

# Java Bytecode Compilation

Punit

## Abstract

High-level languages used for programming like Java, C, C++ etc. Compiles a program to its equivalent low level code which can be understood and executed by the machine. Here we will discuss about the Java compilation process, i.e., how Java compiles a code and what stages it have in compilation process. We basically focus on the Java bytecode and its advantages over native code. As we know the purpose of compilation is mainly to produce the executable version of a program. We also discuss the various approaches of compilation in java. The first approach is to compiling the java bytecode, second is the two stages process going through the java bytecode to java native code, the third approach is to go through bypassing the java bytecode and going directly to native machine code.

**Keywords:** Java Bytecode, Compilation process, Native code, JIT(Just in time), conditions in javabytecode, methods in java bytecode.

## 1. Introduction

The Java is the most popular programming languages now a days that we all know. Let us discuss about some basic concepts of java like Compilation process and its bytecode. As we know we write the java source code and the compiler(JVM) which is platform independent translates it into bytecode. Bytecode are the machine language of the Java virtual machine. When a Java virtual machine loads the class file, it gets one stream of bytecodes for each method in the class. The bytecode for a method are executed after the method is called during the running the program. They can be executed by interpretation, JIT

compiling or the technique chosen by the designer of a particular JVM. The Java bytecode allows us to use loops, conditional statement, methods etc. We will discuss further in detail about Java bytecode and other statements like conditional statement and methods in Java bytecode.

## 2. Java Compilation Process

The main purpose of compilation is to produce an executable file of a program. Java requires each class placed in its own source file and file must be same as class name with extension .java.

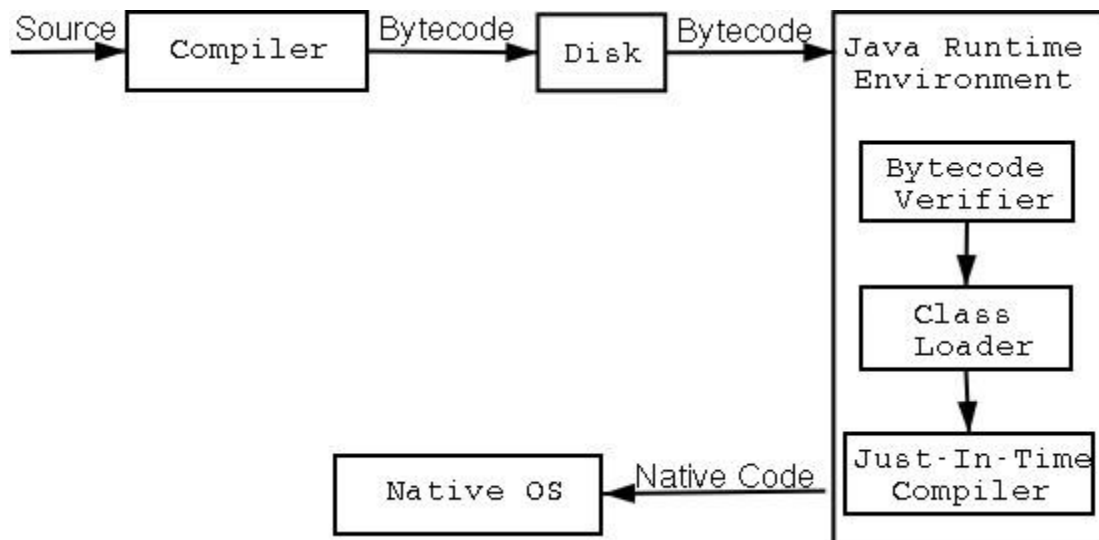


Fig. 1

When we start compiling source code, each class is placed in its own .class file that contain the bytecode. After finishes compiling all source files, the result class file will be equal to the source files, then which will combine to form your Java program. This is the stage where the class loader comes along with the verifier as shown in Fig 1.

The class loader is used for loading each class bytecode. When class is needed by the JVM finds the file that contains the bytecode for the class, then it is read into memory and passed to the defineClass. After that the verifier is invoked and process the bytecode in four passes to ensure it is safe. After the class is successfully verified, its loading is completed and it is available for use by the runtime. The Java bytecode allow us to easily decompile class files to source.

