Extension dynamique et réflexion

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1

Extension dynamique

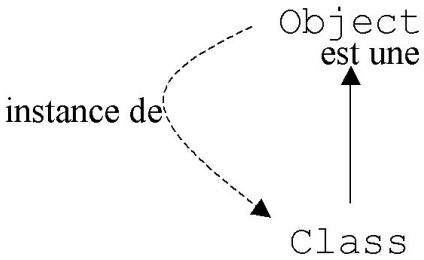
Java offre la possibilité d'avoir des programmes qui s'étendent dynamiquement.
Cela permet un programme P de charger des classes qui n'existaient pas au moment de la programmation/compilation de P et de créer et manipuler des instances de ces classes.

Réflexion

Avec Java il est possible dans un programme de manipuler et d'analyser dynamiquement les objets du programme eux-mêmes

introspection

Schéma objet méta-circulaire



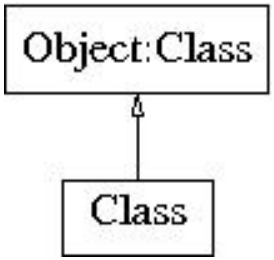
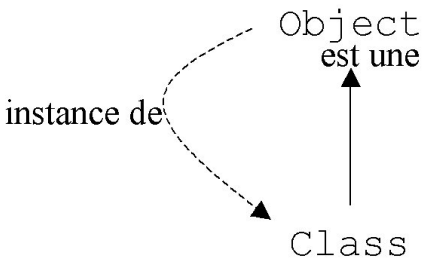
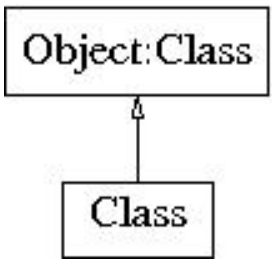


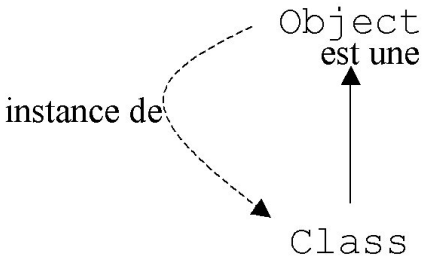
Schéma objet méta-circulaire

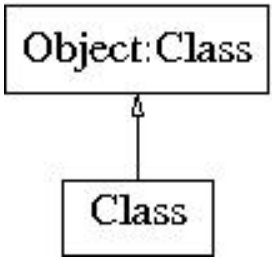




Classe
nom : String # mesAttributs : Map<String,Attribut<?>> # mesMethodes : Map<String,Methode>
+Classe(nom : String, attributs : Collection<Attribut<?>>, methodes : Collection<Methode>) +creerInstance(args : Object...) : Object +getMethode(nom : String) : Methode +getAttribut(nom : String) : Attribut<?> ...

Schéma objet méta-circulaire





Classe
nom : String # mesAttributs : Map<String,Attribut<?>> # mesMethodes : Map<String,Methode>
+Classe(nom : String, attributs : Collection<Attribut<?>>, methodes : Collection<Methode>) +creerInstance(args : Object...) : Object +getMethode(nom : String) : Methode +getAttribut(nom : String) : Attribut<?> ...

avec (très approximativement)

Methode
nom : String ...
+invoquer(instance : Object, args : Object...)

Attribut<T>
nom : String # value : T
+getValeur() : T +setValeur(value : T)

- les classes sont des instances de `Class` et donc des objets
  ↪ possèdent attributs : les attributs des instances, leurs méthodes, etc.
  ↪ possèdent méthodes (on peut donc leur envoyer des messages)
  - création d'instances,
  - connaître la valeur de ses attributs
  - appeler des méthodes
- la classe `Object` est une classe et donc une instance de `Class`, elle peut donc être manipulée aussi

- les classes sont des instances de `Class` et donc des objets
  ↪ possèdent attributs : les attributs des instances, leurs méthodes, etc.
  ↪ possèdent méthodes (on peut donc leur envoyer des messages)
  - création d'instances,
  - connaître la valeur de ses attributs
  - appeler des méthodes
- la classe `Object` est une classe et donc une instance de `Class`, elle peut donc être manipulée aussi

- en quelque sorte (pas tout à fait en Java) :

```
class UneClasse { ...} ≡ Class UneClasse = new Class(...);
```

