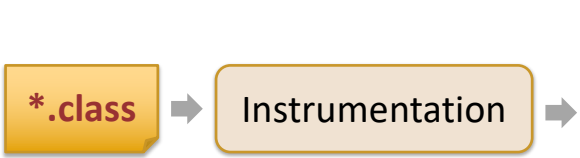
Joker<?>

Lambda compilation and other gimmiks of modern JDKs

Марк Хоффман
mtrail GmbH
@marcandsweep

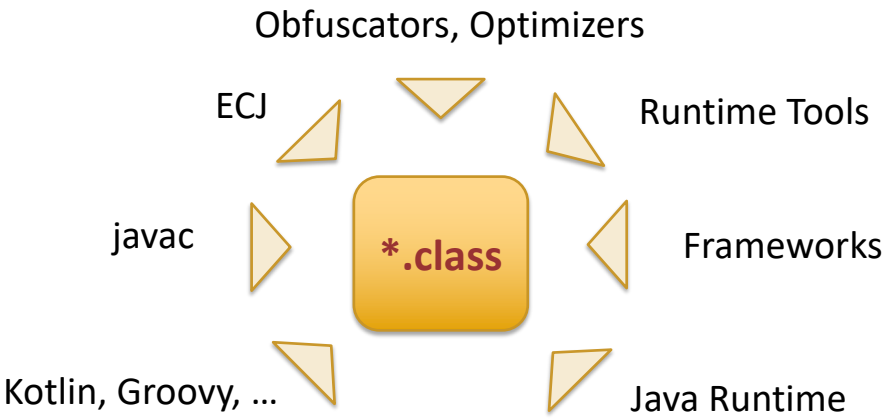
Why do I talk about JDK internals?

- Disclaimer: I‘m not a JDK developer!
- I‘m a JDK explorer
- Working on JaCoCo, a bytecode based code coverage tool




```
120.    public static IRuntime createFor(final Instrumentation
121.            final String className, final String accessFiel
122.            throws ClassNotFoundException {
123.        final ClassFileTransformer transformer = new ClassF
124.            public byte[] transform(final ClassLoader loade
125.                    final String name, final Class<?> class
126.                    final ProtectionDomain protectionDomain
127.                    throws IllegalClassFormatException {
128.                if (name.equals(className)) {
129.                    return instrument(source, accessFieldNa
130.                }
131.                return null;
132.            }
133.        };
134.        inst.addTransformer(transformer);
135.        final Class<?> clazz = Class.forName(className.repl
136.        inst.removeTransformer(transformer);
137.        try {
138.            clazz.getField(accessFieldName);
139.        } catch (final NoSuchFieldException e) {
140.            throw new RuntimeException(format(
141.                    "Class %s could not be instrumented.",
142.        }
143.        return new ModifiedSystemClassRuntime(clazz, access
144.    }
```

Java Classfile Creation



Some Bytecode Warm-up

```
static int add(int a, int b) {
    return a + b;
}
```

```
$> javap -v ByteCodeExamples


static int add(int, int);
    iload_0
    iload_1
    iadd
    ireturn
```

Method Invocations

```
static void execute(Runnable job) {
    job.run();
}
```

```
$> javap –v ByteCodeExamples

static void execute(java.lang.Runnable);
    aload_0
    invokeinterface java/lang/Runnable.run:()V
    return
```

Implicit Runtime API Dependencies

Java 8

```
static String conc(String a, String b) {
    return a + b;
}
```

```
$> javap –v ByteCodeExamples

static java.lang.String conc(java.lang.String, java.lang.String);
    new java/lang/StringBuilder
    dup
    aload_0
    invokestatic  java/lang/String.valueOf:(Ljava/lang/Object;)Ljava/lang/String;
    invokespecial java/lang/StringBuilder."<init>":(Ljava/lang/String;)V
    aload_1
    invokevirtual java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang…
    invokevirtual  java/lang/StringBuilder.toString:()Ljava/lang/String;
    areturn
```

Homework

Identify all runtime API dependencies that javac may create.

