Class files and Bytecode

Bytecode is an intermediate representation of a Java program after the source code compilation. It is stored in .class files. When someone runs a program, JVM executes bytecode, and the program works. Bytecode is also a kind of a language that programmers can directly read, understand, and even modify, but it is more complicated than using Java.

Compiling the source file:

First, let's consider the source code of a small program inside the Main.java file.

```
public class Main {

    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        System.out.println(a + b);
    }
}
```

As you can see, this program just prints 3.

Let's compile it using javac:

```
javac Main.java
```

This command will create the Main.class file in the same directory.

This is a structured binary file that contains bytecode instructions of the program.

It can be run directly by executing this:

```
java -cp . Main
```

The -cp **(classpath)** option tells JVM to search class files in the current folder; Main is the name of the class.

Disassembling bytecode

All instructions in .class files are written in bytecode machine language. To make a .class file readable for humans, you should disassemble it. It's possible to do that using the javap disassembler embedded in JDK. It has the following path:

```
<JDK installation folder>/bin/javap
```

Let's disassemble our file:

```
javap -c Main.class
```

The -c argument means that we need to print out disassembled code, that is, the instructions that comprise Java bytecode for each of the methods in the class.

Here is our bytecode:

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

  public static void main(java.lang.String[]);
    Code:
       0: iconst_1
       1: istore_1
       2: iconst_2
       3: istore_2
       4: getstatic     #2  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: iload_1
       8: iload_2
       9: iadd
      10: invokevirtual #3  // Method java/io/PrintStream.println:(I)V
      13: return
}
```

You can see that the bytecode is quite readable. The file has a regular structure which is common for all .class files. It is interesting that Java compiler added the default no-arg constructor Main() for the class.

There is another argument -v for the javap command. It allows you to see more information about the class, file metadata, and values from the constant pool. Here is a part of the output:

```
Classfile /../../Main.class
  Last modified Oct 8, 2019; size 392 bytes
  MD5 checksum 7c6f013dc34260456bdde418433a1029
  Compiled from "Main.java"
```

```
public class Main
  minor version: 0
  major version: 55
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #4                    // Main
  super_class: #5                   // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:
   #1 = Methodref          #5.#14    // java/lang/Object."<init>":()V
   #2 = Fieldref           #15.#16  // java/lang/System.out:Ljava/io/PrintStream;
   #3 = Methodref          #17.#18  // java/io/PrintStream.println:(I)V
... a lot of other constants ...
```

We reduced the pool of constants since it was too long. Values from this pool are used during the program execution.

Bytecode instructions

Each bytecode instruction consists of a one-byte operation code: **opcode** followed by zero or more **operands**. There are about 200 bytecode instructions currently in use: the full list can be found on Wikipedia.

Many instructions have prefixes and/or suffixes referring to the types of operands they operate on: i for integer, l for long, s for short, b for byte, c for a character, f for float, d for double, a for a reference

Let's consider some of the most used in programs instructions:

- aload_0 loads a reference onto the stack from local variable 0;
- iconst_0, iconst_1, iconst_2 loads the int value 0, 1, or 2 onto the stack;
- istore_0, istore_1, istore_2 stores int value into the variable 0, 1, 2;
- iload_0, iload_1, iload_2 loads an int value from local variable 0, 1, 2;
- iadd, isub, imul, idiv performs basic arithmetic operations with integers;
- invokespecial invokes instance method on object *objectref* and puts the result on the stack;
- invokevirtual invokes virtual method on object *objectref* and puts the result on the stack;
- getstatic gets a static field *value* of a class, where the field is identified by field reference in the constant pool *index;*
- return returns void from a method.

Many instructions use stack since JVM works as a stack machine for calculations.

Now, we can read bytecode of the main method.

```
iconst_1         // push 1 onto the stack
istore_1         // assign 1 to the variable 1 (a)
iconst_2         // push 2 onto the stack
istore_2         // assign 2 to the variable 2 (b)
getstatic     #2  // Field java/lang/System.out:Ljava/io/PrintStream;
iload_1          // loads 1 from a
iload_2          // loads 2 from b
iadd             // calculate 1 + 2
invokevirtual #3  // Method java/io/PrintStream.println:(I)V
return           // return from the method main
```

Here, the command invokevirtual #3 takes an argument from the constant pool.

Many instructions use stack since JVM works as a stack machine for calculations.

In a Java class file, the first four bytes represent the magic number. The magic number is a fixed value that uniquely identifies the file as a Java class file. It's used by the Java Virtual Machine (JVM) to verify that the file being loaded is indeed a valid Java class file.

The magic number for Java class files is represented in hexadecimal as 0xCAFEBABE. In the file itself, it's stored as four individual bytes in big-endian order:

- Byte 1: CA (hexadecimal)
- Byte 2: FE (hexadecimal)
- Byte 3: BA (hexadecimal)
- Byte 4: BE (hexadecimal)

So, when you open a Java class file in a hex editor, the first four bytes you'll see are CA FE BA BE. This sequence of bytes serves as an identifier to the JVM, indicating that the file is a valid Java class file. If these bytes are not present or are incorrect, the JVM will not recognise the file as a valid class file.