

Le décompilateur JAD

par Gildas Cuisinier ([Hikage](#)) ([Blog](#))

Date de publication : 27/06/2007

Dernière mise à jour : 27/06/2007


Cet article a pour but d'expliquer ce qu'est un décompilateur Java.
Il présentera le décompilateur JAD et son intégration dans Eclipse grâce à un plugin
Il finira par une légère introduction à la notion d'obfuscation, et ce par un petit exemple avec le logiciel ProGuard

- I - Qu'est-ce qu'un décompilateur
 - I-A - Petit rappel théorique
 - I-B - A quoi ressemble du bytecode ?
 - I-C - Décompilateur
- II - Le décompilateur JAD
 - II-A - Installation
 - II-B - Exemples d'utilisation
 - II-B-1 - Décompiler un fichier class unique
 - II-B-2 - Décompiler dans un fichier
 - II-B-3 - Décompiler tout un package en gardant la hiérarchie des répertoires
 - II-B-4 - Autres options intéressantes
- III - Jadclipse
 - III-A - Présentation
 - III-B - Installation
 - III-C - Configuration
 - III-D - Utilisation
- IV - Annexe A : Obfuscation
 - IV-A - Petit exemple avec ProGuard
- V - Remerciements

I - Qu'est-ce qu'un décompilateur

I-A - Petit rappel théorique

Contrairement à des langages tels que le C ou le C++, qui sont des langages compilés, Java fait partie d'une autre famille qu'on appelle langages semi-compilés.

C'est à dire que le compilateur Java ne produira pas du code binaire, qui aurait pu être compris directement par le système d'exploitation, mais un code intermédiaire communément appelé  **bytecode**.

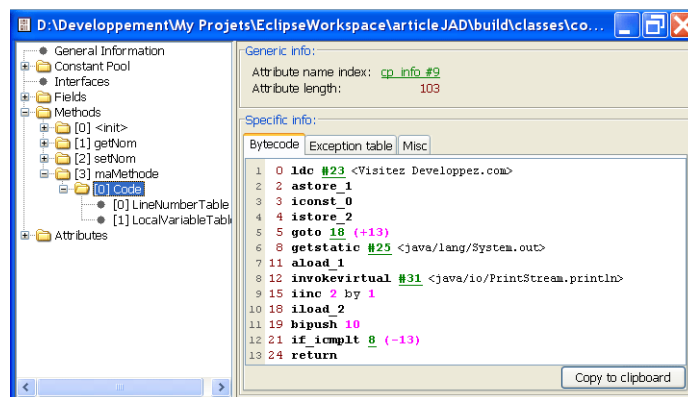
La conséquence de cela est qu'un programme Java n'est pas "autoexécutable", et qu'il a besoin d'une machine virtuelle Java, dont le rôle est d'interpréter ce bytecode et d'exécuter les appels systèmes correspondants.

Ce bytecode est stocké dans les fichiers .class que génère le compilateur Java.

I-B - A quoi ressemble du bytecode ?

Afin de mieux comprendre, nous allons compiler la classe `com.developpez.hikage.bytecode.ByteCode`, et visualiser le fichier .class grâce à un outil.

Dans ce screenshot, on peut voir différentes sections, telles que Fields, Constants, Methods, Interfaces ...



ByteCode dans Bytecode Viewer

Etudions un peu le bytecode de la méthode `maMethode` :

Bytecode de `maMethode`

```
0 ldc #23 <Visitez Developpez.com>
2 astore_1
3 iconst_0
4 istore_2
5 goto 18 (+13)
8 getstatic #25 <java/lang/System.out>
11 aload_1
12 invokevirtual #31<java/io/PrintStream.println>
15 iinc 2 by 1
18 iload_2
19 bipush 10
21 if_icmplt 8 (-13)
24 return
```

Bytecode de maMethode

Ce qui donne comme suite d'exécution :

