Mastering the mechanics of Java method invocation

Special bytecodes make calling methods particularly efficient. Knowing how they operate reveals how the JVM executes your code.

by Ben Evans

February 26, 2021

[Java Magazine is pleased to republish this article from Ben Evans, published in 2017, about Java Virtual Machine internals.

—Ed.]

This article explains how the JVM executes methods in Java 8 and Java 9. [And later. —Ed.] This is a fundamental topic about the internals of the JVM that is essential background for anyone who wants to understand the JVM's just-in-time (JIT) compiler or tune the execution of applications.

To get started, look at the following simple bit of Java code:

```java
long time = System.currentTimeMillis();
HashMap<String, String> hm = new HashMap<>();
hm.put("now", "bar");
Map<String, String> m = hm;
m.put("foo", "baz");
```

To see the bytecode that the Java compiler produces for this code, use the `javap` tool with the `-c` switch to display the decompiled code, as follows (indented lines are merely continued from the previous line):

```
0: invokestatic #2 // Method java/lang/System.currentTimeMillis()
3: lstore_1
4: new #3 // class java/util/HashMap
7: dup
8: invokespecial #4 // Method java/util/HashMap
```
Java programmers who are new to looking at code at the JVM level might be surprised to learn that Java method calls are actually turned into one of several possible bytecodes of the form `invoke*`, `invokestatic`, `invokevirtual`, or `invokeinterface`.

Let's take a closer look at the first part of the decompiled code.

```
0: invokestatic #2 // Method java/lang/System
3: lstore_1
```

The static call to `System.currentTimeMillis()` is turned into an `invokestatic` opcode that appears at position 0 in the bytecode. This method takes no parameters, so nothing needs to be loaded onto the evaluation stack before the call is dispatched.

Next, the two bytes 00 02 appear in the byte stream. These are combined into a 16-bit number (#2, in this case) that is used as an offset into a table—called the constant pool—within the class file. All constant pool indices are 16 bits, so whenever an opcode needs to refer to an entry in the pool, there will always be two subsequent bytes that encode the offset of the entry.

The decompiler helpfully includes a comment that lets you know which method offset #2 corresponds to. In this case, as expected, it's the method `System.currentTimeMillis()`. In the decompiled output, `javap` shows the name of the called method, the types of parameters the method takes (in parentheses), followed by the return type of the method.

Upon return, the result of the call is placed on the stack, and at offset 3 you see the single, argument-less opcode, `lstore_1`, which saves the return value in a local variable of type `long`.

Human readers are, of course, able to see that this value is never used again. However, one of the design goals of the Java compiler is to represent the contents of the Java source code as
faithfully as possible—whether it makes logical sense or not. Therefore, the return value of System.currentTimeMillis() is stored, even though it is not used after this point in the program.

Now look at the next chunk of the decompiled code.

```
4: new #3 // class java/util/HashMap
7: dup
8: invokespecial #4 // Method java/util/HashMap
11: astore_3
12: aload_3
13: ldc #5 // String now
15: ldc #6 // String bar
17: invokevirtual #7 // Method java/util/HashMap
        (Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
19: astore_3
20: pop
```

Bytecodes 4 to 10 create a new HashMap instance before instruction 11 saves a copy of it in a local variable. Next, instructions 12 to 16 set up the stack with the HashMap object and the arguments for the call to put(). The actual invocation of the put() method is performed by instructions 17 to 19.

The invoke opcode used this time is invokevirtual. This differs from a static call, because a static call does not have an instance on which the method is called; such an instance is sometimes called the receiver object. (In bytecode, an instance call must be set up by placing the receiver and any call arguments on the evaluation stack and then issuing the invoke instruction.) In this case, the return value from put() is not used, so instruction 20 discards it.

