JAVA AGENTS AND BYTECODE

eMag Issue 42 - May 2016

InfoQueue

ARTICLE
Java Bytecode: Bending the Rules

ARTICLE
Secrets of the Bytecode Ninjas

ARTICLE
Java's secret weapon: invokedynamic

Living In The Matrix With Bytecode Manipulation

Ashley Puls examines three common byte code manipulation frameworks: ASM, CGLib, and Javassist (Java Programming Assistant), showing how these tools work and why frameworks like Spring use them.

Easily Create Java Agents with Byte Buddy

In this article Rafael Winterhalter, creator of the bytecode manipulation tool Byte Buddy, provides detailed guidance on how to easily create Java agents using Byte Buddy.

Java Bytecode: Bending the Rules

Few developers ever work with Java bytecode directly, but bytecode format is not difficult to understand. In this article Rafael Winterhalter takes us on a tour of Java bytecode & some of its capabilities

Five Advanced Debugging Techniques Every Java Developer Should Know

With architectures becoming more distributed and code more asynchronous, pinpointing and resolving errors in production is harder than ever. In this article we investigate five advanced techniques that can help you get to the root cause of painful bugs in production more quickly, without adding material overhead.

Java's secret weapon: `invokedynamic`

invokedynamic was the first new Java bytecode since Java 1.0 and was crucial in implementing the "headline" features of Java 8 (such as lambdas and default methods). In this article, we take a deep dive into invokedynamic and explain why it is such a powerful tool for the Java platform and for JVM languages such as JRuby and Nashorn.

Secrets of the Bytecode Ninjas

Java is defined by the Java Language Spec, but the resulting bytecode is defined by a completely separate standard. This article looks at the structure of class files and how to create them directly.

FOLLOW US

facebook.com /InfoQ
@InfoQ
google.com /+InfoQ
linkedin.com company/infoq

CONTACT US

GENERAL FEEDBACK feedback@infoq.com
ADVERTISING sales@infoq.com
EDITORIAL editors@infoq.com

VICTOR GRAZI is the Java queue lead at InfoQ. Inducted as an Oracle Java Champion in 2012, Victor works at Nomura Securities on building core tools, and as a technical consultant and Java evangelist. He is also a frequent presenter at technical conferences. Victor hosts the "Java Concurrent Animated" open source project on SourceForge.

A LETTER FROM THE EDITOR

Java bytecode programming is not for the faint of heart, but in a world where new JVM languages, fancy profilers, and proxying frameworks are prevalent, it can be a powerful tool not just for reengineering existing code, but for creating clean, reusable, and reduced coupling architectures.

In this eMag we have curated articles on bytecode manipulation, including how to manipulate bytecode using three important frameworks: Javassist, ASM, and ByteBuddy, as well as several higher level use cases where developers will benefit from understanding bytecode.

We start with "Living in the Matrix with Bytecode Manipulation", where we capture in detail two popular bytecode manipulation frameworks - Javassist and ASM, following a presentation by New Relic's Ashley Puls.

Next, Byte Buddy creator Rafael Winterhalter gives us a detailed recipe for easily creating Java agents using Byte Buddy in "Easily Create Java Agents with Byte Buddy".

Takipi's Tal Weiss is next up, in an accessible tutorial on valuable production debugging techniques in "5 Advanced Java Debugging Techniques Every Developer Should Know About". With architectures becoming more distributed and code more asynchronous, pinpointing and resolving errors in production is a difficult challenge. Weiss shows us how to do this quickly and painlessly without a lot of overhead.

Then Rafael Winterhalter returns with a gentler touch in "Java Bytecode: Bending the Rules", providing some standard Java coding anomalies that can be affected by knowledge of bytecode - for example, throwing checked exceptions without declaration or changing final fields.

In "Secrets of the Bytecode Ninjas", Ben Evans uses ASM to give us a preliminary exposure to the structure of a class file and what bytecode actually looks like, and shows us how to manipulate it through examples.

Finally, Ben Evans tells us how to use Java bytecode's newest instruction in "invokedynamic - Java's Secret Weapon". invokedynamic was the first new Java bytecode since Java 1.0, and was crucial in implementing the "headline" features of Java 8 (such as lambdas and default methods). It is also a must-have tool for the Java platform and for JVM languages such as JRuby and Nashorn.

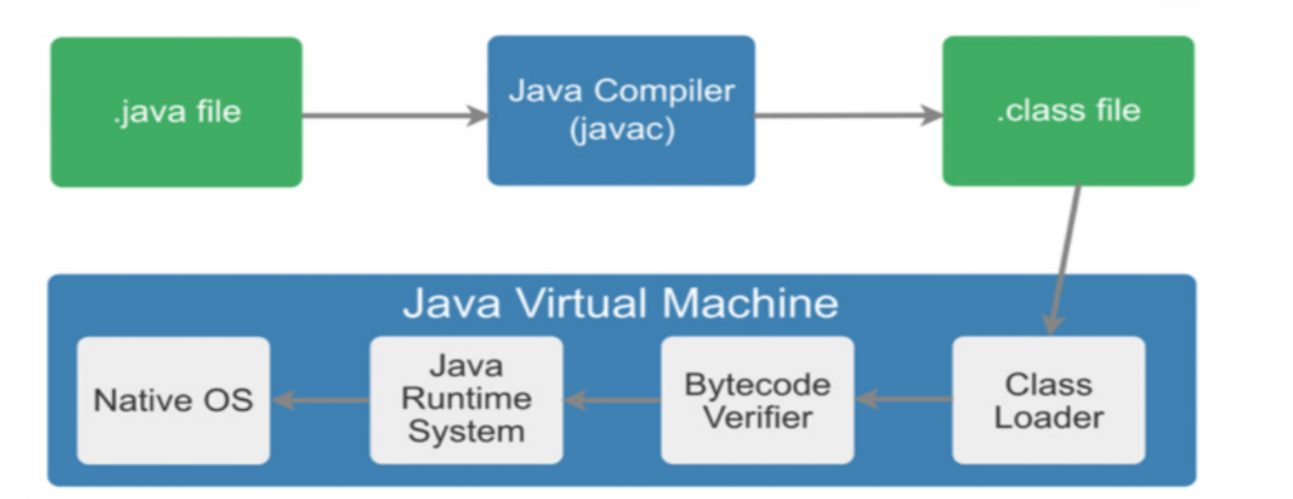
Watch on InfoQ

Living In The Matrix With Bytecode Manipulation

Presentation summary by Victor Grazi

You are probably all too familiar with the following sequence: You input a .java file into a Java compiler, (likely using javac or a build tool like ANT, Maven or Gradle), the compiler grinds away, and finally emits one or more .class files.

What is Java Bytecode?



Source: http://www.techlila.com/write-programs-linux/

Figure 1: What is Java bytecode?

If you run the build from the command line with verbose enabled, you can see the output as it parses your file until finally it prints your .class file.

```
001 javac -verbose src/com/example/spring2gx/BankTransactions.java
```

The generated .class file contains the bytecode, essentially the instruction set for the Java virtual machine (JVM), and is what gets loaded by the Java runtime class loader when a program executes.

This article will investigate Java bytecode and how to manipulate it, and we will see why anyone would ever want to do so. .

Bytecode Manipulation Frameworks

The popular frameworks for manipulating bytecode have been:

- ASM,
- AspectJ,
- BCEL,
- Byte Buddy,
- CGLIB,
- Cojen,
- Javassist, and
- Serp.

This article focuses on Javassist and ASM.

Why should you care about manipulating bytecode?

Many common Java libraries such as Spring and Hibernate, as well as most JVM languages and even your IDEs, use bytecode-manipulation frameworks. For that reason, and because it's really quite fun, you might find bytecode manipulation a valuable skillset to have. You can use bytecode manipulation to perform many tasks that would be difficult or impossible to do without it, and once you learn it, the sky's the limit.

One important use case is program analysis. For example, the popular FindBugs bug-locator tool uses ASM under the hood to analyze your bytecode and locate bug patterns. Some software shops have code-complexity rules such as a maximum number of if/else statements in a method or a maximum method size. Static analysis tools analyze your bytecode to determine the code complexity.

Another common use is class generation. For example, ORM frameworks typically use proxies based on your class definitions. Or consider security applications that provide syntax for adding authorization annotations.

JVM languages such as Scala, Groovy, and Grails all use a bytecode-manipulation framework.

Consider a situation where you need to transform library classes without having the source code, a task routinely performed by Java profilers. For example, at New Relic, bytecode instrumentation is used to time method executions.

With bytecode manipulation, you can optimize or obfuscate your code, or you can introduce functionality such as adding strategic logging to an application. This article will focus on a logging example, which will provide the basic tools for using these bytecode manipulation frameworks.

Our example

Sue is in charge of ATM coding for a bank. She has a new requirement: add key data to the logs for some designated important actions.

Here is a simplified bank-transactions class. It allows a user to log in with a username and password, does some processing, withdraws a sum of money, and then prints out "transactions completed." The important actions are the login and withdrawal.

```
001 public void login(String password,
      String accountId, String userName)
      {
002     // login logic
003 }
004 public void withdraw(String
      accountId, Double moneyToRemove) {
005     // transaction logic
006 }
```

To simplify the coding, Sue would like to create an @ImportantLog annotation for those method calls, containing input parameters that represent the indexes of the method arguments she wants to record. With that, she can annotate her login and withdraw methods.

```
001 /**
002  * A method annotation which should be used to
     indicate
003  * important methods whose invocations should be
     logged.
004  */
005  public @interface ImportantLog {
006    /**
007     * The method parameter indexes whose values
     should be logged.
008     * For example,if we have the method
009     * hello(int paramA, int paramB, int paramC), and
     we
010     * wanted to log the values of paramA and paramC,
     then fields
011     * would be [“0”,”2”]. If we only want to log the
     value of
012     * paramB, then fields would be [“1”].
013     */
014    String[] fields();
015  }
```

For login, Sue wants to record the account ID and the username so her fields will be set to “1” and “2”, (she doesn’t want to display the password!) For the withdraw method, her fields are “0” and “1” because she wants to output the first two fields: account ID and the amount of money to remove. Her audit log ideally will contain something like this:

```
@ImportantLog(fields = { "1", "2" })
public void login(String password, String accountId, String userName) {
    // login logic
}
A call was made to "login" on "com/example/spring2gx/BankTransactions"
    Important params:
        Index 1 value: ${accountId}
        Index 2 value: ${userName}
@ImportantLog(fields = { "0", "1" })
public void withdraw(String accountId, Double moneyToRemove) {
    // transaction logic
}
A call was made to "withdraw" on "com/example/spring2gx/BankTransactions"
    Important params:
        Index 0 value: ${accountId}
        Index 1 value: ${moneyToAdd}
```

To hook this up, Sue is going to use a Java agent. Introduced in JDK 1.5, Java agents allow you to modify the bytes that comprise the classes in a running JVM, without requiring any source code.

Without an agent, the normal execution flow of Sue’s program is:

1. Run Java on a main class, which is then loaded by a class loader.
2. Call the class’s main method, which executes the defined process.
3. Print “transactions completed.”

When you introduce a Java agent, a few more things happen — but let’s first see what’s required to create an agent. An agent must contain a class with a premain method. It must be packaged as a JAR file with a properly constructed manifest that contains a Premain-Class entry. There is a switch that must be set on launch to point to the JAR path, which makes the JVM aware of the agent.

```
001 java -javaagent:/to/agent.jar com/example/spring2gx/
     BankTransactions
```

Inside `premain`, register a `Transformer` that captures the bytes of every class as it is loaded, makes any desired modifications, and returns the modified bytes. In Sue's example, `Transformer` captures `BankTransaction`, which is where she makes her modifications and returns the modified bytes. Those are the bytes that are loaded by the class loader, and which the `main` method will execute to perform its original functionality in addition to Sue's required augmented logging.

When the agent class is loaded, its `premain` method is invoked before the application `main` method.

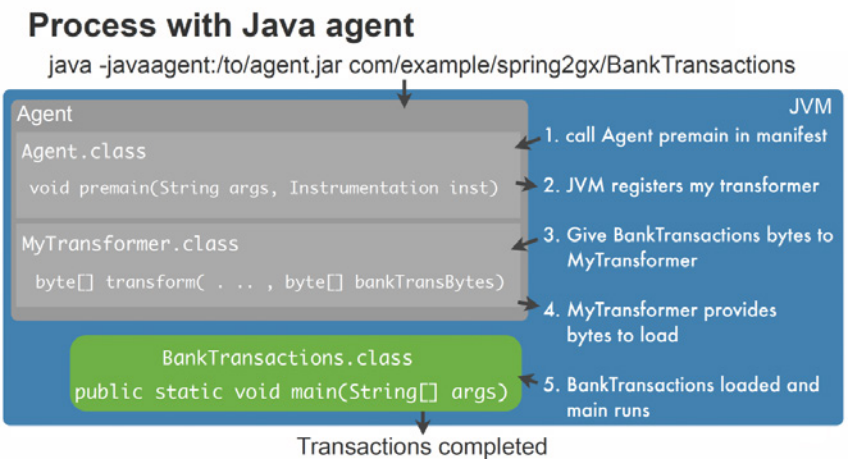


Figure 2: Process with Java agent.

It's best to look at an example.

The Agent class doesn't implement any interface, but it must contain a premain method, as follows:

Code for Java agent

```
Manifest:
 Premain-Class: com.example.spring2gx.agent.Agent
```

```
package com.example.spring2gx.agent;

public class Agent {

    public static void premain(String args, Instrumentation inst) {
        System.out.println("Starting the agent");
        inst.addTransformer(new ImportantLogClassTransformer());
    }
}
```

The `Transformer` class contains a `transform` method, whose signature accepts a `ClassLoader`, class name, `Class` object of the class being redefined, `ProtectionDomain` defining permissions, and the original bytes of the class. Returning `null` from the `transform` method tells the runtime that no changes have been made to that class.

```
package com.example.spring2gx.agent;

public class ImportantLogClassTransformer
  implements ClassFileTransformer {

  public byte[] transform(ClassLoader loader, String className,
    Class classBeingRedefined, ProtectionDomain protectionDomain,
    byte[] classfileBuffer) throws IllegalClassFormatException {
        System.out.println(" Loading class: " + className);
        return null;
    }
}
```

To modify the class bytes, supply your bytecode manipulation logic in transform and return the modified bytes.

```
package com.example.spring2gx.agent;

public class ImportantLogClassTransformer
  implements ClassFileTransformer {

  public byte[] transform(ClassLoader loader, String className,
    Class classBeingRedefined, ProtectionDomain protectionDomain,
    byte[] classfileBuffer) throws IllegalClassFormatException {
    // manipulate the bytes here
      return modified bytes;
    }
}
```

Javassist

A subproject of JBoss, Javassist (short for "Java Programming Assistant") consists of a high-level object-based API and a lower-level one that is closer to the bytecode. The more object-based one enjoys more community activity and is the focus of this article. For a complete tutorial, refer to the Javassist website.

In Javassist, the basic unit of class representation is the `CtClass` ("compile time class"). The classes that comprise your program are stored in a `ClassPool`, essentially a container for `CtClass` instances.

The `ClassPool` implementation uses a `HashMap`, in which the key is the name of the class and the value is the corresponding `CtClass` object.

