Introspection en JAVA, présentation de l'API Reflection

Par Ricky81

Date de publication : 28 juin 2004

L'introspection consiste en la découverte, de façon dynamique, des informations propres à une classe Java. Ces informations, que nous appellerons méta données, décrivent de façon exhaustive les caractéristiques d'une classe Java (champs, méthodes, ...).
Ce mécanisme est utilisé par la machine virtuelle Java en cours d'exécution, mais également par les outils de développements. L'API Java qui permet ceci est l'API Reflection, nous allons voire à travers cet article à quoi celle-ci peut servir, mais surtout comment vous pourrez l'utiliser pour vos développements Java.

Avant-propos

Il peut parfois arriver de devoir déterminer une caractéristique d'un objet de façon dynamique, ou de pouvoir agir de façon générique sur un objet. Cela pose évidemment des difficultés puisqu'on risque d'établir une liste assez conséquente de traitements devant pouvoir être gérés alors qu'un code générique pourrait résoudre le problème et apporter de la robustesse à la solution mise en place en cas d'évolution des besoins.
Mais pour ce faire, il est nécessaire de disposer d'informations sur les méta données des classes et de pouvoir agir sur un objet en ne connaissant pas le champs ou la méthode concernés au moment de la compilation. L'introspection et l'API Reflection répondent à ces besoins, et cet article va vous présenter les concepts et les possibilités qui s'offrent au développeur.

Cet article se veut donc formateur, j'espère donc que les explications qui vont suivre seront claires. Dans le cas contraire, je suis ouvert à toute critique ou remarque **constructives** ainsi qu'à des propositions d'enrichissement de l'article.
Merci de privilégier les **forums** pour vos questions sur le sujet.

Remerciements

Mes remerciements vont droit au membres de la section Java pour leurs conseils et leurs remarques.

1 - Le byte code et l'introspection

Les binaires créés en Java ne se limitent pas à de simples instructions exécutables. Les binaires (byte code) contiennent également de nombreuses informations plus proches du code Java à l'origine du fichier .class : ce sont les méta données.
Ces informations concernent notamment les différentes classes : classe parente, interface implémentée, champs, méthodes, ... Nous allons détailler dans ce qui suit l'utilité de ce type d'informations.

1.1 - Conséquences au niveau de l'exécution

Le domaine d'application auquel on peut penser en premier est l'exécution. Même si on ne voit pas à première vue à quoi l'utilisation des méta données au cours de l'exécution peut servir, on peut naturellement convenir que si elles n'ont pas d'utilité à l'exécution l'intérêt de les stocker risque d'être fortement critiqué.

Prenons un exemple simple : vous disposez d'une bibliothèque (sous la forme d'un jar) et vous l'utilisez pour écrire un petit programme. Vous compilez et tout fonctionne correctement, jusqu'au jour où une nouvelle version de la bibliothèque devient disponible. Vous vous empressez de remplacer le jar et de relancer votre programme. Et là stupeur ... votre programme ne fonctionne plus et vous obtenez un message équivalent au suivant :

```
Java.lang.NoSuchMethodError
 At MonProg.main(MonProg.java:36)
```

Vous comprenez très facilement qu'une méthode que vous utilisiez a dû disparaître de la librairie. C'est justement grâce aux méta données que la machine virtuelle peut vous renvoyer avec précision le type d'erreur et la localiser.

Cette particularité permet donc un style de programmation dynamique permettant de faire abstraction du contenu de classes et d'écrire un code générique pouvant gérer n'importe quelle classe. Il n'est pas nécessaire de chercher très loin un tel programme ... il suffit de regarder ce que proposent les environnements de développement en matière d'aide à la programmation ou de complétion de code. C'est également le cas pour d'autres langages me direz vous. Et bien c'est beaucoup plus simple en Java, contrairement à ces autres langages pour lesquels les environnements de développement sont obligés de construire leur propre banque d'informations sur les méta données pour proposer ce service.

1.2 - Description des méta données

Pour obtenir des détails sur le format de méta données dans un binaire, vous pouvez vous reporter à des documents de spécification de la machine virtuelle Java.