There exists three different approaches to Java compilation :

1. Compilation to Java bytecode
2. Two-stage process going through bytecode to native code
3. Bypasses Java bytecode and going directly to native machine code.

### 2.1 Just-in-time compilation:

A just-in-time compiler (a JIT) compiles Java bytecode to the native machine code of the machine which we are using (for example, an Intel Pentium processor or a SUN Sparc chip). This compilation is performed at the time when the request is made to execute the program (hence the name "just in time"). Java virtual machine include a JIT compiler which first compiles the Java bytecodes to native code and then runs the native code.

It would seem that this method of compilation combines the benefits of the byte code approach with the speed of the native code approach but still it has a drawback. A JIT runs in "user time", which means that the time taken to compile the bytecodes to native code is seen by the user as a delay in starting the execution of the program. This visibility to the user severely constrains the amount of time which the JIT compiler can take to run. However, the work of producing very efficient native code requires a number of different time-consuming analyses. A JIT cannot take the time to perform these so it must settle for producing compiled native code whose run-time performance could have been improved, had more time been available. Because the time to compile the program from bytecodes is perceived as part of the run time of the application and because the compiled native code cannot be highly optimised, it might be that the final run time of the application is not much better than if the program had been interpreted as

bytecodes in the first place, saving the overhead of the time taken by JIT compilation. For this reason many Java virtual machines (such as SUN's java and Microsoft's jview ) provide JIT compilation as an option, which can be turned off.

### 2.2 Native compilation:

If we are willing to give up the portability and compact size of executable program provided by Java bytecode then an option is to compile our Java source code directly to native code for our chosen machine. In this case the time taken by the compilation is performed only while the program is being developed and not every time that the program is executed. In this case it is possible to spend more time running the compiler, performing the time-consuming analyses which may lead to a more highly optimised native code program as a result. Such a compiler for Java is gcj, the GNU compiler for Java. This is part of a family of compiler programs which includes compilers for the languages C and C++. Some parts of the code of these compilers are shared and development tools such as the program debugger can be used on programs written in these languages and compiled with one of the GNU family of compilers.

### 2.3 Java Bytecode compilation:

The standard approach to Java compilation is to compile to Java bytecode and then to execute this on a Java Virtual Machine (JVM). The JVM interprets the Java bytecodes to run the program. Compilation to bytecode brings several genuine advantages. One advantage is that compiled programs are portable in the sense that they can be run on a number of different computer systems (such as Linux, Windows, Solaris, and many others) for which a JVM is available. The Java language is the only programming language in widespread use today which provides this capability. At best, programs written in other programming languages need to be recompiled if they are moved from one machine to another. More often, they need to be modified as well. The portability which Java provides is brought about because only the JVM needs to be ported from one machine to another in order to be able to port any Java bytecode program between machines.

## 3. Java Bytecode in detail

Let us discuss the bytecode in detail with small examples. As we know java bytecode having the extension .class after compilation of source code file. Let understand the bytecode concept with a small example: a pojo with one field, a setter and a getter

```
public class Foo { private
String bar;
```

```

public String getBar(){
return bar;
}
public void setBar(String bar) {
this.bar = bar; }

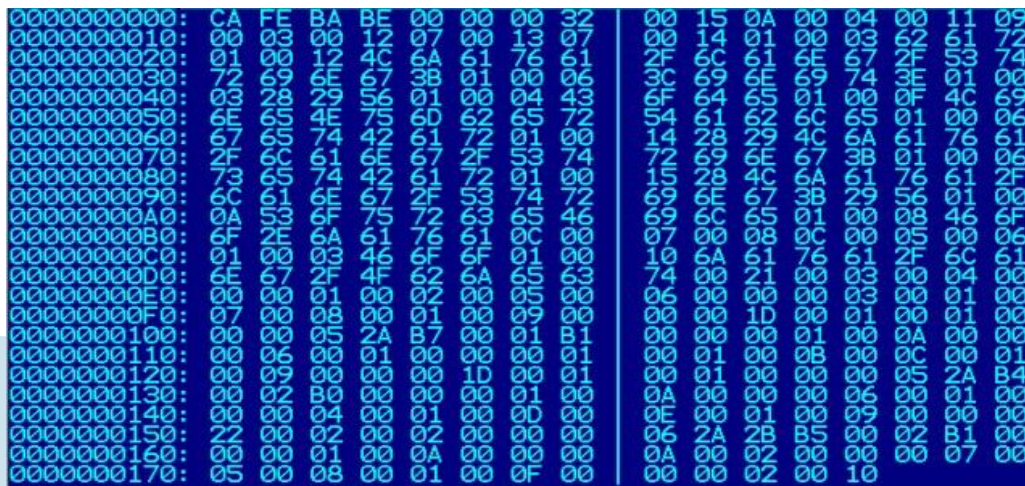
```

```

0: aload_0
}

