Java Magazine

Coding, JVM Internals, Tools

Understanding the constant pool inside a Java class file

Andrew Binstock | February 18, 2022

Taking a deep dive into the class file's symbol table

Download a PDF of this article

The venerable class file is the JVM's fundamental unit of execution. Every Java class produces a file that contains considerable data as well as executable bytecodes that implement the behavior. Class files are generated by the `javac` compiler and can be run in the following three modes:

- Standalone mode, provided they contain a `main()` method
- Bundled with other classes into JAR, EAR, or WAR files, which are simply zipped collections of classes and data items
- Bundled into modules and then into executable images via tools such as `jlink`

When it comes time to execute a specific class, the JVM locates the class file and loads it. The loading process involves parsing the file into its various fields and then placing the parsed data in a convenient format into the JVM's *method area*. This is an area shared across threads where variables and methods, among other items, can be looked up.

The class file format has changed little over the releases of Java: It consists of a file header, some bytes identifying the Java version that generated the class file, a significant area called the constant pool (which I discuss shortly), additional data fields, the methods, and finally a series of attributes.

I discuss shortly), additional data fields, the methods, and finally a series of attributes.

Over the years, the changes brought by various releases have not altered the layout of the class file, but they have changed the data items and especially the attributes stored in the class file. Because the original layout was extensible, these changes happened without disruption. To ensure safe execution, the bytes identifying the Java version in the file prevent older versions of the runtime from executing class files with newer features.

Introducing the constant pool

One of the most important sections of a class file is the *constant pool*, which is a collection of entries that serve as a symbol table of sorts for the class. The constant pool contains the names of classes that are referenced, initial values of strings and numeric constants, and other miscellaneous data crucial to proper execution.

You can learn a lot by looking at the constant pool for the following simple class:

```
class Hello {
    public static void main( String[] args ) {
        for( int i = 0; i < 10; i++ )
            System.out.println( "Hello from Hello.main!" );
  }
}
```

Copy code snippet

To see the constant pool, use the `javap` class file disassembler included in the JDK. Running `javap` with the verbose `-v` option prints a wealth of detail about the class, including the constant pool and the bytecode for all the methods. Running `javap -v Hello.class`, I get a listing of 83 lines. Here is the constant pool portion of that output.

```
   #1 = Methodref          #6.#16         // java/lang/Object."<init>":()V
   #2 = Fieldref           #17.#18        // java/lang/System.out:Ljava/io/Pr
   #3 = String             #19            // Hello from Hello.main!
   #4 = Methodref          #20.#21        // java/io/PrintStream.println:(Lja
   #5 = Class              #22            // Hello
   #6 = Class              #23            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               main
  #12 = Utf8               ([Ljava/lang/String;)V
  #13 = Utf8               StackMapTable
```

```
    #13 = Utf8               StackMapTable
    #14 = Utf8               SourceFile
    #15 = Utf8               Hello.java
    #16 = NameAndType        #7:#8          // "<init>":()V
    #17 = Class              #24            // java/lang/System
    #18 = NameAndType        #25:#26        // out:Ljava/io/PrintStream;
    #19 = Utf8               Hello from Hello.main!
    #20 = Class              #27            // java/io/PrintStream
    #21 = NameAndType        #28:#29        // println:(Ljava/lang/String;)V
    #22 = Utf8               Hello
    #23 = Utf8               java/lang/Object
    #24 = Utf8               java/lang/System
    #25 = Utf8               out
    #26 = Utf8               Ljava/io/PrintStream;
    #27 = Utf8               java/io/PrintStream
    #28 = Utf8               println
    #29 = Utf8               (Ljava/lang/String;)V
```

Copy code snippet

The format of this data will look familiar from other discussions in this magazine, especially in JVM-focused articles by Ben Evans, such as "Understanding Java method invocation with invokedynamic."

The numbers on the left are simply the entry number for the items. Note that the first entry is `#1`, not `#0`. There is a dummy entry implicit at slot 0 that is never referenced. The second column gives the type of entry, the third column contains the value of the entry, and the data after the `//` characters is `javap`'s helpful way of telling you what's being referred to. I'll clarify this shortly.

Perhaps the most striking thing about this listing is the frequency of entry type `Utf8`, which is an internal-only format used in the Java class to represent string data. (Technically, this entry slightly diverges from the UTF-8 standard to eliminate 0x00 bytes; for all other intents, it's a UTF-8 encoding.)

All the other entries in this constant pool eventually resolve to one of the `Utf8` entries. Let's see that in practice: There are four entries of type `Class`. Let's take the first one (entry `#5`). That entry points to entry `#22`, which is a `Utf8` entry with the value `Hello`, which is the name of the class being examined. As mentioned in the previous paragraph, by showing it after the double slashes of entry `#5javap`, helpfully saves you from having to jump around to get this value.

