Réflexivité - Introspection Java

Module M4105C

Jean-François Brette

jean-francois.brette@parisdescartes.fr

Réflexivité - réflexion

capacité d'un programme de se connaître
et de se modifier **à l'exécution**

Les 3 niveaux de réflexivité :

- **Classe des objets lors de l'exécution**,
RTTI, cast avec exception, instanceof
- **Introspection**,
ensemble des méthodes, champs, classes internes,
appel dynamique, modification, bypass la sécurité
- **Intercession**,
changer la façon d'effectuer l'appel de méthode,
ajouter des champs,
émuler en Java par modification du byte-code

Les classes au runtime
java.lang.Class<T>

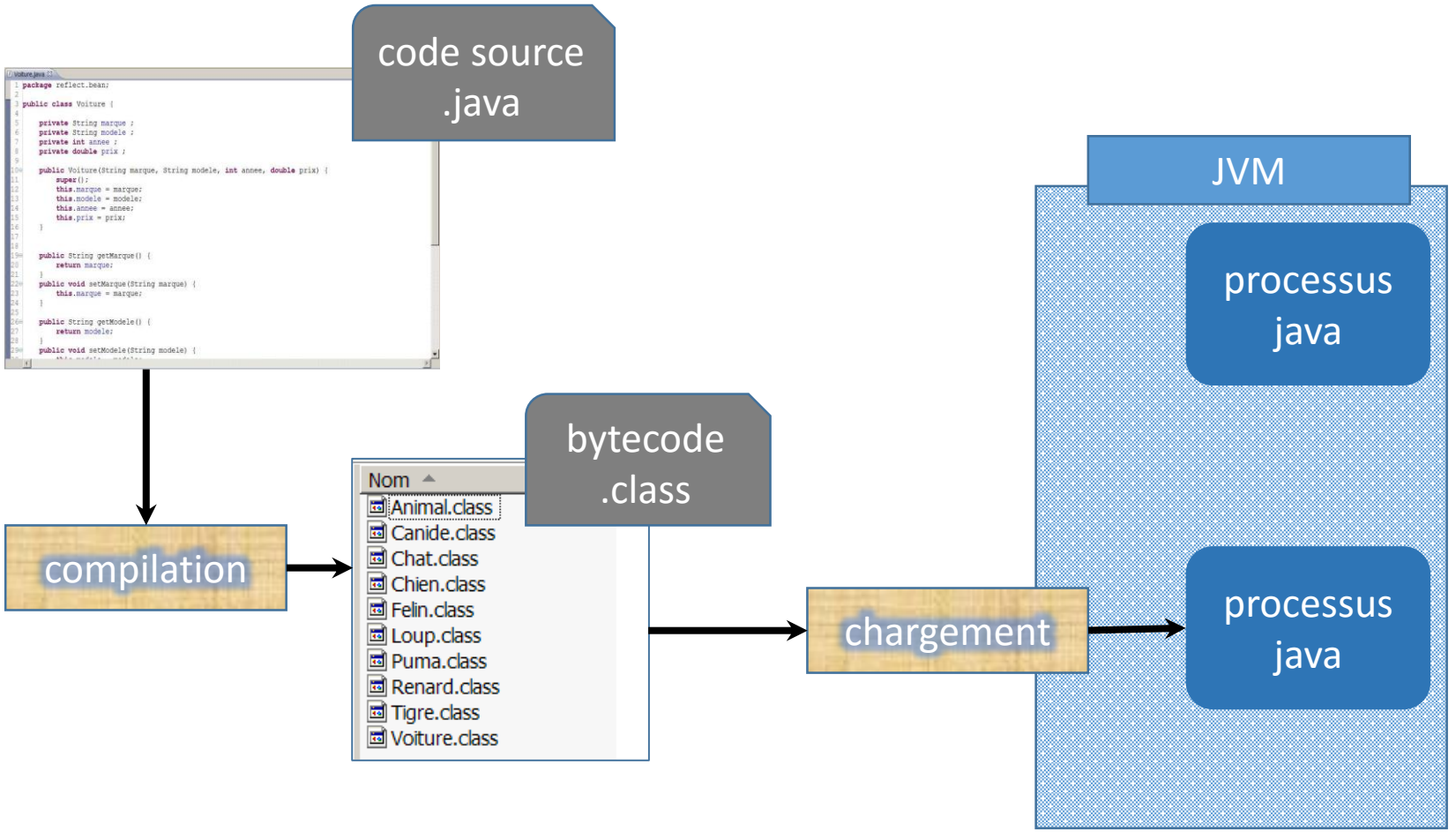
La classe java.lang.Class

Pendant l'exécution, chaque classe est représentée par une instance de java.lang.Class

- Cette instance est unique pour une classe donnée
- Elle est créée automatiquement lors du chargement-initialisation de la classe
- 3 façons d'obtenir cet objet : Class<?> classe =

La méthode getClass()	**(new ArrayList()).getClass()**
La syntaxe .class	**java.util.ArrayList.class**
La méthode Class.forName(String)	**Class.forName(«java.util.ArrayList»)**

Cycle de vie des classes



Chargement d'une classe (java.lang.ClassLoader)

Chaque classe est chargée et initialisée UNE seule fois

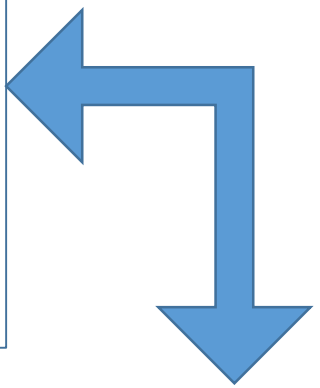
Statique : toutes les classes nécessaires au processus sont chargées au lancement du processus (C++)

