Programmation avancée et répartie en Java : Programmation dynamique et références faibles

Frédéric Gava

L.A.C.L
Laboratoire d'**A**lgorithmique, **C**omplexité et **L**ogique

Cours de M1 MIAGE
(d'après les notes de cours de Fabrice Mourlin)

Plan

1 Programmation dynamique

2 Class Loader

3 Références faibles

Plan

1. Programmation dynamique
2. Class Loader
3. Références faibles

Plan

1. Programmation dynamique
2. Class Loader
3. Références faibles

Déroulement du cours

1. Programmation dynamique
2. Class Loader
3. Références faibles

Qu'est-ce ?

La classe Class et le package java.lang.reflect sont les bases de la programmation dynamique.

Problèmematique

La "découverte" et l'utilisation au runtime d'objets d'un type "inconnu". La définition de protocoles auquels doivent adhérer ces objets inconnus (convention de Bean) permet de créer des outils capables de manipuler dynamiquement des catégories d'objets.

Le compilateur

Java étant un langage compilé puis interprété (JVM), les codes que l'on écrit opèrent sur des classes connues au "compile-time". Il est pourtant possible d'écrire des programmes qui opèrent sur des classes "inconnues" du compilateur.

Pourquoi ?

- Du fait du polymorphisme, des opérations définies par des types connus du compilateur (classes ou interfaces) peuvent en réalité opérer sur des objets ayant un type effectif non apparent ; les classes correspondantes peuvent même avoir été découvertes et chargées dynamiquement à l'exécution.
- Il peut s'avérer utile de découvrir l'API d'une classe à l'exécution et de pouvoir l'utiliser. C'est en particulier le cas de certains outils Java : génération de protocoles (comme la linéarisation des objets), outils de déploiement, assemblages interactifs de composants, composants graphiques génériques s'adaptant à des "modèles" prédéfinis, *etc.*

Vocabulaire

Réflexion : se connaitre soi même. Les 3 niveaux de réflexion :

- Classe des objets lors de l'exécution, 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 (prog par aspect, java.lang.instrument)

La classe des classes

Pour pouvoir opérer à un niveau interprétatif sur un objet on a besoin d'une instance de la classe java.lang.Class représentant la classe de l'objet. Une telle instance peut s'obtenir par :

- Chargement dynamique d'une classe au travers d'un ClassLoader ; on doit alors fournir une chaîne donnant le nom complet de la classe (hiérarchie de package + nom de classe) :

  ```
  Class laClasse = Class.forName(nom);
  ```

- Obtention de la classe d'une instance (p. ex. un transfert distant) : `Class laClasse = instance.getClass();`
- Consultation d'un champ statique connu au compile-time :

  ```
  Class PanelClass = JPanel.class;
  Class tabCharClass = char[].class;
  Class typeInt = int.class;
  Class typeInt = Integer.TYPE;
  ```

  cas particulier des scalaires primitifs : on ne peut pas faire des opérations "objet" avec. Mais (voir plus tard)...

Obtention d'informations

À partir d'une instance de Class (qui ne répond pas à la condition isPrimitive()) on peut obtenir des informations sur les constituants :

- Quels sont les champs déclarés dans la classe ?
  ```
  Field[] champs = maClasse.getDeclaredFields();
  ```
  si la classe représente une tableau (isArray() répond true), il est possible de connaître le type des composants :
  ```
  Class typeComposants = classeTableau.getComponentType();
  ```
- Quelles sont les méthodes définies dans la classe ?
  ```
  Method[] méthodes = maClasse.getDeclaredMethods();
  ```
- Quels sont les constructeurs définis dans la classe ?
  ```
  Constructor[] constructeurs=maClasse.getDeclaredConstructors();
  ```
  L'utilisation des classes correspondantes (Field, Method, Constructor, *etc.*) relève du package java.lang.reflect.

