

Wykład 12

Plik .class

ASM

Plik .class

Wiadomo, że jak się skompiluje program w javie kompilatorem `javac` to z kodu źródłowego o rozszerzeniu `java` powstaje nam zwykle jeden plik z rozszerzeniem `.class`. (Czasem może być więcej plików, jeżeli ta klasa zawiera jakieś klasy zagnieżdżone).

Struktura Pliku `.class` w javie:

```
ClassFile {  
    u4 magic;  
    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];  
}
```

u4 — czterobajtowa sekwencja

u2 — dwubajtowa sekwencja

Teraz omówimy co znaczą kolejne bajty w tym pliku:

Jak skonstruowany jest Plik .class ?

Przykład dla kodu:

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

Na początku każdego pliku .class bajtkodu javy, 4 pierwsze bajty to:

CA FE BA BE — tzw. "*magic*", identyfikator formatu pliku

Kolejne 4 określają wersję wirtualnej maszyny javy JVM która będzie potrzebna do uruchomienia programu:

00 00 00 33 — numer wersji JVM: (51.0)

Bajty od prawej są bardziej znaczące. Zapis w systemie szesnastkowym.

$$3 * 16 + 3 = 51$$

Czyli zgodna wersja to JVM: (51.0) — oznacza to zgodność z Java 1.7

Ogólna Zasada

JVM 1.k (k>1) obsługuje klasy od 45.0 do 44+k.0 włącznie

00 22 — ilość deklarowanych elementów (Constant Pool).

Constant Pool — to specjalna sekcja w pliku .class Javy, która zawiera stałe potrzebne do wykonania kodu konkretnej klasy.

$$2 * 16 + 2 = 34$$

Stałych jest zawsze o jeden mniej niż constant_pool_count!

Po tej deklaracji występują następująco kolejne, 33 elementy:

1. **07 00 02** – **klasa** o nazwie w elemencie nr 2
2. **01 00 04** – **string UTF-8** o długości 4 — następnie jest ten string
4D 61 69 6E — *Main*
3. **07 00 04** – **klasa** o nazwie w elemencie nr 4
4. **01 00 10** – **string UTF-8** o długości 16
6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74 - *java/lang/Object*
5. **01 00 06** *<init>* — Już bez pisania tych 16 bajtów
6. **01 00 03** *()V*
7. **01 00 04** *Code*
8. **0A 00 03 00 09** – **klasa 3** posiada **metodę 9**
9. **0C 00 05 00 06** – **metoda** o nazwie 5 i sygnaturze 6 (brak argumentów (), zwraca void V)
10. **01 00 0F** *LineNumberTable*
11. **01 00 12** *LocalVariableTable*
12. **01 00 04** *this*
13. **01 00 06** *LMain;*
14. **01 00 04** *main*
15. **01 00 16** *(Ljava/lang/String;)V*
16. **09 00 11 00 13** – **klasa 17** posiada atrybut 19
17. **07 00 12** – **klasa** o nazwie 18
18. **01 00 10** *java/lang/System*
19. **0C 00 14 00 15** – **atrybut** o nazwie 20 i typie 21
20. **01 00 03** *out*
21. **01 00 15** *Ljava/io/PrintStream;*
22. **08 00 17** – 23 element to **stała tekstowa**
23. **01 00 0C** *Hello world!*
24. **0A 00 19 00 1B** – **klasa 25** posiada **metodę 27**
25. **07 00 1A** – **klasa** o nazwie 26
26. **01 00 13** *java/io/PrintStream*
27. **0C 00 1C 00 1D** – **metoda 28** o sygnaturze 29
28. **01 00 07** *println*
29. **01 00 15** *(Ljava/lang/String;)V*
30. **01 00 04** *args*
31. **01 00 13** *[Ljava/lang/String;*
32. **01 00 0A** *SourceFile*
33. **01 00 09** *Main.java*

Po *Constant Pool*-u mamy poniższe bajty:

00 21 - modyfikatory dostępu dla klasy: ACC_PUBLIC (00 01) | ACC_SUPER (00 20) – ze względu na kompatybilność ze starszymi JVM.