Dynamique en Java : lors de sa première apparition dans l'exécution du processus

- implicite : Client c = new Client(….) ;
- explicite : Class.forName(aString); (ClassNotFoundException)

Initialisation d'une classe

```
public class Texte {
private final static String texte = "hello";
  public static String affiche() {
   return texte;
  }
}
```



bloc static

```
public class Texte {
private final static String texte;
static {
  texte = "hello";
}
  public static String affiche() {
    return texte;
  }
}
```

Initialisation de constantes par lecture fichier

```
public class Data {
private final static String nomFichier ;
public static int SOLDE_MAX ;

static {
  nomFichier = "c:\file\fichier.txt";
  try {
     FileInputStream f = newFileInputStream(nomFichier);
     SOLDE_MAX = f.read(); // IOException
  } catch (IOException e) {
     SOLDE_MAX = 0;
     // e.printStackTrace(); ???
  }
}
}
```

Singleton thread-safe

```
public class Singleton {
static { // le bloc static est thread-safe
        instance = new Singleton(……);
}
private static Singleton instance;
private Singleton(…….) {
// initialisation des variables d'instance si nécessaire……
}
public static Singleton getInstance() {
// le test <if (instance==null)> est inutile et n'est pas thread-safe
// à moins d'un synchronized(Singleton.class) --> baisse permanente de performance
        return instance;
}
// méthodes d'instances.....
……
}
```

RTTI : Run-Time Type Identification

Java maintient l'Identification de Type au Run-Time sur tous les objets

Permet de connaître le **type dynamique** d'une référence

Pas nécessairement le même que le **type statique**

Permet d'implémenter la liaison dynamique

Type statique **Type dynamique**

```
ObjetGraphique objGraph = new Cercle();
objGraph.draw();
```

Compilation : type statique (référence déclarée)

Exécution : type dynamique (objet instancié)

Classe et type

- Un objet de la classe **Class** est paramétré par le type qu'elle représente :

```
Class<String> c1 = String.class;
```

- Polymorphisme : **Class<? extends TypeStatique>**

```
Class<? extends ObjetGraphique> classe = objGraph.getClass();
```

```
// class Enseignant extends Personne
Personne p = new Enseignant("JF Brette") ;
Class<? extends Personne> classe = p.getClass();
classe == Personne.class → false
classe == Enseignant.class → true
```

- **Cas le plus général : Class<?> (classe, interface, type de base… même void (Void)**

```
public class EncapsuleClasse {
// juste une classe qui encapsule un attribut Class<?>
private Class<?> laClass;
public EncapsuleClasse(Class<?> laClass) {
        this.laClass = laClass;
}
@Override
public String toString() {
        return "EncapsuleClasse " + this.laClass;
}
}

**************************************************
Quelques exemples d'exécution

        System.out.println(new EncapsuleClasse(String.class));
        System.out.println(new EncapsuleClasse(int.class));
        System.out.println(new EncapsuleClasse(Runnable.class));
        System.out.println(new EncapsuleClasse(void.class));

Résultats affichés

   EncapsuleClasse class java.lang.String
   EncapsuleClasse int
   EncapsuleClasse interface java.lang.Runnable
   EncapsuleClasse void
```

Instancier sans new : méthode newInstance() de Class

public T newInstance()
throws InstantiationException, IllegalAccessException

IllegalAccessException la classe ou son constructeur vide n 'est pas accessible
InstantiationException – la classe n'est pas instanciable
ExceptionInInitializerError – l'initialisation a échoué
SecurityException – liée au SecurityManager

```
Class<Personne> cl = Personne.class;
Personne p = cl.newInstance();
```

```
Class< ? extends Personne> cl = Enseignant.class ;
Personne [] personnes = new Personne[10] ;
personnes [0] = cl.newInstance();
```

```java
public class EncapsuleRunnable {
// une classe qui encapsule un attribut d'une classe qui implémente Runnable
public EncapsuleRunnable(Class<? extends Runnable> laClass) {
          this.laClassRunnable = laClass;
}

private Class<? extends Runnable> laClassRunnable;

public void lancer() throws InstantiationException, IllegalAccessException {
          new Thread(this.laClassRunnable.newInstance()).start();
}
@Override
public String toString() {
          return "Test [laClass=" + laClassRunnable + "]";
}

**********************************************************
Quelques exemples d'exécution

          //new EncapsuleRunnable(String.class); -> erreur de compilation
          System.out.println(new EncapsuleRunnable(objetRunnable.Producteur.class));
          System.out.println(new EncapsuleRunnable(objetRunnable.Activite.class));
          new EncapsuleRunnable(objetRunnable.Producteur.class).lancer();

Résultats affichés

      Test [laClass=class objetRunnable.Producteur]
      Test [laClass=interface objetRunnable.Activite]
      Dans le run() du producteur
```

Un serveur découplé du service

Le code de la classe serveur du td « serveur inversion » est lié au nom de la classe de service :

```
class Serveur implements Runnable {
……
public void run() {
try {
while(true)
          new ServiceInversion(listen_socket.accept()).lancer();
}
……
}
```

Changement de service → code du serveur à modifier !

