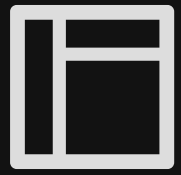
Things I didn’t want to know about JVM bytecode but learned anyway

Jakub Kozłowski | The Art Of Scala 2nd edition | 16.11.2022

Disclaimer

- I don’t work directly on JVM bytecode during the workweek
- This was purely for fun and learning
- No JVMs were hurt while making this talk

Agenda



Classfile overview

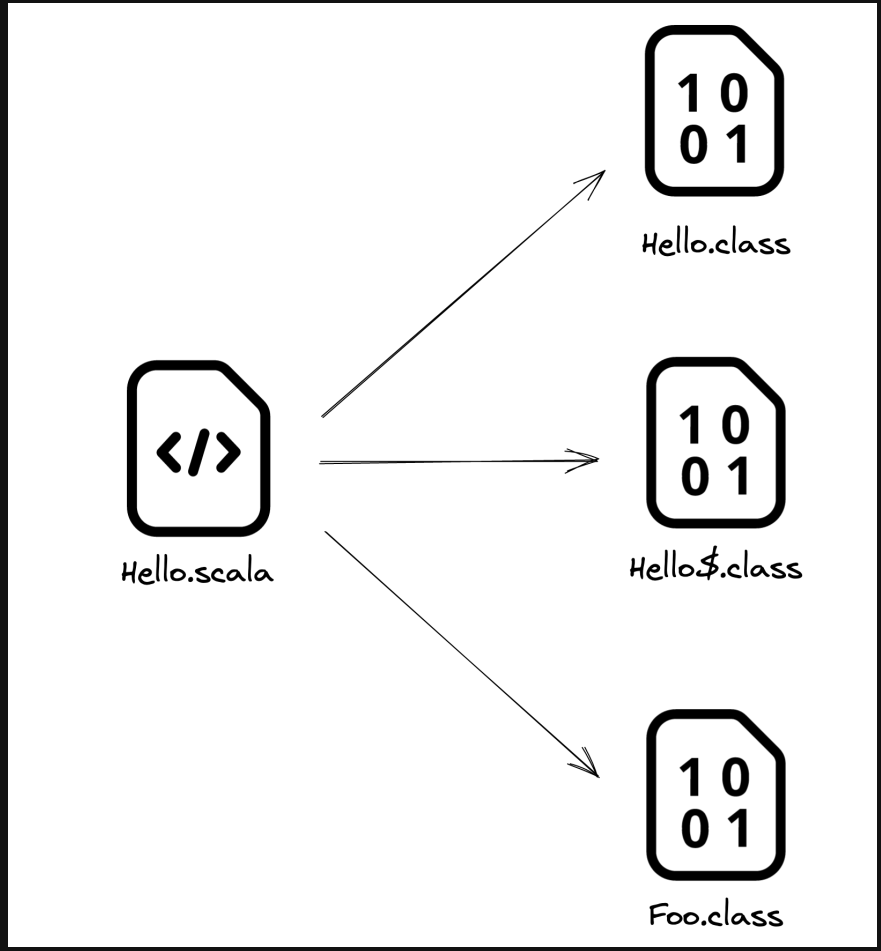
Binary files 101

Classfile encoding

Classfile overview

What’s a classfile

- binary file
- **output** of a compiler
- a single source can produce 0…n classes
- **input** format for the JVM
- represents **one** class/interface/~~module~~



Classfile structure

```
case class ClassFile(
  minorVersion: Int,
  majorVersion: Int,
  constants: ConstantPool,
  accessFlags: Set[ClassAccessFlag],
  thisClass: String,
  superClass: String,
  interfaces: List[InterfaceName],
  fields: List[FieldInfo],
  methods: List[MethodInfo],
  attributes: Map[AttributeName, AttributeInfo],
)
```

everything in this talk is based on the Java SE 17 spec.

Binary files 101

Binary files 101

- binary (non-text) file: sequence of bits - e.g. `0b1100101010110101100`
- bit: single digit in a base-2 numeric system - (`0` or `1`)
- byte: a group of 8 bits (*usually*) - e.g. `0b10011111`, or `0x9f`

Classfile encoding

Classfile binary format

- `u1`: 1 unsigned byte
- `u2`: 2 unsigned bytes
- `u4`: …

Magic number

`0b11001010111111101011101010111110`

!?

`0xCAFEBABE` 💀

```
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];
}
```

JVM version

Minor, major version

- minor goes first (for some reason)
- major - minimum JVM version required to run this
- minor - since JDK 12, either:
  - `0x0000` (0, normal classfile) or
  - `0xffff` (65 535) - experimental features required

```
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];
}
```

Major versions

- 52 (Java 8)
- 55 (Java 11),
- 61 (Java 17)…
- …enough space (`u2`) to let us have Java 65491 :)

If you get it wrong:

```
Error: LinkageError occurred while loading main class Foo
  java.lang.UnsupportedClassVersionError: Foo has been compiled by a more recent version
  of the Java Runtime (class file version 61.0), this version of the Java Runtime
  only recognizes class file versions up to 55.0
```

Major version compatibility table

Constant pool

- an ordered list of reusable constants
- contains all literals, class/method/field/type names etc.
- prefixed with 2 bytes for the pool size
- each constant is prefixed with a discriminator byte (`tag`)
- indices and sizes are **off by one (start at 1)**

```
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];
}
```

