Java, bytecode et JVM (et réciproquement)

Nous n'aurons en pratique quasi jamais besoin de ce qui va suivre, l'IDE le faisant pour nous. Cependant, savoir comment tout cela fonctionne permet épisodiquement de se sortir de mauvais pas.

Prérequis

- Vous avez suivi la partie utiliser Intellij et avez un programme "hello world!" fonctionnel.
- vous connaissez la différence entre JDK, JRE et JVM.
- pour les exemples on supposera que l'on a accès à un terminal (il en existe sous linux, mac et Windows)

Principe

Cette vidéo montre comment fonctionne un ordinateur et comment il peut exécuter du code. De là les différences entre les approches de Java et d'autres langages comme le C.

En gros, il existe 3 grand paradigmes pour l'exécution de programme :

- **interprétation** : un fichier écrit par un humain est "lu" par un programme qui traduit une à une chaque ligne en langage machine (python fonctionne "en gros" comme ça)
- **compilation** : un fichier écrit par un humain est tout d'abord traduit complètement en langage machine par un compilateur et c'est la traduction machine que l'on exécute (les langage C/C++ par exemple)
- **machine virtuelle** : un fichier écrit par un humain est tout d'abord traduit en un langage intermédiaire (le bytecode en Java) et c'est ce fichier traduit qui est exécuté par une machine virtuelle qui le transforme par bloc (ce qu'on appelle le JIT, Just In Time compilation) en langage machine (le langage Java par exemple)

L'approche par machine virtuelle est une approche intermédiaire entre l'interprétation pure et dure et la compilation. Ceci permet de mitiger les principaux défauts des approches par compilation et interprétation :

- **compilation** : il faut compiler chaque programme pour toutes les architectures et systèmes d'exploitations sur le quel on veut faire tourner son programme (mac, linux, windows, téléphone, etc...). Ce qui n'est pas le cas avec l'approche interprétée/machine virtuelle puisque seul l'interpréteur ou la machine virtuelle doit être compilée sur chaque architecture machine.
- **interprétation** : comme chaque ligne est lue puis compilée puis exécutée, c'est comparativement lent par rapport à un programme compilé. Le byte code de java est un langage intermédiaire plus simple à convertir en langage machine.

Interprété

Programme python `hello.py` :

```
print("Hello World!")
```

On l'exécute via l'interpréteur qui transforme chaque ligne une à une en langage machine :

```
~ $ python3 hello.py
Hello World!
~ $
```

Compilé

Programme C "hello.c" :

```
#include <stdio.h>

int main()
{
  printf("hello world!\n") ;
}
```

On le compile en exécutable (ici mac) avec un compilateur, ici gcc:

```
~ $ gcc hello.c
~ $
```

Le compilateur a créé un fichier qui s'appelle par défaut `a.out` et qui est un fichier exécutable (la commande file permet de connaitre le type d'une fichier) :

```
~ $ file a.out
a.out: Mach-O 64-bit executable x86_64
~ $
```

Ce fichier peut donc être exécuté sur notre machine :

```
~ $ ./a.out
hello world!
~ $
```

Bytecode Java

Fichier `Hello.java`

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

On le transforme en bytecode grâce à la commande `javac`. Le fichier de sortie se nomme `Hello.class` :

```
~ $ javac Hello.java
~ $
```

C'est un fichier java *compilé* (du bytecode, pas du langage machine) :

```
~ $ file Hello.class
Hello.class: compiled Java class data, version 52.0
~ $
```

On peut exécuter la classe Hello via la JVM (par défaut, la commande `java` va chercher un fichier `Hello.class` pour exécuter la classe Hello) :

```
~ $ java Hello
Hello World!
~ $
```

Distribuer son code Java

Lorsque l'on a beaucoup de fichiers, resources, etc, et ça va arriver très vite, on a coutume de :

- placer son code dans des répertoires différents via des packages
- distribuer son code via une archive jar

Pour la démonstration on aura un unique fichier `Hello.java` dans le package `com.mco` (il est donc dans le répertoire `./com/mco/` si le répertoire courant (`./`) est la racine du projet) :

```java
package com.mco;

public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

On compile le java :

```
~ $ javac com/mco/java.java
~ $
```

On execute le bytecode depus le répertoire racine du projet :

```
~ $ java com/mco/Hello
Hello World!
~ $
```

En supposant que l'on ait beaucoup de fichiers classes on les regroupe en un unique fichier jar.

Souvent l'IDE le fera pour vous mais comme on est là pour apprendre on le fait à la main en suivant l'aide de la documentation :

```
jar cfe hello.jar com.mco.Hello com/mco/Hello.class
```

Un jar n'est rien d'autre qu'un gros fichier zip avec nos classes, nos ressources si nécessaires (image, video, etc) et un répertoire `META-INF` contenant ses paramètres (le fichier `MANIFEST.MF` par exemple qui contient la classe à exécuter par défaut) :

```
~ $ jar tf hello.jar
META-INF/
META-INF/MANIFEST.MF
com/mco/Hello.class
~ $
```

On peut ensuite l'exécuter :

```
~ $ java -jar hello.jar
Hello World!
~ $
```

Ou, si l'on veut exécuter une classe particulière du jar :

