BT

Java 8 Lambdas - A Peek Under the Hood

Java 8 was released in March 2014 and introduced lambda expressions as its flagship feature. You may already be using them in your code base to write more concise and flexible code. For example, you can combine lambda expressions with the new Streams API to express rich data processing queries:

```
int total = invoices.stream()
                    .filter(inv -> inv.getMonth() == Month.JULY)
                    .mapToInt(Invoice::getAmount)
                    .sum();
```

This example shows how to calculate the total amount due in July from a collection of invoices. Pass a lambda expression to find invoices whose month is July, and a method reference to extract the amount from the invoice.

You may be wondering how the Java compiler implements lambda expressions and method references behind the scenes and how the Java virtual machine (JVM) deals with them. For example, are lambda expressions simply syntactic sugar for anonymous inner classes? After all, the code above could be translated by copying the body of the lambda expression into the body of the appropriate method of an anonymous class (we discourage you from doing this!):

```
int total = invoices.stream()
                    .filter(new Predicate<Invoice>() {
                        @Override
                        public boolean test(Invoice inv) {
                            return inv.getMonth() == Month.JULY;
                        }
                    })
                    .mapToInt(new ToIntFunction<Invoice>() {
                        @Override
                        public int applyAsInt(Invoice inv) {
                            return inv.getAmount();
                        }
                    })
                    .sum();
```

This article will explain why the Java compiler doesn't follow this mechanism, and will shed light on how lambda expressions and method references are implemented. We will look at the bytecode generation and briefly analyze lambda performance in the lab. Finally, we will discuss the performance implications in the real world.

Why are anonymous inner classes unsatisfactory?

Anonymous inner classes have undesirable characteristics that can impact the performance of your application.

First, the compiler generates a new class file for each anonymous inner class. The filename usually looks like ClassName$1, where ClassName is the name of the class in which the anonymous inner class is defined, followed by a dollar sign and a number. The generation of many class files is undesirable because each class file needs to be loaded and verified before being used, which impacts the startup performance of the application. The loading may be an expensive operation, including disk I/O and decompressing the JAR file itself.

If lambdas were translated to anonymous inner classes, you'd have one new class file for each lambda. As each anonymous inner class would be loaded it would take up room in the JVM's meta-space (which is the Java 8 replacement for the Permanent Generation). If the code inside each such anonymous inner class is compiled to machine code by the JVM, it would be stored inside a code cache. In addition, these anonymous inner classes would be instantiated into separate objects. As a consequence, anonymous inner classes would increase the memory consumption of your application. It is potentially helpful to introduce a caching mechanism in order to reduce all of this memory overhead, which motivates the introduction of some kind of abstraction layer.

Most importantly, choosing to implement lambdas using anonymous inner class from day one would have limited the scope of future lambda implementation changes, as well as the ability for them to evolve in line with future JVM improvements.

Let's take a look at the following code:

```
import java.util.function.Function;
public class AnonymousClassExample {
    Function<String, String> format = new Function<String, String>() {
        public String apply(String input){
            return Character.toUpperCase(input.charAt(0)) + input.substring(1);
        }
    };
}
```

You can examine the bytecode generated for any class file using the command

```
javap -c -v ClassName
```

The corresponding generated bytecode for the Function created as an anonymous inner class will be something along the lines of this:

```
0: aload_0
1: invokespecial #1 // Method java/lang/Object."<init>":()V
4: aload_0
5: new           #2 // class AnonymousClassExample$1
8: dup
9: aload_0
10: invokespecial #3 // Method AnonymousClass$1."<init>":(LAnonymousClassExample;)V
13: putfield      #4 // Field format:Ljava/util/function/Function;
16: return
```

This code shows the following:

> 5: An object of type AnonymousClassExample$1 is instantiated using the byte code operation new. A reference to the newly created object is pushed on the stack at the same time.
> 8: The operation dup duplicates that reference on the stack.
> 10: This value is then consumed by the instruction invokespecial, which initializes the anonymous inner class instance.
> 13: The top of the stack now still contains a reference to the object, which is stored in the format field of the AnonymousClassExample class using the putfield instruction.

AnonymousClassExample$1 is the name generated by the compiler for the anonymous inner class. If you want to reassure yourself, you can inspect the AnonymousClassExample$1 class file as well, and you'll find the code for the implementation of the Function interface.

Translating lambda expressions to anonymous inner classes would limit possible future optimisations (e.g. caching) as they would be tied to the anonymous inner class bytecode generation mechanism. As a consequence, the language and JVM engineers needed a stable binary representation that provided enough information while allowing alternative possible implementation strategies by the JVM in the future. The next section explains just how this is possible!