Solution : ajouter un paramètre au serveur, la classe du service

```
          new Serveur(serveurinverse.ServiceInversion.class, DEFAULT_PORT).lancer();
```

(voir le code complet sur Eclipse)

Une factory plus évolutive

```
import java.util.Collection; // couplage à l'interface - normal

import java.util.ArrayList;  // couplage aux implémentations possibles
import java.util.Vector;     // → pas évolutif et lié à creeCollection()

public class FabriqueCollection {

public static Collection creeCollection(String typeDeLObjet) throws Exception {
  switch (typeDeLObjet) {
          case "java.util.ArrayList" : return new ArrayList();
          case "java.util.Vector" : return new Vector() ;
          //etc
  }
  throw new Exception (typeDeLObjet+" non géré par la fabrique");
} // fin creeCollection

public static Collection creeCollectionParIntrospection(String typeDeLObjet)
                    throws Exception {
try {
          Class<?> classe = Class.forName(typeDeLObjet);
          // tester que cette clase implémente Collection
          return (Collection) classe.newInstance();
} catch (Exception e) {
          throw new Exception (typeDeLObjet+" non géré par la fabrique");
}
} // fin creeCollectionParIntrospection
}
 (test sur Eclipse)
```

Introspection
java.lang.reflect

java.lang.reflect

contient les classes Field, Method, Constructor,…

- Toutes possèdent **`getName()`**
- **`Field`** possède **`getType()`** qui renvoie un objet de type **`Class`**
- **`Method`** et **`Constructor`** ont des méthodes pour obtenir le type de retour et le type des paramètres et surtout des méthodes pour les exécuter

Modifier : un utilitaire de décodage

Field, Method et Constructor possèdent **`getModifiers()`** qui renvoie un **`int`** (4 octets) dont les bits à 0 ou à 1 signifient static, public, private, etc…

On utilise les méthodes de **`java.lang.reflect.Modifier`** pour interpréter cette valeur à l'aide de méthodes toutes **static**

- **`boolean isFinal(int)`**
- **`boolean isPublic(int)`**
- **`boolean isPrivate(int)`**
- **`boolean isSynchronized(int)`**

```
Modifier.isPublic(String.class.getModifiers())
                                              true
Modifier.isInterface(Runnable.class.getModifiers())
                                              true
```

Obtenir les méthodes de la classe

Récupérer toutes les méthodes :

Method[] getMethods : méthodes publiques (de la classe et héritées)
Method[] getDeclaredMethods : méthodes déclarées dans la classe (quelque soit le droit d'accès) mais pas les méthodes héritées.

Récupérer une seule méthode : il faut préciser la signature de la méthode :

```
public Method getMethod(String name, Class<?>... parameterTypes)
```

Remarque : Le principe est le même pour obtenir les **constructeurs** ou les **attributs** d'une classe (Method est remplacé par **Constructor** ou **Field**).

Thread.class.getConstructor(Runnable.class)
Thread.class.getDeclaredField(« target »)

Instancier en passant les paramètres

```
public class Personne {
public Personne(String nom) {
          this(nom, 0);
}
public Personne(String nom, int age) {
          this.nom = nom;
          this.age = age;
}……
}
```

```
private static void test(Class<?> classe) {
     Constructor<?> constr1 = classe.getConstructor(String.class);
     Constructor<?> constr2 = classe.getConstructor(String.class, int.class);
     System.out.println(constr1.newInstance("brette"));
     // System.out.println(constr1.newInstance("brette",25)); // erreur
     System.out.println(constr2.newInstance("brette",25));
}
```

Instancier un service en passant la socket

```
private Class<? extends Service> serviceClass;
private Constructor<? extends Service> constr;

public Serveur(Class<? extends Service> serviceClass, int port)
          throws IOException, NoSuchMethodException, SecurityException {
  listen_socket = new ServerSocket(port);
  this.serviceClass = serviceClass;
  this.constr = serviceClass.getDeclaredConstructor(java.net.Socket.class);
  if (!Modifier.isPublic(constr.getModifiers()))
    throw new NoSuchMethodException("Le constructeur n'est pas public");
  this.thread = new Thread(this);
}

public void run() {
  while(true) {
    Socket client_socket = listen_socket.accept();
    Service service = this.constr.newInstance(client_socket);

    this.init(service);
    service.start();
}
}
```

Invoquer une méthode avec ses paramètres

```
public class Personne {

private int age;

public boolean majeur() {
          return this.majeur(18);
}
public boolean majeur(int ageMajeur){
          return (this.age>=ageMajeur);
}
```

```
Class<Personne> classeP = Personne.class;
Personne brette = new Personne(« Brette », 25);

Method method = classeP.getMethod("majeur");
System.out.println(method.invoke(brette));// --> true

Method method2 = classeP.getMethod("majeur", int.class);
System.out.println(method2.invoke(brette, 30));// --> false

Method method = classeP.getMethod("majeur« , String.class);
          → NoSuchMethodException
```

Incrémenter un attribut int à l'aide d'accesseurs get/set

```
public class Personne {

private int porteMonnaie = 0;

public int getPorteMonnaie() {
return porteMonnaie;
}
public void setPorteMonnaie (int
porteMonnaie) {
this.porteMonnaie = porteMonnaie;
}
}
```

```
public class Vol {

private int nbPlacesDispos = 150;

public int getNbPlacesDispos() {
return nbPlacesDispos;
}
public void setNbPlacesDispos(int
nbPlacesDispos) {
this.nbPlacesDispos = nbPlacesDispos;
}
}
```

```
Personne p = new Personne();
incrementerPropriete(p, «porteMonnaie», 10);

Vol v=new Vol(150);
incrementerPropriete(v,«nbPlacesDispos», -1);
```

Incrémenter un attribut int à l'aide d'accesseurs get/set

```
private static void incrementerPropriete(Object objet, String nom, int quantité)
          throws …….. {

String nomProp = Character.toUpperCase(nom.charAt(0))+ nom.substring(1);

Class<?> classe = objet.getClass();

// Récupérer la valeur de l'attribut
Method accesseur = classe.getMethod("get" + nomProp);
Object valeur = accesseur.invoke(objet);

// Incrémenter la valeur
int nlleValeur = ((Integer) valeur).intValue() + quantité;

// Affecter la nouvelle valeur
Method modifieur = classe.getMethod("set" + nomProp, int.class);
modifieur.invoke(objet, nlleValeur);
}
```

Principe des Java Beans (1)

Un "Java Bean" est une classe Java qui peut être utilisé par un programme qui ne connait rien de lui.