@marcandsweep

Lambda Types

```
List<String> greetings = Arrays.asList("Hello", "Lambda");

greetings.forEach((item) -> System.out.println(item));

greetings.forEach(System.out::println);

Consumer<String> printer = System.out::println;
greetings.forEach(printer);
```

Functional Interfaces

- SAM = Single Abstract Method
- Java uses strict types for lambdas
- No generic method type (like in other languages)

Lambda vs. Inner Classes

```
List<String> greetings = Arrays.asList("Hello", "Lambda");

greetings.forEach(new Consumer<String>() {
   @Override
   public void accept(String item) {
      System.out.println(item);
   }
});
```

- New instance on every usage
- Implicit capture of enclosing instance

Desugaring

```
greetings.forEach((item) -> System.out.println(item));
```

```
$> javap -v LambdaExample

private static synthetic void lambda$0(java.lang.String item);
    getstatic java/lang/System.out:Ljava/io/PrintStream;
    aload_0
    invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
    return
```

Lambda Instance Creation

```
greetings.forEach((item) -> System.out.println(item));
```

```
$> javap –v LambdaExample


aload_1
invokedynamic InvokeDynamic #0:accept()Ljava/util/function/Consumer;
invokeinterface java/util/List.forEach(Ljava/util/function/Consumer;)V
```

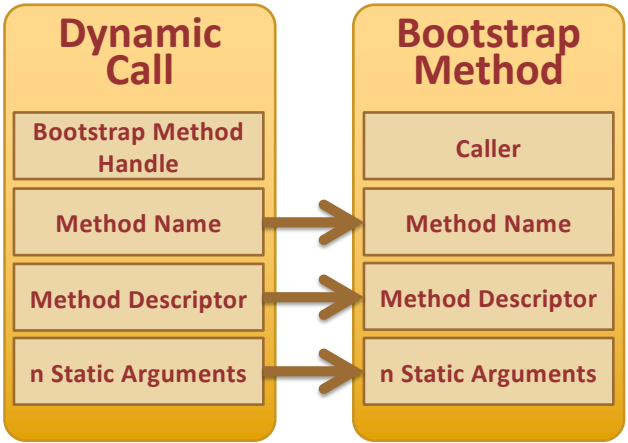
Method Invocations in Bytecode

```
invokevirtual java/io/PrintStream.println(C)V
invokeinterface java/lang/Runnable.run()V
invokestatic java/lang/System.exit(I)V
invokespecial java/lang/Object.<init>()V
```

- All statically typed
- `invokedynamic` added with Java 7


CallSite
Target Method
Bootstrap Method
invokedynamic

Bootstrap Method Parameters



java.lang.invoke.LambdaMetafactory

- Bootstrap methods for lambda instance creation
- Part of the JDK API
- Why „Meta“?
  Factory for methods which create the lambda instances

java.lang.invoke.LambdaMetafactory

```
public static CallSite metafactory(
    MethodHandles.Lookup caller,
    String invokedName,
    MethodType invokedType,
    MethodType samMethodType,
    MethodHandle implMethod,
    MethodType instantiatedMethodType) {

 …

}
```

Fixed bootstrap method parameters

Lambda specific parameters

java.lang.invoke.LambdaMetafactory

```
public static CallSite altMetafactory(
    MethodHandles.Lookup caller,
    String invokedName,
    MethodType invokedType,
    Object... args) { … }
```

Fixed bootstrap method parameters

n Lambda specific parameters

- Serializable lambdas
- Additional marker interfaces
- Bridge methods

Don‘t Ever Do This in Your Production Code!
(At least don‘t blame me)

```
// Parameters to the lambda factory:
Lookup caller = MethodHandles.lookup();
String invokedName = "accept";
MethodType invokedType = MethodType.methodType(Consumer.class);
MethodType samMethodType = MethodType.methodType(Void.TYPE, Object.class);
MethodHandle implMethod = caller.findStatic(
       LambdaMetaFactoryTest.class, "myLambda", samMethodType);

// Create the factory:
CallSite factory = LambdaMetafactory.metafactory(
        caller, invokedName, invokedType, samMethodType, implMethod,
samMethodType);

// Create the actual lambda instance:
Consumer<String> printer = (Consumer<String>) factory.dynamicInvoker().invoke();

// Use the lambda instance:
List<String> greetings = Arrays.asList("Hello", "Lambda");
greetings.forEach(printer);
```