```
~ $ java -cp hello.jar com.mco.Hello
Hello World!
~ $
```

Voir du bytecode en vrai

On se basera sur l'excellent article
http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/

Pour l'exercice, on utilisera notre Hello.java sans package que l'on aura compilé en Hello.class.

Le bytecode se voit en utilisant la commande `javap`:

```
~ $ javap -c Hello.class
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method
java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
```

```
    Code:
       0: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Hello World!
       5: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
~ $
```

Nous avons 7 instructions et le fichier pèse 416 octet. C'est beaucoup plus que le fichier initial qui pèse lui environ 110 octets. Regardons ces 416 octets.

Pour voir un fichier exécutable, on ne peut pas l'ouvrir avec un éditeur de texte qui va essayer de le lire comme du texte (souvent en unicode). Nous voulons voir les 0 et les 1 qui le forment.

Pour cela on utilisera un éditeur permettant de voir un fichier sous forme de suite de nombres. On utilise les nombres Hexadécimaux car pratiques pour séparer les octets (8 bits, de 0 à 255 en base 10 et de 0 à FF en base 16) qui sont l'unité de base pour un processeur.

Il en existe plein (j'utilise hexdump sous linux et Hex Fiend sous mac).

Le fichier `Hello.class` est le suivant :

```
~ $ hexdump Hello.class -C
00000000  ca fe ba be 00 00 00 33  00 1d 0a 00 06 00 0f 09
|.......3........|
00000010  00 10 00 11 08 00 12 0a  00 13 00 14 07 00 15 07
|................|
00000020  00 16 01 00 06 3c 69 6e  69 74 3e 01 00 03 28 29
|.....<init>...()|
00000030  56 01 00 04 43 6f 64 65  01 00 0f 4c 69 6e 65 4e
|V...Code...LineN|
00000040  75 6d 62 65 72 54 61 62  6c 65 01 00 04 6d 61 69
|umberTable...mai|
00000050  6e 01 00 16 28 5b 4c 6a  61 76 61 2f 6c 61 6e 67
|n...([Ljava/lang|
00000060  2f 53 74 72 69 6e 67 3b  29 56 01 00 0a 53 6f 75
|/String;)V...Sou|
00000070  72 63 65 46 69 6c 65 01  00 0a 48 65 6c 6c 6f 2e
|rceFile...Hello.|
00000080  6a 61 76 61 0c 00 07 00  08 07 00 17 0c 00 18 00
|java............|
00000090  19 01 00 0c 48 65 6c 6c  6f 20 57 6f 72 6c 64 21  |....Hello
World!|
000000a0  07 00 1a 0c 00 1b 00 1c  01 00 05 48 65 6c 6c 6f
|...........Hello|
```

```
000000b0  01 00 10 6a 61 76 61 2f  6c 61 6e 67 2f 4f 62 6a
|...java/lang/Obj|
000000c0  65 63 74 01 00 10 6a 61  76 61 2f 6c 61 6e 67 2f
|ect...java/lang/|
000000d0  53 79 73 74 65 6d 01 00  03 6f 75 74 01 00 15 4c
|System...out...L|
000000e0  6a 61 76 61 2f 69 6f 2f  50 72 69 6e 74 53 74 72
|java/io/PrintStr|
000000f0  65 61 6d 3b 01 00 13 6a  61 76 61 2f 69 6f 2f 50
|eam;...java/io/P|
00000100  72 69 6e 74 53 74 72 65  61 6d 01 00 07 70 72 69
|rintStream...pri|
00000110  6e 74 6c 6e 01 00 15 28  4c 6a 61 76 61 2f 6c 61
|ntln...(Ljava/la|
00000120  6e 67 2f 53 74 72 69 6e  67 3b 29 56 00 21 00 05
|ng/String;)V.!..|
00000130  00 06 00 00 00 00 00 02  00 01 00 07 00 08 00 01
|................|
00000140  00 09 00 00 00 1d 00 01  00 01 00 00 00 05 2a b7
|..............*.|
00000150  00 01 b1 00 00 00 01 00  0a 00 00 00 06 00 01 00
|................|
00000160  00 00 01 00 09 00 0b 00  0c 00 01 00 09 00 00 00
|................|
00000170  25 00 02 00 01 00 00 00  09 b2 00 02 12 03 b6 00
|%...............|
00000180  04 b1 00 00 00 01 00 0a  00 00 00 0a 00 02 00 00
|................|
00000190  00 03 00 08 00 04 00 01  00 0d 00 00 00 02 00 0e
|................|
000001a0
~ $ ls -la
```

La différence de taille s'explique par le format d'un fichier class. Remarquez la petite blague du magic number des fichiers classes.

Recompiler en Java

Pour recompiler en java un fichier class il faut passer du bytecode au java. Cela ne peut pas se faire sans perte. Pour cela il nous faudra un désassembleur. Nous utiliserons cfr qui a le bon goût d'être écrit en java :

```
~ $ java -jar cfr_0_110.jar Hello.class
/*
 * Decompiled with CFR 0_110.
 */
import java.io.PrintStream;
```

```
public class Hello {
    public static void main(String[] arrstring) {
        System.out.println("Hello World!");
    }
}
~ $
```