Il doit donc respecter certaines conventions :

** La classe est **publique**
** La classe **implémente "java.io.Serializable"**
** Il existe un **constructeur public sans paramètre**
** Pour chaque attribut, on dispose de deux accesseurs de nommage standard **get**<NomAttribut> et **set**<NomAttribut>

Principe des Java Beans (2)

Les conventions de nommage des JavaBeans permettent d'utiliser la réflexivité pour adresser dynamiquement leurs propriétés.

Principe :

Pour chaque propriété (attribut) du JavaBean on détermine un "nom symbolique" à partir des méthodes "getXxx" et "setXxx"

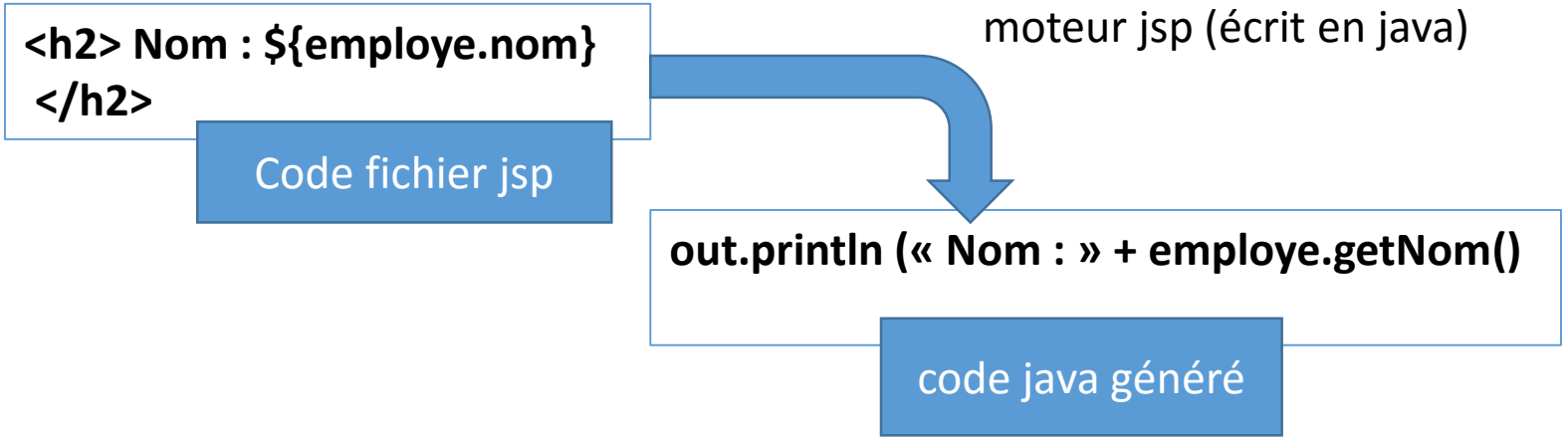
Le nom symbolique correspond à la suite de caractères qui suit les préfixes "get" et "set". Le premier caractère est converti en minuscule, les autres restent inchangés

getAge / setAge ← "age"
getFirstName / setFirstName ← "firstName"

Java Beans et JSP avec EL

JSP : Java Server Page (pages Web dynamiques)
EL : Expressions Languages
manipulations de données plus lisibles que les scriptlets

Principe : un bean 'employe' de la classe Employe



Les proxys dynamiques java.reflect.Proxy

Principe du proxy

Un proxy est un objet qui contrôle l'accès aux méthodes d'un objet

Transparent pour le code client (via une interface)
Invoque les méthodes - de l'objet contrôlé - par délégation
Implémente **toute** l'interface de l'objet contrôlé

Une classe tiers (factory) crée le proxy

Code du client de la fabrique :