Extension dynamique

- 2 manières de charger (dynamiquement) une classe en mémoire :
  - `Class.forName(String)`
  - `loadClass(String)` dans `ClassLoader` (sous-classes de)

puis création dynamique d'instances

`newInstance(...)`

Extension dynamique

- 2 manières de charger (dynamiquement) une classe en mémoire :
  - `Class.forName(String)`
  - `loadClass(String)` dans `ClassLoader` (sous-classes de)

puis création dynamique d'instances

`newInstance(...)`

- `public static Class<?> forName(String className) throws ClassNotFoundException`
- `public T newInstance() throws InstantiationException, IllegalAccessException`

```
package essais.dynamic;
interface Bidule {
   public void doIt();
}
class Truc implements Bidule {
   public void doIt() { System.out.println("bidule truc"); }
}
class Machin implements Bidule {
   public void doIt() { System.out.println("bidule machin"); }
}
public class TestForName {
   public static void main(String[] args)
         throws ClassNotFoundException,IllegalAccessException,
                                 InstantiationException {
      Class c = Class.forName("essais.dynamic."+args[0]);
      Bidule b = (Bidule) c.newInstance();
      b.doIt();
   }
} // TestForName
```

```
+---------------------------------------
| java TestForName Truc --> bidule truc
| java TestForName Machin --> bidule machin
```

Introspection

- dans `Object` : méthode

```
public Class<?> getClass()
```

renvoie l'objet représentant la classe dont `this` est instance.

Introspection

- dans `Object` : méthode

  `public Class<?> getClass()`

  renvoie l'objet représentant la classe dont `this` est instance.

- à partir de là accès à (paquetage `java.lang.reflect`) :
  - constructeurs `Constructor`
  - attributs `Field`
  - méthodes `Method`

Introspection

- dans `Object` : méthode

  `public Class<?> getClass()`

  renvoie l'objet représentant la classe dont `this` est instance.

- à partir de là accès à (paquetage `java.lang.reflect`) :
  - constructeurs `Constructor`
  - attributs `Field`
  - méthodes `Method`

- Accès "statique" à l'objet `Class` représentant une classe :

  `nomClasse``.class` exemple : `String.class`, `Carotte.class`

- il existe un objet `Class` identifiant les types primitifs.

  `<wrapperClasse>.TYPE`

  exemple : `Integer.TYPE` pour `int`, `Character.TYPE` pour `char`, etc.
  à noter, l'existence d'une classe `Void` (et donc de `Void.TYPE`).

Informations accessibles.

Méthodes de `java.lang.Class<T>` (non exhaustif)

- nom de classe (type) (`getName()`)

Informations accessibles.

Méthodes de `java.lang.Class<T>` (non exhaustif)

- nom de classe (type) (`getName()`)
- super classe (`getSuperClass()`)

Informations accessibles.

Méthodes de `java.lang.Class<T>` (non exhaustif)

- nom de classe (type) (`getName()`)
- super classe (`getSuperClass()`)
- paquetage (`getPackage()`) (objet `Package`)

Informations accessibles.

Méthodes de `java.lang.Class<T>` (non exhaustif)

- nom de classe (type) (`getName()`)
- super classe (`getSuperClass()`)
- paquetage (`getPackage()`) (objet `Package`)
- modificateurs de classes (`public`, `abstract`) (`getModifiers()`)

Informations accessibles.

Méthodes de `java.lang.Class<T>` (non exhaustif)

- nom de classe (type) (`getName()`)
- super classe (`getSuperClass()`)
- paquetage (`getPackage()`) (objet `Package`)
- modificateurs de classes (`public`, `abstract`) (`getModifiers()`)
- le nom des interfaces implémentées (`getInterfaces()`)

Informations accessibles.

Méthodes de `java.lang.Class<T>` (non exhaustif)

- nom de classe (type) (`getName()`)
- super classe (`getSuperClass()`)
- paquetage (`getPackage()`) (objet `Package`)
- modificateurs de classes (`public`, `abstract`) (`getModifiers()`)
- le nom des interfaces implémentées (`getInterfaces()`)
- les attributs (`get[Declared]Field[s]()`) – objet `Field` : nom, type, modificateur

Informations accessibles.

Méthodes de `java.lang.Class<T>` (non exhaustif)

- nom de classe (type) (`getName()`)
- super classe (`getSuperClass()`)
- paquetage (`getPackage()`) (objet `Package`)
- modificateurs de classes (`public`, `abstract`) (`getModifiers()`)
- le nom des interfaces implémentées (`getInterfaces()`)
- les attributs (`get[Declared]Field[s]()`) – objet `Field` : nom, type, modificateur
- les constructeurs (`get[Declared]Constructor[s]()`) – objet `Constructor`

Informations accessibles.