```

First of all compile the file and get the file Foo.class which is the bytecode file. Open the bytecode file using hexeditor and you will see some hexcodes like as shown in fig. 2 below :



The image shows a hex editor view of the Foo.class file. The left column shows addresses from 00000000 to 00000017. The right column shows the corresponding hex bytes. The bytes are: CA FE BA BE 00 00 00 32 00 15 0A 00 04 00 11 09 00 03 00 00 03 62 61 72 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 74 3E 53 74 67 69 6E 67 3B 01 00 06 3C 69 6E 69 74 3E 01 00 06 6F 64 65 01 00 0F 4C 00 69 54 61 62 6C 65 72 6E 14 28 29 4C 6A 61 76 61 76 61 72 69 6E 67 3B 01 00 06 15 28 4C 6A 61 76 61 2F 69 6E 67 3B 29 56 01 00 69 6C 65 01 00 08 08 46 6F 07 00 08 0C 00 05 00 06 10 6A 61 76 61 2F 6C 61 74 00 21 00 03 00 04 00 06 00 00 00 03 00 01 00 00 00 00 1D 00 01 00 00 00 01 00 01 B1 00 00 00 01 00 0A 00 00 00 06 00 01 00 0E 00 01 00 09 00 00 06 2A 2B B5 00 02 B1 00 0A 00 02 00 00 07 00 00 02 00 10

Fig. 2 Hexcode of Foo.class file

```

1: aload_1
2: putfield    #2; //Field bar:Ljava/lang/String;
5: return
}

```

Each pair of hex numbers(a byte) is actually the tranlatable to opcodes, but it is too hard to read these hexcodes in binary format. Lets proceed to the mnemonical representation of these codes. Executing the javap -c Foo will printout the original bytecode stored in Foo.class file as following :

```

public class Foo extends java.lang.Object {
public    Foo();
Code:
0: aload_0
1:                invokespecial
    #1;    //Method
java/lang/Object.<init>:()V
4: return

public    java.lang.String    getBar();
Code:
0: aload_0
1: getfield    #2; //Field bar:Ljava/lang/String;
4: areturn

public    void    setBar(java.lang.String);
Code:

```

The class is very simple and it is easy to see the relation between the sourced code and the generated bytecode. First of all we notice that in the bytecode version of the class the compiler inferred the default constructor (as promised by the JVM spec). Secondly, if we study the Java bytecode instructions (in our example aload\_0 and aload\_1), we can see that some of the instructions have prefixes like aload\_0 or istore\_2. This is related to the type of the data that the instruction operates with. The prefix 'a' means that the opcode is manipulating an object reference. The prefix 'i' means the opcode is manipulating an integer.