Lambda Creation Process

Lambda Implementation?

```
List<String> l = Arrays.asList("Hi");

l.forEach((item) -> { throw new RuntimeException("Here we are!"); });
```

```
$> java LambdaCallStack

Exception in thread "main" java.lang.RuntimeException: Here we are!
     at LambdaCallStack.lambda$0(LambdaCallStack.java:16)
     at java.base/java.util.Arrays$ArrayList.forEach(Arrays.java:4389)
     at LambdaCallStack.main(LambdaCallStack.java:16)
```

Lambda Implementation?

```
List<String> l = Arrays.asList("Hi");

l.forEach((item) -> { throw new RuntimeException("Here we are!"); });
```

```
$> java -XX:+UnlockDiagnosticVMOptions -XX:+ShowHiddenFrames LambdaCallStack

Exception in thread "main" java.lang.RuntimeException: Here we are!
     at LambdaCallStack.lambda$0(LambdaCallStack.java:16)
     at LambdaCallStack$$Lambda$1/142638629.accept(<Unknown>:1000004)
     at java.base/java.util.Arrays$ArrayList.forEach(Arrays.java:4389)
     at LambdaCallStack.main(LambdaCallStack.java:16)
```

Lambda Implementation Classes I

- Bytecode created on-the-fly
  `jdk.internal.misc.Unsafe.defineAnonymousClass()`
- Can be dumped with
  `-Djdk.internal.lambda.dumpProxyClasses=<path>`

Lambda Implementation Classes II

- Members
  - SAM implementation
  - Bridge methods
  - Constructor to retrieve captures
  - Members to store captures
  - Static factory method

Inspecting Lambda Implementation Classes

```
Class<?> c = instance.getClass();
show("toString()",      instance);
show("Class Name",      c.getName());
show("Class Modifier“, Modifier.toString(c.getModifiers()));
show("Super Class",     c.getSuperclass().getName());
show("Interfaces",      c.getInterfaces());
show("Class Loader",    c.getClassLoader());
show(“Field",           c.getDeclaredFields());
show(“Constructor",     c.getDeclaredConstructors());
show(“Method",          c.getDeclaredMethods());
```

Inspecting: Simple Lambda

```
Runnable r = () -> System.out.println("Lambda");
```

```
toString()     : LambdaInstanceInspection$$Lambda$1/821270929@2f2c9b19
Class Name     : LambdaInstanceInspection$$Lambda$1/821270929
Class Modifier : final
Super Class    : java.lang.Object
Interface      : interface java.lang.Runnable
Class Loader   : sun.misc.Launcher$AppClassLoader@1540e19d
Constructor    : private .../821270929()
Method         : public void .../821270929.run()
```

Inspecting: Lambda with Capture

```
String capturedValue = "Lambda";
Runnable r = () -> System.out.println(capturedValue);
```

```
Interface      : interface java.lang.Runnable
Field          : private final java.lang.String .../303563356.arg$1
Constructor    : private .../303563356(java.lang.String)
Method         : public void .../303563356.run()
Method         : private static java.lang.Runnable
                           .../303563356.get$Lambda(java.lang.String)
```

Inspecting: Serializable Lambda

```
Runnable r = (Runnable & Serializable) () -> System.out.println();
```

```
Interface      : interface java.lang.Runnable
Interface      : interface java.io.Serializable
Class Loader   : sun.misc.Launcher$AppClassLoader@1540e19d
Constructor    : private .../250421012()
Method         : public void .../250421012.run()
Method         : private final java.lang.Object
                               .../250421012.writeReplace()
```

java.lang.invoke.SerializedLambda

```
public final class SerializedLambda implements Serializable {

    private final Class<?> capturingClass;
    private final String functionalInterfaceClass;
    private final String functionalInterfaceMethodName;
    private final String functionalInterfaceMethodSignature;
    private final String implClass;
    private final String implMethodName;
    private final String implMethodSignature;
    private final int implMethodKind;
    private final String instantiatedMethodType;
    private final Object[] capturedArgs;

    ...
}
```

Inspecting: Bridge Methods

```
static interface StringSupplier extends Supplier<String> {
   @Override
   String get();
}

StringSupplier s = () -> "Lambda";
```

```
Constructor    : private .../1096979270()
Method         : public java.lang.String .../1096979270.get()
Method         : public java.lang.Object .../1096979270.get()
```