A normal Java class contains fields, constructors, and methods. The corresponding `CtClass` represents those as `CtField`, `CtConstructor`,

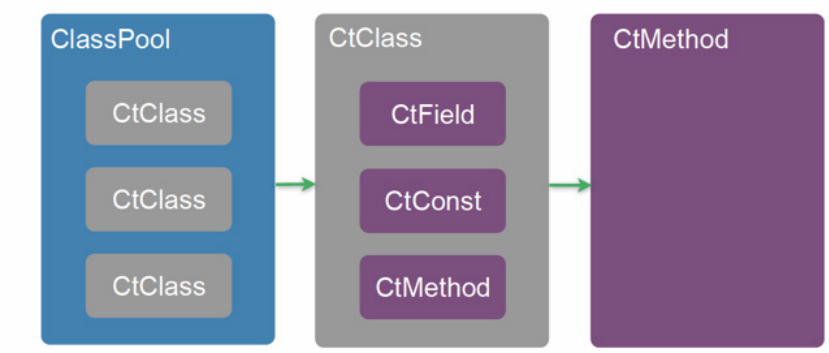
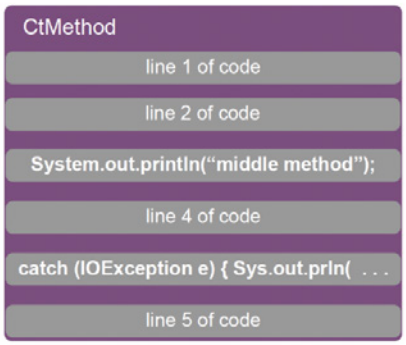


Figure 3

`CtMethod` contains lines of code for the associated method. We can insert code at the beginning of the method using the `insertBefore` command. The great thing about Javassist is that you write pure Java, albeit with one caveat: the Java must be implemented as quoted strings. But most people would agree that's much better than having to deal with bytecode! (Although, if you happen to like coding directly in bytecode, stay tuned for the ASM section.) The JVM includes a bytecode verifier to guard against invalid bytecode. If your Javassist-coded Java is not valid, the bytecode verifier will reject it at runtime.

Similar to `insertBefore`, there's an `insertAfter` to insert code at the end of a method. You can also insert code in the middle of a method by using `insertAt` or add a catch statement with `addCatch`.

```
putMethod.insertAt(3, true, "System.out.println(\"middle method\");");
      CtClass exceptionType = classpool.get("java.io.IOException");
putMethod.addCatch("{System.out.println($e); throw $e}", exceptionType);
```



Let's kick off your IDE and code your logging feature. We start with an Agent (containing premain) and our ClassTransformer.

```
001 package com.example.spring2gx.agent;
002 public class Agent {
003  public static void premain(String args,
     Instrumentation inst) {
004    System.out.println("Starting the agent");
005    inst.addTransformer(new
     ImportantLogClassTransformer());
006  }
007 }
008
009 package com.example.spring2gx.agent;
010 import java.lang.instrument.ClassFileTransformer;
011 import java.lang.instrument.
     IllegalClassFormatException;
012 import java.security.ProtectionDomain;
013
014 public class ImportantLogClassTransformer
015    implements ClassFileTransformer {
016
017  public byte[] transform(ClassLoader loader, String
     className,
018                          Class classBeingRedefined,
     ProtectionDomain protectionDomain,
019                          byte[] classfileBuffer)
     throws IllegalClassFormatException {
020 // manipulate the bytes here
021      return modified_bytes;
022  }
```

To add audit logging, first implement transform to convert the bytes of the class to a CtClass object. Then, you can iterate its methods and capture ones with the @ImportantLogin annotation on them, grab the input parameter indexes to log, and insert that code at the beginning of the method.

```
001  public byte[] transform(ClassLoader loader, String className,
002                          Class classBeingRedefined, ProtectionDomain
     protectionDomain,
003                          byte[] cfbuffer) throws IllegalClassFormatException {
004 // convert byte array to a ct class object
005     pool.insertClassPath(new ByteArrayClassPath(className,classfileBuffer));
006 // convert path slashes to dots.
007     CtClass cclass = pool.get(className.replaceAll("/", "."));
008     if (!cclass.isFrozen()) {
009 // check each method of ct class for annotation @ImportantLog
010       for (CtMethod currentMethod : cclass.getDeclaredMethods()) {
011 // Look for annotation @ImportantLog
012         Annotation annotation = getAnnotation(currentMethod);
013         if (annotation != null) {
014 // if @ImportantLog annotation present on method, then
015
016 // get important method parameter indexes
017           List<String> parameterIndexes = getParamIndexes(annotation);
018 // add logging statement to beginning of the method
019           currentMethod.insertBefore(createJavaString(currentMethod, className,
     parameterIndexes));
020         }
021       }
022       return cclass.toBytecode();
023     }
024     return null;
025   }
```

Javassist annotations can be declared as "invisible" or "visible". Invisible annotations, which are only visible at class loading time and compile time, are declared by passing in the `RententionPolicy.CLASS` argument to the annotation. Visible annotations (`RententionPolicy.RUNTIME`) are loaded and visible at run time. For this example, you only need the attributes at compile time, so make them invisible.

The `getAnnotation` method scans for your `@ImportantLog` annotation and returns null if it doesn't find the annotation.

```
001  private Annotation
     getAnnotation(CtMethod method) {
002     MethodInfo methodInfo = method.
     getMethodInfo();
003     AnnotationsAttribute attInfo =
     (AnnotationsAttribute) methodInfo
004
     .getAttribute(AnnotationsAttribute.
     invisibleTag);
005     if (attInfo != null) {
006       return attInfo.
     getAnnotation("com.example.spring.
     mains.ImportantLog");
007     }
008     return null;
009   }
```

With the annotation in hand, you can retrieve the parameter indexes. Using Javassist's `ArrayMemberValue`, the member value fields are returned as a `String` array, which you can iterate to obtain the field indexes you had embedded in the annotation.

```
001  private List<String>
     getParamIndexes(Annotation
     annotation) {
002     ArrayMemberValue fields =
     (ArrayMemberValue) annotation.
     getMemberValue("fields");
003     if (fields != null) {
004       MemberValue[] values =
     (MemberValue[]) fields.getValue();
005       List<String> parameterIndexes =
     new ArrayList<String>();
006       for (MemberValue val : values)
     {
007         parameterIndexes.
     add(((StringMemberValue) val).
     getValue());
008       }
009       return parameterIndexes;
010     }
011     return Collections.emptyList();
012   }
```

You are finally in a position to insert your log statement in `createJavaString`.

```
001     private String createJavaString(CtMethod currentMethod, String
      className,
002                                     List<String> indexParameters) {
003       StringBuilder sb = new StringBuilder();
004       sb.append("{StringBuilder sb = new StringBuilder");
005       sb.append("(\"A call was made to method '\");");
006       sb.append("sb.append(\"");
007       sb.append(currentMethod.getName());
008       sb.append("\");sb.append(\"' on class '\");");
009       sb.append("sb.append(\"");
010       sb.append(className);
011       sb.append("\");sb.append(\"'.\");");
012       sb.append("sb.append(\"\\n Important params:\");");
013
014       for (String index : indexParameters) {
015         try {
016           int localVar = Integer.parseInt(index) + 1;
017           sb.append("sb.append(\"\\n Index: \");");
018           sb.append("sb.append(\"");
019           sb.append(index);
020           sb.append("\");sb.append(\" value: \");");
021           sb.append("sb.append($" + localVar + ");");
022         } catch (NumberFormatException e) {
023           e.printStackTrace();
024         }
025       }
026       sb.append("System.out.println(sb.toString());}");
027       return sb.toString();
028     }
029   }
```

Your implementation creates a `StringBuilder`, appending some preamble followed by the required method name and class name. One thing to note is that if you're inserting multiple Java statements, you need to surround them with squiggly brackets (see lines 4 and 26).

(Brackets are not required for just a single statement.)

That pretty much covers the code for adding audit logging using Javassist. In retrospect, the positives are:

- Because it uses familiar Java syntax, there's no bytecode to learn.
- There wasn't too much programming to do.
- Good documentation on Javassist exists.

The negatives are:

- Not using bytecode limits capabilities.
- Javassist is slower than other bytecode-manipulation frameworks.

ASM

ASM began life as a Ph.D. project and was open-sourced in 2002. It is actively updated, and supports Java 8 since the 5.x version. ASM consists of an event-based library and an object-based one, similar in behavior respectively to SAX and DOM XML parsers. This article will focus on the event-based library. Complete documentation can be found at http://download.forge.objectweb.org/asm/asm4-guide.pdf.

A Java class contains many components, including a superclass, interfaces, attributes, fields, and methods. With ASM, you can think of each of these as events; you parse the class by providing a `ClassVisitor` implementation, and as the parser encounters each of those components, a corresponding "visitor" event-handler method is called on the `ClassVisitor` (always in this sequence).

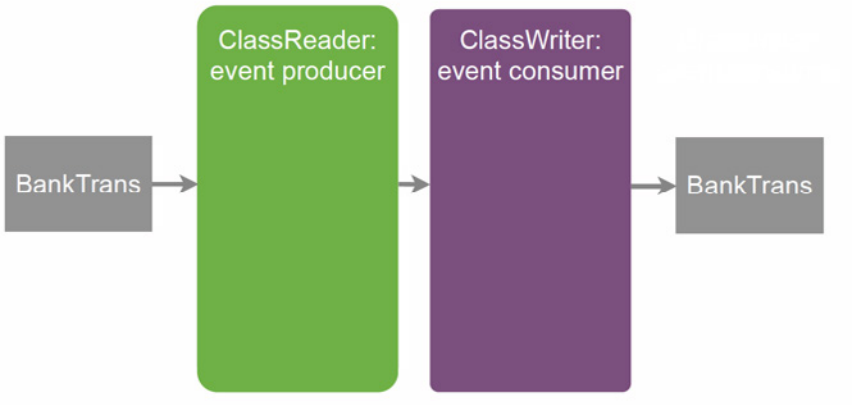
```
001 package com.sun.xml.internal.ws.org.objectweb.asm;
002    public interface ClassVisitor {
003         void visit(int version, int access, String
      name, String signature, String superName, String[]
      interfaces);
004         void visitSource(String source, String debug);
005         void visitOuterClass(String owner, String
      name, String desc);
006         AnnotationVisitor visitAnnotation(String desc,
      boolean visible);
007         void visitAttribute(Attribute attr);
008         void visitInnerClass(String name, String
      outerName, String innerName, int access);
009         FieldVisitor visitField(int access, String
      name, String desc, String signature, Object value);
010         MethodVisitor visitMethod(int access, String
      name, String desc, String signature, String[]
      exceptions);
011         void visitEnd();
012    }
```

Let's get a feel for the process by passing Sue's `BankTransaction` (defined at the beginning of the article) into a `ClassReader` for parsing.

Again, start with the `Agent premain`:

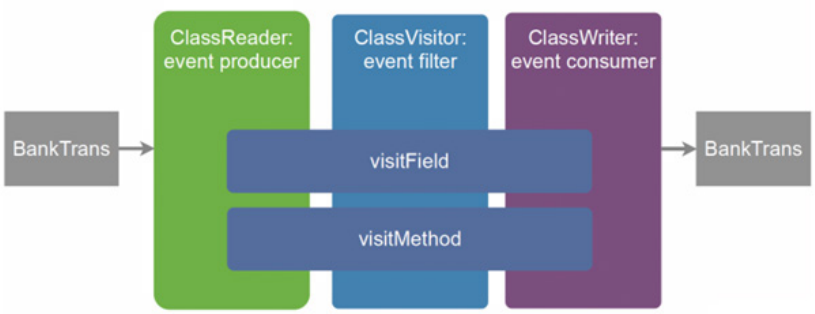
```
001 import java.lang.instrument.Instrumentation;
002 public class Agent {
003   public static void premain(String args,
      Instrumentation inst) {
004     System.out.println("Starting the agent");
005     inst.addTransformer(new
      ImportantLogClassTransformer());
006   }
007 }
```

Then pass the output bytes to a no-op ClassWriter to put the parsed bytes back together in the byte array, producing a rehydrated BankTransaction that as expected is virtually identical to our original class.



```
001 import jdk.internal.org.objectweb.asm.ClassReader;
002 import jdk.internal.org.objectweb.asm.ClassWriter;
003 import jdk.internal.org.objectweb.asm.ClassVisitor;
004 import java.lang.instrument.ClassFileTransformer;
005 import java.lang.instrument.
     IllegalClassFormatException;
006 import java.security.ProtectionDomain;
007
008 public class ImportantLogClassTransformer implements
     ClassFileTransformer {
009   public byte[] transform(ClassLoader loader, String
     className,
010                           Class classBeingRedefined,
     ProtectionDomain protectionDomain,
011                           byte[] classfileBuffer)
     throws IllegalClassFormatException {
012     ClassReader cr = new ClassReader(classfileBuffer);
013     ClassWriter cw = new ClassWriter(cr, ClassWriter.
     COMPUTE_FRAMES);
014     cr.accept(cw, 0);
015     return cw.toByteArray();
016   }
017 }
```

Now, modify your `ClassWriter` to do something a little more useful by adding a `ClassVisitor` (named `LogMethodClassVisitor`) to call your event-handler methods, such as `visitField` or `visitMethod`, as the corresponding components are encountered during parsing.




```
001 public byte[] transform(ClassLoader loader, String
     className,
002                           Class classBeingRedefined,
     ProtectionDomain protectionDomain,
003                           byte[] classfileBuffer)
     throws IllegalClassFormatException {
004     ClassReader cr = new ClassReader(classfileBuffer);
005     ClassWriter cw = new ClassWriter(cr, ClassWriter.
     COMPUTE_FRAMES);
006     ClassVisitor cv = new LogMethodClassVisitor(cw,
     className);
007     cr.accept(cv, 0);
008     return cw.toByteArray();
009   }
```

For your logging requirement, you want to check each method for the indicative annotation and add any specified logging. You only need to overwrite `ClassVisitor visitMethod` to return a `MethodVisitor` that supplies your implementation. Just like there are several components of

a class, there are several components of a method, corresponding to the method attributes, annotations, and compiled code. ASM's `MethodVisitor` provides hooks for visiting every opcode of the method, so you can get pretty granular in your modifications.

```
001 public class LogMethodClassVisitor extends
     ClassVisitor {
002   private String className;
003   public LogMethodIfAnnotationVisitor
     (ClassVisitor cv, String className) {
004     super(Opcodes.ASM5, cv);
005     this.className = className;
006   }
007   @Override
008   public MethodVisitor visitMethod(int access, String
     name, String desc,
009                                    String signature,
     String[] exceptions) {
010 // put our logic in here
011   }
012 }
```

Again, the event handlers are always called in the same predefined sequence, so you always know all of the attributes and annotations on the method before you have to actually visit the code. (Incidentally, you can chain together multiple instances of `MethodVisitor`, just like you can chain multiple instances of `ClassVisitor`.) So in your `visitMethod`, you're going to hook in the `PrintMessageMethodVisitor`, overriding `visitAnnotations` to capture your annotations and insert any required logging code.

```
public class PrintMessageMethodVisitor extends MethodVisitor {

  @Override
  public AnnotationVisitor visitAnnotation(String desc, boolean visible) {
    1. check method for annotation @ImportantLog
    2. if annotation present, then get important method param indexes
  }


  @Override
  public void visitCode() {
    3. if annotation present, add logging to beginning of the method
  }

}
```

Your `PrintMessageMethodVisitor` overrides two methods. First comes `visitAnnotation`, so you can check the method for your `@ImportantLog` annotation. If present, you need to extract the field indexes from that field's property. When `visitCode` executes, the presence of the annotation has already been determined and so it can add the specified logging. The `visitAnnotation` code hooks in an `AnnotationVisitor` that exposes the field arguments on the `@ImportantLog` annotation.

```
001 public AnnotationVisitor visitAnnotation(String desc,
     boolean visible) {
002     if (“Lcom/example/spring2gx/mains/ImportantLog;”.
     equals(desc)) {
003       isAnnotationPresent = true;
004       return new AnnotationVisitor(Opcodes.ASM5,
005         super.visitAnnotation(desc, visible)) {
006         public AnnotationVisitor visitArray(String
     name) {
007           if (“fields”.equals(name)){
008             return new AnnotationVisitor(Opcodes.
     ASM5,
009               super.visitArray(name)) {
010               public void visit(String name, Object
     value) {
011                 parameterIndexes.add((String) value);
012                 super.visit(name, value);
013               }
014             };
015           }else{
016             return super.visitArray(name);
017           }
018         }
019       };
020     }
021     return super.visitAnnotation(desc, visible);
022   }
```