The internal referencing between entries can become complicated. For example, the `MethodRef` entry for `#4` is a reference to a method. It eventually points to the `println()` method, which is called in the original code. The `MethodRef` entry points to two additional entries, as follows:

- First is `#20`, which is the class name. This name is shown using the JVM's internal format in which the dots that usually separate the parts of a class's name are replaced by forward slashes.
- The second is `#21`, which points to a `NameAndType` record. This record points to two other `Utf8` entries, which give the name of the method and its signature.

When you string together the class name, method name, and signature, you get the following output, which is also shown to the right of the double slashes for slot #4:

```
java/io/PrintStream.println(Ljava/lang/String;)V
```

(I removed the colon, which was inserted by `javap` for readability.) This string is worthy of careful examination. The first thing that stands out is the part in parentheses that begins with an *L* and ends with a semicolon. The parentheses express the type of parameters the method accepts. The descriptors for primitive types are

- `B` (byte)
- `C` (char)
- `D` (double)
- `F` (float)
- `I` (int)
- `J` (long)
- `S` (short)
- `Z` (Boolean)

If the code used the form of `println()` that accepted a float, the signature would be `(F)V`. This example, however, sends a string to `println()`; the string is not a primitive but rather an object. Objects are expressed by `L` followed by the name of the class that defines their type, followed by a semicolon. Therefore, the method expects to be passed a string. The method returns nothing, which is communicated by `V` (for void) after the closing parenthesis.

Note that arrays are shown using opening brackets for each dimension, followed by the type of the array. The signature for `main()`, which takes an array of strings, is

```
([Ljava/lang/String;)V
```

Note the opening bracket inside the opening parenthesis.

Returning to the program, you'll notice that the Java source code called `System.out.println()`, while the function pointed to here is `Java.io.PrintStream.println()`. That is because `System.out` is a static field. It's of type `PrintStream`, whose `println()` static method is being called here. This transmutation is conveyed by entry #2 in the constant pool.

Now, look at entry #1, a `MethodRef` (method reference) to an `<init>()V` method in `java.lang.Object`. This is the default constructor that the compiler inserts into classes that don't declare their own constructor. When a class fails to have a constructor, the compiler goes to its superclass looking for one. If the superclass doesn't have one, the JVM looks at that superclass' superclass, and so on up the chain. If no superclass has a constructor, the compiler will eventually get to the topmost object in the hierarchy, `java.lang.Object`, and use its constructor.

The constructor has the name `<init>`, which you'll notice is a name that cannot appear in Java code, because method names are not allowed to begin with the < symbol. This last constraint is a signal that this call was inserted by the compiler and not derived from user code. An interesting side note is that the compiler creates this constructor for a class automatically—even for a class whose only method is the static `main()`, meaning the constructor is not called.

In the bytecode, the `Utf8`, `MethodRef`, and so forth are single-byte tags that identify the record that follows. There are more tags than shown in the listing. These include tags for each of the primitive types (typically used to initialize static versions of the data item), method handles, module and package entries, and specialized entries to handle `invokedynamic` instructions. The complete list of constant pool tags is in the *JVM Specification* document section 4.4.

Constant pool entries refer to each other using a two-byte unsigned value. In theory, this would imply that the largest constant pool could have 65,534 entries (the unsigned two-byte maximum less the initial dummy entry), but such is not the case. Entries for longs and doubles occupy two slots in the constant pool, with the second slot being unused and inaccessible (this is enforced by the JVM).

This design added complexities to JVM implementations and is gently lamented in the *JVM Specification* document section 4.4.5: "In retrospect, making 8-byte constants take two constant pool entries was a poor choice." This kind of candor makes the documentation a pleasure to read.

In practice, constant pools rarely if ever threaten to breach the maximum. For example, the `java.math.BigDecimal` class is a massive entity with an amazing 167 methods and 37 initialized fields. Its constant pool has 1,533 entries, which is a lot but nowhere near the permitted maximum.

Using the constant pool

Earlier in this article, I referred to the constant pool as a symbol table of sorts, which is how it's used by executing methods. Look at the bytecode for the `main()` method in the previous program.

```
 0: iconst_0
 1: istore_1
 2: iload_1
 3: bipush        10
 5: if_icmpge     22
 8: getstatic     #2  // Field java/lang/System.out:Ljava/io/PrintStream;
11: ldc           #3  // String Hello from Hello.main!
13: invokevirtual #4  // Method java/io/PrintStream.println:(Ljava/lang/Strin
16: iinc          1, 1
19: goto          2
22: return
```

Copy code snippet

Methods consist of one-byte instructions (hence the term *bytecode*) followed by zero or more arguments. The arguments are either values or references to entries in the constant pool. This listing, which is generated by `javap`, consists of a left column that shows the location of the current bytecode instruction as an offset from 0, a second column showing the bytecode instruction, and a third column that contains any arguments; also, as before, `javap` has put helpful resolutions of values behind double slashes.