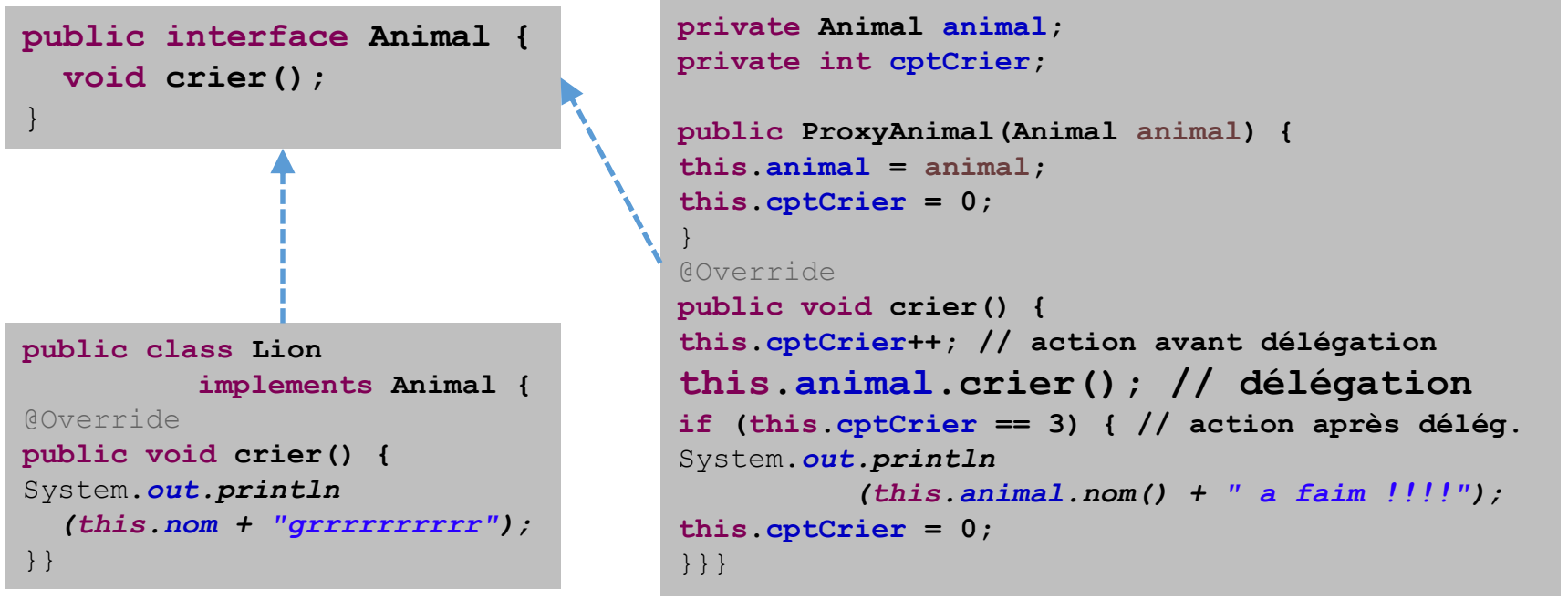
```
Animal unLion = FabriqueAnimal.créerAnimal("brette");
```

```
public class FabriqueAnimal {
        public static Animal créerAnimal (String arg) {
                return new ProxyAnimal(new Lion(arg));
        }
}
```

Lion et ProxyAnimal implémentent l'interface Animal

Proxy : interface, délégation, factory

Le proxy effectue un traitement lié au contexte (domaine, méthode)



```java
public interface Animal {
  void crier();
}
```

```java
public class Lion
          implements Animal {
@Override
public void crier() {
System.out.println
  (this.nom + "grrrrrrrrrr");
}}
```

```java
public class ProxyAnimal implements Animal {

private Animal animal;
private int cptCrier;

public ProxyAnimal(Animal animal) {
this.animal = animal;
this.cptCrier = 0;
}
@Override
public void crier() {
this.cptCrier++; // action avant délégation
this.animal.crier(); // délégation
if (this.cptCrier == 3) { // action après délég.
System.out.println
          (this.animal.nom() + " a faim !!!!");
this.cptCrier = 0;
}}}
```

Proxy à traitement standard (trace, surveillance d'activité,…)

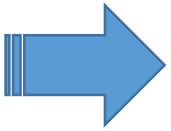
Pour chaque interface : 1 classe de proxy

```
public class TraceAnimal implements Animal {

private Animal animal;

public ProxyAnimal(Animal animal) {
  this.animal = animal;
}
@Override
public void crier() {
  System.out.println(« avant crier » + animal );
   this.animal.crier(); // délégation
  System.out.println(« après crier » + animal);
}}}
```

redondance pour chaque méthode de Animal

et même pour toute méthode de tout interface !

Comment avoir du code de proxy indépendant de l'interface et des méthodes ?



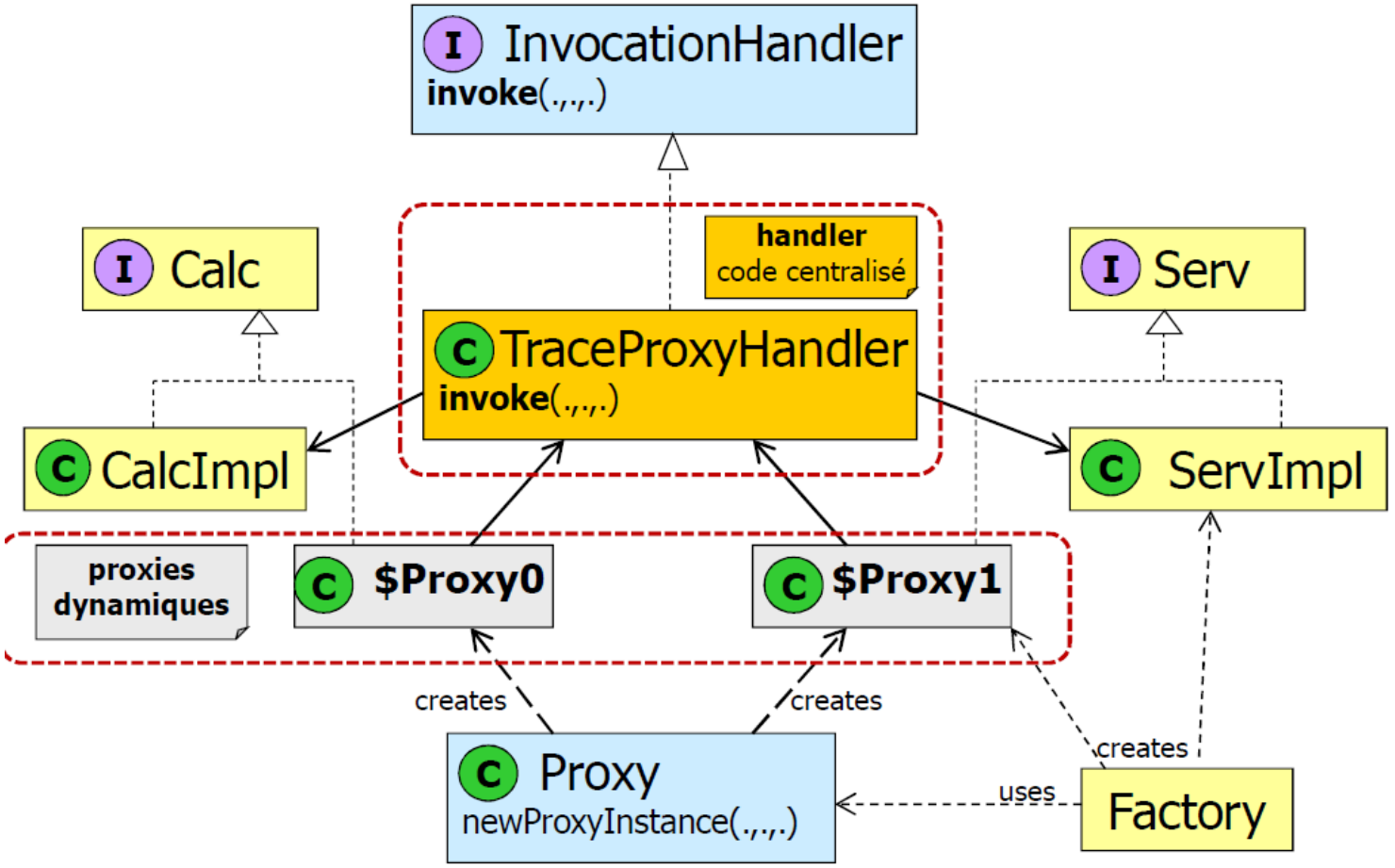
introspection !