Now, let’s look at the `visitCode` method. First, it must check if the `AnnotationVisitor` flagged the annotation as present. If so, then add your bytecode.

```
001 public void visitCode() {
002     if (isAnnotationPresent) {
003 // create string builder
004       mv.visitFieldInsn(Opcodes.GETSTATIC, “java/
     lang/System”, “out”,  “Ljava/io/PrintStream;”);
005       mv.visitTypeInsn(Opcodes.NEW, “java/lang/
     StringBuilder”);
006       mv.visitInsn(Opcodes.DUP);
007 // add everything to the string builder
008       mv.visitLdcInsn(“A call was made to method
     \””);
009       mv.visitMethodInsn(Opcodes.INVOKESPECIAL,
     “java/lang/StringBuilder”, “<init>”,
010         “(Ljava/lang/String;)V”, false);
011       mv.visitLdcInsn(methodName);
012       mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
     “java/lang/StringBuilder”, “append”,
013         “(Ljava/lang/String;)Ljava/lang/
     StringBuilder;”, false);
014
015     }
```

This is the scary part of ASM — you actually have to write bytecode, so that’s something new to learn. You have to know about the stack, local variables, etc. It’s a fairly simple language, but if you just want to hack

around, you can actually get the existing bytecode pretty easily with `javap`:

```
001 javap -c com/example/spring2gx/mains/PrintMessage
```

```
public void print();
    Code:
       0: new           #38                 // class java/lang/StringBuilder
       3: dup
       4: invokespecial #40                 // Method java/lang/
StringBuilder."<init>":()V
       7: astore_1
       8: aload_1
       9: ldc           #41                 // String A call was made to method \"
      11: invokevirtual #43                 // Method java/lang/
StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
      14: pop
      15: aload_1
      16: aload_0
      17: getfield      #14                 // Field methodName:Ljava/lang/String;
      20: invokevirtual #43                 // Method java/lang/
StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

I recommend writing the code you need in a Java test class, compiling that, and running it though `javap -c` to see the exact bytecode. In the code sample above, everything in blue is actually the bytecode. On each line, you get a one-byte opcode followed by zero or more arguments. You will need to determine those arguments for the target code, and they can usually be extracted by doing a `javap-c -v` on the original class (`-v` for verbose, which displays the constant pool).

I encourage you to look at the JVM spec, which defines every opcode. There are operations like `load` and `store` (which move data between your operand stack and your local variables), overloaded for each parameter type. For example, `ILOAD` moves an integer value from the stack into a local variable field whereas `LLOAD` does the same for a long value.

There are also operations like `invokeVirtual`, `invokeSpecial`, `invokeStatic`, and the recently added `invokeDynamic`, for invoking standard instance methods, constructors, static methods, and dynamic methods in dynamically typed JVM languages, respectively. There are also operations for creating new classes using the `new` operator, or to duplicate the top operand on the stack.

In sum, the positives of ASM are:

- It has a small memory footprint.
- It's typically pretty quick.
- It's well documented on the web.
- All of the opcodes are available, so you can really do a lot with it.
- There's lots of community support.

The really only one negative, but it's a big one: you're writing bytecode, so you have to understand what's going on under the hood and as a result developers tend to take some time to ramp up.

Lessons learned

- When you're dealing with bytecode manipulation, it's important to take small steps. Don't write lots of bytecode and expect it to immediately pass verification and work. Write one line at a time, think about what's in your stack, think about your local variables, and then write another line. If it's not passing the verifier, change one thing at a time; otherwise you'll never get it to work. Also keep in mind that besides the JVM verifier, ASM maintains a separate bytecode verifier, so it's good to run both and verify that your bytecode passes both of them.
- It's important to think about class loading when you're modifying classes. When you use a Java agent, its transformer will touch every class as it is loaded into the JVM, no matter which class loader is loading it. So you need to make sure that the class loader can also see that object. Otherwise, you're going to run into trouble.
- If you're using Javassist and an application server that has multiple class loaders, you have to be concerned about your class pool being able to see your class objects. You might have to register a new classpath to your class pool to get it to see your class objects. You can chain your class pools like Java chains class loaders, so if it doesn't find the `CTClass` object in its class pool, it can go look at its parents.
- Finally, it's important to note that the JDK has its own capability to transform classes, and some limitations will apply to any class that the JDK has already transformed; you can modify the implementation of methods but, unlike original transformations, re-transformations are not permitted to change the class structure, for example by adding new fields or methods, or by modifying signatures.

Bytecode manipulation can make life easier. You can find bugs, add logging (as discussed), obfuscate source code, perform preprocessing like Spring or Hibernate, or even write your own language compiler. You can restrict your API calls, analyze code to see if multiple threads are accessing a collection, lazy-load data from the database, and find differences between JARs by inspecting them.

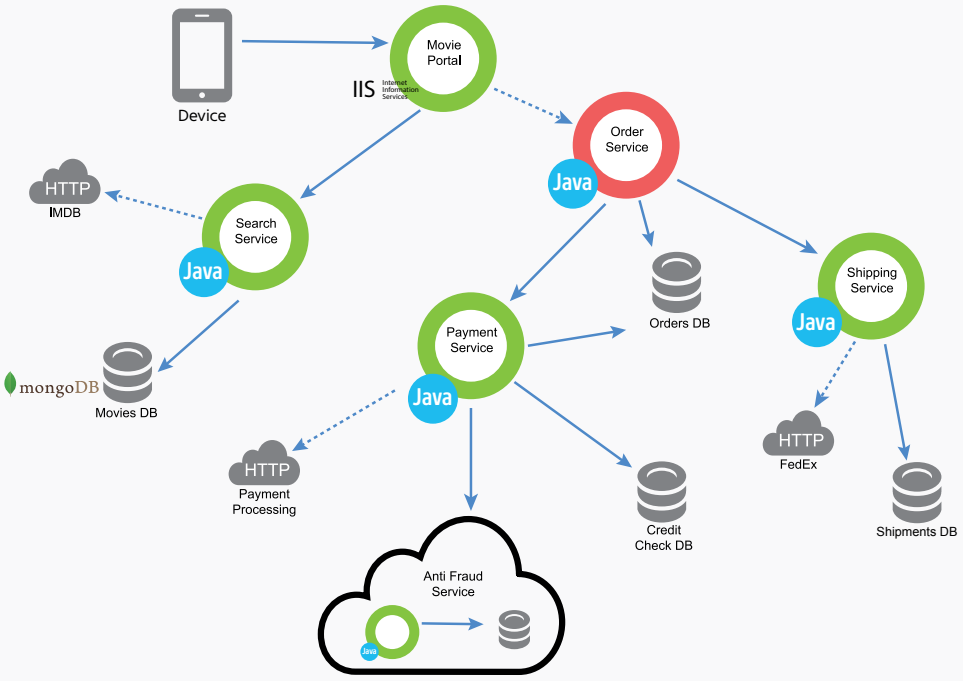
So I encourage you to make a bytecode-manipulation framework your friend. Someday, one might save your job.

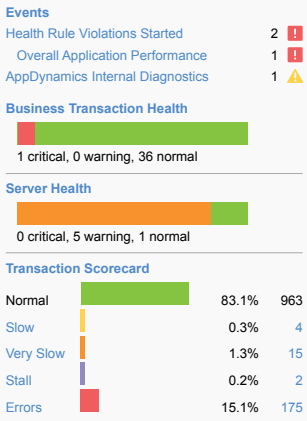
This article is extracted from a presentation by New Relic's Ashley Puls at Spring One. The full presentation can be viewed and downloaded here: http://www.infoq.com/presentations/asm-cglib-javassist-2

APPDYNAMICS

There's nothing about Java that AppDynamics doesn't see.

AppDynamics gives you the visibility to take command of your Java application's performance, no matter how complicated or distributed your environment is.





When your business runs on Java, count on AppDynamics to give you the complete visibility you need to be sure they are delivering the performance and business results you need — no matter how complex, distributed or asynchronous your environment, live 'in production' or during development.

See every line of code. Get a complete view of your environment with deep code diagnostics and auto-discovery. Understand performance trends with dynamic baselining. And drastically reduce time to root cause and remediation.

See why the world's largest Java deployments rely on the AppDynamics Application Intelligence Platform. Sign up for a FREE trial today at www.appdynamics.com/java.

appdynamics.com

Easily Create Java Agents with Byte Buddy

Rafael Winterhalter works as a software consultant in Oslo, Norway. He is a proponent of static typing and a JVM enthusiast with particular interests in code instrumentation, concurrency, and functional programming. Rafael blogs about software development, regularly presents at conferences, and was pronounced a JavaOne Rock Star. When coding outside of his work place, he contributes to a wide range of open-source projects and often works on Byte Buddy, a library for simple runtime code generation for the Java virtual machine. For this work, Rafael received a Duke's Choice award.

A Java agent is a Java program that executes just prior to the start of another Java application (the "target" application), affording that agent the opportunity to modify the target application or the environment in which it runs. In the most basic use case, a Java agent sets application properties or configures a certain environment state, enabling the agent to serve as a reusable and pluggable component. The following example describes such an agent, which sets a system property that becomes available to the actual program.

```
001 public class Agent {
002   public static void
      premain(String arg)
      {
003     System.
      setProperty("my-
      property", "foo");
004   }
005 }
```

As demonstrated by the above code, a Java agent is defined like any other Java program, except that `premain` replaces the main method as the entry point. As the name suggests, this method is executed before the main method of the target application. There are no other specific rules for writing an agent other than the standard rules that apply to any Java program. One minimal difference is that a Java agent receives a single, optional argument instead of an array of zero or more arguments.

To launch your agent, you must bundle the agent classes and resources in a JAR, and set the `Agent-Class` property in the

JAR manifest to the name of your agent class containing the `premain` method. (An agent must always be bundled as a JAR file; it cannot be specified in an exploded format.) Next, you must launch the application by referencing the JAR file's location via the `javaagent` parameter on the command line.

```
001 java -javaagent:myAgent.jar -jar
     myProgram.jar
```

You can also prepend optional agent arguments to this location path. The following command starts a Java program and attaches the given agent that provides the value `myOptions` as the argument to the `premain` method:

```
001 java -javaagent:myAgent.jar=myOptions
     -jar myProgram.jar
```

It is possible to attach multiple agents by repeating the `javaagent` command.

A Java agent is capable of much more than only altering the state of an application's environment; you can grant a Java agent access to the Java instrumentation API, allowing it to modify the code of the target application. This little known feature of the Java virtual machine offers a powerful tool that facilitates the implementation of aspect-oriented programming.

You apply such modifications to a Java program by adding a second parameter of type `Instrumentation` to the agent's `premain` method. You can use the `Instrumentation` parameter to perform a range of tasks, from determining an object's size in bytes to modifying class implementations by registration of `ClassFileTransformers`. After it is registered, a `ClassFileTransformer` is invoked by any class loader upon loading a class. When invoked, a class-file transformer has the opportunity to transform or to even fully replace any class file before the represented class is loaded. In this way, it is possible to enhance or modify a class's behavior before it is put to use, as exemplified by the following example.

```
001 public class Agent {
002  public static void premain(String
     argument, Instrumentation inst) {
003    inst.addTransformer(new
     ClassFileTransformer() {
004      @Override
005      public byte[] transform(
006        ClassLoader loader,
007        String className,
008        Class<?> classBeingRedefined,
     // null if class was not previously
     loaded
009        ProtectionDomain
     protectionDomain,
010        byte[] classFileBuffer) {
011        // return transformed class
     file.
012      }
013    });
014  }
015 }
```

After registering the above `ClassFileTransformer` with an `Instrumentation` instance, the transformer is invoked every time a class loads. For this purpose, the transformer receives a binary representation of the class file and a reference to the class loader that is attempting to load this class.

A Java agent can also be registered during the run time of a Java application. In this case, the instrumentation API allows for the redefinition of already loaded classes, a feature called "*HotSwap*". Unfortunately, redefining loaded classes is limited to replacing method bodies. No members may be added or removed and no types or signatures may change when redefining a class. This limitation does not apply when a class is loaded for the first time, and in those cases the `classBeingRedefined` parameter is set to null.

Java bytecode and the class-file format

A class file represents a Java class in its compiled state. A class file contains the bytecode representation of program instructions originally coded as Java source code. Java bytecode can be considered to be the language of the Java virtual machine (JVM). In fact, the JVM has no notion of Java as a programming language, but exclusively processes bytecode. As a result of the binary representation, bytecode consumes less space than a program's source code. Also, representing a program as bytecode allows for easier compilation of languages other than Java — Scala or Clojure for example — for running on the JVM. Without bytecode as an intermediate language, it would have been necessary to translate any program into Java source code before running it.

In the context of code manipulation, this abstraction comes at a price. When applying a `ClassFileTransformer` to a Java class, this class can no longer be processed as Java source code, even if that transformed code had been written in Java in the first place. To make things worse, when transforming the class, the reflection API for introspecting the class's members or annotations is also off limits as access to the API would require the class to already be loaded, which would not happen before the transformation process completed.

Fortunately, Java bytecode is a comparatively simple abstraction with a relatively small number of operations, and you can generally learn it with little

effort. The JVM executes a program by processing values as a stack machine. A bytecode instruction typically indicates to the virtual machine that it should pop values from an operand stack, perform some operations, and push a result back on the stack.

Let's consider a simple example: adding the numbers 1 and 2. The JVM first pushes both numbers onto the operand stack by executing the bytecode instructions `iconst_1` and `iconst_2`. `iconst_1` is a one-byte convenience operator that pushes the number 1 onto the stack. Similarly, `iconst_2` pushes the number 2 onto the stack. Subsequently, executing the `iadd` instruction pops the two latest values off the stack and pushes back the sum of those numbers. Within a class file, each instruction is not stored by its mnemonic name but rather as a single byte that uniquely identifies a specific instruction, thus the term "*bytecode*". The above bytecode instructions and their impact on the operand stack are visualized in Figure 1.

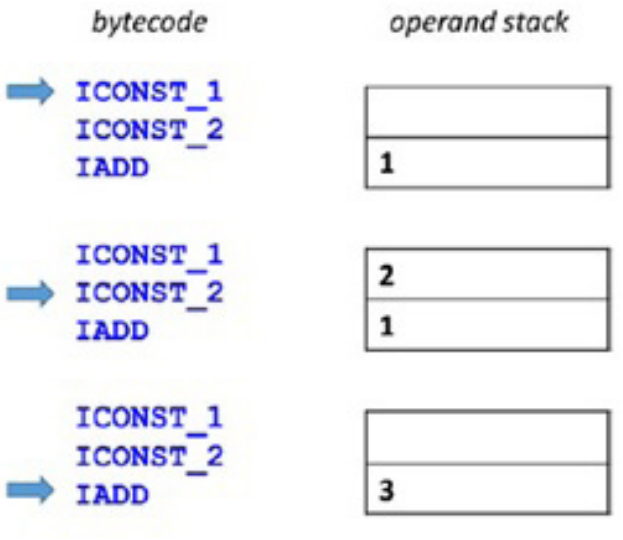


Figure 1

Fortunately for humans, who do better with source than with bytecode, the Java community has created several libraries that parse class files and expose compacted bytecode as a stream of named instructions. For example, the popular ASM library offers a simple visitor API that dissects a class file into members and method instructions, operating in a similar fashion to a SAX parser for reading XML files. Using ASM, the bytecode for the above example can be implemented as demonstrated by the following code (where the `visitIns` instructions are ASM's way of providing the revised method implementation).

```
001 MethodVisitor methodVisitor = ...
002 methodVisitor.visitIns(Opcodes.
      ICONST_1);
003 methodVisitor.visitIns(Opcodes.
      ICONST_2);
004 methodVisitor.visitIns(Opcodes.IADD);
```

It should be noted that the bytecode specification merely serves as a metaphor and that a Java virtual machine is allowed to translate a program into optimized machine code as long as the program's outcome remains correct. Thanks to the simplicity of bytecode, it is straightforward to replace or to modify the instructions within an existing class. Therefore, using ASM and understanding the fundamentals of Java bytecode already suffices for implementing a class-transforming Java agent by registering a `ClassFileTransformer` that processes its arguments using this library.

Overcoming the bytecode metaphor

For a practical application, parsing a raw class file still implies a lot of manual work. Java programmers are often interested in a class in the context of its type hierarchy. For example, a Java agent might be required to modify any class that implements a given interface. To determine information about a class's super types, it no longer suffices to parse the class file that is provided by a `ClassFileTransformer`, which only contains the names of the direct super type and interfaces. A programmer would still have to locate the class files for these types in order to resolve a potential super-type relationship.

Another difficulty is that making direct use of ASM in a project requires any developer on a team to learn about the fundamentals of Java bytecode. In practice, this often excludes many developers from changing any code that is concerned with bytecode manipulation. In such a case, implementing a Java agent imposes a threat to a project's long-term maintainability.

To overcome these problems, it is desirable to implement a Java agent using a higher-level abstraction than direct manipulation of Java bytecode. Byte Buddy is an open-source, Apache 2.0-licensed library that addresses the complexity of bytecode manipulation and the instrumentation API. Byte Buddy's declared goal is to hide explicit bytecode generation behind a type-safe domain-specific language. Using Byte Buddy, bytecode manipulation hopefully becomes intuitive to anybody who is familiar with the Java programming language.

Introducing Byte Buddy

Byte Buddy is not exclusively dedicated to the generation of Java agents. It offers an API for the generation of arbitrary Java classes, and on top of this class generation API, Byte Buddy offers an additional API for generating Java agents.

The following example demonstrates how Byte Buddy generates a simple class that subclasses `Object` and overrides the `toString` method to return "Hello World!" As with raw ASM, `intercept` instructs Byte Buddy to supply the method implementation using the intercepted instructions.

