JAVA SE

Behind the scenes: How do lambda expressions really work in Java?

Look into the bytecode to see how Java handles lambdas.

by Ben Evans

September 25, 2020

Download a PDF of this article

What does a lambda expression look like inside Java code and inside the JVM? It is obviously some type of value, and Java permits only two sorts of values: primitive types and object references. Lambdas are obviously not primitive types, so a lambda expression must therefore be some sort of expression that returns an object reference.

Let's look at an example:

```
public class LambdaExample {
    private static final String HELLO = "Hell

    public static void main(String[] args) th
        Runnable r = () -> System.out.println
        Thread t = new Thread(r);
        t.start();
        t.join();
    }
}
```

Programmers who are familiar with inner classes might guess that the lambda is really just syntactic sugar for an anonymous implementation of `Runnable`. However, compiling the above class generates a single file: `LambdaExample.class`. There is no additional class file for the inner class.

This means that lambdas are *not* inner classes; rather, they must be some other mechanism. In fact, decompiling the bytecode via `javap -c -p` reveals two things. First is the fact that the lambda body has been compiled into a private static method that appears in the main class:

```
private static void lambda$main$0();
    Code:
       0: getstatic     #7                  /
       3: ldc           #9                  /
       5: invokevirtual #10                 /
       8: return
```

You might guess that the signature of the private *body method* matches that of the lambda, and indeed this is the case. A lambda such as this

```
public class StringFunction {
    public static final Function<String, Inte
}
```

will produce a body method such as this, which takes a string and returns an integer, matching the signature of the interface method

```
private static java.lang.Integer lambda$stati
    Code:
       0: aload_0
       1: invokevirtual #2                  /
       4: invokestatic  #3                  /
       7: areturn
```

The second thing to notice about the bytecode is the form of the main method:

```
public static void main(java.lang.String[]) t
    Code:
       0: invokedynamic #2,  0              /
       5: astore_1
       6: new           #3                  /
       9: dup
      10: aload_1
      11: invokespecial #4                  /
      14: astore_2
      15: aload_2
      16: invokevirtual #5                  /
      19: aload_2
      20: invokevirtual #6                  /
      23: return
```

Notice that the bytecode begins with an `invokedynamic` call. This opcode was added to Java with version 7 (and it is the only opcode ever added to JVM bytecode). I discussed method

invocation in "Real-world bytecode Handling with ASM" and in "Understanding Java method invocation with invokedynamic" which you can read as companions to this article.

The most straightforward way to understand the `invokedynamic` call in this code is to think of it as a call to an unusual form of the factory method. The method call returns an instance of some type that implements `Runnable`. The exact type is not specified in the bytecode and it fundamentally does not matter.

The actual type does not exist at compile time and will be created on demand at runtime. To better explain this, I'll discuss three mechanisms that work together to produce this capability: call sites, method handles, and bootstrapping.

Call sites

A location in the bytecode where a method invocation instruction occurs is known as a *call site*.

Java bytecode has traditionally had four opcodes that handle different cases of method invocation: static methods, "normal" invocation (a virtual call that may involve method overloading), interface lookup, and "special" invocation (for cases where overload resolution is *not* required, such as superclass calls and private methods).

Dynamic invocation goes much further than that by offering a mechanism through which the decision about which method is actually called is made by the programmer, on a per-call site basis.

Here, `invokedynamic` call sites are represented as `CallSite` objects in the Java heap. This isn't strange: Java has been doing similar things with the Reflection API since Java 1.1 with types such as `Method` and, for that matter, `Class`. Java has many dynamic behaviors at runtime, so there should be nothing surprising about the idea that Java is now modeling call sites as well as other runtime type information.

When the `invokedynamic` instruction is reached, the JVM locates the corresponding call site object (or it creates one, if this call site has never been reached before). The call site object contains a *method handle*, which is an object that represents the method that I actually want to invoke.

The call site object is a necessary level of indirection, allowing the associated invocation target (that is, the method handle) to change over time.

There are three available subclasses of `CallSite` (which is abstract): `ConstantCallSite`, `MutableCallSite`, and `VolatileCallSite`. The base class has only package-private constructors, while the three subtypes have public constructors. This means that `CallSite` cannot be directly subclassed by

user code, but it is possible to subclass the subtypes. For example, the JRuby language uses `invokedynamic` as part of its implementation and subclasses `MutableCallSite`.

Note: Some `invokedynamic` call sites are effectively just lazily computed, and the method they target will never change after they have been executed the first time. This is a very common use case for `ConstantCallSite`, and this includes lambda expressions.