- ligne 0 : chargement de la constante 23 (une chaine "Visitez Developpez.com") sur la pile
- ligne 2 : stockage de la variable de la pile (donc la chaine précédente) dans la première variable locale
- ligne 3 : chargement de la valeur entière 0 sur la pile
- ligne 4 : stockage de la variable de la pile (donc la valeur 0) dans la deuxième variable locale
- ligne 5 : saut à la ligne 18
- ligne 18 : chargement d'une valeur entière provenant de la variable 2 vers la pile
- ligne 19 : chargement de la valeur 10 sur la pile (valeur sur la pile : 10 suivi de la valeur de la variable 2)
- ligne 21 : si la dernière valeur de la pile est plus petite que l'avant dernière valeur de la pile, alors saut en ligne 8
- ligne 8 : chargement du champ "out" de la classe java.lang.System
- ligne 11 : chargement de la valeur de la variable 1 ("Visitez Developpez.com") sur la pile
- ligne 12 : appel de la méthode println de la classe java.io.PrintStream (le champ out de ligne 8)
- ligne 15 : incrémentation de la variable 2 par 1
- ligne 24 : fin de la méthode

En français, on pourrait dire que la logique de cette méthode est de boucler sur un compteur allant de 0 à 10, et d'appeler à chaque itération la méthode **println(String texte)** de l'objet System.out.

Pourriez-vous tenter de réécrire cette méthode en Java ?

Code de la classe java

```
package com.developpez.hikage.bytecode;

public class ByteCode {

    private String nom;

    public String getNom() { return nom; }

    public void setNom(String nom) { this.nom = nom; }

    public void maMethode(){

        String texte = "Visitez Developpez.com";

        for(int i = 0 ; i < 10 ; i++){
            System.out.println(texte);
        }
    }
}
```

Aviez-vous trouvé? ;-)

I-C - Décompilateur

Dès que l'on connaît les opérations bytecode, ainsi que le langage Java, et que l'on est doté d'une bonne logique, il devient très simple de traduire du bytecode en code source Java.

Et bien c'est exactement ce que fait un décompilateur.

De plus, hormis la logique, d'autres informations sur la classe sont stockées dans le fichier .class : le nom des champs, le nom des méthodes, les exceptions qu'une méthode est susceptible de lancer, etc ...

Grâce à toutes ces informations, le code source peut être restitué exactement !!


Enfin presque. En effet, les commentaires ne sont pas enregistrés lors de la compilation, et sont réellement perdus. Cela dit, le code est parfois plus clair ainsi.

Un seul soucis apparait lors de debug. En effet, du fait de la disparition des commentaires, les points d'arrêts risquent d'être mal placés.

Par exemple si vous mettez un point d'arrêt à la ligne 5 dans un code décompilé, il est tout à fait possible que cette ligne correspondait au départ à une ligne de commentaire, et donc le point d'arrêt ne sera jamais atteint.

Facheux ?

Non, car les informations sur l'emplacement d'une ligne de code sont aussi stockées dans le fichier .class, et donc certains décompilateurs (dont JAD) permettent de recréer le fichier source avec la même structure que le fichier original, en remplaçant les lignes de commentaires par des lignes blanches.

 *Il faut cependant avouer que le code restitué n'est plus "totalement" le même avec Java 5 dans le cas particulier des génériques.*

En effet, les génériques de Java sont vérifiés durant la phase de compilation (et généreront une erreur si un type ne correspond pas à celui attendu), mais sont ensuite traduits en Object. Autrement dit une List<String> sera une simple List dans le bytecode compilé, il devient donc impossible à JAD de rendre la notion de génériques.

II - Le décompilateur JAD

II-A - Installation

JAD est disponible ici : **JAD**.

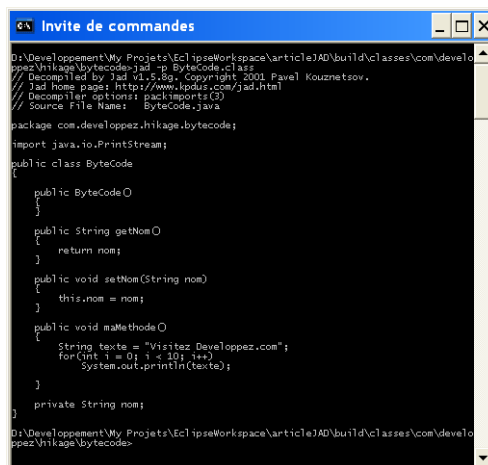
C'est un simple exécutable sous forme de fichier .exe. Je conseille de le décompresser dans c:\windows\system32\ ou dans le répertoire bin de votre JRE/JDK afin d'être directement dans le PATH de windows.