```
001 Class<?> dynamicType = new
      ByteBuddy()
002    .subclass(Object.class)
003    .method(ElementMatchers.
      named("toString"))
004    .intercept(FixedValue.value("Hello
      World!"))
005    .make()
006    .load(getClass().getClassLoader(),
007          ClassLoadingStrategy.Default.
      WRAPPER)
008    .getLoaded();
```

In the code above, Byte Buddy implements a method in two steps. First, a programmer needs to specify an `ElementMatcher` that is responsible for identifying one or several methods to implement. Byte Buddy offers a rich set of predefined interceptors that are exposed in the `ElementMatchers` class. In the above case, the `toString` method is matched by its exact name, but we could also match on more complex code structure such as type or annotations.

Whenever Byte Buddy generates a class, it analyzes the class hierarchy of the generated type. In the above example, Byte Buddy determines that the generated class inherits a single method named `toString` from its superclass `Object`, where the specified matcher instructs Byte Buddy to override that method by the subsequent `Implementation` instance — `FixedValue` in our example.

When creating a subclass, Byte Buddy always intercepts a matched method by overriding the method in the generated class. However, as we will see later, Byte Buddy is also capable of redefining existing classes without sub-classing. In such cases, Byte Buddy replaces an existing method with generated code while copying the original code into another, synthetic method.

In our sample code above, the matched method is overridden with an implementation that returns the fixed value "Hello World!" The `intercept` method accepts an argument of type Implementation and Byte Buddy ships with several predefined implementations such as the selected `FixedValue` class. If required, you can implement a method as custom bytecode using the ASM API discussed above, on top of which Byte Buddy is itself implemented.

After you define the properties of a class, the make method generates it. In our example, the user specified no name so the generated class received a random name. Finally, the generated class loads via a `ClassLoadingStrategy`. Using the default `WRAPPER` strategy, above, a class is loaded by a new class loader that has the environment's class loader as a parent.

After a class is loaded, it is accessible using the Java reflection API. If not specified differently, Byte Buddy generates constructors similar to those of the superclass such that a default constructor is available for the generated class. Consequently, it is possible to validate that the generated class has overridden the `toString` method as demonstrated by the following code.

```
001 assertThat(dynamicType.
      newInstance().toString(),
002          is("Hello World!"));
```

Of course, this generated class is of little practical use. For a real-world application, the return value of most methods is computed at run time and depends on method arguments and object state.

Instrumentation by delegation

A more flexible way to implement a method is with Byte Buddy's `MethodDelegation`. Method delegation can generate an overridden implementation that will invoke another method of a given class or instance. This way, it is possible to rewrite the previous example using the following delegator:

```
001 class ToStringInterceptor {
002   static String intercept() {
003     return "Hello World!";
004   }
005 }
```

With the above POJO interceptor, it becomes possible to replace the previous `FixedValue` implementation with `MethodDelegation.to(ToStringInterceptor.class)`.

```
001 Class<?> dynamicType = new
      ByteBuddy()
002    .subclass(Object.class)
003    .method(ElementMatchers.
      named("toString"))
004    .intercept(MethodDelegation.
      to(ToStringInterceptor.class))
005    .make()
006    .load(getClass().getClassLoader(),
007          ClassLoadingStrategy.Default.
      WRAPPER)
008    .getLoaded();
```

Using this delegator, Byte Buddy determines a *best invokable method* from the interception target provided to the `to` method. In the case of `ToStringInterceptor.class`, the selection process trivially resolves to the only static method that is declared

by this type. In this case, only static methods are considered because a *class* was specified as the target of the delegation. In contrast, it is possible to delegate to an *instance* of a class, in which case Byte Buddy considers all virtual methods. If several such methods are available on a class or instance, Byte Buddy first eliminates all methods that are not compatible with a specific instrumentation. Among the remaining methods, the library then chooses a best match, typically the method with the most parameters. It is also possible to choose a target method explicitly, by narrowing down the eligible methods by handing an `ElementMatcher` to the `MethodDelegation` by invoking the filter method. For example, by adding the following `filter`, Byte Buddy only considers methods named "intercept" as a delegation target.

```
MethodDelegation.
    to(ToStringInterceptor.class)

    .filter(ElementMatchers.
    named("intercept"))
```

After intercepting, the intercepted method still prints "Hello World!" but this time, the result is computed dynamically so that, for example, it is possible to set a breakpoint in the interceptor method that is triggered every time `toString` is called from the generated class.

The full power of the `MethodDelegation` is unleashed when specifying parameters for the interceptor method. A parameter is typically annotated for instructing Byte Buddy to inject a value when calling the interceptor. For example, using the `@Origin` annotation, Byte Buddy provides an instance of the instrumented `Method` as an instance of the class provided by the Java reflection API.

```
class ContextualToStringInterceptor {
  static String intercept(@Origin
    Method m) {
    return "Hello World from " +
    m.getName() + "!";
  }
}
```

When intercepting the `toString` method, the invocation is now instrumented to return "Hello world from toString!"In addition to the `@Origin` annotation, Byte Buddy offers a rich set of annotations. For example, using the `@Super` annotation on a parameter of type `Callable`, Byte Buddy creates and injects a proxy instance that allows invocation of the instrumented method's original code. If the provided annotations are insufficient or impractical for a specific use case, it is even possible to register custom annotations that inject a user-specified value.

Implementing method-level security

As we've seen, it is possible to use a `MethodDelegation` to dynamically override a method at run time using plain Java. That was a simple example but the technique can apply to more practical applications. Consider, for example, the use of code generation to implement an annotation-driven library for enforcing method-level security. In our first iteration, the library will generate subclasses to enforce this security. Then we will use the same approach to implement a Java agent to do the same.

The library uses the following annotation to allow a user to specify that a method be considered secured:

```
@interface Secured {
  String user();
}
```

For example, consider an application that uses the `Service` class to perform a sensitive action that should only be performed if the user is authenticated as an administrator. This is specified by declaring the `Secured` annotation on the method for executing this action.

```
class Service {
  @Secured(user = "ADMIN")
  void doSensitiveAction() {
    // run sensitive code...
  }
}
```

It is of course p\ossible to write the security check directly into the method. In practice, hard-coding crosscutting concerns frequently results in copy/pasted logic that is hard to maintain. Furthermore, directly adding such code does not scale well once an application reveals additional requirements, such as logging, collecting invocation metrics, or result caching. By extracting such functionality into an agent, a method purely represents its business logic, making it easier to read, test, and maintain a codebase.

In order to keep the proposed library simple, the contract of the annotation declares that an `IllegalStateException` should be thrown if the current user is not the one specified by the annotation's user property. Using Byte Buddy, you can implement this behavior with a simple interceptor, such as the `SecurityInterceptor` in the following example, which also keeps track of the user that is currently logged in by its static user field.

```
class SecurityInterceptor {

 static String user = “ANONYMOUS”

 static void intercept(@Origin
Method method) {
   if (!method.
getAnnotation(Secured.class).
user().equals(user)) {
     throw new
IllegalStateException(“Wrong
user”);
   }
 }
}
```

In the above code, the interceptor would not invoke the original method, even were access granted to a given user. Many of the predefined method implementations in Byte Buddy can be chained to overcome this. Using the `andThen` method of the `MethodDelegation` class, the above security check can be prepended to a plain invocation of the original method, as we will see below. Because a failed security check will throw an exception and prevent any further execution, the call to the original method is not be performed if the user is not authenticated.

Putting these pieces together, it is now possible to generate an appropriate subclass of `Service` where all annotated methods are secured appropriately. As the generated class is a subclass of `Service`, the generated class can be used as a substitute for all variables of the `Service` type, without a type casting, and will throw an exception when invoking the `doSensitiveAction` method without proper authentication.`new ByteBuddy()`

```
 .subclass(Service.class)
 .method(ElementMatchers.
isAnnotatedBy(Secured.class))
 .intercept(MethodDelegation.
to(SecurityInterceptor.class)

.andThen(SuperMethodCall.
INSTANCE)))
 .make()
 .load(getClass().getClassLoader(),
       ClassLoadingStrategy.Default.
WRAPPER)
 .getLoaded()
 .newInstance()
 .doSensitiveAction();
```

Unfortunately, because the instrumented subclasses are only created at run time, it is not possible to create such instances without using Java reflection. A factory that encapsulates the complexity of creating a subclass for the purpose of instrumentation should therefore create any instance of an instrumented class. As a result, frameworks that already require the creation of instances by factories commonly use subclass instrumentation — for example, frameworks for dependency-injection like Spring or object-relational mapping like Hibernate. For other types of applications, subclass instrumentation is often too complex to realize.

A Java agent for security

An alternative implementation for the above security framework would be to use a Java agent to modify the original bytecode of a class such as the above `Service` rather than overriding it. By doing so, it would no longer be necessary to create managed instances; simply calling the following line of code

```
new Service().doSensitiveAction()
```

To support this approach to modifying a class, Byte Buddy offers a concept called “*rebasing a class*”. Rebasing a class does not create a subclass but instead merges the instrumented code into the instrumented class to change its behavior. With this approach, you can still access the original code of any method of the instrumented class after instrumenting it, so that instrumentations like `SuperMethodCall` work exactly the same way as when creating a subclass.

Thanks to the similar behavior when either subclassing or rebasing, the APIs for both operations are executed in the same way: by describing a type using the same `DynamicType.Builder` interface. You can access both forms of instrumentation via the `ByteBuddy` class. To make the definition of a Java agent more convenient, Byte Buddy does also offers the `AgentBuilder` class, which is dedicated to solve common use cases in a concise manner. In order to define a Java agent for method-level security, the definition of the following class as the agent’s entry point suffices.

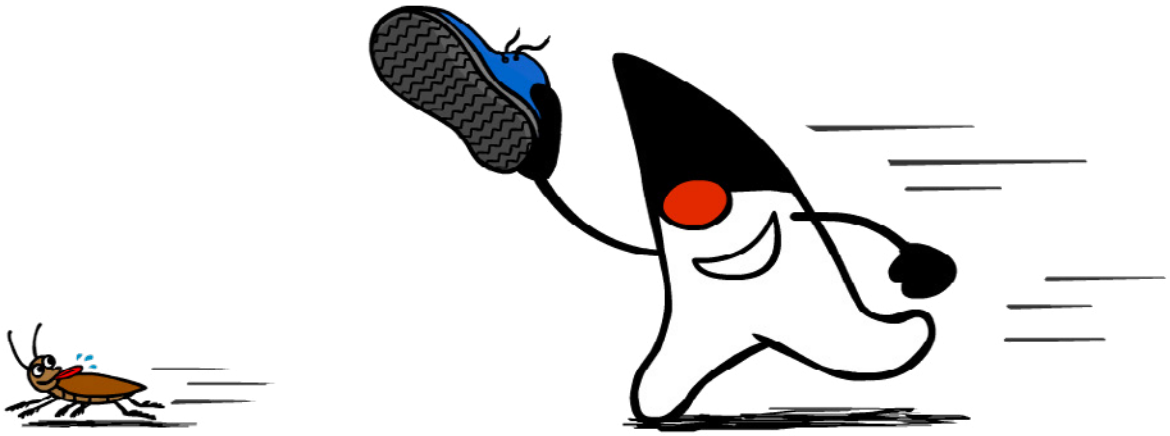
```
001 class SecurityAgent {
002   public static void premain(String
      arg, Instrumentation inst) {
003     new AgentBuilder.Default()
004       .type(ElementMatchers.any())
005       .transform((builder, type) ->
      builder
006         .method(ElementMatchers.
      isAnnotatedBy(Secured.class)
007         .intercept(MethodDelegation.
      to(SecurityInterceptor.class)
008
      .andThen(SuperMethodCall.
      INSTANCE))))
009       .installOn(inst);
010   }
011 }
```

If this agent is bundled in a JAR file and specified on the command line, any type is "transformed" or redefined to secure any methods that specify the `Secured` annotation. Without the activation of the Java agent, the application runs without the additional security checks. Of course, this implies unit tests where the code of an annotated method can be invoked without requiring a specific setup for mocking the security context. As the Java runtime ignores annotation types that cannot be found on the classpath, it is even possible to run the annotated methods after removing the security library from the application entirely.

Another advantage of Java agents is that they are easily stacked. If several Java agents are specified on the command line, each agent has an opportunity to modify a class in the order they occupy on the command line. This would allow, for example, combining frameworks for security, logging, and monitoring without requiring any integration layers among these applications. Using Java agents like this to implement crosscutting concerns allows you to write more modular code without integrating all code against a central framework for managing instances.

The source code to Byte Buddy is freely available on GitHub. A tutorial can be found at http://bytebuddy.net. Byte Buddy is currently available in version 0.7.4 and all code examples are based upon this version. The library won a Duke's Choice award by Oracle in 2015 for its innovative approach and its contribution to the Java ecosystem.

Five Advanced Debugging Techniques Every Java Developer Should Know



Tal Weiss is the CEO of Takipi. Tal has been designing scalable, real-time Java and C++ applications for the past 15 years. He still enjoys analyzing a good bug, though, and instrumenting Java code. In his free time, Tal plays jazz drums.

Production debugging is hard, and it's getting harder. With architectures becoming more distributed and code more asynchronous, pinpointing and resolving errors that happen in production is no child's game. Here are some advanced techniques that can help you to more quickly get to the root cause of painful bugs in production without adding material overhead to your already busy production environment.

Better jstacks

jstack has been around for a long time (about as long as the JVM's been around) and remains a crucial tool in every developer's arsenal. Whenever you're staring at a Java process that's hung or not responding, it's your go-to tool for seeing the stack trace of each thread. Even so, jstack has a couple of disadvantages that detract from its ability to help you debug complex issues in production. The first is that while it tells you what each thread is doing through its stack trace, it doesn't tell what you why it's doing it (the kind of information usually only available through a debugger) or when it started doing it.

Fortunately, there's a great way to fix that and make this good tool even better — by injecting dynamic variable state into each thread's data. The key to this problem lies in one of the most unexpected places. You see, at no point in its execution does jstack query the JVM or the application code for variable state to present. However, there is one important exception, with which we can turn lemons into lemonade, and that is the `Thread.Name()` property, whose value is injected into the stack dump. By set-

ting it correctly, you can move away from uninformative jstack thread printouts that look like:

```
001 pool-1-thread-1" #17 prio=5 os_
      prio=31 tid=0x00007f9d620c9800
      nid=0x6d03 in Object.wait()
      [0x000000013ebcc000]
```

Compare that with the following thread printout that contains a description of the actual work being done by the thread, the input parameters passed to it, and the time in which it started processing the request:

```
001 pool-1-thread- #17: Queue: ACTIVE_
      PROD, MessageID: AB5CAD, type:
      Analyze, TransactionID: 56578956,
      Start Time: 10/8/2014 18:34
```

Here's an example for how we set a stateful thread name.

