

# Le format class

Philippe PRADOS

[pp@philippe.prados.name](mailto:pp@philippe.prados.name)



*Préservez l'environnement,  
n'imprimez pas ce document*

## TABLE DES MATIERES

1.	Les classes .....	3
2.	Le pool de constante .....	4
3.	Les champs .....	5
4.	Les méthodes .....	5
5.	Le code .....	6
6.	Les exceptions.....	6

## Avant propos

*Cet article décrit le format des fichiers class de java. Il explique comment est structuré ces fichiers et comment les analyser.*

Nous avons vu dans des articles précédant comment rédiger un classloader afin d'intégrer des classes venant de différents horizons. Une sous-classe de la classe classloader doit invoquer la méthode `defineClass()` en lui fournissant un tableau d'octet dont le format est celui des fichiers class.

Un fichier class est généré par le compilateur java. Il possède l'ensemble des informations décrivant la classe, et le p-code interprété par la machine virtuelle pour exécuter les traitements.

Nous allons étudier ce format de fichier. Cela nous permettra par exemple de générer dynamiquement des classes et de les injecter dans la machine virtuelle. Une routine produit un tableau au format class, et un classloader l'installe dans le JVM.

Vous trouverez toutes les spécifications de ce format à l'URL suivante :

<http://www.javasoft.com/docs/books/vmspec/2nd-edition/html/ClassFile.doc.html>

La structure de ce fichier est bien faite. En effet, elle est ouverte. Elle permet d'ajouter dans un fichier class des informations supplémentaires, non définies dans les spécifications. Cela permet par exemple de manipuler le fichier avant l'invocation de la méthode `defineClass()` du classloader.

Par exemple, imaginons que nous désirons ajouter des informations sur certaines méthodes pour signaler qu'elles ne font que lire l'instance. Java ne propose pas le concept de méthode constante comme en C++. Il est possible d'écrire un compilateur Java spécifique (à partir d'une souche en open source par exemple), et d'ajouter ce concept. Les méthodes `const` ne font que consulter l'instance. Elles ne peuvent pas les modifier.

```
class MonObjet
{
    private int attribut_;
    public void setAttribut(int attribut)
    {
        attribut_=attribut;
    }
    public int getAttribut() const
    {
        return attribut_;
    }
}
```

La méthode `MonObjet.getAttribut()` ne fait que consulter l'instance.

Un compilateur particulier peut analyser l'attribut `const` indiqué après le nom de la méthode. Il ajoute alors dans le fichier class une information indiquant que la méthode est constante. Le classloader peut alors détecter cela lors du chargement et de l'analyse du fichier class. Il peut transformer dynamiquement le fichier class en mémoire afin, par exemple, d'y injecter dans les méthodes non constantes l'invocation d'une méthode `setDirty()` pour signaler que l'instance a été modifiée. Cela permet à des frameworks de persistance de sauver dans la base de donnée que les objets ayant été modifiés. Les instances dont seules des méthodes constantes ont été invoquées n'ont pas besoin d'être sauveés sur disque. Nous savons qu'elles n'ont pas été modifiées. Sans cette information, le framework ne peut savoir quels sont les objets modifiés.

## 1. LES CLASSES

```
ClassFile
{
    u4 magic; // 0xCAFEBABE
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Regardons maintenant le format des fichiers class. Il commence par un ensemble d'en-tête. Le premier indique, comme souvent, un nombre magique sur quatre octets pour signaler du format du fichier. Il doit avoir la valeur hexadécimale 0xCAFEBABE.

Cela me rappelle une anecdote. Il y a longtemps, je travaillai sur la réalisation d'un terminal graphique avec trucage vidéo en temps réel en architecture 8 bits. L'IBM PC n'existait pas. Pour déterminer la machine, nous avions un débogueur externe. Nous remplacions le microprocesseur par un câble et un périphérique nous permettant d'exécuter pas à pas le boot de la machine, de consulter les registres, etc. Un jour, le programme est parti dans une boucle sans fin. Nous avons alors demandé l'interruption violente du programme. La machine a affiché

l'adresse actuelle d'exécution du programme. Le mot CAFE est apparu. Cela nous a alors fortement incité à faire une pause ! Depuis, je sais que 51.966 est un café en hexadécimal... ;-)

Les champs suivant de l'en-tête indique les numéros de versions majeure et mineur du format du fichier.

Les informations suivantes servent de racine à toutes les informations du fichier. Différents pointeurs permettent de connaître le pool de constante, le nom de la classe, de sa super-classe, des interfaces implémentées, les différents champs, les méthodes et un ensemble d'attributs.

## 2. LE POOL DE CONSTANCE

Pour comprendre le format class, il faut maîtriser le pool de constante. Les classes référencent d'autres classes, invoquent des méthodes, utilisent des constantes numériques. Toutes ces informations peuvent être redondantes. Par exemple, deux méthodes différentes peuvent invoquer la méthode `String.toString()`. Pour éviter d'avoir plusieurs fois la même chaîne de caractères pour référencer la même méthode, un tableau regroupe toutes les informations pouvant être partagées par la classe. Les méthodes n'ont alors qu'à indiquer un numéro de constante dans le pool pour référencer la méthode. Un entier sur 16 bits (`u2`) permet de référencer une information dans le pool. Le champ `this_class` et `super_class` par exemple, possède un index dans le pool de constante. Nous trouvons à l'adresse `constant_pool[this_class-1]` le nom de la classe.

Le format des constantes est le suivant :