The sequence of bytes from 21 to 25 seems rather odd at first glance.

```
21: aload_3
22: astore 4
24: aload 4
26: ldc #8 // String foo
28: ldc #9 // String baz
30: invokeinterface #10, 3 // InterfaceMethod
              (Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
35: pop
```

The HashMap instance that was created at bytecode 4 and saved to a local variable 3 at instruction 11 is now loaded back onto the stack, and then a copy of the reference is saved to local variable 4. This process removes it from the stack, so it must be reloaded (from variable 4) before use.

This shuffling occurs because in the original Java code, an additional local variable (of type Map rather than HashMap) is
created, even though it always refers to the same object as the original variable. This is another example of the bytecode staying as close as possible to the original source code. One of the main reasons for this so-called “dumb bytecode” approach that Java takes is to provide a simple input format for the JVM’s JIT compiler.

After the stack and variable shuffling, the values to be placed in the Map are loaded at instructions 26 to 29. Now that the stack has been prepared with the receiver and the arguments, the call to put() is dispatched at instruction 30. This time, the opcode is invokevirtual—even though the same method is being called. Once again, the return value from put() is discarded, via the pop at instruction 35.

So far, you’ve seen that invokevirtual, invokevirtual, or invokeinterface can be produced by the Java compiler, depending on the context of the call.

**JVM bytecodes for invoking methods**

Look at all the five JVM bytecodes that can be used to invoke methods (see Table 1). In each case, the bytes b0 and b1 are combined into the constant pool offset represented by c1.

<table>
<thead>
<tr>
<th>OP CODE NAME</th>
<th>ARGUMENTS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>invokevirtual</td>
<td>b0 b1</td>
<td>Invokes the method found at copic via virtual dispatch</td>
</tr>
<tr>
<td>invokespecial</td>
<td>b0 b1</td>
<td>Invokes the method found at copic via “special”, that is exact dispatch</td>
</tr>
<tr>
<td>invokeinterface</td>
<td>b0 b1 b0 b0</td>
<td>Invokes the interface method found at copic &amp; using interface offset lookup</td>
</tr>
<tr>
<td>invokestatic</td>
<td>b0 b1</td>
<td>Invokes the static method found at copic</td>
</tr>
<tr>
<td>invokedynamic</td>
<td>b0 b1 b0 b0</td>
<td>Dynamically loads up which method to invoke and calls it</td>
</tr>
</tbody>
</table>

Table 1. JVM bytecodes for invoke methods

It can be a useful exercise to write some Java code, and then disassemble it with javap, to see what circumstances produce each form of the bytecodes.

The most common type of method invocation is invokevirtual, which refers to virtual dispatch. The term virtual dispatch means that the exact method to be invoked is determined at runtime. To understand this, you need to know that each class present in a running application has an area of memory inside the JVM that holds metadata corresponding to that type. This area is called a klass (in Java HotSpot VM, at least) and can be thought of as the JVM’s representation of information about the type.

In Java 7 and earlier, the klass metadata lived in an area of the Java heap called permgen. Because objects within the Java heap must have an object header (called an oop), the classes were known as klassOops. In Java 8 and Java 9, the klass metadata was moved out of the Java heap into the native heap, so the object headers are no longer required. Some of the information from the klass is available to Java programmers via
the `Class<?>` object corresponding to the type—but they are separate concepts.

One of the most important areas of the klass is the vtable. This area is essentially a table of function pointers that point to the implementations of methods defined by the type. When an instance method is called via `invokevirtual`, the JVM consults the vtable to see exactly which code needs to be executed. If a klass does not have a definition for the method, the JVM follows a pointer to the klass corresponding to the superclass and tries again.

This process is the basis of method overriding in the JVM. To make the process efficient, the vtables are laid out in a specific way. Each klass lays out its vtable so that the first methods to appear are the methods that the parent type defines. These methods are laid out in the exact order that the parent type used. The methods that are new to this type and are not declared by the parent class come at the end of the vtable.

The result? When a subclass overrides a method, it will be at the same offset in the vtable as the implementation being overridden. This makes the lookup of overridden methods completely trivial because their offset in the vtable will be the same as the offset of their parent.