```
001 private void processMessage(Message
      message) { //an entry point into
      your code
002    String name = Thread.
      currentThread().getName();
003    try {
004       Thread.currentThread().
      setName(prettyFormat(name,
      getCurrTranscationID(),
005                      message.
      getMsgType(), message.getMsgID(),
      getCurrentTime()));
006       doProcessMessage(message);
007    }
008    finally {
009       Thread.currentThread().
      setName(name); // return to
      original name
010    }
011 }
```

In this example, the thread processes messages out of a queue, and we see the target queue from which the thread is de-queuing messages as well as the ID of the message being processed, the transaction to which it is related (which is critical for reproducing locally), and when the processing of this message began. This last bit of information lets you look at a server jstack with upwards of a hundred worker threads to see which ones started first and are most likely causing an application server to hang. (Figure 1)

The capability works just as well when you're using a profiler, a commercial monitoring tool, a JMX console, or even Java 8's new Mission Control. In all these cases, having stateful thread contexts enhances your ability to look at the live thread state or a historic thread dump, and see exactly what each thread is doing and when it started. (Figure 2)

But the value of `Thread` names doesn't stop there. They play an even bigger role in production debugging — even if you don't use jstack at all. One instance is the global `Thread.uncaughtExceptionHandler` callback, which serves as a last line of defense before an uncaught exception terminates a thread (or is sent back to the thread pool).

By the point an uncaught exception handler is reached, code execution has stopped and both frames and variable state have already been rolled back. The only remaining state that you can use to log the task that thread was processing, its parameters, and starting time is captured by — you guessed it — a stateful `Thread` name (and any additional TLS variables loaded onto it).

It's important to keep in mind that a threading framework might implicitly catch exceptions without you knowing it. A good example is `ThreadPoolExecutorService`, which catches all exceptions in your `Runnable` and delegates them to its `afterExecute` method, which you can override to display something meaningful. Whatever framework you use, be mindful that you still have a chance to log the exception and thread state of a failed thread to avoid tasks disappearing into the ether.

Throughput and deadlock jstacks

Another disadvantage of tools like jstack or Mission Control is that they need to be activated manually on a target JVM that is experiencing issues. This reduces their effectiveness in production, where when issues occur you're not there to debug 99% of the time.

Happily enough, there's a way you can activate jstack programmatically when your application's throughput falls under a specific threshold or meets a specific set of conditions. You can even automatically activate jstack when your JVM deadlocks, to see exactly which threads are deadlocking and what all the other threads are doing (of course, coupled with dynamic variable state for each one, courtesy of stateful thread names). This can be invaluable as deadlocking and concurrency issue are sporadic for the most part and notoriously hard to reproduce. In this case, activating jstack automatically at the moment of deadlock captured the stateful information for each thread, which can enhance your ability to reproduce and solve these kinds of bugs.

Takipi offers more about how to automatically detect deadlocks from within your application.

Capturing live variables

Capturing state from the JVM through thread contexts, however effective, is restricted to variables that you had to format into the thread name in advance.

```
"SQS-staging1_taskforce7_TSK_HITS 5 MsgID: AB5CAD, type: Analyze, queue: ACTIVE_PROD, TID: 5678956, TS:
11/8/20014 18:34" daemon prio=10 tid=0x00007f7fb4572000 nid=0x57b9 runnable [0x00007f7f94ebc000]
   java.lang.Thread.State: RUNNABLE
    at java.net.SocketInputStream.socketRead0(Native Method)
    at java.net.SocketInputStream.read(SocketInputStream.java:152)
    at java.net.SocketInputStream.read(SocketInputStream.java:122)
    at sun.security.ssl.InputRecord.readFully(InputRecord.java:442)
    at sun.security.ssl.InputRecord.read(InputRecord.java:480)
    at sun.security.ssl.SSLSocketImpl.readRecord(SSLSocketImpl.java:927)
    - locked <0x00000000fd0fd7e8> (a java.lang.Object)
    at sun.security.ssl.SSLSocketImpl.readDataRecord(SSLSocketImpl.java:884)
    at sun.security.ssl.AppInputStream.read(AppInputStream.java:102)
    - locked <0x00000000fd0fd048> (a sun.security.ssl.AppInputStream)
    at org.apache.http.impl.io.AbstractSessionInputBuffer.fillBuffer(AbstractSessionInputBuffer.java:160)
    at org.apache.http.impl.io.SocketInputBuffer.fillBuffer(SocketInputBuffer.java:84)
    at org.apache.http.impl.io.AbstractSessionInputBuffer.readLine(AbstractSessionInputBuffer.java:273)
    at org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:92)
    at org.apache.http.impl.conn.DefaultHttpResponseParser.parseHead(DefaultHttpResponseParser.java:62)
    at org.apache.http.impl.io.AbstractMessageParser.parse(AbstractMessageParser.java:260)
    at org.apache.http.impl.AbstractHttpClientConnection.receiveResponseHeader(AbstractHttpClientConnection.
```

Figure 1. An example of how an enhanced jstack shows dynamic variable state for each thread in the dump. Thread start time is marked as TS.

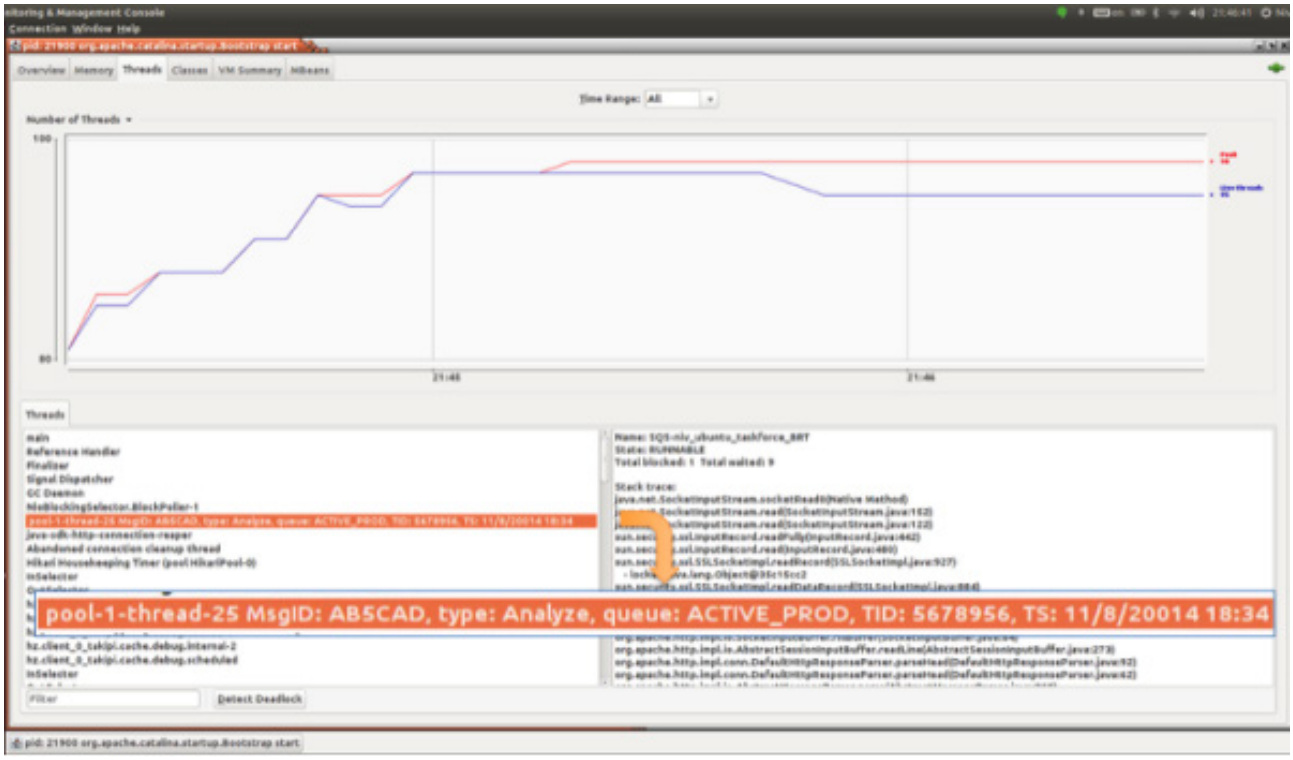


Figure 2. This thread variable state will also be shown by any JDK or commercial debugger or profiler.

Ideally, we want to be able to get the value of any variable from any point in the code from a live JVM without attaching a debugger or redeploying code. BTrace, a great tool that hasn't got the recognition it deserves, lets you do just that.

BTrace lets you run Java-like scripts on top of a live JVM to capture or aggregate any form of variable state without restarting the JVM or deploying new code. This lets you do pretty powerful things like printing the stack traces of threads, writing to a specific file, or printing the number of items of any queue or connection pool.

You do this with BTrace scripting, a Java-like syntax for functions that you inject into the code through bytecode transformation (a process we'll touch on below). The best way to try out the tool is to attach its sample scripts to a live application. Usage is straightforward: from your command line, simply enter `btrace <JVM pid> <script name>`. There's no need to restart your JVM.

BTrace is well documented and comes with many sample scripts to cover various common debugging scenarios around I/O, memory allocation, and class loading. Here are a couple of powerful examples:

- NewArray.java prints whenever a new `char[]` is allocated. You can add your own conditions based on its value. It's handy for selective memory profiling.

- FileTracker.java prints whenever the application writes to a specific file location. It's great for pinpointing the cause of excessive I/O operations.
- Classload.java reacts whenever a target class is loaded into the JVM. It's useful for debugging "JAR Hell" situations.

BTrace was designed as a non-intrusive tool, which means it cannot alter application state or flow control. That's a good thing, as it reduces the chance of negatively interfering with the execution of live code and makes its use in production much more acceptable. But this capability comes with some heavy restrictions: you can't create objects (not even `Strings`!), call into your own or third-party code (to perform actions such as logging), or even do simple things such as looping for fear of creating an infinite loop. To be able to do those, you'll have to write your own Java agent.

A Java agent is a JAR file that provides access by the JVM to an `Instrumentation` object to lets you modify bytecode that has already been loaded into the JVM to alter its behaviour. This essentially lets you "rewrite" code that has already been loaded and compiled by the JVM without restarting the application or changing the `.class` file on disk. Think about it like BTrace on steroids — you can essentially inject new code anywhere in your app, into both your own and third-party code, to capture any information you want.

The biggest downside to writing your own agent is that unlike BTrace, which lets you write Java-like scripts to capture state, Java agents operate at the bytecode level. This means that if you want to inject code into an application, you'll have to create the right bytecode. This can be tricky; bytecode can be hard to produce and read as it follows an operator-stack-like syntax that is similar in many ways to Assembly language. And to make things harder, since bytecode is already compiled, any miscorrelation to the location in which it is injected will be rejected without much fanfare by the JVM's verifier.

Bytecode generation libraries such as JavaAssist and ASM (which is my personal favorite) can assist with this. I often use the Bytecode Outline plugin, which automatically generates the right ASM code for any Java code you type in, then generates its equivalent bytecode, which you can copy and paste into your agent.

```
public class Hook {

    public static void onAllocation() {
        Monitor.onAllocation();
    }
```

Outline | Bytecode | Task List

```
com/sparktale/bugtale/meta/amagent/Hook
{
mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "onAllocation", "()V", null, null);
mv.visitCode();
Label l0 = new Label();
mv.visitLabel(l0);
mv.visitLineNumber(17, l0);
mv.visitMethodInsn(INVOKESTATIC, "com/sparktale/bugtale/meta/amagent/Monitor", "onAllocation", "
Label l1 = new Label();
mv.visitLabel(l1);
mv.visitLineNumber(18, l1);
mv.visitInsn(RETURN);
mv.visitMaxs(0, 0);
mv.visitEnd();
}
```

Figure 3. Dynamically generating bytecode generation scripts from your IDE using the ASM Bytecode Outline plugin.

Click here for a real-world example of a sample agent we used to detect and fix sporadic memory leaks coming from third-party code on our server, correlating it to application state. (Figure 3)

The last technique I'd like to touch on is building native JVM agents. This approach uses the JVM TI C++ API layer, which gives you unprecedented control of and access to the internals of the JVM. This includes things like getting callbacks whenever GC starts and stops, new threads spawn, monitors are acquired, and many more low-level events. This is by far the most powerful approach to acquire state from running code, as you are essentially running at the JVM level.

But with great power comes great responsibility, and some pretty complex challenges make this approach harder to implement. Since you're running at the JVM level you're no longer writing in cross-platform Java but in low-level, platform-dependent C++. A second disadvantage is that the APIs themselves, while extremely powerful, are hard to use and can significantly impact performance, depending on the specific set of capabilities you're consuming.

On the plus side, this layer provides terrific access to parts of the JVM that would otherwise be closed to you in your search for the root cause of production bugs. When we began writing Takipi for production debugging, I'm not sure we knew the extent to which TI would play a crucial role in our ability to build the tool. Through the use of this layer, you're able to detect exceptions, detect calls into the OS, and map application code without manual user instrumentation. If you have the time to take a look at this API layer, I highly recommend it, as it opens a window into the JVM not many of us use.

Java Bytecode: Bending the Rules

Rafael Winterhalter works as a software engineer in Oslo, Norway. He is a proponent of static typing and a JVM enthusiast with particular interests in code instrumentation, concurrency, and functional programming. Rafael blogs about software development, regularly presents at conferences, and was pronounced a JavaOne Rock Star. When coding outside of his workplace, he contributes to a wide range of open-source projects and often works on Byte Buddy, a library for simple runtime code generation for the Java virtual machine. For this work, Rafael received a Duke's Choice award.

When the javac compiler compiles a Java program, it translates the program not to executable machine code but rather yields Java bytecode, which serves as an intermediate format that describes a program to the Java virtual machine (JVM). Despite their similar names, the Java virtual machine has no notion of the Java programming language, but exclusively processes bytecode instructions.

One of the original intentions of Java bytecode was to reduce a Java program's size. As an emerging language in the fledgling days of the World Wide Web, Java required applets with a minimal download time. Thus, sending single bytes as instructions was preferred to transmitting human-readable words and symbols. But despite that translation, a Java program expressed as bytecode still largely resembles the original source code.

Over time, developers of languages besides Java created compilers to translate those languages into Java bytecode. Today, the list of language-to-Java-bytecode compilers is almost endless and nearly every programming language has become executable on the Java virtual machine. Moreover, the recent introduction of the `invokedynamic` bytecode instruction in Java 7 has increased the capabilities of efficiently running dynamic languages on the JVM.

Beyond the `invokedynamic` instruction, however, Java bytecode remains fairly Java specific. As of this writing, the set

of bytecode instructions largely reflects the feature set of the Java programming language. For example, any value in a Java program remains strictly typed, even though a dynamic JVM language might suggest the opposite. In addition, the virtual machine strictly enforces the object-oriented paradigm so any function of a JVM language that is not semantically associated with an object still needs to be represented as a method inside some class.

Nevertheless, the Java virtual machine grows increasingly polyglot. And despite its Java-centric nature, the virtual machine still supports the composition of bytecode instructions that a Java compiler would reject.

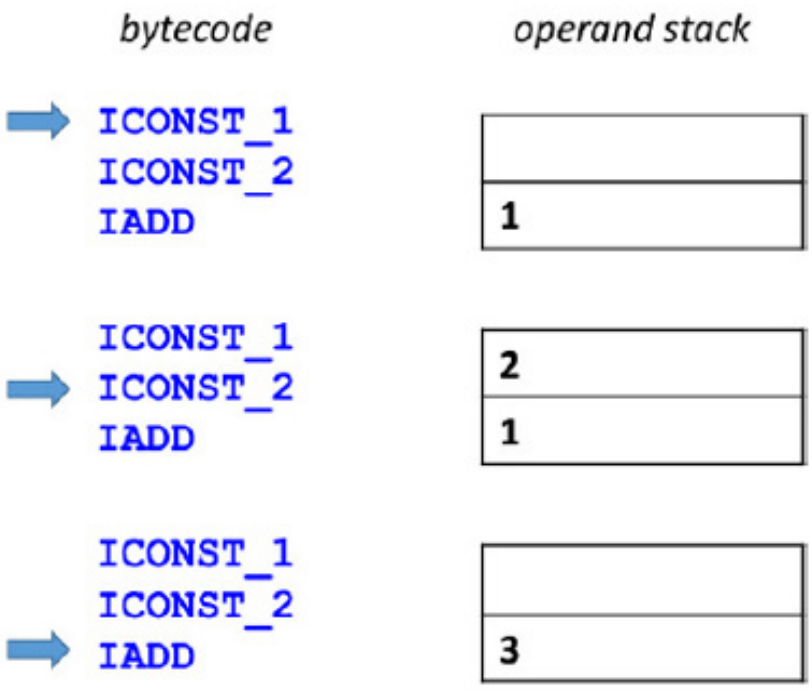


Figure 1

Java bytecode fundamentals

Few developers ever work with Java bytecode directly and at first glance this virtual-machine language might seem overly complicated. In fact, the bytecode format is quite trivial to understand. As its name suggests, the JVM represents a computer that normally exists not as actual hardware but only virtually as a program of its own. Java bytecode serves as the set of legal machine-code instructions for this virtual computer.

Every program maintains an internal stack of operands, and any bytecode instruction operates on that operand stack for the currently executing method. To process any values, those values must first be pushed onto the stack. Once on the stack, those values can serve as inputs to an operation. Suppose we want to add the numbers 1 and 2. Both values must first get pushed onto the operand stack by executing the bytecode instructions `iconst_1` and `iconst_2`. With these two numbers on the stack, the `iadd` instruction can now consume both values by popping them off the top of the stack. As a result, the instruction pushes the sum of the consumed values back onto the stack. (Figure 1)

Similarly, a method gets executed by first pushing its arguments onto the operand stack. When executing the method, the JVM then pops all arguments off the stack to hand them to the invoked method. Non-static methods additionally receive the instance of the invoked method as an implicit first argument. After returning, a non-void method finally pushes back its return value onto the operand stack.

To make sure that no method is invoked with incompatible arguments, the JVM executes a verifier whenever it loads new bytecode. Among other things, the verifier ensures that the virtual machine never arrives at an inconsistent state where an executed bytecode instruction expects different values than those found on the operand stack. If the verifier cannot guarantee such consistency, it rejects the loaded class and throws a `VerifyError`. The Java compiler would of course never create such bytecode. Nevertheless, implementers of other language compilers are equally bound by this static consistency check and must not generate bytecode that would fail the verifier's consistency check.

Partly erased types

However, the verifier does not audit all rules that are imposed by the Java programming language. A famous example for such a rule is the run-time erasure of generic types. When a generic type is translated during compilation, it is reduced to its most general boundary. Of course, this narrows the capabilities of the verifier when asserting bytecode for interacting with generic types. As these generic types were erased during compilation, the verifier can only assure assignability to the erasure of a generic type. Because this compromises the capability of the JVM to verify loaded code, the Java compiler explicitly warns about potentially unsafe usage of generic types.

Note that generic types are not fully erased but are embedded as meta information in a class file. Several frameworks extract this meta information via the reflection API and change their behavior ac-

cording to the discovered information. After all, generic types are only hidden from the JVM, not erased completely.

Unchecking checked exceptions

A lesser-known difference between the JVM and the Java programming language is the treatment of checked exceptions. For a checked exception, the Java compiler normally assures that it is either caught within a method or explicitly declared to be thrown. However, this is only a convention of the Java compiler and not a feature of the JVM. At run time, a checked exception can be thrown independently of any declaration.

By abusing the mentioned erasure of generic types, it is even possible to trick the Java compiler into throwing a checked exception. This can be accomplished by casting a checked exception to a run-time exception. To prevent this casting from producing a type error, it is conducted using generic types, which are removed when translating a method to bytecode. The following example demonstrates how to implement such a generic casting. (Code 1)

To throw such an exception in your code, you would just call the static `doThrow` method, supplying the root `Throwable`, without having to declare an explicit throws clause. The uncheck method is defined to throw a generic exception `T`, which the compiler must allow since the `T` generic parameter, being a subclass of `Throwable`, *might be* a `RuntimeException`.

Since the generic information is erased during compilation, the casting to `T` does not translate into a bytecode instruction. The Java compiler warns about this unsafe use of generics but here this warning is intentionally ignored. This unsafe operation can now be used to trick the compiler into "casting" any `Throwable` to a run-time exception, thereby obviating any explicit throws declaration.