Les noms des classes utilisés par la JVM sont différents de ceux utilisés par le développeur dans son fichier .java. En effet, la classe **java.lang.String** y sera gérée sous la forme java/lang/String. Pour des raisons de distinction d'un nom de classe au sein d'un flux, il peut arriver que la JVM rajoute les caractères 'L' et ';', on obtiendrait ainsi Ljava/lang/String;.

Les champs et méthodes sont identifiés par leur nom et les types sont codés de la manière suivante :

Type	Identifiant JVM
byte	B
char	C
double	D
float	F
int	I
long	J
short	S
boolean	Z
type[]	[type
paquetage.MaClasse	Lpaquetage.MaClasse;

Pour ce qui est des méthodes, la JVM sait par exemple directement localiser la portion de byte code correspondant à l'implémentation d'une méthode à partir du nom et de la signature de cette dernière.

1.3 - L'utilitaire javap

Il existe bien entendu un certain nombre d'utilitaires permettant d'ordonner et de rendre les informations disponibles dans le byte code compréhensibles par une personne.
L'un de ces outils, **javap**, est disponible dans le SDK et consiste en un décompilateur élémentaire. Il est présent dans le répertoire jdk\bin et sa syntaxe de base est la suivante :

```
javap NomClasse
```

Où NomClasse correspond au nom de la classe (pas d'extension .java) précédée ou non du nom de paquetage. Ainsi, si on se place dans un répertoire classes contenant le paquetage monPaquetage, il faudra appeler :

```
javap monPaquetage.NomClasse
```

alors que si on se place directement dans le répertoire monPaquetage, il suffira d'utiliser la première syntaxe.

Nous pouvons par exemple regarder l'appel suivant sur java.lang.Object, lancé en ligne de commande :

```
E:\>javap java.lang.Object
Compiled from "Object.java"
public class java.lang.Object{
    public native int hashCode();
    static {};
    public java.lang.Object();
    protected void finalize();
       throws java/lang/Throwable
    public final native void notify();
    public final native void notifyAll();
    public final void wait();
       throws java/lang/InterruptedException
    public final native void wait(long);
       throws java/lang/InterruptedException
    public final void wait(long,int);
       throws java/lang/InterruptedException
    public final native java.lang.Class getClass();
    protected native java.lang.Object clone();
       throws java/lang/CloneNotSupportedException
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
}
```

Mieux qu'un simple exemple, testez le par vous même. Vous aurez accès au options de la commande par un classique :

```
javap -help
```

2 - L'API Java Reflection

Nous avons vu jusque là les informations que gérait la JVM, les raisons du stockage de méta données dans les binaires, ainsi que les répercussions lors de l'exécution d'un programme.
Nous allons maintenant voir que le développeur peut également accéder à ces informations dans ses programmes, et ceci grâce à l'API Reflection.
Cette API se conforme bien entendu à tout ce qui a été dit précédemment, et tient notamment compte des spécifications de la JVM dans ce qu'elle propose à l'utilisateur. Elle a donc été amenée à évoluer en même temps que la JVM et c'est encore plus le cas pour le passage à Java 1.5.

Voyons pour commencer, avant de l'approfondir, à quoi peut servir cette API.
Elle peut avant tout être utile pour l'introspection, c'est à dire la **découverte des caractéristiques d'une classe** (classe mère, champs, méthodes, ...). D'une utilité un peu plus avancée, elle permet d'**instancier des classes de manière dynamique**, en agissant sur ses champs, en appelant ses méthodes ...
La réflexion est également potentiellement intéressante pour l'écriture d'un générateur de code générique pour un ensemble de classe.
Cette API sert également dans le **processus de sérialisation** d'un bean Java ou dans la génération du code de création d'un table en base de données pour la persistance de la classe cible.
Enfin, cette API est très utile pour tout outil devant faire abstraction des spécificités d'une application en proposant un service générique pour n'importe quelle classe. On peut par exemple penser à un outil de log (**voir l'article sur l'API Logging par Hugo Etiévant**).

Nous allons approfondir cette API dans les paragraphes qui suivent.

3 - Présentation du paquetage java.lang.reflect

Intéressons nous maintenant au paquetage **java.lang.reflect** dans lequel nous allons trouver les éléments indispensables pour utiliser l'introspection. Mais commençons par évoquer la classe **java.lang.Class** qui est la classe centrale sur laquelle repose la réflexivité.