Méthodes de `java.lang.Class<T>` (non exhaustif)

- nom de classe (type) (`getName()`)
- super classe (`getSuperClass()`)
- paquetage (`getPackage()`) (objet `Package`)
- modificateurs de classes (`public`, `abstract`) (`getModifiers()`)
- le nom des interfaces implémentées (`getInterfaces()`)
- les attributs (`get[Declared]Field[s]()`) – objet `Field` : nom, type, modificateur
- les constructeurs (`get[Declared]Constructor[s]()`) – objet `Constructor`
- les méthodes (`get[Declared]Method[s]()`)

Informations accessibles.

Méthodes de `java.lang.Class<T>` (non exhaustif)

- nom de classe (type) (`getName()`)
- super classe (`getSuperClass()`)
- paquetage (`getPackage()`) (objet `Package`)
- modificateurs de classes (`public`, `abstract`) (`getModifiers()`)
- le nom des interfaces implémentées (`getInterfaces()`)
- les attributs (`get[Declared]Field[s]()`) – objet `Field` : nom, type, modificateur
- les constructeurs (`get[Declared]Constructor[s]()`) – objet `Constructor`
- les méthodes (`get[Declared]Method[s]()`)
  - objet `Method` : nom, type de retour, type args, exception, modificateurs, ...

Informations accessibles.

Méthodes de `java.lang.Class<T>` (non exhaustif)

- nom de classe (type) (`getName()`)
- super classe (`getSuperClass()`)
- paquetage (`getPackage()`) (objet `Package`)
- modificateurs de classes (`public`, `abstract`) (`getModifiers()`)
- le nom des interfaces implémentées (`getInterfaces()`)
- les attributs (`get[Declared]Field[s]()`) – objet `Field` : nom, type, modificateur
- les constructeurs (`get[Declared]Constructor[s]()`) – objet `Constructor`
- les méthodes (`get[Declared]Method[s]()`)
  - objet `Method` : nom, type de retour, type args, exception, modificateurs, ...
- `isInstance(Object)`, `isAssignableFrom(Class)`, `isInterface`

Actions

- créer une instance : `newInstance()`
  ou utiliser les constructeurs :
  `Constructor : newInstance(Object... initargs)`

Actions

- créer une instance : `newInstance()`
  ou utiliser les constructeurs :
  `Constructor` : `newInstance(Object... initargs)`
- récupérer une valeur d'attribut : `Field` : `get(Object)`

Actions

- créer une instance : `newInstance()`
  ou utiliser les constructeurs :
  `Constructor` : `newInstance(Object... initargs)`
- récupérer une valeur d'attribut : `Field` : `get(Object)`
- invoquer une méthode : `Method` : `invoke(Object, Object...)`

Création d'un instance

```
package counter;
public class SimpleCounter implements Counter {
   public SimpleCounter(int v) { ...  }
   ...
}
```

```
package counter;
public class TestCounter {
   public static void main(String[] args)
          throws ClassNotFoundException,IllegalAccessException, InstantiationException {
      Class counterClass = Class.forName("counter."+args[0]);
      Constructor construct = counterClass.getConstructor(Integer.TYPE);
      Counter aCounter = construct.newInstance(new Integer(args[1]));
      new GraphicalCounter(aCounter);
   }
}
```

```
utilisation :
...> java TestCounter SimpleCounter 5
...> java TestCounter AnotherCounter 12
```

Invoquer une méthode

```
import java.lang.reflect.Method;

Class c = ...
Method meth = c.getMethod(nom_méthode, les classes des params /*Class...*/ });
Object result = meth.invoke(obj,{ les paramètres } /*Object...*/);
        // obj est une référence vers une instance de type ''c''
        // ''Object result ='' seulement si méthode avec résultat...
```

le paramètre *obj* est ignoré si la méthode est statique

NB : `IllegalAccessException,IllegalArgumentException, InvocationTargetException`

Invoquer une méthode

```
import java.lang.reflect.Method;

Class c = ...
Method meth = c.getMethod(nom_méthode, les classes des params /*Class...*/ });
Object result = meth.invoke(obj,{ les paramètres } /*Object...*/);
        // obj est une référence vers une instance de type ''c''
        // ''Object result ='' seulement si méthode avec résultat...
```

le paramètre *obj* est ignoré si la méthode est statique

NB : `IllegalAccessException,IllegalArgumentException, InvocationTargetException`

```
ArrayList list = new ArrayList();
        //invocation de :  "list.add(0,"premier")"
Class c = java.util.ArrayList.class;
Method meth = c.getMethod("add", Integer.TYPE, Object.class );
meth.invoke(list, 0, "premier");                          +------------
System.out.println(list.size()+" -> "+list.get(0));       |
```