```
001 static void doThrow(Throwable throwable) {
002   class Unchecker<T extends Throwable> {
003     @SuppressWarnings("unchecked")
004       private void uncheck(Throwable throwable)
      throws T {
005         throw (T) throwable;
006       }
007   }
008   new Unchecker<RuntimeException>().
      uncheck(throwable);
009 }
```

Code 1

```
001 void doSomething() throws Exception {
002   Arrays
003     .asList("foo", "bar")
004     .forEach((ExceptionConsumer<String>) s ->
      doSomethingWith(s));
005 }
006
007 void doSomethingWith(String s) throws Exception;
```

Code 2

```
001 interface ExceptionConsumer<T> extends Consumer<T> {
002
003   void doAccept(T t) throws Throwable;
004
005   @Override
006   default void accept(T t) {
007     try {
008       doAccept(t);
009     } catch (Throwable throwable) {
010       doThrow(throwable);
011     }
012   }
013 }
```

Code 3

At first glance, this might not seem very useful. However, unchecking checked exceptions can be an interesting option when dealing with lambda expressions. Most functional interfaces that ship with the Java class library do not declare checked exceptions. Therefore, checked exceptions need to always be caught within a lambda expression, preempting the benefits of concision generally associated with such functional expressions.

This is especially inconvenient if the method using a lambda expression intends to escalate a checked exception to its caller. To achieve such escalation, the checked exception needs to be wrapped inside of an unchecked exception. This wrapper exception needs then to be caught outside of the lambda such that the wrapped exception can be re-thrown. By unchecking the checked exception using the

trick from above, it is possible to largely avoid such boilerplate. (Code 2)

Given this helper interface that unchecks a checked exception, it is now possible to operate with methods that throw checked exceptions even inside of a lambda expressions. (Code 3)

As demonstrated in the above code, unchecking the checked exception allows it to propagate to the outer method. On the downside, the Java compiler is no longer able to verify that the lambda expression's outer method declares the checked exception. Therefore, unchecking checked exceptions needs to be handled with great care.

Return-type overloading

The Java programming language does not consider a method's return type to be part of that method's signature. Therefore, it is not possible to define two methods with the same name and parameter types within the same class even if they return values of different types. The rationale behind this decision is a situation in which a method is invoked for its side effect while ignoring its return value. In this case, the resolution of a method call would become ambiguous if a Java method was overloaded by its return type.

In Java bytecode, any method is instead identified by a signature that does include the method's return type, so a Java class file can include two methods that only differ in their return types. Consequently, bytecode must refer to a specific return type when invoking a method. For this reason, a Java program needs to be recompiled if the return type of an invoked method changes, even if the Java source code does not require any changes. Without this recompilation, the JVM is not able to link a call site to the changed method because the bytecode instruction no longer references the correct return type.

A hybrid type system

Because methods are referred to by their exact signature, a compiled Java class needs to preserve the types that are defined by the Java programming language. However, when executing a method, the JVM applies a slightly different type system for primitive types than the Java programming language. In a way, the JVM applies a hybrid type system with which it distinguishes types for describing values from types that exist during execution.

When loading a value onto the operand stack, the JVM treats primitive types that are smaller than an integer as if they were integers. Consequently, it makes little difference to a program's bytecode representation if a method variable is represented by, for example, a `short` instead of an `int`. Instead, the Java compiler inserts additional bytecode instructions that crop a value to the allowed range of a `short` when assigning values. Using shorts over integers in a method's body can therefore result in additional bytecode instructions rather than presumably optimizing a program.

This equalization does not apply however when storing values on the heap, either in the form of a field or as the value of an array. However, Boolean values still do not exist even on this level but are rather encoded as the single-byte values zero and one representing `false` and `true`. This is attributable to the fact that most hardware does not allow explicit access to a single bit, and so Boolean values are instead represented as bytes.

Breaking the constructor chain

The Java compiler requires any constructor to invoke another constructor as its first instruction, either implicitly or explicitly. To be valid, this invoked constructor must be declared by the same class or the direct superclass. Within Java bytecode, this restriction is only partially enforced. Instead, the JVM's verifier asserts that another valid constructor is called eventually. Additionally, it verifies that any method calls and field reads on the constructed instance occur only after this constructor was called. Other than that, it is perfectly legal to execute any code before invoking another constructor. It is even possible to write values to fields of the constructed instance before calling another constructor.

To make this rule less erratic, the JVM's type system includes a special type named undefined. In addition, the construction of an object is split into two separate bytecode instructions; the first instruction creates the undefined object and the second instruction completes the definition by calling a constructor on it. In bytecode, a constructor is represented as an instance method named `<init>`, which is also displayed in stack traces.

As long as an object is still considered undefined, the JVM's verifier forbids any meaningful interaction with the instance besides writing to its fields. Other than that, to the JVM, it is nothing more than calling an ordinary method. When fully deactivating the JVM's verifier, the virtual machine is even capable of not invoking a constructor or invoking constructors multiple times on the same instance.

Final-ish field

Another convention of constructors that the Java programming

```
001 class Foo {
002   final int bar;
003   Foo() {
004     this(43);
005     System.out.println(bar);
006     bar = 42;
007   }
008   Foo(int value) {
009     bar = value;
010   }
011 }
```

Code 4

```
001 void someMethod() {
002   invokedynamic[Bootstrapper::bootstrap]
003 }
004
005 class Bootstrapper {
006   static CallSite bootstrap(MethodHandles.Lookup
    lookup, Object... args) {
007     String name = new Random().nextBoolean() ?
    "foo": "bar";
008     MethodType methodType = MethodType.
    methodType(void.class);
009     return new ConstantCallSite(lookup.
    findStatic(Bootstrapper.class,
010                                 name, methodType));
011   }
012
013   static void foo() {
014     System.out.println("foo!");
015   }
016
017   static void bar() {
018     System.out.println("bar!");
019   }
020 }
```

Code 5

language strictly enforces is the one and only one-time assignment to final fields. A final field's value cannot be changed once it is assigned. At the same time, a final field must be defined within a constructor's body for a Java program to compile.

While the JVM is aware of final fields, the JVM's verifier enforces different rules. So it is legal in bytecode to reassign a final field any number of times as long as this reassignment is conducted within a constructor call of the class that declares the field. If a constructor invokes another constructor of the same class, it is even possible for the other constructor to reassign a final field. Thus, the following Java class can be legally expressed as bytecode. (Code 4)

After invoking the no-argument constructor of the above class, the only field is assigned the value 42 but the value 43 is printed to the console.

At the same time, it is also possible to skip assigning a value to a final field altogether. In this case, the field is initialized with a default value. Interestingly, such rules are not enforced at all by the bytecode verifier when dealing with static final fields. Static final fields can be reassigned arbitrarily within the declaring class.

Delaying compilation decisions until run time

The introduction of the `invokedynamic` instruction brought several major changes to the implementation of the JVM. Many Java developers did not perceive the introduction of this new bytecode instruction as significant, probably due to the inability to explicitly define dynamic method invocations using the Java programming language.

With explicit bytecode generation, it is however possible to use `invokedynamic` to delay the decision of selecting the invocation target of an arbitrary call site until its first execution. This is especially useful for dynamic languages when the information about which method should be invoked cannot be determined until the method is actually executed. At first glance, using `invokedynamic` can often appear equivalent to using Java's reflection API. By using dynamic method invocation, however, it is possible to explicitly link a method invocation to a call site. With Java 9, this linkage will yield a stack trace that completely hides the dynamic invocation. Besides allowing for more efficient treatment of primitive types, `invokedynamic` also allows for better handling of security as the JVM only allows binding of invocation targets that are visible to the class that declares a dynamic call site.

Whenever an `invokedynamic` call site is created in

bytecode, it references a so-called bootstrap method for deciding what method should be invoked. A bootstrap method is implemented in plain Java and serves as a lookup routine for locating the method that should be called for the `invokedynamic` instruction. This lookup is only performed once, but it is possible to manually rebind a dynamic call site at a later time.

By using some pseudo syntax, an `invokedynamic` instruction can implement a call site that either calls a method `foo` or `bar`, depending on a randomized state. (Code 5)

When the method containing the `invokedynamic` instruction is called for the first time, the referenced bootstrap method is invoked. As a first argument, it receives a lookup object for looking up methods that are visible to the class enclosing an `invokedynamic` call site. Depending on the outcome of the randomized name assignment, the `invokedynamic` call site is either bound to invoking `foo` or `bar` by returning the appropriate method. After binding the method, the bootstrap method is never again called for the same call site. At the same time, the `invokedynamic` call site is from then on handled as if the method call were hard-coded to the formerly dynamic call site.

Secrets of the Bytecode Ninjas

Ben Evans is the CEO of jClarity, a Java/JVM performance-analysis startup. In his spare time he is one of the leaders of the London Java Community and holds a seat on the Java Community Process Executive Committee. His previous projects include performance testing the Google IPO, financial trading systems, writing award-winning websites for some of the biggest films of the '90s, and others.

The Java language is defined by the Java Language Specification (JLS). The executable bytecode of the Java virtual machine (JVM), however, is defined by a separate standard, the Java Virtual Machine Specification (usually referred to as the VMSpec).

JVM bytecode is produced by javac from Java source-code files, and the bytecode is significantly different from the language. For example, some familiar high-level Java-language features have been compiled away and don't appear in the bytecode at all.

One of the most obvious examples of this would be Java's loop keywords (for, while, etc.), which are compiled away and replaced with bytecode branch instructions. This means that bytecode's flow control inside a method consists only of if statements and jumps (for looping).

This article assumes that the reader has a grounding in bytecode. If you require some background, see *The Well-Grounded Java Developer* (Evans and Verburg, Manning 2012) or this report from RebelLabs (signup required for PDF).

Let's look at an example that often puzzles developers who are new to JVM bytecode. It uses the javap tool, which ships with the JDK or JRE and which is effectively a Java bytecode disassembler. In our example, we will discuss a simple class that implements the Callable interface:

```
001 public class
      ExampleCallable
      implements
      Callable<Double> {
002     public Double
      call() {
003         return
      3.1415;
004     }
005 }
```

We can disassemble this as shown, using the simplest form of the javap tool. (Code 1)

```
001 $ javap kathik/java/bytecode_examples/
    ExampleCallable.class
002 Compiled from "ExampleCallable.java"
003 public class kathik.java.bytecode_examples.
    ExampleCallable
004         implements java.util.concurrent.Callable<java.
    lang.Double> {
005   public kathik.java.bytecode_examples.
    ExampleCallable();
006   public java.lang.Double call();
007   public java.lang.Object call() throws java.lang.
    Exception;
008 }
```

Code 1

This disassembly looks wrong — after all, we wrote one call method, not two. Even if we had tried to write it as such, javac would have complained that there were two methods with the same name and signature that differ only in return type, and so the code would not have compiled. Nevertheless, this class was generated from the real, valid Java source file shown above.

This clearly shows that Java's familiar ambiguous return-type restriction is a Java language constraint rather than a JVM bytecode requirement. If the thought of javac inserting code that you didn't write into your class files is troubling, it shouldn't be: we see it every day! One of the first lessons a Java programmer learns is that if you don't provide a constructor, the compiler adds a simple one for you. In the output from javap, you can see the constructor provided even though we didn't write it.

These additional methods illustrate how the requirements of the language spec are stricter than the details of the VM spec. There are a number of "impossible" things that can be done if we write bytecode directly — legal bytecode that no Java compiler will ever emit.

For example, we can create classes with genuinely no constructor. The Java language spec requires that every class has at least one constructor, and javac will insert a simple void constructor automatically if we fail to provide one. If we write bytecode directly, however, we are free to omit one. Such a class could not be instantiated, even via reflection.

Our final example is one that almost works, but not quite. In bytecode, we can write a method that attempts to call a private method belonging to another class. This is valid bytecode, but it will fail to link correctly if any program attempts to load it. This is because the class loader's verifier will detect the access control restrictions on the call and reject the illegal access.

Introduction to ASM

If we want to create code that can implement some of these non-Java behaviours, then we will need to produce a class file from scratch. As the class file format is binary, it makes sense to use a library that enables us to manipulate an abstract data structure, then convert it to bytecode and stream it to disc.

There are several such libraries to choose from but we will focus on ASM. This is a common library that appears (in slightly modified form) in the Java 8 distribution as an internal API. For user code, we want to use the general open-source library instead of the JDK's version, as we should not rely upon internal APIs.

The core focus of ASM is to provide an API that while somewhat arcane (and occasionally crufty), corresponds to the bytecode data structures in a fairly direct way.

The Java runtime is the result of design decisions made over a number of years, and you can clearly see the resulting accretion in successive versions of the class file format.

ASM seeks to model the class file fairly closely, and so the basic API breaks down into a number of fairly simple sections for methods (although ones that model binary concerns).

If you wish to create a class file from scratch, you need to understand the overall structure of a class file, and this does change over time. Fortunately, ASM handles the slight differences in class file format among Java versions, and the strong compatibility requirements of the Java platform also help.

In order, a class file contains:

- magic number (in the traditional Unix sense - Java's magic number is the rather dated and sexist 0xCAFEBABE),
- version number of the class file format in use,
- constant,
- access-control flags (e.g. is the class public, protected, package access, etc.),
- type name of this class,
- superclass of this class,
- interfaces that this class implements,
- fields that this class possesses (over and above those of superclasses),
- methods that this class possesses (over and above those of superclasses), and
- attributes (class-level annotations).

The main sections of a JVM class file can be recalled using the following mnemonic: (Figure 1)

ASM offers two APIs, and the easiest to use relies heavily upon the visitor pattern. In a common

Figure 1

formulation, ASM starts from a blank slate, with the `ClassWriter`. (When getting used to working with ASM and direct bytecode manipulation, many developers find the `CheckClassAdapter` a useful starting point — this is a `ClassVisitor` that checks its methods in a similar manner to the verifier that appears in Java's classloading subsystem.)

Let's look at some simple class-generation examples that follow a common pattern:

- Spin up a `ClassVisitor` (in our cases, a `ClassWriter`).
- Write the header.
- Generate methods and constructors as needed.
- Convert the `ClassVisitor` to a byte array and write it out.

Examples