The first three instructions load a zero, which is the count of completed loops, into a local variable and onto the Java stack, and then `bipush` pushes a 10 onto the stack. This 10 represents the maximum number of loop iterations in the original Java code. A comparison between the two is made, as follows:

- If the counter is greater than or equal to 10, the code jumps to instruction `#22`, which is a `return` statement that exits the function. Because the function being returned is `main()`, it ends the program.
- However, if 10 iterations of the loop have not occurred, the logic drops to the following `getstatic` instruction at position `#8`, which fetches the object `System.out`. (All that the bytecode knows is that it's fetching a field specified by constant pool entry `#2`.) Looking this up, as shown previously, refers to constant pool entries `#17` and `#18`, which point to a `java.io.PrintStream` object named `out`.

The next instruction, `ldc`, loads the string located at constant pool entry `#3` and puts it onto the stack.

The following instruction, `invokevirtual`, calls the `println()` method of `java.io.PrintStream` as specified in constant pool entry #4. Per the previous discussion of this entry, it expects a string, which it finds on the stack of the calling function. After printing the string, the remaining code increments the counter previously initialized to 0 and runs through the loop again.

The use of comparisons and jumps that depend on the result of the `compare` instruction is how the Java compiler encodes loops. This will be familiar to assembly language programmers.

Class files in practice

Everything I've described so far is roughly how the JVM works in its *initial* parse of a class and first run through the `main()` method. Subsequent runs are far more optimized than that process. Why? A performant JVM could not afford on every iteration of the loop to look up constant pool entry `#2`, then from there jump to entries `#17` and `#18`, and from there go to fetch the pointed-to object.

In fact, there is a runtime constant pool, which is an optimized representation of the parsed entries. For example, it might replace the reference of entry `#2` with a direct link to the object to be fetched, saving several lookups in the process.

Likewise, method lookups are accelerated by a variety of tricks. For example, let's return to the `BigDecimal` class I mentioned earlier. It has 137 methods, which are accessed in the method area of the JVM, where they were placed by the class loader.

A problem is that the constant pool tells you only the name and the signature of the method. To find it initially, the JVM must search to find the method it's looking for. To accelerate the search, the JVM creates a data structure called the *MTable* (for *method table*), which holds the name of the methods and pointers to their bytecodes. The MTable also contains the names of all methods in superclasses with the corresponding pointers. In this way, the climb through superclasses discussed earlier is resolved by a single lookup.

The MTable can be used for other things. For example, I'm presently working on the Jacobin project, which is writing a more-than-minimal JVM in the Go language. The project uses the MTable for purposes such as redirecting the Jacobin JVM to use a function written in Go rather than in its Java counterpart. This is done by sticking an entry into the MTable using the method name and signature as the key and a

This is done by sticking an entry into the Mtable using the method name and signature as the key and a pointer to the Go code as the value. This is useful especially for operating system APIs (which are often written in native code) and for performance optimization.

Conclusion

Navigating decompiled Java classes is a useful skill and the mark of an advanced understanding of Java programming. It is particularly helpful when a bit of code does not do what you expect. By popping open the class with `javap` and examining the bytecode, you can tell how the compiler interpreted your instructions.

By the way, many of the behind-the-scenes details exposed by *Java Magazine* quizzes can be understood by using this approach though, of course, you can't run `javap` while taking the certification exam.

Dig deeper

- Understanding Java method invocation with invokedynamic
- Mastering the mechanics of Java method invocation
- Behind the scenes: How do lambda expressions really work in Java?
- How the JVM locates, loads, and runs libraries
- Containerizing apps with jlink
- Four common pitfalls of the BigDecimal class and how to avoid them

Andrew Binstock

Andrew Binstock (@platypusguy) was formerly the editor in chief of *Java Magazine*. Previously, he was the editor of *Dr. Dobb's Journal*. He co-founded the company behind the open-source iText PDF library, which was acquired in 2015. His book on algorithm implementation in C went through 16 printings before joining the long tail. Previously, he was the editor in chief of *UNIX Review* and, earlier, the founding editor of the *C Gazette*. He lives in Silicon Valley with his wife. When not coding or editing, he studies piano.

Resources for

About

Careers

Developers

Investors

Partners

Startups

Why Oracle

Analyst Reports

Best CRM

Cloud Economics

Corporate Responsibility

Diversity and Inclusion

Security Practices

Learn

What is Customer Service?

What is ERP?

What is Marketing Automation?

What is Procurement?

What is Talent Management?

What is VM?

What's New

Try Oracle Cloud Free Tier

Oracle Sustainability

Oracle COVID-19 Response

Oracle and SailGP

Oracle and Premier League

Oracle and Red Bull Racing Honda

Contact Us

US Sales 1.800.633.0738

How can we help?

Subscribe to Oracle Content

Try Oracle Cloud Free Tier

Events

News