Pierwszy atrybut oznacza, że klasa jest publiczna a ten drugi ze względu na kompatybilność ze starszymi JVM

00 01 – numer elementu określający klasę definiowaną w tym pliku

Dla przypomnienia:

1. **07 00 02** – klasa o nazwie w elemencie nr 2
2. **01 00 04** – string UTF-8 o długości 4
4D 61 69 6E — *Main*

00 03 – numer nadklasy

Dla przypomnienia:

3. **07 00 04** – klasa o nazwie w elemencie nr 4
4. **01 00 10** – string UTF-8 o długości 16
6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74 - *java/lang/Object*

00 00 – liczba interfejsów

Jeśli byłyby jakieś interfejsy teraz następowałaby tablica wszystkich interfejsów

00 00 – liczba pól (atrybutów)

Analogicznie z polami, byłaby tablica

00 02 – liczba metod

Jak wygląda struktura `method_info` ?

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

00 01 – modyfikatory dostępu: ACC_PUBLIC (00 01)
00 05 – indeks elementu z nazwą metody — *<init>*
00 06 – indeks elementu z sygnaturą metody — *()V*
00 01 – liczba dodatkowych atrybutów

Teraz zaczyna się jednoelementowa tablica:

00 07 – indeks elementu z nazwą atrybutu (*Code*)
00 00 00 2F – długość atrybutu
00 01 00 01 – rozmiar stosu I rozmiar tablicy zmiennych lokalnych

Metoda *<init>*

- Pisze ją za nas kompilator

Metoda wywołana na obiekcie `object` inicjalizuje (wywołuje konstruktor) nadklasę.

00 00 00 05 – długość kodu, wykonywanych instrukcji — będzie 5 bajtów instrukcji

Poniższe instrukcje mogą być różne, ale takie są dla metody *init*: — działa to trochę jak w *assemblerze*

2A — *ALOAD_0* — zmienna lokalna o adresie 0 jest wstawiana na stos
B7 00 08 — *INVOKESPECIAL java/lang/Object.<init>()V*; — wywołuje metodę `void Object.<init>()`;
B1 — *RETURN* — zwraca typ `void`

Te instrukcje zawsze znajdują się w bajtkodzie!

00 00 – długość tablicy wyjątków
00 02 – liczba dodatkowych atrybutów
00 0A – tablica numerów linii
00 00 00 06 – długość atrybutu
00 01 – długość tabeli
00 00 – indeks instrukcji w tabeli (*Code*)
00 01 – nr lini w pliku źródłowym

00 0B – tablica zmiennych lokalnych
00 00 00 0C – długość atrybutu
00 01 – długość tabeli
00 00 – początek zmiennej
00 05 – długość zmiennej
00 0C – indeks nazwy zmiennej

00 0D – indeks nazwy typu zmiennej
00 00 - indeks zmiennej w tabeli zmiennych lokalnych

Metoda *main*

00 09 – modyfikatory dostępu: ACC_PUBLIC (00 01) | ACC_STATIC (00 08)
00 0E – indeks elementu z nazwą metody (*main*)
00 0F – sygnatura (*[Ljava/lang/String;)V*)

(*[Ljava/lang/String;)V* — Przyjmowana tablica Stringów a typ zwracany to void, zgadza się

```
public static void main(String[] args)
```

00 01 – liczba dodatkowych atrybutów
00 07 – indeks elementu *Code*
00 00 00 37 – długość elementu
00 02 – rozmiar stosu
00 01 – rozmiar tablicy zmiennych lokalnych
00 00 00 09 – długość kodu

B2 00 10 — *GETSTATIC java/lang/System.out : Ljava/io/PrintStream;* — inicjalizacja klasy/obiektu *System.out* i odłożenie go na stos
12 16 — *LDC Hello world!* — załadowanie na stos stałej tekstowej
B6 00 18 — *INVOKEVIRTUAL java/io/PrintStream.println(Ljava/lang/String;)V* — wywołuje metodę *System.out.println(String)*
B1 — *RETURN* — zwraca typ void

00 00 - długość tablicy wyjątków
00 02 – liczba dodatkowych atrybutów
00 0A – tablica numerów linii
00 00 00 0A 00 02 00 00 00 04 00 08 00 05

W tej tablicy chodzi o to któremu elementowi kodu odpowiada który numer linii

00 0B – tablica zmiennych lokalnych
00 00 00 0C 00 01 00 00 00 09 00 1E 00 1F 00 0
00 01 – liczba atrybutów
00 20 – *SourceFile*
00 00 00 02 – długość atrybutu
00 21 - *Main.java*

Warto zapamiętać:

1. **Kompilator dopisuje nam tą metodę `init`**

Czyli my możemy mieć w klasie jedną metodę. Natomiast w wyniku kompilowania, w pliku `class` będziemy mieć 2 metody.