Class et type paramétré

- Un objet de type `Class` est paramétré par le type représenté

```
Class<String> clazz=String.class;
String s=clazz.newInstance();
```

- Pour getClass(), à cause du sous-typage, la règle est Class<? **extends** erasure(type déclaré)>

```
String s="toto";
Object o=s;
Class<? extends Object> clazz2=o.getClass();
clazz2==s.getClass() // true
```

- Permet de charger une classe par son nom complet (avec le nom du paquetage), pratique pour un système de plugin

```
public static void main(String[] args) throws ClassNotFoundException,
                        InstantiationException, IllegalAccessException {
 Class<?> clazz=Class.forName(args[0]);
 Object obj=clazz.newInstance();
 for(Field field:clazz.getFields())
  {System.out.println("=>field "+field.getName()+" "+field.get(obj));}
//gava@gava-laptop:~$ java reflect.GetFields java.awt.Point
// =>field x 0
// =>field y 0
```

Pour les types primitifs

- les types primitifs (int, void, etc.) ont des objets Class correspondants.
- type_primitif.**class** est typé Class<Wrapper> et pour void, **void**.**class** est typé Class<Void>

```
public static void main(String[] args) throws ... {
  Class<Integer> clazz=Integer.class;
  Class<Integer> clazz2=int.class;
  Integer i=clazz.getConstructor(clazz2).newInstance(3);
}
```

- On voit bien que **new** n'est qu'un raccourci pour la méthode newInstance()

Codes reflexifs

La manipulation des champs, méthodes, constructeurs peut poser des problèmes de droit d'accès. Pourtant certains mécanismes doivent pouvoir agir indépendamment des privilèges de responsabilité (private, protected, *etc.*) :

- Membres et constructeurs de la classe dérivent tous de java.lang.reflect.AccessibleObject qui dispose d'une méthode setAccessible(**boolean**) qui permet de passer outre aux privilèges de responsabilité.
- Bien entendu n'importe quel code ne peut réaliser ces opérations sans disposer d'une autorisation de sécurité adequate : java.lang.reflect.ReflectPermission ("suppressAccessChecks")

Codes reflexifs : les champs

À partir d'une instance de Field on peut obtenir :

- son nom : String nom = champs[ix].getName();
- son type : Class type = champs[ix].getType();
- d'autres informations comme les modificateurs utilisés.

On peut également consulter la valeur de ce membre (ou la modifier). Les assesseurs "get/setXXX" permettent de lire ou modifier une valeur sur une instance passée en paramètre :

```
Class laClasse = instance.getClass() ;
Field leChamp0 = laClasse.getDeclaredFields() [0];
.... // obtention informations de type
.... // fixation evt. des droits d'accès
.....// création d'un objet de ce type: valeurObjet
leChamp0.set(instance, valeurObjet);
```

Si champ statique, le premier argument est ignoré (peut être **null**).

Propriété de la class Class

Savoir si la classe est :

- une annotation `isAnnotation()`
- une classe anonyme `isAnonymousClass()`
- Un tableau `isArray()`
- Un type énuméré `isEnum()`
- Une interface `isInterface()`
- Une classe locale à une méthode `isLocalClass()`
- Une classe interne `isMemberClass()`
- Une classe d'un type primitif `isPrimitive()`
- Générée par le compilateur `isSynthetic()`

Hiérarchie de classes

Il est possible pour une classe d'obtenir :

- sa superclass Class<? **super** T> getSuperClass()
- ses interfaces Class<?>[] getInterfaces()

De plus, on peut :

- Tester si un objet est de cette classe isInstance(Object o)
- Tester si une classe est sous-type d'une autre isAssignableFrom(Class<?> clazz)
- Caster une référence vers la classe T cast(Object o)
- Voir **this** comme une sous-classe du paramètre <U> Class<? **extends** U> asSubclass(Class<U> clazz)

Typesafe generics

cast() et asSubclass() sont utilisé pour vérifier à runtime des casts non vérifiables à cause de l'erasure

```
public static <T> T createNullProxy(Class<T> interfaze) {
  Object o=Proxy.newProxyInstance(interfaze.getClassLoader(),
                                  new Class<?>[]{ interfaze },EMPTY_HANDLER);
  //return (T)o; // unsafe
  return interfaze.cast(o); // ok safe
}
```

```
public static Class<? extends LookAndFeel> getLafClass(String className) {
  Class<?> lafClass=Class.forName(className);
  //return (Class<? extends LookAndFeel>)o; // unsafe
  return lafClass.asSubClass(LookAndFeel.class); // ok safe
}
```

Créer un objet à partir de sa Class

- La classe Class<T> possède une méthode newInstance() qui renvoie un objet de type T

```
public static void main(String[] args) throws ... {
  Class<String> clazz=String.class;
  String s=clazz.newInstance();
  ...
```