```
001 public class Simple implements ClassGenerator {
002   // Helpful constants
003   private static final String GEN_CLASS_NAME = "GetterSetter";
004   private static final String GEN_CLASS_STR = PKG_STR + GEN_CLASS_NAME;
005
006   @Override
007   public byte[] generateClass() {
008     ClassWriter cw = new ClassWriter(0);
009     CheckClassAdapter cv = new CheckClassAdapter(cw);
010     // Visit the class header
011     cv.visit(V1_7, ACC_PUBLIC, GEN_CLASS_STR, null, J_L_O, new String[0]);
012     generateGetterSetter(cv);
013     generateCtor(cv);
014     cv.visitEnd();
015     return cw.toByteArray();
016   }
017
018   private void generateGetterSetter(ClassVisitor cv) {
019     // Create the private field myInt of type int. Effectively:
020     // private int myInt;
021     cv.visitField(ACC_PRIVATE, "myInt", "I", null, 1).visitEnd();
022
023     // Create a public getter method
024     // public int getMyInt();
025     MethodVisitor getterVisitor =
026         cv.visitMethod(ACC_PUBLIC, "getMyInt", "()I", null, null);
```

```
027     // Get ready to start writing out the bytecode for the method
028     getterVisitor.visitCode();
029     // Write ALOAD_0 bytecode (push the this reference onto stack)
030     getterVisitor.visitVarInsn(ALOAD, 0);
031     // Write the GETFIELD instruction, which uses the instance on
032     // the stack (& consumes it) and puts the current value of the
033     // field onto the top of the stack
034     getterVisitor.visitFieldInsn(GETFIELD, GEN_CLASS_STR, "myInt", "I");
035     // Write IRETURN instruction - this returns an int to caller.
036     // To be valid bytecode, stack must have only one thing on it
037     // (which must be an int) when the method returns
038     getterVisitor.visitInsn(IRETURN);
039     // Indicate the maximum stack depth and local variables this
040     // method requires
041     getterVisitor.visitMaxs(1, 1);
042     // Mark that we've reached the end of writing out the method
043     getterVisitor.visitEnd();
044
045     // Create a setter
046     // public void setMyInt(int i);
047     MethodVisitor setterVisitor =
048         cv.visitMethod(ACC_PUBLIC, "setMyInt", "(I)V", null, null);
049     setterVisitor.visitCode();
050     // Load this onto the stack
051     setterVisitor.visitVarInsn(ALOAD, 0);
052     // Load the method parameter (which is an int) onto the stack
053     setterVisitor.visitVarInsn(ILOAD, 1);
054     // Write the PUTFIELD instruction, which takes the top two
055     // entries on the execution stack (the object instance and
056     // the int that was passed as a parameter) and set the field
057     // myInt to be the value of the int on top of the stack.
058     // Consumes the top two entries from the stack
059     setterVisitor.visitFieldInsn(PUTFIELD, GEN_CLASS_STR, "myInt", "I");
060     setterVisitor.visitInsn(RETURN);
061     setterVisitor.visitMaxs(2, 2);
062     setterVisitor.visitEnd();
063   }
064
065   private void generateCtor(ClassVisitor cv) {
066     // Constructor bodies are methods with special name <init>
067     MethodVisitor mv =
068         cv.visitMethod(ACC_PUBLIC, INST_CTOR, VOID_SIG, null, null);
069     mv.visitCode();
070     mv.visitVarInsn(ALOAD, 0);
071     // Invoke the superclass constructor (we are basically
072     // mimicing the behaviour of the default constructor
073     // inserted by javac)
074     // Invoking the superclass constructor consumes the entry on the top
075     // of the stack.
076     mv.visitMethodInsn(INVOKESPECIAL, J_L_O, INST_CTOR, VOID_SIG);
077     // The void return instruction
078     mv.visitInsn(RETURN);
079     mv.visitMaxs(2, 2);
080     mv.visitEnd();
081   }
082
083   @Override
084   public String getGenClassName() {
085     return GEN_CLASS_NAME;
086   }
087 }
```

This uses a simple interface with a single method to generate the bytes of the class, a helper method to return the name of the generated class, and some useful constants.

```
interface ClassGenerator {
public byte[] generateClass();

public String getGenClassName();

// Helpful constants
public static final String PKG_STR = “kathik/java/bytecode_examples/”;
public static final String INST_CTOR = “<init>”;
public static final String CL_INST_CTOR = “<clinit>”;
public static final String J_L_O = “java/lang/Object”;
public static final String VOID_SIG = “()V”;
}
```

To drive the classes we'll generate, we use a harness, called `Main`. This provides a simple class loader and a reflective way to call back onto the methods of the generated class. For simplicity, we also write out our generated classes into the Maven target directory into the right place to be picked up on the IDE's classpath.

```
public class Main {
public static void main(String[] args) {
   Main m = new Main();
   ClassGenerator cg = new Simple();
   byte[] b = cg.generateClass();
   try {
     Files.write(Paths.get(“target/classes/” + PKG_STR +
       cg.getGenClassName() + “.class”), b, StandardOpenOption.CREATE);
   } catch (IOException ex) {
     Logger.getLogger(Simple.class.getName()).log(Level.SEVERE, null, ex);
   }
   m.callReflexive(cg.getGenClassName(), “getMyInt”);
}
```

The following class just provides a way to get access to the protected `defineClass()` method so we can convert a `byte[]` into a class object for reflective use.

```
private static class SimpleClassLoader extends ClassLoader {
 public Class<?> simpleDefineClass(byte[] clazzBytes) {
   return defineClass(null, clazzBytes, 0, clazzBytes.length);
 }
}

private void callReflexive(String typeName, String methodName) {
 byte[] buffy = null;
 try {
   buffy = Files.readAllBytes(Paths.get(“target/classes/” + PKG_STR +
     typeName + “.class”));
   if (buffy != null) {
     SimpleClassLoader myCl = new SimpleClassLoader();
     Class<?> newClz = myCl.simpleDefineClass(buffy);
     Object o = newClz.newInstance();
     Method m = newClz.getMethod(methodName, new Class[0]);
     if (o != null && m != null) {
       Object res = m.invoke(o, new Object[0]);
       System.out.println(“Result: “ + res);
     }
   }
 } catch (IOException | InstantiationException | IllegalAccessException |
         NoSuchMethodException | SecurityException |
         IllegalArgumentException | InvocationTargetException ex) {
   Logger.getLogger(Simple.class.getName()).log(Level.SEVERE, null, ex);
 }
}
```

This setup makes it easy, with minor modifications, to test different class generators to explore different aspects of bytecode generation.

The non-constructor class is similar. For example, here is a how to generate a class that has a single static field with getters and setters for it (this generator has no call to `generateCtor()`):

```
001 private void generateStaticGetterSetter(ClassVisitor cv) {
002 // Generate the static field
003   cv.visitField(ACC_PRIVATE | ACC_STATIC, “myStaticInt”, “I”, null,
004      1).visitEnd();
005
006   MethodVisitor getterVisitor = cv.visitMethod(ACC_PUBLIC | ACC_STATIC,
007                                       “getMyInt”, “()I”, null, null);
008   getterVisitor.visitCode();
009   getterVisitor.visitFieldInsn(GETSTATIC, GEN_CLASS_STR, “myStaticInt”, “I”);
010
011   getterVisitor.visitInsn(IRETURN);
012   getterVisitor.visitMaxs(1, 1);
013   getterVisitor.visitEnd();
014
015   MethodVisitor setterVisitor = cv.visitMethod(ACC_PUBLIC | ACC_STATIC, “setMyInt”,
016                                       “(I)V”, null, null);
017   setterVisitor.visitCode();
018   setterVisitor.visitVarInsn(ILOAD, 0);
019   setterVisitor.visitFieldInsn(PUTSTATIC, GEN_CLASS_STR, “myStaticInt”, “I”);
020
021 }
022
023 setterVisitor.visitInsn(RETURN);setterVisitor.visitMaxs(2,2);setterVisitor.
     visitEnd();
```

Note how the methods are generated with the `ACC_STATIC` flag set and how the method arguments are first in the local variable list (as implied by the `ILOAD 0` pattern — in an instance method, this would be `ILOAD 1`, as the “this” reference would be stored at the 0 offset in the local variable table).

Using javap, we can confirm that this class genuinely has no constructor.

```
001 $ javap -c kathik/java/bytecode_examples/StaticOnly.class
002 public class kathik.StaticOnly {
003 public static int getMyInt(); Code:
004 0: getstatic    #11                 // Field myStaticInt:I
005 3: ireturn
006
007 public static void setMyInt(int); Code:
008 0: iload_0
009 1: putstatic    #11                 // Field myStaticInt:I
010 4: return
011 }
```

Working with generated classes

Until now, we have worked reflexively with the classes we’ve generated via ASM. This helps to keep the examples self-contained, but in many cases we want to use the generated code with regular Java files. This is easy enough to do. The examples helpfully place the generated classes into the Maven target directory.

```
001 $ cd target/classes
002 $ jar cvf gen-asm.jar kathik/java/bytecode_examples/GetterSetter.class kathik/java/
     bytecode_examples/StaticOnly.class
003 $ mv gen-asm.jar ../../lib/gen-asm.jar
```

Now we have a JAR file that can be used as a dependency in some other code. For example, we can use our `GetterSetter` class:

```
001 import kathik.java.bytecode_examples.GetterSetter;
002 public class UseGenCodeExamples {
003  public static void main(String[] args) {
004    UseGenCodeExamples ugcx = new UseGenCodeExamples();
005    ugcx.run();
006  }
007
008  private void run() {
009    GetterSetter gs = new GetterSetter();
010    gs.setMyInt(42);
011    System.out.println(gs.getMyInt());
012  }
013 }
```

This won't compile in the IDE as the `GetterSetter` class is not on the classpath. However, if we drop down to the command line and supply the appropriate dependency on the classpath, everything works fine.

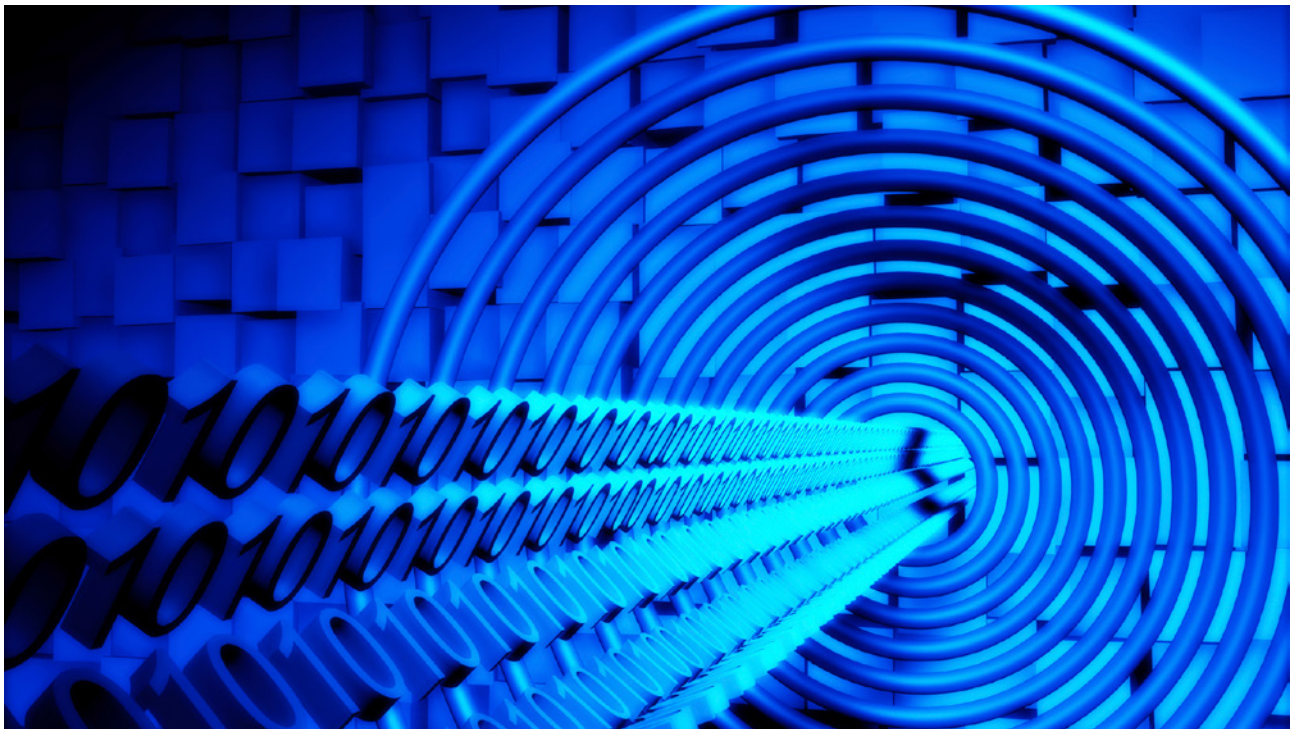
```
001 $ cd ../../src/main/java/
002 $ javac -cp ../../../lib/gen-asm.jar kathik/java/bytecode_examples/withgen/
     UseGenCodeExamples.java
003 $ java -cp .:../../../lib/gen-asm.jar kathik.java.bytecode_examples.withgen.
     UseGenCodeExamples
004 42
```

Conclusion

In looking at the basics of generating class files from scratch, using the simple API from the ASM library, we've seen some of the differences between the requirements of Java language and bytecode, and seen that some of the rules of Java are actually only conventions from the language that the runtime does not enforce. We've also seen that you can use a correctly written class file directly from the language, just as though it had been produced by javac. This is the basis of Java's interoperability with non-Java languages, such as Groovy or Scala.

There are a number of much more advanced techniques available, but this should get you started with deeper investigations of the JVM runtime and how it operates.

Java's secret weapon: `invokedynamic`



by Ben Evans

The release of Java 7 contained several new features that seemed at first sight to be of limited use to Java developers (see InfoQ's "Java 7 Features Which Enable Java 8").

One feature in particular, invokedynamic, proved crucial in implementing the headline features of Java 8 (such as lambdas and default methods). It is a powerful tool for the Java platform, and for JVM languages such as JRuby and Nashorn.

The earliest work on `invokedynamic` goes back to at least 2007 with the first successful dynamic invocation taking place August 26, 2008. This predates Oracle's acquisition of Sun and shows that this feature has been in progress for a long time, by the standards of most developers.

`invokedynamic` is remarkable in that it was the first new bytecode since Java 1.0. It joined the existing invoke bytecodes `invokevirtual`, `invokestatic`, `invokeinterface`, and `invokespecial`. These four existing opcodes implement all forms of method dispatch that Java developers are typically familiar with, specifically:

1. `invokevirtual` is the standard dispatch for instance methods.
2. `invokestatic` is used to dispatch static methods.
3. `invokeinterface` is used to dispatch a method call via an interface.
4. `invokespecial` is used when non-virtual (i.e. "exact") dispatch is required.

Some developers may be curious as to why the platform requires all four opcodes, so let's look at a simple example that uses the different invoke opcodes, to demonstrate the difference between them.