 *Il est à noter que JAD est gratuit pour un usage non commercial, dans le cas contraire, il faut contacter son auteur à l'adresse jad@kpdus.com*

II-B - Exemples d'utilisation

II-B-1 - Décompiler un fichier class unique

```
jad -p maclasse.class
```



```
Invite de commandes
D:\Developpement\My_Projets\EclipseWorkspace\articleJAD\build\classes\com\develo
pp2\hikage\bytecode>jad -p ByteCode.class
// Decompiled by JAD v1.5.89, Copyright 2001 Pavel Kouznetsov.
// JAD home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   ByteCode.java

package com.developpez.hikage.bytecode;
import java.io.PrintStream;
public class ByteCode
{
    public ByteCode()
    {
    }

    public String getNom()
    {
        return nom;
    }

    public void setNom(String nom)
    {
        this.nom = nom;
    }

    public void maMethod()
    {
        String texte = "Visitez Developpez.com";
        for(int i = 0; i = 10; i++)
            System.out.println(texte);
    }

    private String nom;
}
D:\Developpement\My_Projets\EclipseWorkspace\articleJAD\build\classes\com\develo
pp2\hikage\bytecode>
```

II-B-2 - Décompiler dans un fichier

```
jad -sjava -dsrc maclasse.class
```

- -s : extension du fichier, par défaut .jad
- -d : répertoire de destination

II-B-3 - Décompiler tout un package en gardant la hiérarchie des répertoires

```
jad.exe -r -sjava -dsrc ./**/*.class
```

- -s : extension du fichier, par défaut .jad

- -d : répertoire de destination
- -r : recrée la hiérarchie de répertoire
- ./**/*.class : pattern style ant

II-B-4 - Autres options intéressantes

- -o : Ecrase un fichier s'il existe déjà
- -lnc : Ajoute en commentaire le numéro de la ligne de code du fichier original
- -ff : Met les champs de classe avant la définition des méthodes (sinon les champs seront tout en bas du fichier)
- -t<num> : remplace les tabs par 'num' espaces
- -t : remplace les espaces par des tabulations
- -8 : convertit des chaînes Unicode vers des chaînes ANSI

III - Jadclipse

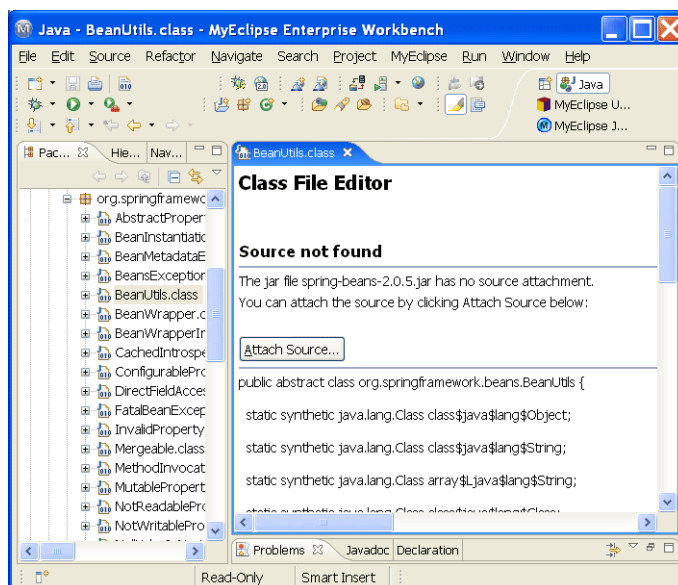
III-A - Présentation

Ne vous est-il jamais arrivé d'utiliser des bibliothèques Open Source et en pleine exécution de debug, vouloir comprendre ce que fait une d'entre elles ?