### 3.1 Conditionals in Java bytecode:

Of course different programs, even if they achieve the same effect, will usually give rise to different bytecode sequences when compiled. We consider now two different ways of implementing a method to compute the absolute value of an integer (the absolute value of a negative integer  $-n$  is  $n$  whereas the absolute value of a positive integer  $n$  is  $n$  itself). We write two versions of a method to compute the absolute value of  $n$ . These versions are called absFirst() and absSecond(). The

difference between them is whether we test for being negative or test for being positive. In the first case we place the value  $-n$  on the true limb of the conditional and the value  $n$  on the false limb. In the second case we instead place  $n$  on the true limb of the conditional and  $-n$  on the

The control operators to break out of a loop or to continue with the next iteration have simple translations in Java bytecode instructions. Each causes a transfer of control, the break to the next statement after the loop and the

<pre>int absFirst(int n) {     if (n &lt; 0)         return -n;     else         return n; }</pre>	<pre>int absSecond(int n) {     if (n &gt; 0)         return n;     else         return -n; }</pre>
--	---

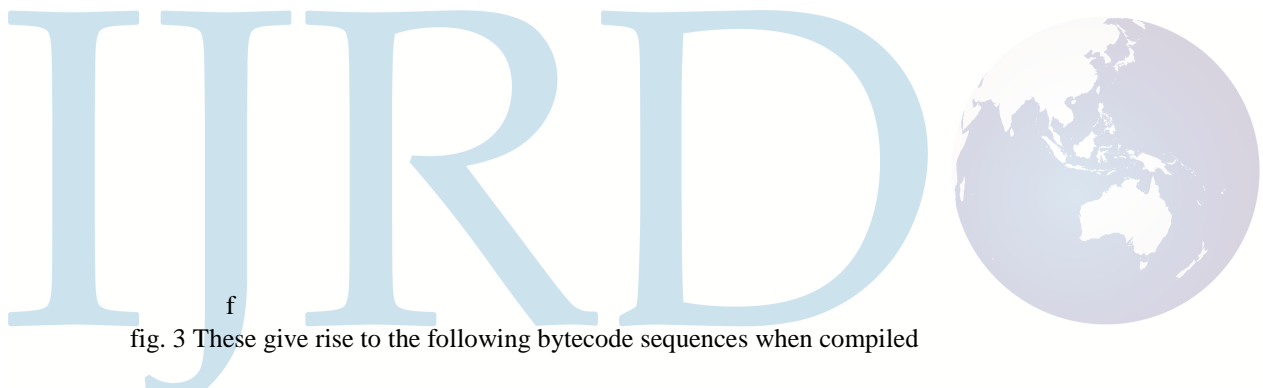


fig. 3 These give rise to the following bytecode sequences when compiled

<b>Method</b> <i>int absFirst(int)</i>	<b>Method</b> <i>int absSecond(int)</i>
0 <i>iload_1</i>	0 <i>iload_1</i>
1 <b><i>ifge</i></b> 7	1 <b><i>ifle</i></b> 6
4 <i>iload_1</i>	4 <i>iload_1</i>
5 <i>ineg</i>	5 <b><i>ireturn</i></b>
6 <b><i>ireturn</i></b>	6 <i>iload_1</i>
7 <i>iload_1</i>	7 <i>ineg</i>
8 <b><i>ireturn</i></b>	8 <b><i>ireturn</i></b>

Fig 4

continue to the update operation which precedes the loop condition evaluation. The following example illustrates this process.

The case of comparing with zero occurs so commonly in programs that specialised versions of the comparison operators are provided for this, *ifeq*, *ifne*, *iflt*, *ifge*, *ifgt*, *ifle*. The instruction *ineg* negates the integer on the top of the stack. The instruction *ireturn* returns the integer result on top of the stack.

### 3.2 Break and continue:

```

void breakContinue() {
    for (int i = 0 ; i < 99 ; i++) {
        if (i < 90) continue;
        else break;
    }
}

```

```

class Methods {
    static int first() {
        return second();
    }
    static int second() {
        return third();
    }
    static int third() {
        return 3;
    }
}

```

Fig. 5

### 3.3 Simple method invocation:

The simplest type of method to invoke in Java is a static method with no parameters. Below we show the Java source code for a class with three static methods

Seen from inside the method, formal parameters are simply numbered, just as local variables are. Thus the methods `add2()` and `add1()` refer to the integer variable numbered zero (using `iload 0`).