Proxy défini par introspection (proxy dynamique)

Objectif : créer une classe de proxy autour d'un objet obj
Contexte : obj est d'un type inconnu

Connaître la classe de l'objet
Class<?> classe = obj.getClass()
Connaître les interfaces implémentées par obj
classe.getInterfaces()
Trouver une méthode invoquée par le client
classe.getMethod(obj,args)
Dans Class *public Method getMethod(String name, Class<?>... parameterTypes)*
Invoquer une méthode d'une interface par délégation
method.invoke(obj,args)
Créer la classe du proxy avec son attribut, ses méthodes, la délégation,etc --→ *classe java.lang.reflect.Proxy*

Proxy dynamique
séparation délégation/contrôle



Proxy.newProxyInstance(……..)

La factory renvoie le proxy
instance de la classe créée dynamiquement

Code du client de la fabrique :

```
Animal unLion = FabriqueAnimal.créerAnimal("brette");
```

```
public class FabriqueAnimal {
  public static Animal créerAnimal (String arg) {
    return
    (Animal) Proxy.newProxyInstance (
                Lion.class.getClassLoader(),
                Lion.class.getInterfaces(),
                new AlerteActivite(new Lion("brette"))
             );
  }
}
```

InvocationHandler : contrôle d'accès

Le « handler » reçoit tous les messages des interfaces via le proxy

```
public class AlerteActivite implements InvocationHandler {

private final Object objet;
private int activite;

public AlerteActivite(Object obj) {
  this.objet = obj;
  this.activite = 0;
}
@Override
public Object invoke(Object proxy, Method method, Object[] args)
         throws Throwable {
  this.activite++;
  Object obj = method.invoke(objet, args);
  alerte(); // modalités à définir
  return obj;
}}
```

Moniteur de Hoare dynamique

Le moniteur enrobe les messages d'un *synchronized*

```
public class Moniteur implements InvocationHandler {
private Object ressource;
public Moniteur(Object ressource) {
        this.ressource = ressource;
}
public Object invoke(Object obj, Method method, Object[] args) {
        Object result = null;
  synchronized (ressource) {
    try { result = method.invoke(this.ressource, args);
    } catch (Exception e) {System.err.println(e);}
  }
  return result;
}
}
```

Moniteur de Hoare dynamique

L'application passe une fifo-proxy aux prods-consos

```
//File<Data> fifo = new FIFO<Data> (); // oldschool
File<Data> f = new FIFO<Data> ();
File<Data> fifo =
(File<Data>) Proxy.newProxyInstance (
          f.getClass().getClassLoader(),
          f.getClass().getInterfaces(),
          new Moniteur(f)
        );
Consommateur.setFifo(fifo);
Producteur.setFifo(fifo);
   new Producteur ("brette",20);
   new Producteur ("plombowski", 100);
   new Consommateur ("étud1", 50);
```

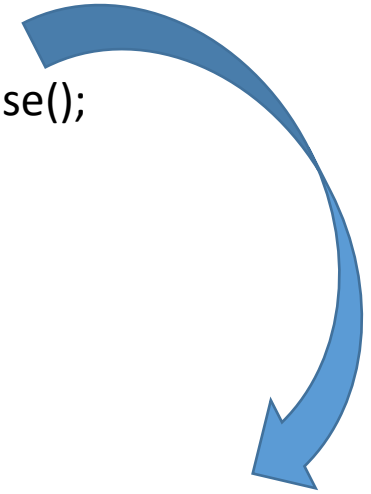
Chargement dynamique de classes java.lang.ClassLoader

ClassLoader : le chargeur de classes

- **Retrouver une classe dans l'environnement d'exécution, la charger/initialiser si elle ne l'est pas déjà et maintenir l'objet Class<?> qui référence cette classe**
- **Un classLoader est un espace d'adressage en soi**
- **Plusieurs classLoaders peuvent co-exister dans une exécution**
- **java.lang.ClassLoader classe abstraite -> sous-classes mais l'arbre d'héritage est très réduit et pas très important**