**Figure 1** shows an example defined by the classes *Pet*, *Cat*, and *Bear* and the interface *Furry*.

![Figure 1. Simple inheritance hierarchy](image)

The vtables for these classes are laid out in Java 7, as shown in **Figure 2**. As you can see, this figure shows the Java 7 layout within permgen, so it refers to klassOops and has the two words of the object header (shown as `m` and `kk` in the figure). As discussed previously, these entries would not be present in Java 8 and Java 9, but all else in the diagram remains the same.
If you call `Cat::feed`, the JVM will not find an override in the `Cat` class and instead will follow the pointer to the class of `Pet`. This klass does have an implementation for `feed()`, so this is the code that will be called. This vtable structure works well because Java implements only single inheritance of classes. This means there is only one direct superclass of any type (except for `Object`, which has no superclass).

In the case of `invokeinterface`, the situation is a little more complicated. For example, the `groom()` method will not necessarily appear in the same place in the vtable for every implementation of `Furry`. The different offsets for `Cat::groom` and `Bear::groom` are caused by the fact that their class inheritance hierarchies differ. The result is that some additional lookup is needed when a method is invoked on an object for which only the interface type is known at compile time.

Note that even though slightly more work is done for the lookup of an interface call, you should not try to micro-optimize by avoiding interfaces. Remember that the JVM has a JIT compiler, and it will essentially eliminate any performance difference between the two cases.

**Example of invoking methods**

Here's another example. Consider this bit of code.

```java
Cat tom = new Cat();
Bear pooh = new Bear();
Furry f;
tom.groom();
pooh.groom();
f = tom;
f.groom();
f = pooh;
f.groom();
```

This code produces the following bytecodes:
The two calls at 27 and 35 look like they are the same, but they actually invoke different methods. The call at 27 will invoke `Cat::groom`, whereas the call at 35 will invoke `Bear::groom`.

With this background on `invokevirtual` and `invokeinterface`, the behavior of `invokespecial` is now easier to understand. If a method is invoked by `invokespecial`, it does not undergo virtual lookup. Instead, the JVM will look only in the exact place in the vtable for the requested method. This means that an `invokespecial` is used for three cases: private methods, calls to a superclass method, and calls to the constructor body (which is turned into a method called `<init> in bytecode). In all three cases, virtual lookup and the possibility of overriding must be explicitly excluded.

**Final methods**

There remains one corner case that should be mentioned: the case of final methods. At first glance, it might appear that calls to final methods would also be turned into `invokespecial` instructions. However, *Java Language Specification* section 13.4.17 has something to say about this case: “Changing a method that is declared `final` to no longer be declared `final` does not break compatibility with pre-existing binaries.”

Suppose a compiler had compiled a call to a final method into an `invokespecial`. If the method then changed to no longer be final, it could be overridden in a subclass. Now, suppose that an instance of the subclass was passed into the compiled code. The `invokespecial` would be executed, and then the wrong implementation of the method would be called. This would be a violation of the rules of Java’s object orientation (strictly speaking, it would violate the Liskov Substitution Principle).
For this reason, calls to final methods must be compiled into `invokevirtual` instructions. In practice, the Java HotSpot VM contains optimizations that allow final methods to be detected and executed extremely efficiently.

**Conclusion**

I have now examined four of the five invocation instructions that the JVM supports. The remaining case is `invokedynamic`, and it is such a rich and interesting subject that it requires an article all to itself.

**Dig deeper**

- The `javap` command
- The Java Virtual Machine instruction set
- Real-world bytecode handling with ASM

---

**Ben Evans**

Ben Evans (@kittylyst) is a Java Champion and Principal Engineer at New Relic. He has written five books on programming, including *Optimizing Java* (O'Reilly). Previously he was a founder of jClarity (acquired by Microsoft) and a member of the Java SE/EE Executive Committee.