- Lève les exceptions suivantes :
  - InstantiationException, si la classe est abstraite ou qu'il n'existe pas de constructeur sans paramètre
  - IllegalAccessException, si le constructeur n'est pas publique
  - InvocationTargetException, si une exception est levée par le constructeur, celle-ci est stockée dans la cause de l'exception

java.lang.reflect

À partir d'une classe on peux accéder à ses membres :

- les constructeurs d'objets de type T sont représentés par des instances de la classe Constructor<T>.
- les champs sont représentés par des instances de Field
- les méthodes sont représentées par des instances de Method
- les classes internes sont représentées des instances de Class
- Avec "XXX", Constructor, Field, Method ou Class :
  - getDeclaredXXXs(), qui renvoie tous les "XXX" (privés inclus) qui sont déclarés par la classe, c'est-à-dire non hérités ;
  - getXXXs() retourne tous les "XXX" public (même hérités).
  - getDeclaredXXX(param) renvoie le "XXX" déclarés, dont le nom et/ou les types (objets Class) des paramètres sont donnés en argument ;
  - la méthode getXXX(param), qui retourne le "XXX" public, hérité ou non, dont le nom et/ou les types des paramètres sont donnés en argument ;

java.lang.reflect, remarques

- Les instances des classes Class, Constructor, Method ou Field ne sont pas liées à un objet particulier.
- Lorsque l'on veut changer la valeur d'un champs d'un objet ou appeler une méthode sur celui-ci, on doit fournir l'objet en paramètre
- Pour les méthodes ou les champs statiques, ce paramètre peut (doit) être **null**.

Classe Constructor

- On obtient un Constructor<T> à partir de la classe en utilisant getConstructor(Class... types)
- La méthode T newInstance(Object... args) permet d'appeler le constructeur avec des arguments

```
public static void main(String[] args) throws NoSuchMethodException, InstantiationExc
                                              IllegalAccessException, InvocationTarge
  Class<Point> point = Point.class;
  Constructor<Point> c = point.getConstructor(int.class, int.class);
  Point p = c.newInstance(1,2);
  System.out.println(p); // (1,2)
}
```

- Pour les types primitifs, il faut fournir leur enveloppe (vue avant)

Classe Field

Une instance de Field représente un champs d'un objet

- Object get∗(Object target) permet d'obtenir la valeur
- **void** set∗(Object target) permet de changer la valeur

```
public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
  Field xField=Point.class.getField("x");
  Point p=new Point(1,1);
  System.out.println(xField.getInt(p)); // 1
  xField.setInt(p,12);
  System.out.println(p); // (12,1);
  xField.set(p,Integer.valueOf(13));
  System.out.println(p); // (13,1);
}
```

Classe Method

- Une instance de Method obtenu avec getDeclaredMethod(String,Class<?>...) permet d'appeler une méthode sur tous les objets d'un même type (et sous-types) ;
- La méthode Object invoke(Object receiver, Object... args) permet d'appeler la méthode avec des arguments

```
public static void main(String[] args) throws NoSuchMethodException,
                                    IllegalAccessException, InvocationTargetException {
  Class<PrintStream> print=PrintStream.class;
  Method method = print.getDeclaredMethod("println",char.class);
  method.invoke(System.out,'a'); // a
}
```

classes internes & constructeur

Pour les classes internes non statiques, il faut penser à ajouter un argument aux constructeurs correspondant à une instance de la classe englobante.

```
public class InnerTest {
  class Inner { // pas public ici
     public Inner(int value) { this.value=value; }
     @Override public String toString() {return String.valueOf(value);}
     private final int value;
  }
 public static void main(String[] args) throws InstantiationException, IllegalAccessException,
                                               InvocationTargetException, NoSuchMethodExcep
   Class<?> innerClass=InnerTest.class.getDeclaredClasses()[0];
   Object o=innerClass.getConstructor(InnerTest.class,int.class).
   newInstance(new InnerTest(),3);
   System.out.println(o);
 }
}
```