This means that a nonconstant call site can have many different method handles as its target over the lifetime of a program.

Method handles

Reflection is a powerful technique for doing runtime tricks, but it has a number of design flaws (hindsight is 20/20, of course), and it is definitely showing its age now. One key problem with reflection is performance, especially since reflective calls are difficult for the just-in-time (JIT) compiler to inline.

This is bad, because inlining is very important to JIT compilation, not the least of which is because it's usually the first optimization applied and it opens the door to other techniques (such as escape analysis and dead code elimination).

A second problem is that reflective calls are linked every time the call site of `Method.invoke()` is encountered. That means, for example, that security access checks are performed. This is very wasteful because the check will typically either succeed or fail on the first call, and if it succeeds, it will continue to do so for the life of the program. Yet, reflection does this linking over and over again. Thus, reflection incurs a lot of unnecessary cost by relinking and wasting CPU time.

To solve these problems (and others), Java 7 introduced a new API, `java.lang.invoke`, which is often casually called *method handles* due to the name of the main class it introduced.

A method handle (MH) is Java's version of a type-safe function pointer. It's a way of referring to a method that the code might want to call, similar to a `Method` object from Java reflection. The MH has an `invoke()` method that actually executes the underlying method, in just the same way as reflection.

At one level, MHs are really just a more efficient reflection mechanism that's closer to the metal; anything represented by an object from the Reflection API can be converted to an equivalent MH. For example, a reflective `Method` object can be converted to an MH using `Lookup.unreflect()`. The MHs that are created are usually a more efficient way to access the underlying methods.

MHs can be adapted, via helper methods in the `MethodHandles` class, in a number of ways such as by

composition and the partial binding of method arguments (currying).

Normally, method linkage requires exact matching of type descriptors. However, the `invoke()` method on an MH has a special polymorphic signature that allows linkage to proceed regardless of the signature of the method being called.

At runtime, the signature at the `invoke()` call site should look like you are calling the referenced method directly, which avoids type conversions and autoboxing costs that are typical with reflected calls.

Because Java is a statically typed language, the question arises as to how much type-safety can be preserved when such a fundamentally dynamic mechanism is used. The MH API addresses this by use of a type called `MethodType`, which is an immutable representation of the arguments that a method takes: the signature of the method.

The internal implementation of MHs was changed during the lifetime of Java 8. The new implementation is called *lambda forms*, and it provided a dramatic performance improvement with MHs now being better than reflection for many use cases.

Bootstrapping

The first time each specific `invokedynamic` call site is encountered in the bytecode instruction stream, the JVM doesn't know which method it targets. In fact, there is no call site object associated with the instruction.

The call site needs to be *bootstrapped*, and the JVM achieves this by running a bootstrap method (BSM) to generate and return a call site object.

Each `invokedynamic` call site has a BSM associated with it, which is stored in a separate area of the class file. These methods allow user code to programmatically determine linkage at runtime.

Decompiling an `invokedynamic` call, such as that from my original example of a `Runnable`, shows that it has this form:

```
0: invokedynamic #2,  0
```

And in the class file's constant pool, notice that entry #2 is a constant of type `CONSTANT_InvokeDynamic`. The relevant parts of the constant pool are

```
#2 = InvokeDynamic      #0:#31
   ...
  #31 = NameAndType        #46:#47        //
```

```
   #46 = Utf8               run
   #47 = Utf8               ()Ljava/lang/Runna
```

The presence of 0 in the constant is a clue. Constant pool entries are numbered from 1, so the 0 reminds you that the actual BSM is located in another part of the class file.

For lambdas, the `NameAndType` entry takes on a special form. The name is arbitrary, but the type signature contains some useful information.

The return type corresponds to the return type of the `invokedynamic` factory; it is the target type of the lambda expression. Also, the argument list consists of the types of elements that are being captured by the lambda. In the case of a stateless lambda, the return type will always be empty. Only a Java closure will have arguments present.

A BSM takes at least three arguments and returns a `CallSite`. The standard arguments are of these types:

- `MethodHandles.Lookup`: A lookup object on the class in which the call site occurs
- `String`: The name mentioned in the `NameAndType`
- `MethodType`: The resolved type descriptor of the `NameAndType`

Following these arguments are any additional arguments that are needed by the BSM. These are referred to as *additional static arguments* in the documentation.

The general case of BSMs allows an extremely flexible mechanism, and non-Java language implementers use this. However, the Java language does not provide a language-level construct for producing arbitrary `invokedynamic` call sites.

For lambda expressions, the BSM takes a special form and to fully understand how the mechanism works, I will examine it more closely.