Integer_info constant

- content: 4-byte (32-bit) **signed** integer
- example: `48` -> `0x00000030`

```
CONSTANT_Integer_info {
    u1 tag;
    u4 bytes;
}
```

Utf8_info constant

- content:
  - `u2 length`: the amount of data bytes
  - `u1` x `length`: "modified UTF-8"-encoded data

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

Modified UTF-8 encoding

- pretty complex, but space efficient: ASCII characters only use 1 byte each
- can be encoded/decoded with JDK's `DataOutputStream`/`DataInputStream`
- also used in Java Serialiation

Examples:

input	length	data
`"hello"`	`0x0005`	`0x68656c6c6f`
`"łódź"`	`0x0007`	`0xc582c3b364c5ba`

Class_info constant

- content: `u2 name_index`: index to the constant pool
  - **must** target a `UTF8_Info` constant (class name)
- example: index 2 (third item in pool) -> `0x03`

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

Real example

`javap -v Hello`

```
public class Hello
  minor version: 0
  major version: 48
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #2                          // Hello
  super_class: #4                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 1, attributes: 1
Constant pool:
   #1 = Utf8               Hello
   #2 = Class              #1             // Hello
   #3 = Utf8               java/lang/Object
   #4 = Class              #3             // java/lang/Object
...
```

(last one I promise) Long_info constant

- content:
  - `u4`: high bytes
  - `u4`: low bytes

but wait!

```
CONSTANT_Long_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

Long_info / Double_info

> All 8-byte constants take up two entries in the constant_pool table of the class file

e.g. with this constant pool

index	1	2	3	4	5	6
contents	utf8	class	int	**long**	❌	utf8

the pool's size is still 6 (encoded as `` `7` ``)!

Access flags

- `u2` (16-bit) bitmask
- info about class modifiers under non-overlapping bits:

```
Public -> 0x0001,
Final -> 0x0010,
Super -> 0x0020,
Interface -> 0x0200,
Abstract -> 0x0400,
Synthetic -> 0x1000,
Annotation -> 0x2000,
Enum -> 0x4000,
```

example: `Public` + `Enum` == `0x4001`

Not a lot of space left!

```
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];
}
```

Skip ahead

- we've talked about class names already
- one piece of trivia: if `super_class` contains `0` (non-existent index), it means you're looking at `java/lang/Object`
- interfaces/fields out of scope

```
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];
}
```

Methods

- prefixed with `u2` for the amount of methods

```
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];
}
```

Methods inside

- `u2`: access flags (similar to those of classes)
- `u2`: name index in constant pool (`Utf8_info`)
- `u2`: **descriptor** index in constant pool (`Utf8_info`)
- attributes, ignoring (classes have identical layout)

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

Descriptors

Some examples:

type	descriptor
`Int`	`I`
`Double`	`D`
`String`	`Ljava/lang/String;`
`Unit`	`V`
`def f(a: Int, d: Double): Unit`	`(ID)V`
`(Int, Double) => Unit`	`Lscala/Function2;`

Attributes

- A "second class" construct
- key-value mapping (think `` `Map[String, Attribute]` ``)
- prefixed with `` `u2` `` for the amount of attributes
- each attribute has variable length

```
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];
}
```

Attributes inside

- `u2`: **name** constant pool index (`Utf8_Info` constant)
- `u4`: data length
- `u1` x `length`: data

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

Example attributes - Code

- method attribute containing the actual "bytecode" of the method
- it has its own attributes!

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    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];
}
```

Attribute trivia

- The spec defines 30 standard attributes
  - 7 **must** be supported by JVMs (e.g. `Code`)
  - 10 **must** be supported by JDK libraries (e.g. `LineNumberTable`)
  - 13 are non-critical metadata (e.g. `Deprecated`)

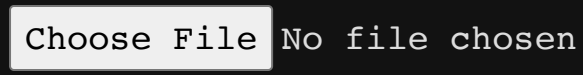
- Unrecognized attributes **must** be ignored

- JVMs **must** ignore attributes that don't exist in a given classfile format
  - e.g. the `Record` attribute will be ignored in old JVMs even if the file targets an old major version

Instruction trivia

- in JRE 17, there are ~202 opcodes
- most are constant-size but some are not
  - e.g. `tableswitch` (`0xaa`) has optional 0-3 byte padding after the opcode to ensure alignment

What’s that? A file input? 🤔



Built with scodec

```
val classFile: Codec[ClassFile] =
  (
    ("magic number " | constant(hex"CAFEBABE")) ::
      ("minor version" | u2) ::
      ("major version" | u2) ::
      constantPool ::
      classAccessFlags ::
      ("this class" | constantPoolIndex[Constant.ClassInfo]) ::
      ("super class" | constantPoolIndex[Constant.ClassInfo]) ::
      listOfN(
        "interface count" | u2,
        "interface index" | constantPoolIndex[Constant.ClassInfo],
      ) ::
      listOfN(
        "field count" | u2,
        fieldInfo,
      ) ::
      listOfN(
        "method count" | u2,
        methodInfo,
      ) ::
      attributes
  ).dropUnits.as[ClassFile]
```

Resources

- Java SE 17 spec (classfile format)
- Inside the JVM book
- Mateusz's JVM book :)

Tools

- `javap`, `javap -v` (Metals can show these too!)
- `xxd` / `hexdump` / a dozen other binary editors/viewers
- `java.io` `DataInput`/`DataOutput`

Thank you

- Watch my YouTube! yt.kubukoz.com
- Contact + slides + YT: linktr.ee/kubukoz