Visibilité des élements

La classe java.lang.reflect.Modifier permet d'interpréter un entier renvoyé par les éléments comme modificateurs de visibilité.

```
for(Field field:String.class.getDeclaredFields()) {
  int modifiers=field.getModifiers();
  StringBuilder builder=new StringBuilder();
  if (Modifier.isPrivate(modifiers)) builder.append("private ");
  if (Modifier.isProtected(modifiers)) builder.append("protected ");
  if (Modifier.isPublic(modifiers)) builder.append("public ");
  if (Modifier.isStatic(modifiers)) builder.append("static ");
  if (Modifier.isFinal(modifiers)) builder.append("final ");
  System.out.println(builder.append(field.getType().getName())
                                          .append(' ').append(field.getName()));
}
```

java.lang.reflect et Sécurité

Par défaut, la reflexion vérifie la sécurité lors de l'exécution, on ne peut alors effectuer les opérations que si l'on a les droits.

```
public static void main(String[] args) throws NoSuchMethodException, InvocationTargetExcep
                                              InstantiationException, IllegalAccessExceptio
  // access to package private constructor which shares value array
  // beware it's OS specific !!
   Constructor<String> c=String.class.getDeclaredConstructor(int.class,int.class,char[].class)
   char[] array="hello".toCharArray();
   String name=c.newInstance(0,array.length,array);
}
```

```
Exception in thread "main" java.lang.IllegalAccessExceptio
 Class reflect.UnsafeTest can not access a member of class
  java.lang.String with modifiers ""
  at sun.reflect.Reflection.ensureMemberAccess(Reflection.j
  at java.lang.reflect.Constructor.newInstance(Constructor.
  at reflect.UnsafeTest.main(UnsafeTest.java:16)
```

Passer “outre” la Sécurité

Les Constructor, Field et Method héritent de AccessibleObject qui possède setAccessible et qui permet d'éviter de faire le test de sécurité (accélérer le code). On peut alors appeler des méthodes privées, changer la valeur d'un champ final (pas static final).

```
public final String i = "toto";
public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
 FiField f = new FiField();
 Field xField=FiField.class.getField("i");
 xField.setAccessible(true);
 xField.set(f,"bobo");
 System.out.println(f.i); // toto !?
 System.out.println(xField.get(f)); // bobo
 // access to package private constructor which shares value array
 // beware it's OS specific !! Parametrage du Sandbox
 Constructor<String> c=String.class.getDeclaredConstructor(int.class,int.class,char[].class);
 c.setAccessible(true);
 char[] array="hello".toCharArray();
 String name=c.newInstance(0,array.length,array);
 System.out.println(name); // hello
 array[4]='\n';
 System.out.println(name); // hell
```

Les tableaux par reflexion

- java.lang.reflect.Array permet de créer où de manipuler des tableaux de type primitif ou d'objet
- Création d'un tableau :
  static Object newInstance(Class<?>componentType, **int** length)
- Obtenir la valeur d'une case
  static Object get*(Object array, **int** index)
- Changer la valeur d'une case
  static **void** set*(Object array, **int** index, Object value)

Reflexion et Type paramétré

- Par defaut, les méthodes de reflexion retournent les types érasés par le compilateur et non les types paramétrés définies par l'utilisateur
- Il est possible d'obtenir :
  - Les variable de types des types et méthodes paramétrées
  - La signature exacte des méthodes
- L'ensemble des méthodes de reflexion possède donc deux versions :
  - une version érasée utilisant la classe Class pour représenter les types, ex : getExceptionTypes()
  - une version paramétré préfixé par getGenerics... utilisant l'interface Type, ex : getGenericsExceptionTypes()

L'interface Type

L'interface Type définie l'interface de base de tout les types Java :

- Les types paramétrés ⇒ ParameterizedType
- Les variables de type (T) ⇒ TypeVariable<D> où D correspond au type sur lequel il est déclaré
- Les widcards ⇒ WildcardType
- Les classes ⇒ Class
- Les tableaux de type paramétré (List<String>[]) ou de variable de type (T[]) ⇒ GenericArrayType

ParameterizedType et TypeVariable

ParameterizedType

Repésente les types paramétrés List<Integer> avec les méthodes :

- Les arguments (ici,Integer) ⇒ Type[] getActualTypeArguments()
- Le type raw (ici, List) ⇒Type getRawType()
- L'englobant (classes internes) ou **null** ⇒ Type getOwnerType()

TypeVariable

TypeVariable<D> représente une variable de type déclarée par D.