Lambdas and invokedynamic

To address the concerns explained in the previous section, the Java language and JVM engineers decided to defer the selection of a translation strategy until run time. The new invokedynamic bytecode instruction introduced with Java 7 gave them a mechanism to achieve this in an efficient way. The translation of a lambda expression to bytecode is performed in two steps:

generate an invokedynamic call site (called *lambda factory*), which when invoked returns an instance of the **Functional Interface** to which the lambda is being converted;
convert the body of the lambda expression into a method that will be invoked through the invokedynamic instruction.

To illustrate the first step let's inspect the bytecode generated when compiling a simple class containing a lambda expression such as:

```
import java.util.function.Function;

public class Lambda {
    Function<String, Integer> f = s -> Integer.parseInt(s);
}
```

This will translate to the following bytecode:

```
 0: aload_0
 1: invokespecial #1 // Method java/lang/Object."<init>":()V
 4: aload_0
 5: invokedynamic #2, 0 // InvokeDynamic
                  #0:apply:()Ljava/util/function/Function;
10: putfield #3 // Field f:Ljava/util/function/Function;
13: return
```

Note that method references are compiled slightly differently because javac does not need to generate a synthetic method and can refer to the method directly.

How the second step is performed depends on whether the lambda expression is *non-capturing* (the lambda doesn't access any variables defined outside its body) or *capturing (*the lambda accesses variables defined outside its body).

Non-capturing lambdas are simply desugared into a static method having exactly the same signature of the lambda expression and declared inside the same class where the lambda expression is used. For instance the lambda expression declared in the Lambda class above can be desugared into a method like this:

```
static Integer lambda$1(String s) {
    return Integer.parseInt(s);
}
```

Note: $1 is not an inner class, it is just our way of representing compiler generated code

The case of a capturing lambda expression is a bit more complex because the captured variables have to be passed to the method implementing the body of the lambda expression together with the formal arguments of the lambda. In this case the common translation strategy is to prepend the arguments of the lambda expression with an additional argument for each of the captured variables. Let's look at a practical example:

```
int offset = 100;
Function<String, Integer> f = s -> Integer.parseInt(s) + offset;
```

the corresponding method implementation could be generated asy:

```
static Integer lambda$1(int offset, String s) {
    return Integer.parseInt(s) + offset;
}
```

However this translation strategy is not set in stone because the use of the invokedynamic instruction gives the compiler the flexibility to choose different implementation strategies in the future. For instance, the captured values could be boxed in an array or, if the lambda expression reads some fields of the class where it is used, the generated method could be an instance one, instead of being declared static, thus avoiding the need to pass those fields as additional arguments.

Performance in the Lab

The main advantage of this approach is the performance characteristics. It would be lovely to just think of these as being reducible to a single number, but there are actually multiple operations involved here.

The first part is the linkage step, which corresponds to the lambda factory step mentioned above. If we were comparing the performance to anonymous inner classes, then the equivalent operation would be the class loading of the anonymous inner class. Oracle have published a **performance analysis** onto this tradeoff by Sergey Kuksenko and you can see Kuksenko **deliver a talk on the topic** at the 2013 JVM Language Summit**[3]**. The analysis shows that it takes time to warm up the lambda factory approach, during which it is initially slower. Performance comes into line with class loading when there are enough call sites linked, if the code is on a hot path (i.e., one called frequently enough to get JIT compiled). On the other hand if it's a cold path the lambda factory approach can be up to 100x faster.

The second step is capturing the variables from the surrounding scope. As we've already mentioned, if there are no variables to capture then this step can be optimised automatically to avoid allocating a new object with the lambda factory based implementation. In the anonymous inner class approach we would be instantiating a new object. In order to optimize the equivalent case you would have to manually optimise the code by creating a single object and hoisting it into a static field. For example:

```
// Hoisted Function
public static final Function<String, Integer> parseInt = new Function<String, Integer>() {
    public Integer apply(String arg) {
        return Integer.parseInt(arg);
    }
};

// Usage:
int result = parseInt.apply("123");
```

The third step is calling the actual method. At the moment both anonymous inner classes and lambda expressions perform the exact same operation so there is no difference in performance here. The out of the box performance for non-capturing lambda expressions is already ahead of the hoisted anonymous inner class equivalent. The implementation of capturing lambda expressions is a similar ballpark to the performance of allocating an anonymous inner class in order to capture these fields.

What we've seen in this section is that broadly the implementation of lambda expressions performs well. Whilst anonymous inner classes need manual optimisation to avoid allocation the most common case of this (a lambda expression that doesn't capture its arguments) is already being optimized for us by the JVM.

Performance in the Field

Of course its all well and good to understand the overall performance model, but how do things stack up in practice? We've been using Java 8 in a few software projects with generally positive results. The automatic optimisation of non-capturing lambdas can provide a nice benefit. There's one particular example identified which raises some interesting questions about future optimisation directions.