Lambda Instance Re-Use

```
Runnable createLambda() {
   return LambdaInstancesTest::staticMethod;
}

@Test
public void same_call_site() {
   Runnable r1 = createLambda();
   Runnable r2 = createLambda();
   assertSame(r1, r2);
   assertSame(r1.getClass(), r2.getClass());
}
```

One Implementation per Call Site

```
@Test
public void different_call_site() {
   Runnable r1 = LambdaInstancesTest::staticMethod;
   Runnable r2 = LambdaInstancesTest::staticMethod;
   assertNotSame(r1, r2);
   assertNotSame(r1.getClass(), r2.getClass());
}
```

Capturing Lambdas

```
Runnable createLambdaWithCapture() {
   return System.out::println;
}

@Test
public void same_call_site_with_capture() {
   Runnable r1 = createLambdaWithCapture();
   Runnable r2 = createLambdaWithCapture();
   assertNotSame(r1, r2);
   assertSame(r1.getClass(), r2.getClass());
}
```

Bonus Question:

To Capture or Not to Capture?

```
Runnable createLambdaWithCaptureOrNot() {
   return () -> System.out.println();
}

@Test
public void same_call_site_with_capture_or_not() {
   Runnable r1 = createLambdaWithCaptureOrNot();
   Runnable r2 = createLambdaWithCaptureOrNot();
   assertSame(r1, r2);
   assertSame(r1.getClass(), r2.getClass());
}
```

More InvokeDynamic Gimmiks

Java 9

```
static String directions(String title, String room) {
    return "Talk " + title + " is in " + room + ".";
}
```

```
$> javap –v ByteCodeExamples

static java.lang.String directions(java.lang.String, java.lang.String);
    aload_0
    aload_1
    invokedynamic makeConcatWithConstants:(L…String;L…String;)L…String;
    areturn
```

More InvokeDynamic Gimmiks

Java 9

```
static String directions(String title, String room) {
    return "Talk " + title + " is in " + room + ".";
}
```

```
invokedynamic makeConcatWithConstants:(L…String;L…String;)L…String;

1: #29 REF_invokeStatic
   java/lang/invoke/StringConcatFactory.makeConcatWithConstants:(
       Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;
       Ljava/lang/invoke/MethodType;Ljava/lang/String;
       [Ljava/lang/Object;)Ljava/lang/invoke/CallSite;
   Method arguments:
      #32 Talk \u0001 is in \u0001.
```

java.lang.invoke.StringConcatFactory

```
public static CallSite makeConcat(
    MethodHandles.Lookup lookup,
    String name,
    MethodType concatType
) throws StringConcatException {

…

}
```

Fixed bootstrap method parameters

java.lang.invoke.StringConcatFactory

```
public static CallSite makeConcatWithConstants(
    MethodHandles.Lookup lookup,
    String name,
    MethodType concatType,
    String recipe,
    Object... constants
) throws StringConcatException {

…

}
```

Fixed bootstrap method parameters

Concat configuration parameters

Multiple Strategies

Configurable with system property **`java.lang.invoke.stringConcat`**

- BC_SB
- BC_SB_SIZED
- BC_SB_SIZED_EXACT

Bytecode Generation

- MH_SB_SIZED
- MH_SB_SIZED_EXACT
- MH_INLINE_SIZED_EXACT

Method Handle Chaining

Bytecode Generation

- Can be dumped with
  `-Djava.lang.invoke.stringConcat.dumpClasses=<path>`

tl;dl

- Syntactical sugar compiles to funny bytecode
- Compilation might create runtime API dependencies
- Instead of optimizing at compile time InvokeDynamic defers possible optimizations to the runtime

References

https://www.eclipsecon.org/na2015/session/lambda-mechanics

- Code Examples: https://github.com/marchof/lambda-mechanics
- JLS8: Functional Interfaces, Chapter 9.8
- JLS8: Compile-Time Step 3: Is the Chosen Method Appropriate? Chapter 15.12.3
- JLS8: Runtime-Evaluation of Lambda Expressions, Chapter 15.27.4
- The Java Virtual Machine Specification, Java SE 8 Edition
- Java 8 API: LambdaMetafactory
- Brian Götz: Lambda - A Peek Under The Hood
- Anton Arihpov: Java 8 Revealed
- Aleksey Shipilev: The Lord of the Strings

Questions?

Please find me at the discussion zone #3.

Марк Хоффман
mtrail GmbH
@marcandsweep