## 4. Advantages of Java Bytecode

There are three advantages of Java using bytecode instead of going to the native code of the system:

**4.1 Portability:** Each kind of computer has its unique instruction set. While some processors include the instructions for their predecessors, it's generally true that a program that runs on one kind of computer won't run on any other. Add in the services provided by the operating system, which each system describes in its

```

Method void breakContinue()
0  iconst_0
1  istore_1
2  goto 17
5  iload_1
6  bipush 90
8  if_icmpge 23
11 goto 14
14 itnc 1 1
17 iload_1
18 bipush 99
20 if_icmplt 5
23 return

```

```

class Methods
Method int first()
0  invokestatic #2
3  ireturn

Method int second()
0  invokestatic #3
3  ireturn

Method int third()
0  iconst_3
1  ireturn

```

Fig. 6

and the relevant part of the compiled bytecode for this class. The methods of the class are referred to by number so that method `first()` is #1, method `second()` is #2 and method `third()` is #3. Returning an integer result from an integer method is achieved by leaving the integer result on top of the operand stack.

own unique way, and you have a compatibility problem. In general, you can't write and compile a program for one kind of system and run it on any other without a lot of work. Java gets around this limitation by inserting its virtual machine between the application and the real environment (computer + operating system). If an application is compiled to Java bytecode and that bytecode is interpreted the same way in every environment then you can write a single program which will work on all the different platforms where Java is supported. (That's the theory, anyway. In practice there are always small incompatibilities lying in wait for the programmer.)

**4.2 Security:** One of Java's virtues is its integration into the Web. Load a web page that uses Java into your

browser and the Java code is automatically downloaded and executed. But what if the code destroys files, whether through malice or sloppiness on the programmer's part? Java prevents downloaded applets from doing anything destructive by disallowing potentially dangerous operations. Before it allows the code to run it examines it for attempts to bypass security. It verifies that data is used consistently: code that manipulates a data item as an integer at one stage and then tries to use it as a pointer later will be caught and prevented from executing. (The Java language doesn't allow pointer arithmetic, so you can't write Java code to do what we just described. However, there is nothing to prevent someone from writing destructive bytecode themselves using a hexadecimal editor or even building a Java bytecode assembler.) It generally isn't possible to analyze a program's machine code before execution and determine whether it does anything bad. Tricks like writing self-modifying code mean that the evil operations may not even exist until later. But Java bytecode was designed for this kind of validation: it doesn't have the instructions a malicious programmer would use to hide their assault.

**4.3 Size:** In the microprocessor world RISC is generally preferable over CISC. It's better to have a small instruction set and use many fast instructions to do a job than to have many complex operations implemented as single instructions. RISC designs require fewer gates on the chip to implement their instructions, allowing for more room for pipelines and other techniques to make each instruction faster. In an interpreter, however, none of this matters. If you want to implement a single instruction for the switch statement with a variable length depending on the number of case clauses, there's no reason not to do so. In fact, a complex instruction set is an advantage for a web-based language: it means that the same program will be smaller (fewer instructions of greater complexity), which means less time to transfer across our speed-limited network.

Against its benefits, the use of bytecode brings a disadvantage. It is often argued that interpretation of byte code programs is much slower than execution of native code, compiled only for the machine which we are actually using. Users of computer programs want efficient products: it is frustrating to use a computer program which pauses during execution or which cannot keep up with the speed of user input. The approach used to combine the usefulness of bytecode with the efficiency of native code is called just-in-time compilation.

## 5. References

[1] <http://stackoverflow.com/questions/3406942/how>

- exactly-does-java-compilation-take-place
- [2] <http://www.javaworld.com/jw-09-1996/jw-09bytecodes.html>
- [3] <http://althing.cs.dartmouth.edu/local/www.acm.uiuc.edu/sigmil/RevEng/ch02.html>
- [4] <http://arhipov.blogspot.in/2011/01/java-bytecodefundamentals.html>
- [5] The Java Virtual Machine is described in the book *The Java Virtual Machine Specification* [6] tion by Tim Lindholm and Frank Yellin, Addison-Wesley, Second edition, 1999.