3.1 - La classe Class

En effet, vous avez sans doute déjà rencontré cette classe à l'occasion d'un :

```
Class.forName(maClasse);
```

qui permet de récupérer un objet de type Class correspondant au nom passé en paramètre.

En fait, les instances de la classe Class peuvent être des classes ou des interfaces. Cette classe est indispensable pour pouvoir manipuler des méta données. Nous verrons que cette classe sera présente dans chaque exemple que nous rencontrerons, dont en voici un premier :

```
public class Exemple
{
   public Exemple()
   {
   }

   public String getNom(Object o)
   {
      Class c = o.getClass();
      return c.getName();
   }
}
```

Dans cet exemple, la méthode *getNom* va renvoyer le nom de la classe de l'objet passé en paramètre.

Cette classe dispose des méthodes permettant d'extraire les informations sur une classe quelconque. En voici quelques unes :

java.lang.reflect.Field getField(String name)	Renvoie un objet Field correspondant au champ "name"
java.lang.reflect.Field[] getFields()	Renvoie l'ensemble des champs publics sous la forme d'un tableau
java.lang.reflect.Method getMethod(String name, Class[] parameterTypes)	Renvoie un objet Method correspondant à la méthode "name" avec les paramètres définis par le tableau parameterTypes
java.lang.reflect.Method[] getMethods()	Renvoie l'ensemble des méthodes publiques sous la forme d'un tableau
java.lang.reflect.Constructor getConstructor(Class[] parameterTypes)	Renvoie un objet Constructor correspondant au constructeur avec

	les paramètres defines par le tableau parameterTypes
java.lang.reflect.Constructor[] getConstructors()	Renvoie l'ensemble des constructeurs sous la forme d'un tableau
Class[] getInterfaces()	Renvoie l'ensemble des interfaces implémentées
Class getSuperclass()	Renvoie la classe mère
java.lang.Package getPackage()	Renvoie un objet Package correspondant au paquetage dans lequel se trouve la classe

Il est à noter que les méthodes getXXX() retournant des informations sur les champs ou les méthodes en considérant également les champs et méthodes héritées.
Nous reviendrons sur ces méthodes ainsi que sur d'autres méthodes volontairement omises à ce niveau de l'article.

3.2 - La classe Field

Venons en à notre paquetage **java.lang.reflect**.
Nous avons vu qu'il était possible de récupérer des méta données sur les champs à partir d'une classe en appelant par exemple la méthode *getField*.
Partant d'un tel objet, nous pouvons consulter les informations à l'aide de méthodes de classe :

String getName()	Renvoie le nom du champ
Class getType()	Renvoie le type du champ
int getModifier()	Renvoie un entier codant la visibilité du champ (private, protected, public) mais également d'autres informations comme static

3.3 - La classe Modifier

Nous avons vu précédemment qu'il était possible de connaître la visibilité d'un champ par l'intermédiaire de la méthode *getModifier()*.
Il peut être surprenant de constater qu'il existe une classe **Modifier** alors que la méthode précédente renvoie un entier. Ce qu'il faut bien avoir à l'esprit pour comprendre ceci, c'est qu'il y a plusieurs informations à l'intérieur de cet entier, et il est nécessaire de faire appel à des méthodes de classe pour les distiller (néanmoins on conviendra qu'il pourrait être plus logique d'adapter la classe **Modifier** pour pouvoir faire renvoyer un objet plutôt qu'un entier).

Voici quelques unes des méthodes utiles :

boolean static isFinal(int mod)	Détermine si le code transmis intègre la particularité "final"
boolean static isPublic(int mod)	Détermine si le code transmis intègre une visibilité publique
boolean static isStatic(int mod)	Détermine si le code transmis intègre la particularité "static"

Ainsi, le code suivant permet de déterminer si le champ 'f' possède une visibilité publique :

```
if(java.lang.reflect.Modifier.isPublic(f.getModifier()))
{
   System.out.println("champ à visibilité publique");
```

```
}
```

Néanmoins, il existe une méthode assez pratique, à savoir *String static toString(int mod)*, qui permet de récupérer une description classique complète des informations.

3.4 - La classe Method

Pourquoi ne pas avoir géré les informations disponibles dans la classe **Modifier** directement dans la classe **Field** me direz vous ? De telles informations sont justement également disponibles pour les méthodes et il est donc plus censé de les factoriser dans une classe spécifique.