```
001 public class InvokeExamples {
002     public static void main(String[] args) {
003         InvokeExamples sc = new InvokeExamples();
004         sc.run();
005     }
006
007     private void run() {
008         List<String> ls = new ArrayList<>();
009         ls.add(“Good Day”);
010
011         ArrayList<String> als = new ArrayList<>();
012         als.add(“Dydh Da”);
013     }
014 }
```

This produces bytecode that we can disassemble with the javap tool.

```
001 javap -c InvokeExamples.class
002
003 public class kathik.InvokeExamples {
004   public kathik.InvokeExamples();
005     Code:
006        0: aload_0
007        1: invokespecial #1                  // Method java/lang/Object.”<init>”:()V
008        4: return
009
010   public static void main(java.lang.String[]);
011     Code:
012        0: new           #2                  // class kathik/InvokeExamples
013        3: dup
014        4: invokespecial #3                  // Method “<init>”:()V
015        7: astore_1
016        8: aload_1
017        9: invokespecial #4                  // Method run:()V
018       12: return
019
020   private void run();
021     Code:
022        0: new           #5                  // class java/util/ArrayList
023        3: dup
024        4: invokespecial #6                  // Method java/util/
     ArrayList.”<init>”:()V
025        7: astore_1
026        8: aload_1
027        9: ldc           #7                  // String Good Day
028       11: invokeinterface #8,  2            // InterfaceMethod java/util/List.
     add:(Ljava/lang/Object;)Z
029       16: pop
030       17: new           #5                  // class java/util/ArrayList
031       20: dup
032       21: invokespecial #6                  // Method java/util/
     ArrayList.”<init>”:()V
033       24: astore_2
034       25: aload_2
035       26: ldc           #9                  // String Dydh Da
036       28: invokevirtual #10                 // Method java/util/ArrayList.
     add:(Ljava/lang/Object;)Z
037       31: pop
038       32: return
039 }
```

This showcases three of the four invoke opcodes (and the remaining one, `invokestatic`, is a fairly trivial extension). To start with, we can see that the two calls (at bytes 11 and 28 in the run method):

```
ls.add("Good Day")
```

and

```
als.add("Dydh Da")
```

look very similar in Java source code but are actually represented differently in bytecode.

To javac, the variable `ls` has the static type of `List<String>`, and `List` is an interface. The precise location in the runtime method table (usually called a "vtable") of the `add()` method has not yet been determined at compile time. The source-code compiler therefore emits an `invokeinterface` instruction and defers the actual lookup of the method until run time, when the real vtable of `ls` can be inspected, and the location of the `add()` method found.

By contrast, the call `als.add("Dydh Da")` is received by `als`, and the static type of this is a class type: `ArrayList<String>`. This means that the location of the method in the vtable is known at compile time. javac is therefore able to emit an `invokevirtual` instruction for the exact vtable entry. The final choice of method will still be determined at run time to allow for the possibility of method overriding, but the vtable slot has been determined at compile time.

This example also shows two of the possible use cases of `invokespecial`. This opcode when dispatch should be exactly determined at run time and, in particular, method overriding is neither desired nor possible. The two cases the example demonstrates are private methods and super calls, because such methods are known at compile time and cannot be overridden.

The astute reader will have noticed that all calls to Java methods are compiled to one of these four opcodes, raising the questions of what `invokedynamic` is for and why it is useful to Java developers.

The feature's main goal was to create a bytecode to handle a new type of method dispatch, essentially allowing application-level code to determine which method a call will execute and to do so only when the call is about to execute. This allows language and framework writers to apply much more dynamic programming styles than the Java platform previously supported.

The intent is that user code determines dispatch at run time using the method handles API whilst not suffering the performance penalties and security problems associated with reflection. In fact, the stated aim of `invokedynamic` is to be as fast as regular method dispatch (`invokevirtual`) once the feature has sufficiently matured.

When Java 7 arrived, the JVM supported the execution of the new bytecode but, regardless of the Java code submitted, javac would never produce bytecode that included `invokedynamic`. Instead, the feature was used to support JRuby and other dynamic languages running on the JVM.

This changed in Java 8, which generates `invokedynamic` and uses it under the hood to implement lambda expressions and default methods and as the primary dispatch mechanism for Nashorn. However, there is still no direct way for Java-application developers to do fully dynamic method resolution. That is, the Java language does not have a keyword or library that will create general-purpose `invokedynamic` call sites. This means that the mechanism remains obscure to most Java developers despite its power.

Introduction to method handles

A key concept in `invokedynamic` operation is the method handle. This is a way to represent the method that should be called from the `invokedynamic` call site. Each `invokedynamic` instruction is associated with a special method known as a bootstrap method (BSM). When the interpreter reaches the `invokedynamic` instruction, it calls the BSM, which returns an object (containing a method handle) that indicates which method the call site should execute.

This is somewhat similar to reflection, but reflection has limitations that made it unsuitable for use with `invokedynamic`. Instead, `java.lang.invoke.MethodHandle` (and subclasses) were added to the Java 7 API to represent methods that `invokedynamic` can target. The class `MethodHandle` receives some special treatment from the JVM in order to operate correctly.

One way to think of method handles is as core reflection done in a safe, modern way with maximum possible type safety. They are needed for `MethodHandle` but can also be used alone.

Method types

A Java method can be considered to be made up of four basic pieces:

- name,
- signature (including return type),
- class where it is defined, and
- bytecode that implements the method

If we want to refer to methods, we need an efficient way to represent method signatures, rather than using the horrible `Class<?>[]` hacks that reflection is forced to use.

To put it another way, one of the first building blocks needed for method handles is a way to represent method signatures for lookup. In the Method Handles API, introduced in Java 7, this role is fulfilled by the `java.lang.invoke.MethodType` class, which uses immutable instances to represent signatures. To obtain a `MethodType`, use the `methodType()` factory method. This is a variadic method that takes class objects as arguments.

The first argument is the class object that corresponds to the signature's return type; the remaining arguments are the class objects that correspond to the types of the method arguments in the signature. For example:

```
001 // Signature of toString()
002 MethodType mtToString = MethodType.methodType(String.class);
003
004 // Signature of a setter method
005 MethodType mtSetter = MethodType.methodType(void.class, Object.
      class);
006
007 // Signature of compare() from Comparator<String>
008 MethodType mtStringComparator = MethodType.methodType(int.class,
      String.class, String.class);
```

You can now use the `MethodType`, along with the name and the class that defines the method, to look up the method handle. To do this, we need to call the static `MethodHandles.lookup()` method. This gives us a lookup context that is based on the access rights of the currently executing method (i.e. the method that called `lookup()`).

The lookup-context object has a number of methods that have names that start with "find", e.g. `findVirtual()`, `findConstructor()`, and `findStatic()`. These methods will return the actual method handle, but only if the lookup context was created in a method that could access (call) the requested method. Unlike reflection, there is no way to subvert this access control. In other words, method handles have no equivalent of the `setAccessible()` method. For example:

```
001 public MethodHandle getToStringMH() {
002     MethodHandle mh = null;
003     MethodType mt = MethodType.methodType(String.class);
004     MethodHandles.Lookup lk = MethodHandles.lookup();
005
006     try {
007         mh = lk.findVirtual(getClass(), "toString", mt);
008     } catch (NoSuchMethodException | IllegalAccessException mhx) {
009         throw (AssertionError)new AssertionError().initCause(mhx);
010     }
011
012     return mh;
013 }
```

There are two methods on `MethodHandle` that can be used to invoke a method handle `invoke()` and `invokeExact()`. Both methods take the receiver and call arguments as parameters, so the signatures are:

```
001 public final Object invoke(Object... args) throws Throwable;
002 public final Object invokeExact(Object... args) throws
      Throwable;
```

The difference between the two is that `invokeExact()` tries to call the method handle directly with the precise arguments provided. On the other hand, `invoke()` has the ability to slightly alter the method arguments if needed. `invoke()` performs an `asType()` conversion, which can convert arguments according to this set of rules:

- Primitives will be boxed if required.
- Boxed primitives will be unboxed if required.
- Primitives will be widened if necessary.
- A void return type will be converted to 0 (for primitive return types) or null for return types that expect a reference type.
- Null values are assumed to be correct and passed through regardless of static type.

Let's look at a simple invocation example that takes these rules into account.

```
001 Object rcvr = "a";
002 try {
003     MethodType mt = MethodType.methodType(int.class);
004     MethodHandles.Lookup l = MethodHandles.lookup();
005     MethodHandle mh = l.findVirtual(rcvr.getClass(), "hashCode", mt);
006
007     int ret;
008     try {
009         ret = (int)mh.invoke(rcvr);
010         System.out.println(ret);
011     } catch (Throwable t) {
012         t.printStackTrace();
013     }
014 } catch (IllegalArgumentException | NoSuchMethodException | SecurityException e) {
015     e.printStackTrace();
016 } catch (IllegalAccessException x) {
017     x.printStackTrace();
018 }
```

In more complex examples, method handles can provide a much clearer way to perform the same dynamic programming tasks as core reflection. Not only that, method handles have been designed from the start to work better with the low-level execution model of the JVM and potentially can provide much better performance (although the performance story is still unfolding).

Method handles and invokedynamic

`invokedynamic` uses method handles via the BSM mechanism. Unlike `invokevirtual`, `invokedynamic` instructions have no receiver object. Instead, they act like `invokestatic`, and use a BSM to return an object of type `CallSite`. This object contains a method handle (called the "target") that represents the method to be executed as the result of the `invokedynamic` instruction.

When a class containing `invokedynamic` is loaded, the call sites are said to be in an "unlaced" state, and after the BSM returns, the resulting `CallSite` and method handle are said to be "laced" into the call site.

The signature for a BSM looks like this (note that the BSM can have any name):

```
001 static CallSite
      bootstrap(MethodHandles.Lookup
      caller, String name, MethodType
      type);
```

If we want to create code that contains `invokedynamic`, we will need to use a bytecode-manipulation library (as the Java language doesn't contain the constructs we need). The code examples in the rest of this article use the ASM library to generate bytecode that includes `invokedynamic` instructions.

Here's an ASM-based class for creating a "Hello World" using invokedynamic.

```
001 public class InvokeDynamicCreator {
002
003     public static void main(final String[] args) throws Exception {
004         final String outputClassName = "kathik/Dynamic";
005         try (FileOutputStream fos
006                 = new FileOutputStream(new File("target/classes/" + outputClassName
    + ".class"))) {
007             fos.write(dump(outputClassName, "bootstrap", "()V"));
008         }
009     }
010
011     public static byte[] dump(String outputClassName, String bsmName, String
    targetMethodDescriptor)
012             throws Exception {
013         final ClassWriter cw = new ClassWriter(0);
014         MethodVisitor mv;
015
016         // Setup the basic metadata for the bootstrap class
017         cw.visit(V1_7, ACC_PUBLIC + ACC_SUPER, outputClassName, null, "java/lang/
    Object", null);
018
019         // Create a standard void constructor
020         mv = cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
021         mv.visitCode();
022         mv.visitVarInsn(ALOAD, 0);
023         mv.visitMethodInsn(INVOKESPECIAL, "java/lang/Object", "<init>", "()V");
024         mv.visitInsn(RETURN);
025         mv.visitMaxs(1, 1);
026         mv.visitEnd();
027
028         // Create a standard main method
029         mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main", "([Ljava/lang/String;)
    V", null, null);
030         mv.visitCode();
031         MethodType mt = MethodType.methodType(CallSite.class, MethodHandles.Lookup.
    class, String.class,
032                 MethodType.class);
033         Handle bootstrap = new Handle(Opcodes.H_INVOKESTATIC, "kathik/
    InvokeDynamicCreator", bsmName,
034                 mt.toMethodDescriptorString());
035         mv.visitInvokeDynamicInsn("runDynamic", targetMethodDescriptor, bootstrap);
036         mv.visitInsn(RETURN);
037         mv.visitMaxs(0, 1);
038         mv.visitEnd();
039
040         cw.visitEnd();
041
042         return cw.toByteArray();
043     }
044
045     private static void targetMethod() {
046         System.out.println("Hello World!");
047     }
048
049     public static CallSite bootstrap(MethodHandles.Lookup caller, String name,
    MethodType type) throws NoSuchMethodException, IllegalAccessException {
050         final MethodHandles.Lookup lookup = MethodHandles.lookup();
051         // Need to use lookupClass() as this method is static
```

```
052         final Class<?> currentClass = lookup.lookupClass();
053         final MethodType targetSignature = MethodType.methodType(void.class);
054         final MethodHandle targetMH = lookup.findStatic(currentClass, "targetMethod",
      targetSignature);
055         return new ConstantCallSite(targetMH.asType(type));
056     }
057 }
```

From the point of view of Java applications, these appear as regular class files (although they, of course, have no possible Java source code representation). Java code treats them as black boxes that we can nonetheless call methods on and make use of `invokedynamic` and related functionality.

Here's an ASM-based class for creating a "Hello World" using `invokedynamic`. (See next page)

The code is divided into two sections, the first of which uses the ASM Visitor API to create a class file called `kathik.Dynamic`. Note the key call to `visitInvokeDynamicInsn()`. The second section contains the target method that will be laced into the call site and the BSM that the `invokedynamic` instruction needs.

Note that these methods are within the `InvokeDynamicCreator` class and not part of our generated `kathik.Dynamic` class. This means that at run time, `InvokeDynamicCreator` must also be on the classpath with `kathik.Dynamic` for the method to be found.

When `InvokeDynamicCreator` runs, it creates a new class file, `Dynamic.class`, which contains an `invokedynamic` instruction, as we can see by using javap on the class.

```
001 public static void main(java.lang.String[]);
002     descriptor: ([Ljava/lang/String;)V
003     flags: ACC_PUBLIC, ACC_STATIC
004     Code:
005       stack=0, locals=1, args_size=1
006          0: invokedynamic #20,  0              // InvokeDynamic #0:runDynamic:()V
007          5: return
```

This example shows the simplest case of `invokedynamic`, which uses the special case of a constant `CallSite` object. This means that the BSM (and lookup) is done only once and so subsequent calls are fast.

More sophisticated usages of `invokedynamic` can quickly get complex, especially when the call site and target method can change during the lifetime of the program.

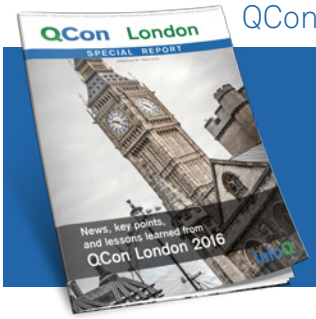
PREVIOUS ISSUES

41

#noprojects

#NoProjects – a number of authors have challenged the idea of the project as a delivery mechanism for information technology product development. The two measures of success and goals of project management and product development don't align and the project mindset is even considered to be an inhibitor against product excellence. This emag presents some alternative approaches.

QCon London 2016 Report



40

This year was the tenth anniversary for QCon London, and it was also our largest London event to date. Including our 140 speakers we had 1,400 team leads, architects, and project managers attending 112 technical sessions across 18 concurrent tracks and 12 in-depth workshops. This eMag brings together InfoQ's reporting of the event, along with views and opinions shared by attendees.

Frugal Innovation

39

In little over a decade, Africa has gone from being a region where it's still hard to find power lines, fixed-line telecom infrastructure, and personal computers to being the second-most mobile-connected continent where about 15% of the billion inhabitants own a cell phone.

Designing Your Culture

38

This eMag brings together a number of articles that explore ways to consciously design your culture, how to nurture and grow attitudes of craftsmanship and professionalism in teams, how to create places which are great to work in that get great outcomes, and how to make a profit.