2. **Faktyczne instrukcje do wykonania to bardzo mały fragment pliku `.class`**

Większość z tych informacji to są klocki, które są potrzebne do wykonania instrukcji kodu naszego programu.

Assembler javowy

Tak samo jak w przypadku kodu maszynowego zamiast używać po prostu liczb, wygodniej do odczytania tego używa się assemblera, czyli takiego zapisu słownego. Dla javy też można. **Istnieją rozszerzenia do środowisk programistycznych, które pozwalają te assembler wyświetlić na bazie bajt kodu.**

jest to ten sam kod co napisany wyżej bajt po bajcie.

```
// Compiled from Main.java (version 1.7 : 51.0, super bit)
public class Main {

    // Method descriptor #6 ()V
    // Stack: 1, Locals: 1
    public Main();
        0  aload_0 [this]
        1  invokespecial java.lang.Object() [8]
        4  return
        Line numbers:
            [pc: 0, line: 1]
        Local variable table:
            [pc: 0, pc: 5] local: this index: 0 type: Main
    // Method descriptor #15 ([Ljava/lang/String;)V
    // Stack: 2, Locals: 1
    public static void main(java.lang.String[] args);
        0  getstatic java.lang.System.out : java.io.PrintStream [16]
        3  ldc <String "Hello world!"> [22]
        5  invokevirtual java.io.PrintStream.println(java.lang.String) : void
[24]
        8  return
        Line numbers:
            [pc: 0, line: 4]
            [pc: 8, line: 5]
        Local variable table:
            [pc: 0, pc: 9] local: args index: 0 type: java.lang.String[]
    }
}
```

Jest to ten sam kod co napisany wyżej bajt po bajcie.

ASM

ASM to biblioteka **ułatwiająca manipulację bytcodeem Javy**. Daje możliwość niskopoziomowej analizy, tworzenia oraz modyfikacji istniejących klas.

Linijka kompilująca pojniższy kod:

```
java -cp asm-4.1.jar:asm-util-4.1.jar:asm-commons-4.1.jar
org.objectweb.asm.util.ASMifier Main.class
```

Przykładowy kod generujący klasę Main wygląda następująco:


```

import org.objectweb.asm.*;

public class MainDump implements Opcodes {
    public static byte[] dump() throws Exception {
        ClassWriter cw = new ClassWriter(0);
        MethodVisitor mv;
        cw.visit(V1_7, ACC_PUBLIC + ACC_SUPER, "Main", null,
                "java/lang/Object", null);
        mv = cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
        mv.visitCode();
        mv.visitVarInsn(ALOAD, 0);
        mv.visitMethodInsn(INVOKESPECIAL, "java/lang/Object", "<init>", "()V");

        mv.visitInsn(RETURN);
        mv.visitMaxs(1, 1);
        mv.visitEnd();

        mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main",
                "([Ljava/lang/String;)V", null, null);
        mv.visitCode();
        mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
                "Ljava/io/PrintStream;");
        mv.visitLdcInsn("Hello world!");
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
                "println", "(Ljava/lang/String;)V");
        mv.visitInsn(RETURN);
        mv.visitMaxs(2, 1);
        mv.visitEnd();
        cw.visitEnd();

        return cw.toByteArray();
    }
}

```

`public static byte[] dump()` — zwraca bytecode klasy *MainDump*

`ClassWriter` — metoda służąca do pisania klas

`MethodVisitor` — obiekt za pomocą którego piszemy metody

`v1_7` — określa wersję JVM

Przykład użycia:

```
public class HelloWorldASM extends ClassLoader{
    public static void main(final String args[]) throws Exception {
        HelloWorldASM loader = new HelloWorldASM();
        byte[] code = MainDump.dump();
        Class c1 = loader.defineClass("Main", code, 0, code.length);
        c1.getMethods()[0].invoke(null, new Object[] { null });
    }
}
```

Uwagi:

1. Można modyfikować bajtkod, dopóki nie wgramy go do pamięci

jak już zainicjujemy tą klasę, (wykonamy instrukcję `defineClass()` w *ClassLoaderze* ✨ "No to już jest koniec" ✨).

2. W bajtkodzie mogą znajdować się adnotacje (mogą być przeczytane przez np. JAXB)