```

```
}
```

```
DOS> javac -classpath . tp7/q1/Exemple.java
```

Exemple2

```
package tp7.q2;
```

```
import tp7.q1.Exemple;
```

```
public class Exemple2{  
    public void methode(){  
    }  
}
```

```
DOS> javac -classpath . tp7/q2/*.java
```

Exemple jar, compressés !!

```
DOS> jar cvf Exemple.jar tp7/q1/*.class
```

```
DOS> javac -classpath Exemple.jar tp7/q2/*.java
```

// un doute → `-verbose`

```
DOS>javac -classpath Exemple.jar -verbose tp7/q2/*.java
```

```
[parsing started tp7/q2/Exemple2.java]
```

```
[parsing completed 160ms]
```

```
[loading Exemple.jar(tp7/q1/Exemple.class)]
```

```
[checking tp7.q2.Exemple2]
```

```
[loading
```

```
  c:\j2sdk1.4.0_01\jre\lib\rt.jar(java/lang/Object.class)]
```

```
[wrote tp7\q2\Exemple2.class]
```

```
[total 701ms]
```

Compilation javac

DOS>javac -verbose -classpath Exemple.jar -bootclasspath c:\j2sdk1.4.0_01\jre\lib\rt.jar tp7/q2/*.java

[parsing started tp7/q2/Exemple2.java]

[parsing completed 160ms]

[loading Exemple.jar(tp7/q1/Exemple.class)]

[checking tp7.q2.Exemple2]

[loading c:\j2sdk1.4.0_01\jre\lib\rt.jar(java/lang/Object.class)]

[wrote tp7\q2\Exemple2.class]

[total 681ms]

Paquetages prédéfinis

- **le paquetage `java.lang.*` est importé implicitement**
 - ce sont les interfaces : *Cloneable*, *Comparable*, *Runnable*
 - et les classes : *Boolean*, *Byte*, *Character*, *Class*, *ClassLoader*, *Compiler*,
Double, *Float*, *InheritableThreadLocal*, *Long*, *Math*, *Number*,
Object, *Package*, *Process*, *Runtime*, *RuntimePermission*,
SecurityManager, *Short*, *StrictMath*, *String*, *StringBuffer*,
System, *Thread*, *ThreadGroup*, *ThreadLocal*,
Throwable, *Void*,
 - toutes les classes dérivées de *Exception*, *ArithmeticException*,
 - et celles de *Error*, *AbstractMethodError*,
- **`java.awt.*` `java.io.*` `java.util.*` **
- **→ documentation du `j2sdk`**

Règle d'accès

	private	défaut	protected	public
même classe	oui	oui	oui	oui
même paquetage et sous-classe	non	oui	oui	oui
même paquetage et classe indépendante	non	oui	oui	oui
paquetages différents et sous-classe	non	non	oui ...	oui
paquetages différents et classe indépendante	non	non	non	oui

Héritage et cumul du comportement

- Une application Android ...
- **public class Main extends Activity{**

```
public void onStart(){  
    super.onStart();  
    ...  
}
```

```
public void onPause(){  
    ...  
    super.onPause();  
    ...  
}
```