Invoquer une méthode

```
import java.lang.reflect.Method;

Class c = ...
Method meth = c.getMethod(nom_méthode, les classes des params /*Class...*/ });
Object result = meth.invoke(obj,{ les paramètres } /*Object...*/);
        // obj est une référence vers une instance de type ''c''
        // ''Object result ='' seulement si méthode avec résultat...
```

le paramètre *obj* est ignoré si la méthode est statique

NB : `IllegalAccessException,IllegalArgumentException, InvocationTargetException`

```
ArrayList list = new ArrayList();
        //invocation de :  "list.add(0,"premier")"
Class c = java.util.ArrayList.class;
Method meth = c.getMethod("add", Integer.TYPE, Object.class );
meth.invoke(list, 0, "premier");                          +------------
System.out.println(list.size()+" -> "+list.get(0));       | 1->premier
```

Accéder à un attribut

```
import java.lang.reflect.Field;

Class c = ...
Field attribut = c.getField(nom_attribut);
Object value = attribut.get(obj);
         // obj est une référence vers une instance de type ''c''
         // getInt(Object), getBoolean(Object), etc.  existent pour
         // attributs à valeur dans types primitifs int, boolean, etc.
```

le paramètre *obj* est ignoré si la méthode est statique

NB : `IllegalArgumentException`, `IllegalAccessException`

Accéder à un attribut

```
import java.lang.reflect.Field;

Class c = ...
Field attribut = c.getField(nom_attribut);
Object value = attribut.get(obj);
        // obj est une référence vers une instance de type ''c''
        // getInt(Object), getBoolean(Object), etc.  existent pour
        // attributs à valeur dans types primitifs int, boolean, etc.
```

le paramètre *obj* est ignoré si la méthode est statique

NB : `IllegalArgumentException`, `IllegalAccessException`

```
Livre sda = new Livre("Le seigneur des Anneaux");
        // accès à :  "sda.titre"
Class c = Livre.class;
Field attribut = c.getField("title");
Object value = attribut.get(sda);            // si accessible !
                                             +------------
System.out.println("-> "+value);             |
```

Accéder à un attribut

```
import java.lang.reflect.Field;

Class c = ...
Field attribut = c.getField(nom_attribut);
Object value = attribut.get(obj);
        // obj est une référence vers une instance de type ''c''
        // getInt(Object), getBoolean(Object), etc.  existent pour
        // attributs à valeur dans types primitifs int, boolean, etc.
```

le paramètre *obj* est ignoré si la méthode est statique

NB : `IllegalArgumentException`, `IllegalAccessException`

```
Livre sda = new Livre("Le seigneur des Anneaux");
        // accès à :  "sda.titre"
Class c = Livre.class;
Field attribut = c.getField("title");
Object value = attribut.get(sda);            // si accessible !
                                             +------------
System.out.println("-> "+value);             | -> Le Seigneur des Anneaux
```

Fichier avec information de typage pour faciliter l'écriture de Factory (peut imposer convention de nommage) :

```
Text                    (pour classe TextAnswer)
Quel est le nom de l'auteur du Seigneur des Anneaux ?
Tolkien
1
YesNo                   (pour classe YesNoAnswer)
Frodo est un Hobbit ?
vrai
1
Numeric                 (pour classe NumericAnswer)
Combien de membres composent la Compagnie de l'Anneau ?
9
2
```

```
public Question litEtCreeQuestion() {
   String typeAnswer = in.readLine();
   ...  lecture text, texteReponse et nbPoints;
   Class c = class.forName(typeAnswer+"Answer");
   Constructor constructor = c.getConstructor(String.class);
   Answser answer = constructor.newInstance( texteReponse);
   Question q = new Question(text, answer, nbPoints);
   return q;
}
```

```
Class c = Class.forName("YesNoAnswer");
Constructor constructor = c.getConstructor( String.class);
Answer answer = constructor.newInstance( "vrai" );
Question q = new Question("texte question", answer, 1);
Class qClass = Question.class; // ou Class.forName("Question")
Method mGetQuestion = qClass.getMethod("getQuestionText");
String qToString = (String) mGetQuestion.invoke(q);
System.out.println(qToString);

Method mSetAnswer = qClass.getMethod("setUserAnswer", String.class);
mSetAnswer.invoke(q, "vrai");

Method mIsCorrect = qClass.getMethod("isCorrect");
boolean result = (Boolean) mIsCorrect.invoke(q);
System.out.println(result);
```

```
Class c = Class.forName("YesNoAnswer");
Constructor constructor = c.getConstructor( String.class);
Answer answer = constructor.newInstance( "vrai" );
Question q = new Question("texte question", answer, 1);
Class qClass = Question.class; // ou Class.forName("Question")
Method mGetQuestion = qClass.getMethod("getQuestionText");
String qToString = (String) mGetQuestion.invoke(q);
System.out.println(qToString);

Method mSetAnswer = qClass.getMethod("setUserAnswer", String.class);
mSetAnswer.invoke(q, "vrai");

Method mIsCorrect = qClass.getMethod("isCorrect");
boolean result = (Boolean) mIsCorrect.invoke(q);
System.out.println(result);
```

