

## 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



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
- 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(...)
```

- `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.
- 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()`)
  - 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)
```
- récupérer une valeur d'attribut : `Field : get(Object)`
- invoquer une méthode : `Method : invoke(Object, Object...)`

`instanceof`

## Applications

"RMI"

## 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`

```
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`

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

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);
```

```
interface Shape {
    void draw();
    void erase();
}

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); // charge dynamiquement
            } catch(ClassNotFoundException e) { throw new BadShapeCreation(id); }
            // regarde si on l'a ajouté :
            if(!factories.containsKey(id)) throw new BadShapeCreation(id);
        }
        return (factories.get(id)).create();
    }
}

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()); }
}
```

## Class Loader

Chaque classe utilisée par un programme Java doit être chargée dans la JVM

accompli par les *class loaders*

- *class loader primaire*
  - (ou système) est propre à chaque JVM
  - un seul par JVM
- *objets "ClassLoader"*
  - est propre à l'application, donc au choix du concepteur de l'application
  - autant que le veut le concepteur

## Espaces de nommage

Dans un JVM, chaque classloader possède son **propre** espace de nommage.

*espace de nommage* = ensemble des noms (qualifiés) des classes chargées

À l'intérieur d'un espace de nommage chaque nom est **unique**

- une même application peut avoir deux espaces de nommage qui contiennent des noms identiques (mais chargés par des class loaders différents) et ne correspondent pas forcément aux mêmes classes.

## Incursion dans la JVM

Quand la JVM charge un type (classe ou interface) :

- l'information concernant ce type est extraite d'un fichier de classe
- cette information est rangée dans la "zone de méthodes"
- un nom complet est ajouté dans l'espace de nommage

tout nom de l'espace de nommage est associé avec des données dans la zone de méthodes

## Edition de liens dynamique

Les fichiers compilés (.class) contiennent des références symboliques sur les autres types dans le *pool de constantes*.

À l'exécution, la JVM réalise **dynamiquement** l'édition de liens lors de la phase de "*résolution de constantes*" en liant les symboles du pool de constantes.

- soit le symbole à résoudre (= nom) est déjà dans l'espace de nommage
- soit ce n'est pas le cas et il faut charger le type et donc choisir un class loader.

### Règle de référencement

la JVM charge une classe référencée avec le même classe loader que la classe qui provoque la référence

## Class Loader

- Création d'espaces de nommage distincts
  - éviter des conflits de noms (cf. plusieurs applets ds 1 seul navigateur)
  - sécurité (protection entre les espaces de nommage)
- Contrôle du chargement des classes
  - sécurité
  - chargement via le réseau.
- pour remplacer dynamiquement une classe déjà chargée
  - mise à jour dynamique de versions de classe sans arrêt de la JVM

## Définition de Class Loader

```
package essais.dynamic;
import java.io.*;
class UneClasse { ... }

class MyClassLoader extends ClassLoader {
    public Class loadClass(String className) throws ClassNotFoundException {
        Class result = findLoadedClass(className);
        // déjà chargée par ce classloader?
        if (result != null) { return result; }
        // si java.* alors utilise le class loader système
        if (className.startsWith("java.")) {
            result = super.findSystemClass(className);
        }
        if (result != null) { return result; }
        byte[] data = getData(className);
        if (data == null) { throw new ClassNotFoundException(); }
        result = defineClass(className, data, 0 ,data.length);
        if (result == null) { throw new ClassNotFoundException(); }
        return result;
    }
}
```

```
private byte[] getData(String className) {
    FileInputStream file;
    String fileName = className.replace('.',File.separatorChar)+".class";
    try { file = new FileInputStream(fileName);
        } catch (FileNotFoundException e) { return null; }
    BufferedInputStream bis = new BufferedInputStream(file);
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    try {
        int c = bis.read(); while (c != -1) { out.write(c); c = bis.read(); }
        } catch (IOException e) { return null; }
    return out.toByteArray();
}
} // MyClassLoader
public class TestCastClassLoader {
    public static void main (String[] args) throws Exception {
        Class c = Class.forName("essais.dynamic.UneClasse");
        MyClassLoader myLoader = new MyClassLoader();
        Class autreC = myLoader.loadClass("essais.dynamic.UneClasse");
        System.out.println("comparaison de classe "+autreC.equals(c));
    }
} // TestCastClassLoader          java TestCastClassLoader --> false
```