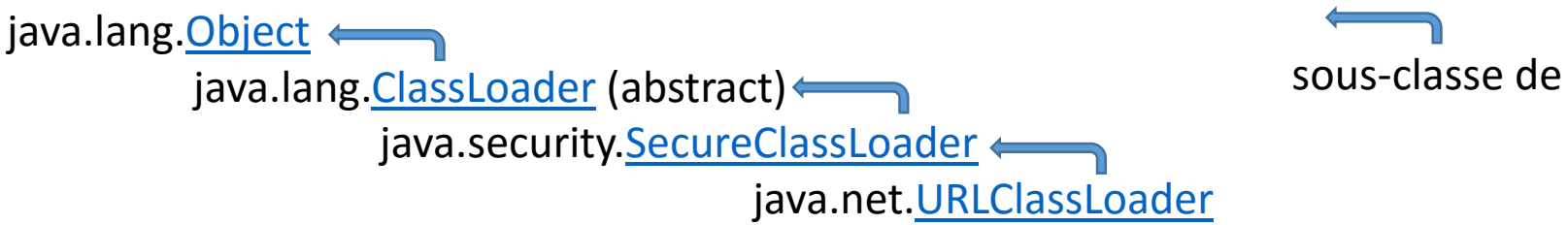
Invoquer une classe lors de l'exécution

```
public class A {
        public void doSomething() {
                B b = new B();
                b.doSomethingElse();
        }
}
```



this.getClass().getClassLoader().loadClass("pack.B")
.newInstance()

ClassLoader : héritage

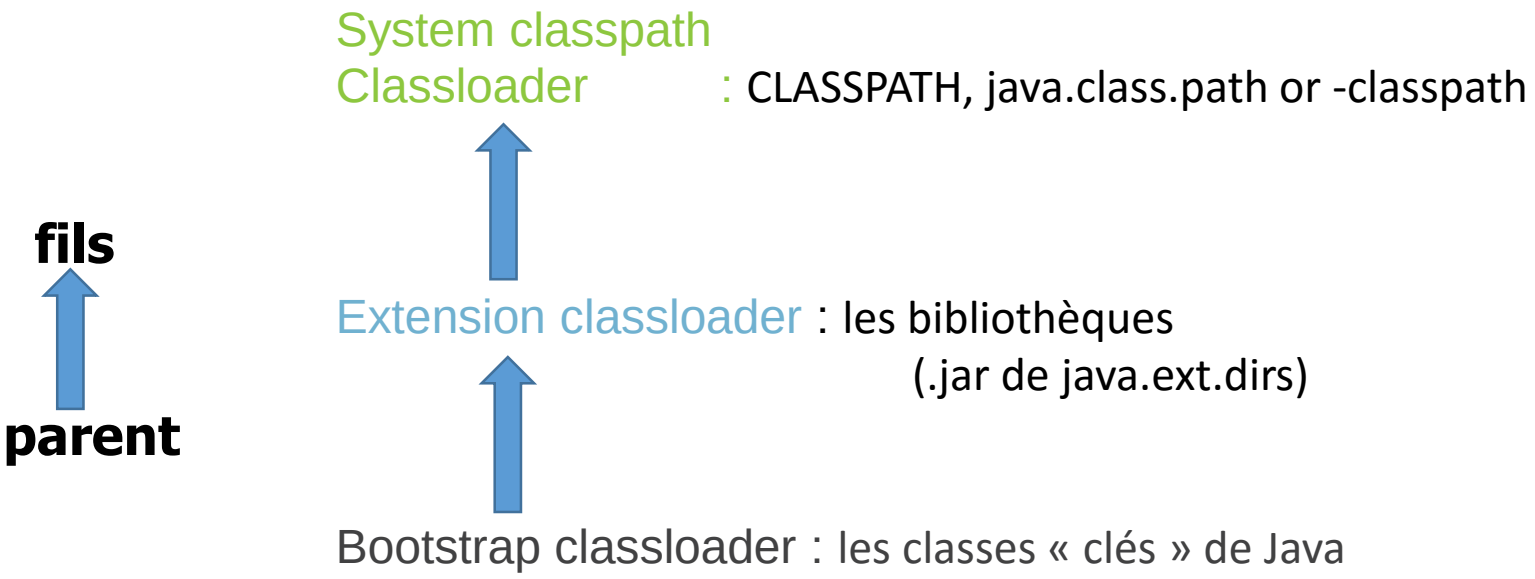


Extrait de la classe ClassLoader :

```
package java.lang;
public abstract class ClassLoader {
        public Class loadClass(String nomClasse)
        protected Class defineClass(byte[] byteCodes)
        public URL getResource(String nomClasse)
        public Enumeration getResources(String name)
        public ClassLoader getParent() → parent de délégation
}
```