- Le nom de la variable (ex : T) ⇒ String getName()
- Ses bornes (ex : Object) ⇒ Type[] getBounds()
- L'élément déclarant (Class, Method ou Constructor) ⇒ D getGenericDeclaration()
- L'interface GenericDeclaration représente un élément pouvant déclarer une variable : ⇒ TypeVariable<?>[] getTypeParameters()

WildcardType et GenericArrayType

WildcardType

WildcardType : un wilcard avec soit une borne supérieur soit une borne inférieur

- Les bornes inférieurs ⇒ Type[] getLowerBounds()
- Les bornes supérieurs ⇒ Type[] getUpperBounds()

GenericArrayType

GenericArrayType : un tableau de type paramétré ou de variable de type. Le type contenu du tableau ⇒ Type getGenericComponentType()

Exemple

```
public static void main(String[] args) {
  System.out.println("List " + Arrays.toString(List.class.getTypeParameters()));
  for(Method m:List.class.getMethods()) {
    System.out.println(m.getGenericReturnType()+" "+m.getName()+" "
                      + Arrays.toString(m.getGenericParameterTypes()));
```

```
List [E]
int hashCode []
void add [int, E]
boolean add [E]
int indexOf [class java.lang.Object]
void clear []
boolean equals [class java.lang.Object]
boolean contains [class java.lang.Object]
int lastIndexOf [class java.lang.Object]
boolean addAll [int, java.util.Collection<? extends E>]
...
```

Déroulement du cours

1. Programmation dynamique
2. Class Loader
3. Références faibles

Proxy et invocation

Un mandataire s'interpose devant un autre objet pour intercepter les messages et modifier la façon dont un service est rendu.

Proxy, un mandataires dynamiques

Proxy permet de construire dynamiquement une instance qui implémente un ensemble d'interfaces : Object newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h)

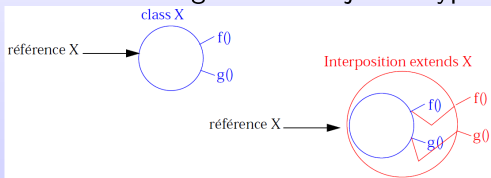
Invocation

L'interface InvocationHandler possède une méthode invoke qui sera appelée pour chaque méthode des interfaces et de Object.
Object invoke(Object proxy, Method method,Object[] args)

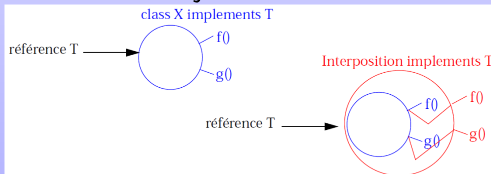
```
public static <T> List<T> traceList(final List<T> list) {
return (List<T>)Proxy.newProxyInstance(ProxyTest.class.getClassLoader(),
                  new Class<?>[]{List.class},new InvocationHandler() {
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
   System.out.println("enter␣"+method);
   try {return method.invoke(list,args);} finally {System.out.println("exit␣"+method);}}};}
```

Réalisation de l'interposition (1)

- Soit avec un objet d'un type qui sous-classe le type de la référence demandée et qui spécialise ses méthodes (pour evenement déléguer à un objet du type demandé



- soit avec un objet qui implante une interface correspondante au type demandé et qui délègue éventuellement à une instance de l'objet initial.



Réalisation de l'interposition (2)

Dans ce dernier cas les mandataires dynamiques (dynamic proxy) permettent de générer dynamiquement des objets d'un type dont la classe effective est définie au runtime. Il peut y avoir plusieurs raisons pour opérer dynamiquement dont, en particulier, le fait que l'interposition ne s'adresse pas forcément à un objet d'un type effectivement connu au compile-time, ou le fait que le comportement rajouté dans l'interposition soit relativement générique (c.a.d s'applique à des ensembles de méthodes).

On pourrait, par exemple, faire de l'interposition dynamique pour faire des appels de méthodes sur des objets distants implantant une certaine interface (une sorte de R.M.I. simplifié sans Stub), ou encore pour interposer des controles avant et après tous les appels de méthodes destinés à un objet (vérification d'invariant de classe, mise en place de transactions,...).