Pour en revenir à nos méthodes, nous avons vu qu'il était possible de récupérer un objet décrivant une méthode, et nous allons voir ce qu'il est possible d'en extraire comme informations :

Class[] getExceptionTypes()	Renvoie un tableau contenant les classes Exception déclarées comme pouvant être lancées
String getName()	Renvoie le nom de la méthode
Class getReturnType()	Renvoie la classe du paramètre retourné par la méthode
Class[] getParameterTypes()	Renvoie un tableau contenant les classes des paramètres de la méthode
int getModifiers()	Renvoie un entier codant la visibilité de la méthode (private, protected, public) mais également d'autres informations comme static

3.5 - La classe Constructor

Enfin, dans le même esprit que pour la classe **Method**, on retrouve un classe **Constructor** qui permet de récupérer des informations sur un constructeur.

Class[] getExceptionTypes()	Renvoie un tableau contenant les classes Exception déclarées comme pouvant être lancées
String getName()	Renvoie le nom de la méthode
Class[] getParameterTypes()	Renvoie un tableau contenant les classes des paramètres de la méthode
int getModifiers()	Renvoie un entier codant la visibilité de la méthode (private, protected, public) mais également d'autres informations comme static

3.6 - Cas particulier des types primitifs

Le lecteur averti aura sans doute une question concernant la compatibilité entre la classe **Class** et les types primitifs de Java comme int, long, char, ...
Ces types n'étant pas des objets, comment se passe la compatibilité avec les méthodes que nous avons vues précédemment et qui renvoient des objets de type **Class** ? Et bien ce sont les types objets enveloppe correspondants (Integer, Long, Char, ...) qui seront renvoyés.
Même remarque lorsque nous tenterons d'appeler une méthode dynamiquement à l'aide de la réflexivité : il faudra passer comme paramètre un objet et non le type primitif qui est utilisé au niveau de la méthode, mais également caster le résultat renvoyé dans le type objet enveloppe correspondant (cela ne devrait plus être nécessaire avec Java 1.5).

Néanmoins, se pose alors une autre question. A savoir : comment distinguer 2 méthodes entièrement identiques ne différant que par un type lequel est pour l'une un type primitif, et pour l'autre le type objet enveloppe correspondant ? Prenons l'exemple du couple Integer/int : pour le premier ce sera *Integer.class* et pour le second *Integer.TYPE*. Ainsi, dans l'exemple suivant :

```
Class c = Class.forName("maClasse");
Class[] p1 = new Class[]{Integer.class, Integer.class);
Class[] p2 = new Class[]{Integer.TYPE, Integer.TYPE);
Method m1 = c.getMethod("somme",p1);
Method m2 = c.getMethod("somme",p2);
```

m1 correspond ici à la méthode *somme(Integer, Integer)* alors que m2 correspond à *somme(int, int)*.

4 - Utilisation de la réflexivité - Exemples d'appels dynamiques

Voyons à présent quelques exemples d'utilisation pratique de la réflexivité.
Nous allons dans un premier temps essayer de modifier le contenu du champ d'un objet de façon dynamique en prenant pour paramètres un objet, le nom du champ qu'on souhaite modifier, et la nouvelle valeur. Puis, nous travaillerons au niveau des méthodes pour vous montrer ce qu'il est possible de réaliser.

4.1 - Edition d'un champ

Imaginez le problème suivant : vous souhaitez pouvoir modifier la valeur du champ d'un objet de façon dynamique. Vous allez donc passer en paramètre le nom de ce champ sous la forme d'un String, mais vous ne pourrez pas agir sur ce champ avec les moyens classiques. Considérons donc une telle méthode :

```
void changeValeur(Object o, String nomChamp, Object val)
```

Il n'est bien entendu pas possible d'accéder au contenu du champ comme en programmation statique par :

```
o.nomChamp;
```

Nous sommes donc obligés de faire appel à la réflexivité.
Voici le code que nous allons par la suite commenter :

```
void changeValeur(Object o, String nomChamp, Object val) throws Exception
{
   Field f = o.getClass().getField(nomChamp);
   f.set(o,val);
}
```

Nous récupérons donc un objet de type Field correspondant au champ concerné par la modification, puis nous faisons appel à la méthode *set* sur ce champ qui permet de modifier le contenu du champ d'un objet (ici o) en lui attribuant la valeur passée en second paramètre.
La méthode *set* est la méthode la plus générale pour faire une affectation sur un objet, mais il existe des méthodes plus restreintes tel que *setDouble(Object obj, double d)* ou *setBoolean(Object obj, boolean z)*.