Dans ce cas, vous étiez obligés de télécharger les sources correspondantes à la version que vous utilisez, et de configurer votre IDE afin de lui indiquer le chemin des sources pour celle-ci.

Mais avec des librairies comme Spring ou Hibernate, cela peut être assez long, tout cela pour voir le fichier source d'une seule classe.

Autre cas possible, vous êtes dans l'impossibilité d'utiliser une connexion Internet, et il vous est alors impossible de télécharger ces sources.



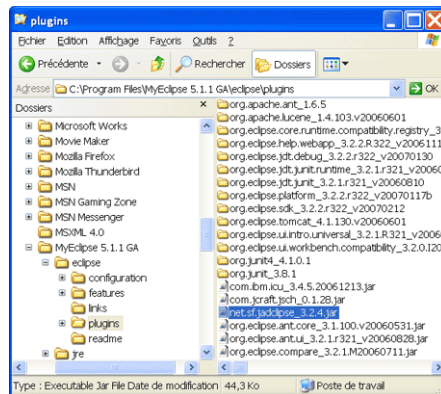
Un fichier class, sans les sources associées

C'est dans des cas pareils que Jadclipse devient très intéressant. En effet, lorsque Eclipse affichera le fichier .class de la bibliothèque, Jadclipse va utiliser JAD pour décompiler la classe et afficher le résultat dans un éditeur Java. Et tout cela de manière totalement transparente.

III-B - Installation

L'installation du plugin est assez simple, c'est un Jar à télécharger [ici](#) . Il existe pour les versions d'Eclipse 3.1, 3.2 et 3.3.

Il suffit de le placer dans le répertoire de plugin d'Eclipse, et de le relancer.

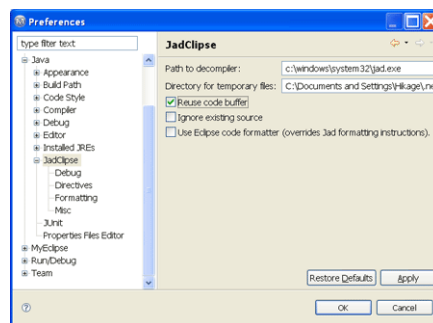


Installation du plugin Jadclipse

III-C - Configuration

Pour configurer Jadclipse, allez dans le menu *Windows* -> *Préférence*, ouvrez la section *Java* -> *Jadclipse*.

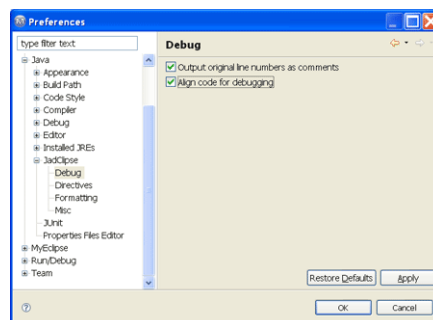
Vérifiez que le chemin vers l'exécutable jad est correct.



Configuration de Jadclipse

Ensuite dans la sous-section *debug*, cochez :

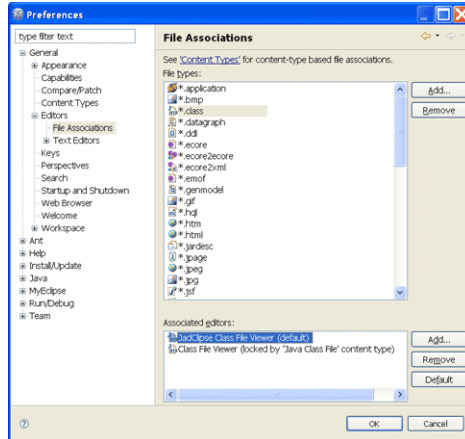
- Output original line numbers as comment
- Align code for debug



Configuration de Jadclipse

Cela permet d'aligner le code tel qu'il était sans le fichier source de départ, en remplaçant les commentaires par des lignes blanches, et ainsi permettre de placer des points d'arrêts correctement.