Decoding the lambda's bootstrap method

Use the `-v` argument to `javap` to see the bootstrap methods. This is necessary because the bootstrap methods live in a special part of the class file and make references back into the main constant pool. For this simple `Runnable` example, there is a single bootstrap method in that section:

```
BootstrapMethods:
  0: #28 REF_invokeStatic java/lang/invoke/La
        (Ljava/lang/invoke/MethodHandles$Look
         Ljava/lang/invoke/MethodType;Ljava/l
         Ljava/lang/invoke/MethodHandle;Ljava
    Method arguments:
      #29 ()V
```

```
        #30 REF_invokeStatic LambdaExample.lamb
        #29 ()V
```

That is a bit hard to read, so let's decode it.

The bootstrap method for this call site is entry #28 in the constant pool. This is an entry of type `MethodHandle` (a constant pool type that was added to the standard in Java 7). Now let's compare it to the case of the string function example:

```
0: #27 REF_invokeStatic java/lang/invoke/Lamb
         (Ljava/lang/invoke/MethodHandles$Look
          Ljava/lang/invoke/MethodType;Ljava/l
          Ljava/lang/invoke/MethodHandle;Ljava
    Method arguments:
      #28 (Ljava/lang/Object;)Ljava/lang/Obje
      #29 REF_invokeStatic StringFunction.lam
      #30 (Ljava/lang/String;)Ljava/lang/Inte
```

The method handle that will be used as the BSM is the same static method `LambdaMetafactory.metafactory( ... )`.

The part that has changed is the method arguments. These are the additional static arguments for lambda expressions, and there are three of them. They represent the lambda's signature and the method handle for the actual final invocation target of the lambda: the lambda body. The third static argument is the erased form of the signature.

Let's follow the code into `java.lang.invoke` and see how the platform uses *metafactories* to dynamically spin the classes that actually implement the target types for the lambda expressions.

The lambda metafactories

The BSM makes a call to this static method, which ultimately returns a call site object. When the `invokedynamic` instruction is executed, the method handle contained in the call site will return an instance of a class that implements the lambda's target type.

The source code for the metafactory method is relatively simple:

```
public static CallSite metafactory(MethodHand
                                   String
                                   Method
                                   Method
                                   Method
                                   Method
            throws LambdaConversionException
        AbstractValidatingLambdaMetafactory m
        mf = new InnerClassLambdaMetafactory(



        mf.validateMetafactoryArgs();
```

```
        return mf.buildCallSite();
}
```

The lookup object corresponds to the context where the `invokedynamic` instruction lives. In this case, that is the same class where the lambda was defined, so the lookup context will have the correct permissions to access the private method that the lambda body was compiled into.

The invoked name and type are provided by the VM and are implementation details. The final three parameters are the additional static arguments from the BSM.

In the current implementation, the metafactory delegates to code that uses an internal, shaded copy of the ASM bytecode libraries to spin up an inner class that implements the target type.

If the lambda does not capture any parameters from its enclosing scope, the resulting object is stateless, so the implementation optimizes by precomputing a single instance—effectively making the lambda's implementation class a singleton:

```
jshell> Function<String, Integer> makeFn() {
   ...>   return s -> s.length();
   ...> }
|  created method makeFn()

jshell> var f1 = makeFn();
f1 ==> $Lambda$27/0x0000000800b8f440@533ddba

jshell> var f2 = makeFn();
f2 ==> $Lambda$27/0x0000000800b8f440@533ddba

jshell> var f3 = makeFn();
f3 ==> $Lambda$27/0x0000000800b8f440@533ddba
```

This is one reason why the documentation strongly discourages Java programmers from relying upon any form of identity semantics for lambdas.

Conclusion

This article explored the fine-grained details of exactly how the JVM implements support for lambda expressions. This is one of the more complex platform features you'll encounter, because it is deep into language implementer territory.

Along the way, I've discussed `invokedynamic` and the method handles API. These are two key techniques that are major parts of the modern JVM platform. Both of these mechanisms are seeing increased use across the ecosystem; for example, `invokedynamic` has been used to implement a new form of string concatenation in Java 9 and above.

Understanding these features gives you key insight into the innermost workings of the platform and the modern frameworks upon which Java applications rely.

Dig deeper

- Java 8: Lambdas, Part 1
- Java 8: Lambdas, Part 2
- Real-world bytecode handling with ASM
- The ASM bytecode framework
- Loop unrolling
- The evolving nature of Java interfaces
- OpenJDK Project Lambda
- Chapter 6. The Java Virtual Machine instruction set

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.

Share this Page