Il y a bien entendu la possibilité d'utiliser des méthodes de consultation, ce qui permet d'écrire une méthode générique d'affichage du contenu d'un champ d'un objet :

```
void afficheValeur(Object o, String nomChamp) throws Exception
{
   Field f = o.getClass().getField(nomChamp);
   System.out.println(f.get(o));
}
```

4.2 - Appel d'une méthode

Après avoir vu qu'il était possible d'agir sur les champs d'un objet à l'aide de la réflexivité, nous allons maintenant nous intéresser aux méthodes. Nous allons ainsi pouvoir lancer une méthode sur un objet de façon entièrement dynamique, en gérant bien entendu le passage des paramètres souhaités à la méthode.

Nous avons vu une certaine catégorie de méthodes pour la classe Method, à savoir ce qui concerne l'interrogation des méta données d'une méthode.

Comme nous avons vu comment récupérer la valeur d'un champ et la modifier, nous allons voir maintenant la méthode à utiliser pour lancer dynamiquement une méthode sur un objet ou une classe. La méthode en question est *invoke* dont voici la déclaration :

```
Object invoke(Object obj, Object[] args)
```

Dans le même esprit des exemples précédents, nous pouvons alors déclarer la méthode suivante :

```
Object lancerMethode(Object o, Object[] args, String nomMethode) throws Exception
{
  Class[] paramTypes = null;
  if(args != null)
  {
    paramTypes = new Class[args.length];
    for(int i=0;i<args.length;++i)
    {
      paramTypes[i] = args[i].getClass();
    }
  }
  Method m = o.getClass().getMethod(nomMethode,paramTypes);
  return m.invoke(o,args);
}
```

Cette méthode permet donc de lancer une méthode sur un objet, et éventuellement de récupérer un résultat, et ceci en récupérant la liste des paramètres et le nom de la méthode. Attention cependant : comme cela a été dit précédemment, la recherche de la méthode par *getMethod* va examiner les types des paramètres et il est donc indispensable de ne pas se tromper au niveau des paramètres. De plus, cela pose quelques problèmes pour les méthodes génériques ayant pour paramètre le type **Object** mais qu'on désire appeler avec un paramètre d'un type dérivé. En effet, *getMethod* cherchera la méthode dont le paramètre a pour type le type dérivé en question, et comme il ne le trouvera pas il renverra une exception.
Je vous laisse chercher une solution à ce problème ...

> Dans le cas d'une méthode de classe, le paramètre *obj* de la méthode *invoke* ne sera pas pris en compte. Vous pouvez tout aussi bien mettre *null* que n'importe quel objet.

5 - Réalisation d'un inspecteur de classe

Voici un exemple très basique de ce qu'on peut faire pour consulter les informations à visibilité publique d'une classe en affichant les informations avec le classique System.out.

```
import java.lang.reflect.*;

public class Explorateur
{
   public Explorateur()
   {
   }

   public void explorerChamps(Object o)
   {
      Field[] f = null;
      Class c = null;

      c = o.getClass();
      f = c.getFields();
      consulterChamps(f,o);
   }

   public void explorerMethodes(Object o)
   {
      Method[] m = null;
      Class c = null;

      c = o.getClass();
      m= c.getMethods();
      consulterMethodes(m);
   }

   private void consulterChamps(Field[] f, Object o)
   {
      for(int i=0;i<f.length;++i)
      {
         System.out.print(Modifier.toString(f[i].getModifiers()));
         System.out.print(" ");
         System.out.print(f[i].getType().getName());
         System.out.print(" ");
         System.out.print(f[i].getName());
         System.out.print(" = ");
         try
         {
            System.out.println(f[i].get(o));
         }
         catch (IllegalAccessException e)
         {
           System.out.println("Valeur non consultable");
         }
      }
   }

   private void consulterMethodes(Method[] m)
   {
      Class[] params = null;
      for(int i=0;i<m.length;++i)
      {
         System.out.print(Modifier.toString(m[i].getModifiers()));
         System.out.print(" ");
         System.out.print(m[i].getReturnType().getName());
         System.out.print(" ");
         System.out.print(m[i].getName());
         System.out.print("(");
         params = m[i].getParameterTypes();
         for(int j=0;j<params.length;++j)
         {
           System.out.print(params[j].getName());
```

```
            }
            System.out.println(")");
        }
    }

}
```