The example in question occurred whilst working on some code for use in a system which required particularly low GC pauses, ideally none. It was thus desirable to avoid allocating too many objects. The project made extensive use of lambdas to implement callback handlers. Unfortunately we still have quite a few callbacks where we capture no local variables, but want to refer to a field of the current class or even just call a method on the current class. Currently this still seems to require allocation. Here's a code example just to clarify what we are talking about:

```
public MessageProcessor() {}

public int processMessages() {
    return queue.read(obj -> {
        if (obj instanceof NewClient) {
            this.processNewClient((NewClient) obj);
        }
        ...
    });
}
```

There is a simple solution to this problem. We hoist up the code into the constructor and assigned it to a field, we then refer directly to the field at the callsite. Here is our previous code example rewritten:

```
private final Consumer<Msg> handler;

public MessageProcessor() {
    handler = obj -> {
        if (obj instanceof NewClient) {
            this.processNewClient((NewClient) obj);
        }
        ...
    };
}

public int processMessages() {
    return queue.read(handler);
}
```

In the project in question this was a serious issue: memory profiling revealed that this pattern was responsible for six of the top eight sites of object allocation and well over 60% of the application’s allocation in total.

As with any potential optimisation applying this approach regardless of context is likely to introduce other problems.

> You are choosing to write non-idiomatic code for purely performance reasons. So there is a readability tradeoff
> There is an allocation tradeoff at stake as well. You're adding a field to MessageProcessor, making it larger to allocate. The creation and capture of the lambda in question also slows down the constructor call to MessageProcessor.

We came across this situation not by looking for the scenario but through memory profiling and had a good business use case which justified the optimisation. We were also in a position where we had objects being allocated once, that reused lambda expressions a lot so the caching was extremely beneficial. As with any performance tuning exercise scientific method is the generally recommended approach.

This is the approach that any other end users seeking to optimise their use of lambda expressions should take as well. Trying to write clean, simple and functional code is always the best first step. Any optimisation, such as this hoisting, should only be done in response to a genuine problem. Writing lambda expressions that capture allocating objects isn't inherently bad - in the same way that writing Java code that calls `new Foo()` isn't inherently bad.

This experience does also suggest that to get the best out of lambda expressions it is important to use them idiomatically. If lambda expressions are used to represent small, pure functions there is little need for them to capture anything from their surrounding scope. As with most things - if you keep it simple things perform well.

Conclusions

In this article we've explained that lambdas aren't just anonymous inner classes under the hood, and why anonymous inner classes would not have been an appropriate implementation approach for lambda expressions. A lot of work has already gone into thinking through the lambda expressions implementation approach. At the moment they are faster than anonymous inner classes for most tasks, but the current state of affairs isn't perfect; there is still some scope for measurement-driven hand optimisation.

The approach used in Java 8 isn't just limited to Java itself though. Scala has historically implemented its lambda expressions by generating anonymous inner classes. In Scala 2.12 though the move has been made to start using the lambda metafactory mechanism introduced in Java 8. Over time its possible that other languages on the JVM may also adopt this mechanism.

About the Authors

Richard Warburton is an empirical technologist and solver of deep-dive technical problems. Recently he has written a book on **Java 8 Lambdas** for O'Reilly and teaches functional code to Java developers with **java8training.com**. He's worked as a developer in many areas including Statistical Analytics, Static Analysis, Compilers and Network Protocols. He is a leader in the London Java Community and runs Openjdk Hackdays. Richard is also a known conference speaker, having talked at Devoxx, JavaOne, JFokus, Devoxx UK, Geecon, Oredev, JAX London and Codemotion. He has obtained a PhD in Computer Science from The University of Warwick.

Raoul-Gabriel Urma is a PhD candidate in Computer Science at the University of Cambridge. He is co-author of the book **Java 8 in Action: Lambdas, streams, and functional-style programming** published by Manning. In addition, Raoul has written over 10 peer-reviewed articles and delivered over 20 technical talks at international conferences. He has worked for large companies such as Google, eBay, Oracle, and Goldman Sachs, as well as for several startup projects. Raoul is also a Fellow of the Royal Society of Arts.
Twitter: @raoulUK

Mario Fusco is a senior software engineer at Red Hat working at the development of the core of Drools, the JBoss rules engine. He has extensive experience as a Java developer, having been involved in (and often leading) many enterprise level projects in industries ranging from media companies to the financial sector. Among his interests are functional programming and domain specific languages. By leveraging these two passions, he created the open source library lambdaj with the purposes of providing an internal Java DSL for manipulating collections and for allowing a bit of functional programming in Java. Twitter: @mariofusco.

Related Editorial

Oracle Releases GraalVM 1.0, a Polyglot Virtual Machine and Platform

Brian Goetz Speaks to InfoQ on Data Classes for Java

Headless Selenium Browsers

Super Charge the Module Aware Service Loader in Java 11

Micronaut for Spring Allows Spring Boot Apps to Run as Micronaut Apps

BT