loadClass(name) *équivalent à* **defineClass(getResource(name).getBytes())**

ClassLoader : délégation



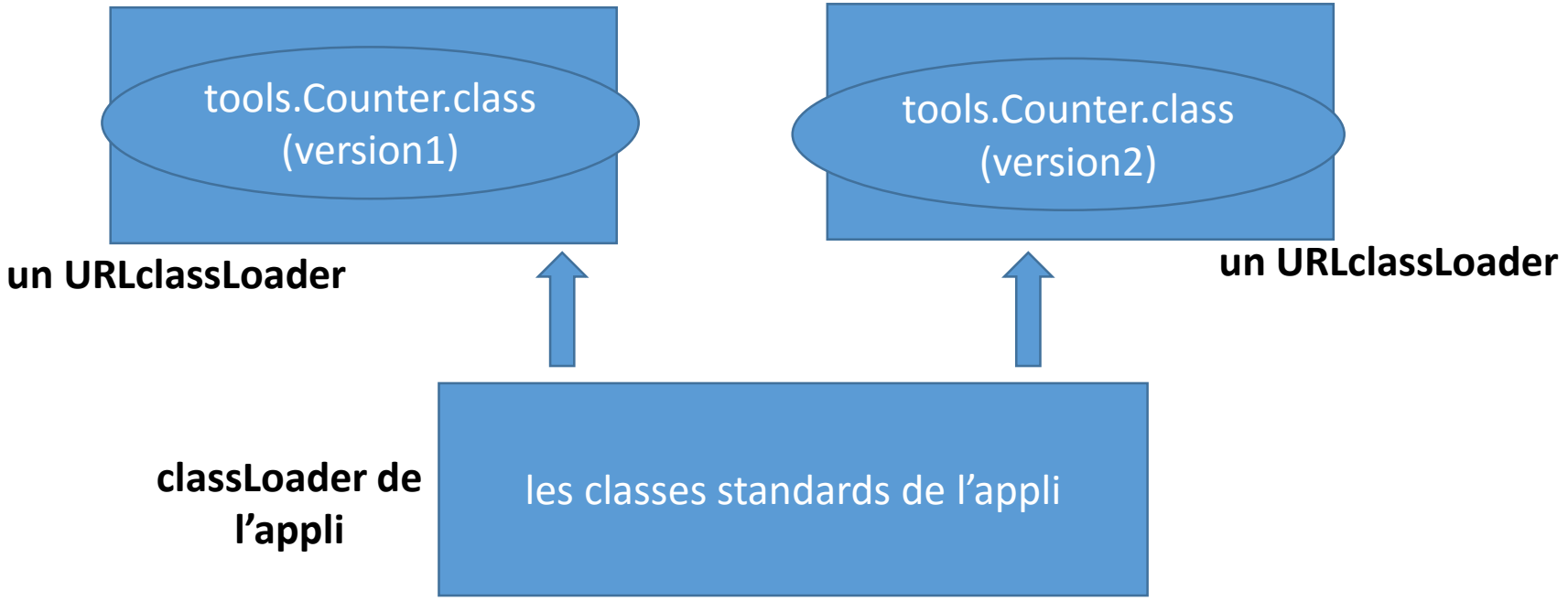
Algorithme de recherche d'une classe (findClass)
(on ne charge que ce qui n'a pas été chargé par un contexte plus général)

- **demande à son parent s'il connait ça (délégation)**
- **sinon, cherche dans son espace d'adressage**

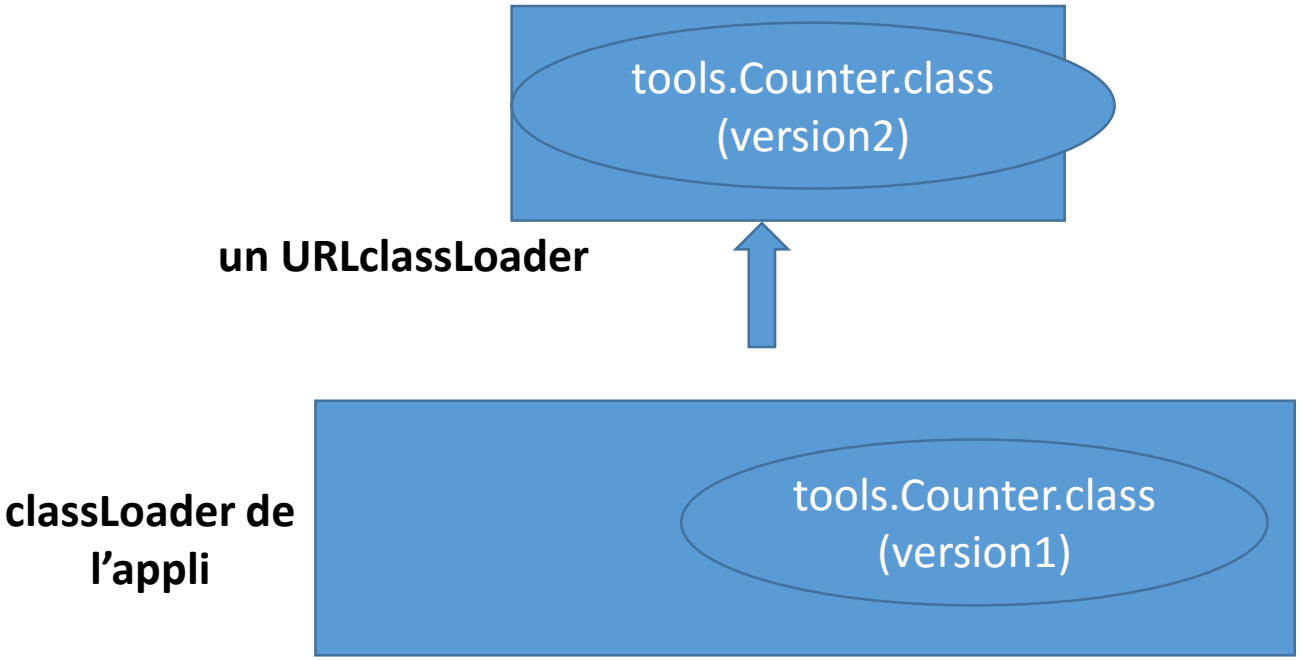
Portée d'une classe et délégation

Une classe est chargée dans un classLoader : sa portée, au vu de l'algorithme de délégation est elle-même et les classloaders fils éventuels

On peut donc avoir 2 (ou plus) versions d'une même classe, du moment que c'est dans 2 classLoaders différents qui ne sont pas fils l'un de l'autre



Mettre à jour une classe dynamiquement



Solution : redéfinir loadClass pour inverser l'ordre de la délégation !

Et maintenant, place au code !