Schéma simplifié d'utilisation pour des appels distants (1)

Une réalisation concrète étant "complexe" nous allons esquisser un mécanisme de mise en place d'appels distants :

```
public interface RemoteService extends Remote {
  Clef remoteGet(Object support, DescripteurMethode factory, Object[] arguments) throws Re
  Object remoteInvoke(Clef support, DescripteurMethode meth, Object[] arguments) throws R
}
```

et maintenant un mécanisme rendant un objet dynamique et basé sur les mécanismes définis par java.lang.reflect.Proxy :

```
public class RemoteFacts {
 private RemoteService server ;
 private class Invoker implements InvocationHandler {
  Clef key ;
  public Object invoke(Object proxy, Method method,Object[] args) throws Throwable {
    // le service demandé sur l'objet est repassé au serveur
    //pour simplifier on n'a pas tenu compte des exceptions
   Object res = server.remoteInvoke(key,
   new DescripteurMethode(method, key), args) ;
   return res ;
  }
 Invoker(Clef key) {this.key = key ;}
}
```

Schéma simplifié d'utilisation pour des appels distants (2)

```
// sert à générer un objet sur le serveur en spécifiant une méthode ''factory'' sur un objet
// le serveur renvoie une ''clef'' qui lui permettra de retrouver l'objet effectif
  public Object remoteObjectAs(Class interf, Object factorySupport,
                               Method factory, Object[] args) throws Exception{
    Clef key = server.remoteGet( factorySupport,new DescripteurMethode(factory), args) ;
    // on génère le Stub dynamique ! (manque les exceptions,etc.
    return Proxy.newProxyInstance (interf.getClassLoader(),
                               new Class[] {interf}, new Invoker(key));
```

Ici l'objet généré par newProxyInstance sera conforme au contrat défini par l'interface demandée, et déléguera des appels de méthodes à un object correspondant connu uniquement sur le serveur. Dans l'exemple c'est l'instance de Invoker qui est chargé d'opérer la délégation. Utilisation :

```
// on ne précise pas ici la nature de l'objet ''banque'' ni le fonctionnement
// des Classes ''Clef'' et ''DescripteurMethode''
CompteDistant cpt = (CompteDistant) rmFact.remoteObjectAs(CompteDistant.class,banque,
                                        getCompteMeth,new Object[]{nomClie
// et maintenant on peut appeler les méthodes de CompteDistant sur l'objet résultant : c'est
// qui implante l'interface demandée
```

Demandez en cours ARCHI WEB pour la notion de BEAN ! ! ! !

java.lang.ClassLoader

Qu'est ce ?

Objet utilisé pour charger dynamiquement les classes Java lorsque celle-ci sont demandées.

Pourquoi ?

Téléchargement d'objets (réseau, fichiers, etc.). Communications d'objets. Accès distants et SGBD.

Par défaut, il existe deux classes loaders :

1. Le classloader primordial des classes du JDK accessibles par le `bootclasspath`, ce classloader n'est pas accessible
2. Le classloader système qui charge les classes de l'application ClassLoader.getSystemClassLoader()

Et on peut de plus créer/instantier son propre classloader

Charger une classe (1)

- Le classloader possède une méthode loadClass() qui permet de charger une classe par son nom (avec le paquetage) :
- si la classe n'est pas trouvée une exception ClassNotFoundException est levée

```
public static void main(String[] args) throws ClassNotFoundException {
  ClassLoader loader=ClassLoader.getSystemClassLoader();
  System.out.println(loader.loadClass("java.lang.Integer"));
}
```

- La classe est chargée mais pas forcément initialisée

Remarque : même plus besoin d'**import** (!)

Charger une classe (2)

Il y a 2 autres façons pour le classloader de la classe courante :

1. Utiliser Class.forName(), qui lève une exception ClassNotFoundException
2. Utiliser la notation **.class**, qui lève une erreur ClassNotFoundError

```
public class Test {
  public static void main(String[] args) throws ClassNotFoundException {
    System.out.println(Integer.class);
    System.out.println(Class.forName("java.lang.Integer"));
    System.out.println(
    Test.class.getClassloader().loadClass("java.lang.Integer"));
```

Il existe aussi URLClassLoader qui peut aller chercher des classes en indiquant plusieurs URLs désignant (des répertoires ou des jars)

```
public static void main(String[] args) throws MalformedURLException, ClassNotFoundExcepti
  URLClassLoader loader = new URLClassLoader(new URL[]{new URL("http://...")});
  Class<?> printerClass = loader.loadClass("reflect.PrinterImpl");
  System.out.println(printerClass); // class reflect.PrinterImpl
```