```
cp_info
{ u1 tag;
  u1 info[];
}
```

Un tag indique le type de la constante, puis les informations suivent. Le Tableau 1 indique les différentes valeurs possibles. On constate que les constantes numériques des classes sont indiquées dans ce tableau. Lorsqu'une méthode utilise par exemple le nombre `3.14`, il se retrouve dans le tableau de constante. Le p-code va référencer cette valeur à l'aide d'un index.

Type	Valeur
<code>CONSTANT_Utf8_info</code>	1
<code>CONSTANT_Integer</code>	3
<code>CONSTANT_Float</code>	4
<code>CONSTANT_Long</code>	5
<code>CONSTANT_Double</code>	6
<code>CONSTANT_Class</code>	7
<code>CONSTANT_String</code>	8
<code>CONSTANT_Fieldref</code>	9
<code>CONSTANT_Methodref</code>	10
<code>CONSTANT_InterfaceMethodref</code>	11
<code>CONSTANT_NameAndType</code>	12

Tableau 1

Utiliser un tag pour qualifier les constantes est une technique pour améliorer la sécurité de Java. En effet, la référence sur le nom d'une classe ne peut pointer vers une `CONSTANT_String` par exemple. Même si les informations sont similaires, la typologie permet de vérifier l'usage des constantes.

Java utilise une grammaire particulière pour décrire les types des objets. Vous pouvez trouver quelques exemples Tableau 2.

Exemple	UTF8
<code>String</code>	<code>Ljava/lang/String;</code>
<code>int</code>	<code>I</code>
<code>int[]</code>	<code>[I</code>
<code>int[][]</code>	<code>[[I</code>
<code>Thread[]</code>	<code>[Ljava/lang/Thread;</code>

Tableau 2

Avec les informations décrites dans les spécifications, il est facile de rédiger un petit programme analysant un fichier class et récupérant toutes les constantes en mémoire. L'analyse de la suite du fichier en est facilitée.

Les en-têtes de `ClassFile` permettent de continuer notre exploration de ce format. Un premier ensemble de drapeau permet de décrire les différents attributs d'une classe. Le Tableau 3 indique les différentes valeurs possibles pour les différents bits. Par exemple, une classe `abstract public` doit avoir la valeur `ACC_PUBLIC|ACC_ABSTRACT=0x0401` dans l'attribut `access_flags`.

Nom du drapeau	Valeur
ACC_PUBLIC	0x0001
ACC_FINAL	0x0010
ACC_SUPER	0x0020
ACC_INTERFACE	0x0200
ACC_ABSTRACT	0x0400

**Tableau 3**

Les deux champs suivants `this_class` et `super_class` indiquent le nom de la classe et de sa super-classe.

Nous trouvons ensuite la taille et un tableau de toutes les interfaces implémentées par la classe. Le tableau possède les références sur les noms des interfaces présentes dans le pool de constante.

### 3. LES CHAMPS

Ensuite, nous avons la description des différents champs de la classe. Chaque champ est décrit par une structure `field_info`.

```
field_info
{
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

`access_flags` décrit les différents drapeaux qualifiant le champ (voir Tableau 4).

Nom du drapeau	Valeur
ACC_PUBLIC	0x0001
ACC_PRIVATE	0x0002
ACC_PROTECTED	0x0004
ACC_STATIC	0x0008
ACC_FINAL	0x0010
ACC_VOLATILE	0x0040
ACC_TRANSIENT	0x0080

**Tableau 4**

Le champ `name_index` référence une constante du pool pour donner un nom à l'attribut de la classe. `descriptor_index` pointe vers une `CONSTANT_Utf8_info` pour décrire le type de l'attribut.

Nous trouvons ensuite la taille et un tableau d'attributs. Nous avons une structure similaire dans la description de la classe. Ce tableau permet d'enrichir les classes, les attributs ou les méthodes avec des informations diverses. C'est ce mécanisme qui permet d'effectuer les traitements décrits au début de l'article. Nous retrouvons généralement dans ces attributs les informations de déverminages.

### 4. LES METHODES

La structure `ClassFile` propose ensuite la liste des méthodes.

```
method_info
{
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Nous retrouvons une structure similaire à `field_info`.

`access_flags` possède les drapeaux de la méthode (voir Tableau 5).

Nom du drapeau	Valeur
ACC_PUBLIC	0x0001
ACC_PRIVATE	0x0002
ACC_PROTECTED	0x0004

ACC_STATIC	0x0008
ACC_FINAL	0x0010
ACC_SYNCHRONIZED	0x0020
ACC_NATIVE	0x0100
ACC_ABSTRACT	0x0400
ACC_STRICT	0x0800

**Tableau 5**

Le `descriptor_index` indique le type de tous ces paramètres et du code retour. Par exemple, la méthode

```
Object mymethod(int i, double d, Thread t)
```

est décrite ainsi

```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```

Nous retrouvons ensuite le tableau des attributs libres.