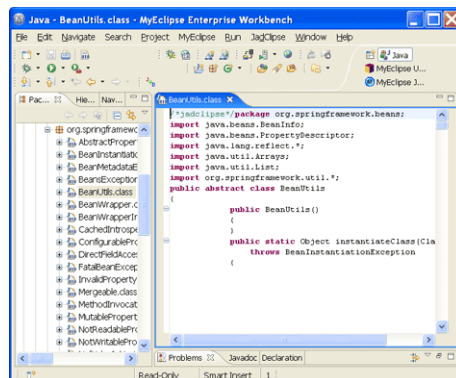
Vérifier ensuite que les fichiers .class sont bien associés à l'éditeur Jadclipse, en allant dans la section *General -> Editor -> File associations*



Configuration de Jadclipse

III-D - Utilisation

Et miracle :



Utilisation de jadclipse

N'est-ce pas merveilleux ?


IV - Annexe A : Obfuscation

Il existe cependant une méthode capable de protéger le code source : l'obfuscation

En gros, on utilise un programme que l'on appelle obfuscateur tel que ProGuard. Ce programme va modifier le bytecode afin de remplacer les noms de méthodes, les noms des champs par des noms totalement abstraits. Cela n'empêche pas un décompilateur de faire son travail, mais le code ainsi régénéré est nettement plus difficile à lire.

Certains obfuscateurs vont même jusqu'à ajouter du bytecode, qui ne sera jamais réellement atteint, afin de tromper le décompilateur qui n'arrive plus à comprendre ce code.

Cependant, ce genre de processus n'est pas possible sur n'importe quel code. En effet, on ne peut pas obfusquer les interfaces publiques d'une API au risque de ne plus permettre son utilisation correcte.

 Une information importante est qu'un Obfuscateur comme ProGuard va, en plus de rendre illisible, optimiser le code. Il en résulte parfois ainsi du code plus léger et plus rapide.

IV-A - Petit exemple avec ProGuard

Voici notre classe ByteCode, ainsi qu'une classe TestMain qui l'utilise simplement.

Byte code, source originale

```
package com.developpez.hikage.bytecode;

public class ByteCode {

    private String nom;

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public void maMethode(){

        String texte = "Visitez Developpez.com";

        for(int i = 0 ; i < 10 ; i++){
            System.out.println(texte);
        }
    }
}
```

TestMain, source originale

```
package com.developpez.hikage.bytecode;

public class TestMain {

    /**
     * @param args
     */
    public static void main(String[] args) {

        ByteCode code = new ByteCode();
    }
}
```

TestMain, source originale

```
code.setNom("test");

code.maMethode();

}

}
```

Et voici le résultat après une obfuscation grace à ProGuard, et une décompilation avec JAD :

ByteCode, après obfuscation

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)

package test;

import java.io.PrintStream;

public final class a
{

    public a()
    {

    }

    private String Code()
    {
        return Code;
    }

    public final void Code(String s)
    {
        Code = s;
    }

    public static void Code()
    {
        String s = "Visitez Developpez.com";
        for(int i = 0; i < 10; i++)
            System.out.println(s);
    }

    public String Code;
}

}
```

TestMain, après obfuscation

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)

package test;

// Referenced classes of package test:
//     a

public final class b
{

    public b()

}
```

TestMain, après obfuscation

```
{  
  
private static void Code()  
{  
    a a1;  
    (a1 = new a()).Code = "test";  
    a.Code();  
}  
}
```

On remarque très rapidement que le code est nettement moins lisible :

- Les méthodes ont été renommées, avec le même nom tant que leur signature ne sont pas identiques
- La variable **privée** nom à été placée en **public**, et les appels aux modificateurs/accesseurs ont été remplacés par des appels sur la variable elle même
- Les noms de package ont été renommés (com.developpez.hikage.bytecode => test)

Mais on peut remarquer aussi que la méthode **static void main (String args[])** a disparue .. et donc il n'y a plus moyen de lancer l'application !

Il existe bien sûr des options dans ProGuard, afin de spécifier quelles modifications sont permises, ainsi que d'exclure certaines classes et méthodes de toutes modifications.

V - Remerciements

Un grand merci à Dutmatlab et ddm pour la relecture orthographique de mon article, ainsi qu'à wichtounet et Ricky81.