Class et cache

Class et ClassLoader

- Chaque classe connaît son classloader ; on peut obtenir avec la méthode clazz.getClassLoader()
- Si la classe est chargée par le classloader primordiale, getClassLoader() renvoie **null**

```
public static void main(String[] args) {
  System.out.println(String.class.getClassLoader()); // null
  System.out.println(ClassLoader.getSystemClassLoader()); // sun.misc.Launcher$AppClassLo
  System.out.println(ClassLoaderTest.class.getClassLoader()); //sun.misc.Launcher$AppClassL
```

ClassLoader et cache

- Le mécanisme garantie que si une même classe est chargée deux fois, celui-ci renverra la même instance.
- Pour cela, un classloader possède un cache de toutes les classes qu'il a déjà chargées

```
public static void main(String[] args) throws ClassNotFoundException {
  System.out.println(String.class==Class.forName("java.lang.String")); // true
```

ClassLoader parent

- Un classloader possède un parent (getParent())
- Le mécanisme des classloaders oblige (normalement) un classloader à demander à son parent s'il ne peut pas charger une classe avant d'essayer de la charger lui même.

```
public static void main(String[] args throws MalformedURLException, ClassNotFoundEx
  URLClassLoader loader = new URLClassLoader(new URL[]{new URL("http://...")});
  Class<?> stringClass = loader.loadClass ("java.lang.String");
  System.out.println(stringClass==String.class); // true
```

- String.**class** est chargée par le classloader primordial
- Ce mécanisme de délégation permet à un classloader de connaître les classes systèmes en délégant leurs chargements au classloader primordial
- Primordial $\Leftarrow$ Système $\Leftarrow$ URLClassLoader

Plusieurs class loaders

Une classe reste attachée au classloader qui l'a chargée donc deux classes chargées par des classloaders différents sont différentes.

```
URL url=new URL("http://...");

URLClassLoader loader = new URLClassLoader(new URL[]{url});
Class<?> c1 = loader.loadClass ("reflect.PrinterImpl");
Object o1 = c1.newInstance();


URLClassLoader loader2 = new URLClassLoader(new URL[]{url});
Class<?> c2 = loader2.loadClass ("reflect.PrinterImpl");
Object o2 = c2.newInstance();


System.out.println(c1==c2); // false
System.out.println(o1.equals(o2)); // false à cause du instanceof
```

ClassLoader API

- Les méthodes de chargement des classes :
  - Charge la classe en demandant au père d'abord ⇒ loadClass(String name)
  - Charge une classe localement ⇒ findClass(String name)
  - Insère une classe dans la VM par son bytecode ⇒ defineClass(String name, **byte**[] b, **int** off,**int** len)
- Si l'on veut respecter le mécanisme de délégation on redéfinit findClass et pas loadClass !
- Les méthodes de chargement de resources :
  - Touver une ressources associé à un classloader
    - URL getResource(String name)
    - InputStream getResourceAsStream(String name)
    - Enumeration<URL> getResources(String name)
  - Trouver une ressources système :
    - **static** URL getSystemResource(String name)
    - **static** InputStream getSystemResourceAsStream(String nam)
    - **static** Enumeration<URL> getSystemResources(String name)

Décharger une classe

- Le fait que tout les objets d'une classe ne soient pas atteignable n'est pas suffisant pour décharger une classe
- Chaque classloader possède un cache de l'ensemble des classes chargées donc il faut que le classloader ayant chargé les classes soit collecté pour décharger les classes chargées
- ne pas confondre avec finalize qui est la méthode appelée par le GC lors de la destruction d'un objet !

```java
URLClassLoader loader = new URLClassLoader(new URL[]{new URL("http://...")});
Class<? extends Printer> printerClass
                    = loader.loadClass("reflect.PrinterImpl").asSubclass(Printer.class);
Printer printer= printerClass.newInstance();
printer=null; System.gc(); printerClass=null; // la classe reste stocké dans le classloader
System.gc(); loader=null; System.gc(); // le classloader est collecté, les classes sont déchargée
```

```
$java -verbose:class UnloadClass
...
[Loaded reflect.Printer from /home/gava/...]
[Loaded reflect.PrinterImpl from http://...]
[Unloading class reflect.PrinterImpl]
```

Déroulement du cours

1. Programmation dynamique
2. Class Loader
3. Références faibles

VOIR COURS !

VOIR COURS !

Au travail !