Mais où est le p-code d'une méthode ? Cette structure ne possède pas de tableau de byte avec le p-code ! En fait, pour le fichier class, le p-code est un attribut au même titre que les informations de déverminage. Ce n'est pas étonnant, car parfois, les méthodes n'ont pas d'implémentation. C'est le cas des méthodes abstraites.

Pour retrouver le p-code des méthodes, il faut alors consulter le tableau des attributs. Ils suivent la structure suivante :

```
attribute_info
{
  u2 attribute_name_index;
  u4 attribute_length;
  u1 info[attribute_length];
}
```

Le nom de l'attribut est présent dans le pool de constante. Certains noms sont définis par les spécifications. Vous pouvez choisir le nom que vous voulez pour un attribut. Pour éviter les conflits de noms, il est recommandé d'utiliser un nom du type : `org.monsite.MonAttribut`.

## 5. LE CODE

Le nom `Code` pour un attribut indique que l'on va trouver le p-code de la méthode dans le tableau `info`. Nous étudierons le format du p-code dans un autre article. Pour le moment, nous trouvons à cet emplacement la structure suivante :

```
Code_attribute
{
  u2 max_stack;
  u2 max_locals;
  u4 code_length;
  u1 code[code_length]; // C'est ici !
  u2 exception_table_length;
  {
    u2 start_pc;
    u2 end_pc;
    u2 handler_pc;
    u2 catch_type;
  } exception_table[exception_table_length];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Nous trouvons dans cette structure des informations indiquant la taille maximum utilisée pour la pile, la taille maximum pour les variables locales à la méthode, le p-code de la méthode et un tableau s'occupant de décrire les exceptions.

Ce tableau est construit lors de l'utilisation des instructions `try` et `catch`. Ces instructions ne sont pas traduites en p-code. Elles alimentent le tableau des exceptions. Pour chaque exception, la structure indique le premier p-code concerné (l'instruction `try`), la dernière instruction concernée (l'accolade fermante suivant le `try`), le p-code s'occupant de traiter l'exception (l'instruction `catch`) et le type de l'exception à capturer.

Nous retrouvons ensuite notre structure d'attribut permettant d'enrichir la description du p-code avec les informations de déverminages. Nous avons alors des attributs libres dans un attribut `Code`.

## 6. LES EXCEPTIONS

Une méthode peut également posséder un attribut `Exceptions`. Il décrit les différentes exceptions générées par la méthode. Il s'agit des exceptions indiquées par l'instruction `throws` après le nom d'une méthode. Ces informations sont utilisées par le compilateur java pour vérifier que toutes les exceptions ont été capturées ou propagées. La machine virtuelle ignore cela. Il est parfaitement possible de produire un fichier class avec une méthode émettant une exception non décrite ici.

Un utilitaire peut supprimer les attributs `Exceptions` afin de réduire la taille d'une archive. Cela peut être utile dans une application en déploiement, mais en phase de développement, cela est équivalent à supprimer toutes les instructions `throws` du code. Le compilateur java ne peut plus vous aider à capturer les exceptions.

Nous avons exploré le format des fichiers class, générés par les compilateurs java. Nous y avons trouvé beaucoup d'informations sur la structure de la classe. Vous pouvez analyser un fichier class en utilisant l'utilitaire `javap` livré avec le JDK. Par exemple, `javap java.lang.String` affiche toutes les méthodes de la classe.

Contrairement aux langages classiques comme le C, le C++, Pascal ou autres Cobol, java effectue l'édition de liens lors de l'installation de la classe dans la JVM. Il est alors nécessaire d'avoir toutes les informations importantes dans les fichiers class. Il est alors généralement possible de dé-compiler une classe java.

Les informations présentes dans chaque classe sont redondantes. Il est possible d'améliorer la taille des archives java en mutualisant les différents pools de constantes et en supprimer les attributs `Exception`. Un utilitaire regroupe tous les pools de toutes les classes d'une archive. Il modifie alors chaque fichier class pour supprimer le pool. Un classloader se charge de charger le pool global de l'archive, et de modifier dynamiquement les fichiers class pour leurs donner un pool local avant l'installation de la classe en mémoire. Ainsi, les archives peuvent être fortement réduites. En effet, le mécanisme de compression utilisé dans les fichiers `jar` compresse chaque fichier, mais pas l'ensemble des fichiers. Il n'est pas capable de détecter les similitudes entre les classes. Optimiser les archives avec cette technique permet d'empêcher la décompilation. En effet, les fichiers class ne respectent plus les spécifications originales. Il est impératif d'utiliser un classloader spécifique pour utiliser les classes. Si l'un d'entre vous rédige cet utilitaire, qu'il me prévienne !

Maintenant que nous savons où le placer, nous regarderons dans un prochain article comment rédiger du p-code java.

Philippe PRADOS  
[philippe@prados.net](mailto:philippe@prados.net)  
[www.philippe.prados.net](http://www.philippe.prados.net)