ce code équivaut (dans son comportement) à celui-ci :

```
Answer answer = new YesNoAnswer("vrai");
Question q = new Question("texte question",answer, 1);
String qToString = q.getQuestionText();
System.out.println(qToString);
q.setUserAnswer("vrai");
boolean result = q.isCorrect();
System.out.println(result);
```

Retour sur “factory method”

Exemples.

```
public abstract class Shape {
   public abstract void draw();
   public abstract void erase();
   public static Shape factory(String type) throws BadShapeCreation {
      if(type.equals("Circle")) return new Circle();
      if(type.equals("Square")) return new Square();
      throw new BadShapeCreation(type);
   }
}

public class Circle extends Shape {
   public Circle() {} // Friendly constructor
   public void draw() { System.out.println("Circle.draw"); }
   public void erase() { System.out.println("Circle.erase"); }
}
```

OCP ?

Exemple (suite)

```
public interface ShapeFactory {
  public Shape createShape();
}
public class AllShapeFactories {
  public static Map<String,ShapeFactory> factories = new HashMap<String,ShapeFactory>();
  public static final Shape createShape(String id) {// "throw BadShapeCreation"
    return factories.get(id).createShape();
  }
public class Circle extends Shape {
  static { ShapeFactory.factories.put("Circle", new CircleFactory()); }
  (...)
}
public class CircleFactory implements ShapeFactory {
  public Shape createShape() { return new Circle(); }
}
public class Rectangle extends Shape {
  static { ShapeFactory.factories.put("Rectangle", new RectangleFactory()); }
  (...)
}
public class RectangleFactory implements ShapeFactory {
  public Shape createShape() { return new Rectangle(); }
}
```

Usage

```
      //création à partir du nom de classe
Shape shape = AllShapeFactories.createShape("Circle");
...
      // création "aléatoire"
int alea = new random.nextInt(ShapeFactory.factories.size());
Iterator<String> it = AllShapeFactories.factories.keySet().iterator();
for (int i = 0 ; i < alea; i++) { it.next(); }
shape = ShapeFactory.createShape(it.next());
```

- éviter la "pollution des classes" : utiliser des classe internes

```
public class Circle extends Shape {
  static { ShapeFactory.factories.put("Circle", new Circle.MaFactory()); }
  (...)
  class MaFactory implements ShapeFactory {           // classe interne
    public Shape createShape() { return new Circle(); }
  }
}
public class Rectangle extends Shape {
  static { ShapeFactory.factories.put("Rectangle", new Rectangle.MaFactory()); }
  class MaFactory implements Factory {
    public Shape createShape() { return new Rectangle(); }
  }
  (...)
}
```

Avec chargement dynamique...

```
public interface Shape {
   public void draw();
   public void erase();
}
public abstract class ShapeFactory {
   protected abstract Shape create();
   static Map<String,ShapeFactory> factories = new HashMap<String,ShapeFactory>();
   // A Template Method:
   public static final Shape createShape(String id) throws BadShapeCreation {
   if(!factories.containsKey(id)) {
      try {
         Class.forName(id);                                    // Chargement dynamique
      } catch(ClassNotFoundException e) { throw new BadShapeCreation(id); }
      // Vérifie si elle a été ajoutée
      if(!factories.containsKey(id)) throw new BadShapeCreation(id);
   }
   return factories.get(id).create();
   }
}
public class Circle implements Shape {
   private Circle() {}
   public void draw() { System.out.println("Circle.draw"); }
   public void erase() { System.out.println("Circle.erase"); }
   private static class Factory extends ShapeFactory {
      protected Shape create() { return new Circle(); }
   }
   static { ShapeFactory.factories.put("Circle", new Circle.Factory()); }
}
```