6 - Au delà des règles de l'encapsulation

Ce que permet de faire l'API Reflection dépasse largement le cadre de l'encapsulation si chère aux langages de programmation orientés objet. En effet, jusqu'à présent je vous ai exposé des éléments permettant de consulter et d'agir sur des champs et méthodes publics. Mais nous étions loin de ce qui est possible puisque cette API permet d'inspecter tous les éléments d'une classe, quelle que soit sa visibilité. Ces possibilités sont à l'origine de nombreux abus dont nous verrons l'un ou l'autre exemple.

6.1 - Champs et méthodes protégés et privés

Nous avons vu notamment au niveau de la classe **Class** qu'il était possible de récupérer un champ ou une méthode à l'aide de *getFields* ou de *getMethods*. Nous étions alors capables de récupérer les champs et méthodes publics.
Vous aurez sans doute remarqué la présence de méthodes semblables appelées *getDeclaredFields* et *getDeclaredMethods* au niveau de la documentation. C'est grâce à ces méthodes que nous allons pouvoir transgresser une première fois les règles de l'encapsulation en étant capable de consulter les caractéristiques des champs et méthodes protégés et privés.

***Remarque importante** : contrairement aux méthodes getFields et getMethods, getDeclaredFields et getDeclaredMethods ne renvoient pas les informations héritées. Dans ce cas, il est nécessaire d'aller interroger la classe mère.*

6.2 - Pourquoi l'API permet ceci ?

Quelles sont les raisons pour lesquelles de telles fonctionnalités sont disponibles dans l'API Reflection ? L'origine du besoin se situe toujours au niveau de la machine virtuelle qui se doit de pouvoir explorer tous les éléments d'une classe, mais surtout la nécessité de disposer des méta données pour pouvoir détecter les erreurs en cours d'exécution.

6.3 - La méthode setAccessible

Encore pire ...
Nous allons transgresser une seconde fois, mais cette fois ci avec des conséquences bien plus graves, en supprimant les verrous en lecture/écriture des éléments protégés et privés.
En effet, même si vous avez pu récupérer les informations sur un champ privé, vous ne serez capables ni de lire, ni de modifier le contenu de ce champ (déclenchement de **IllegalAccessException**).
Néanmoins, il existe pour les différents éléments (Field, Method, Constructor) la méthode *setAccessible(boolean b)* qui permet justement de faire sauter le verrou de sécurité. Grâce à cette méthode, il est maintenant très aisé de consulter l'ensemble des champs pour une classe donnée. Mais il est désormais surtout possible de modifier la valeur d'un champ et de lancer une méthode privée.

6.4 - Un exemple

Prenons une classe **Secret** avec un champ privé *priv* (String).
L'exemple suivant va modifier ce champ privé :

```
void modifierChamp(Secret s, String val)
{
   Field f = s.getClass().getDeclaredField("priv");
   f.setAccessible(true);
   f.set(s,val);
}
```

6.5 - En cas de besoin, des protections existent

Il existe néanmoins des possibilités pour combler ce manque de protection.
En effet, la méthode *setAccessible* est définie dans la classe **AccessibleObject** (dont dérivent notamment les classes **Field**, **Method** et **Constructor**). Il est alors possible de définir qui possède le droit d'appeler la méthode *setAccessible* en définissant un **SecurityManager**.
Ceci dépassant le cadre de cet article, je vous laisse rechercher davantage d'informations en consultant le paquetage **java.security**.

7 - Quelques liens

- **Source de l'explorateur de classe**
- **Source de l'explorateur non limité aux données publiques**

- **L'API Reflection**
- **Présentation de l'API Logging par Cyberzoïde**
- **Documents de spécification de la machine virtuelle Java**
- **La FAQ JAVA**
- **Le forum général JAVA**