Instrumenting Java Bytecode with ASM

In this tutorial, you will learn how to instrument Java .class files using the ASM framework. Part 1 introduces Java bytecode and shows how to read disassembled .class files. Part 2 presents the Visitor pattern, which is used throughout ASM. In part 3 we build a simple call trace instrumentation using ASM.

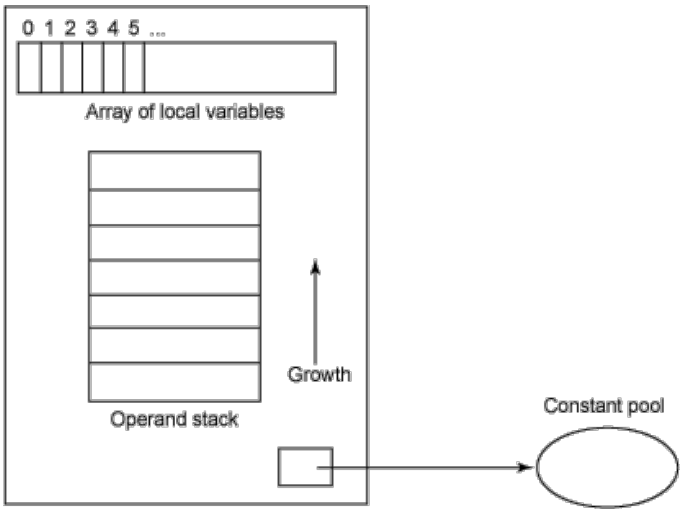
Part 1: Java ByteCode

ASM is a Java bytecode manipulation framework. Let's first figure out what 'Java Bytecode' is. Java bytecode is the instruction set of the Java virtual machine. Each instruction consists of a one-byte opcode followed by zero or more operands. For example, "iadd", which will receive two integers as operand and add them together. You can see all the detail of the instruction list here . Also there is a group list helping us quickly understand what java bytecode has:

- Load and store (e.g. aload_0, istore)
- Arithmetic and logic (e.g. ladd, fcmpl)
- Type conversion (e.g. i2b, d2i)
- Object creation and manipulation (new, putfield)
- Operand stack management (e.g. swap, dup2)
- Control transfer (e.g. ifeq, goto)
- Method invocation and return (e.g. invokespecial, areturn)

Java Virtual Machine:

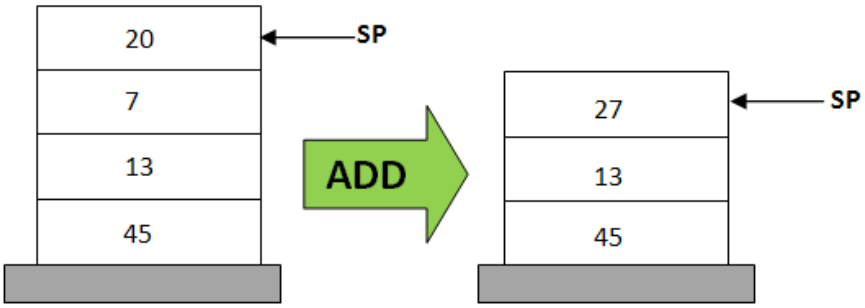
To understand the details of the bytecode, we need to discuss how a Java Virtual Machine (JVM) works regarding the execution of the bytecode. JVM is a platform-independent execution environment that converts Java bytecode into machine language and executes it. A JVM is a stack-based machine. Each thread has a JVM stack which stores *frames*. A frame is created each time a method is invoked, and consists of an operand stack, an array of local variables, and a reference to the runtime constant pool of the class of the current method.



- Learn more about JVM here!

Stack Based Virtual Machines:

We need to know a little about stack based VM to better understand Java Bytecode. A stack based virtual machine the memory structure where the operands are stored is a stack data structure. Operations are carried out by popping data from the stack, processing them and pushing in back the results in LIFO (Last in First Out) fashion. In a stack based virtual machine, the operation of adding two numbers would usually be carried out in the following manner (where 20, 7, and "result" are the operands):



- If you are interested in this part, you could look here to see more detail about stack based VM and register based VM.

Example: Consider the following Java code:

```
public class Test
{
    public static void main(String[] args) {
        printOne();
        printOne();
        printTwo();
    }

    public static void printOne() {
        System.out.println("Hello World");
    }

    public static void printTwo() {
        printOne();
        printOne();
    }
}
```

We use "javac" to compile java program and then use "javap -c" to disassemble the class file, where we can get the java bytecode. A Java class file is a file (with the .class filename extension) containing Java bytecode that can be executed on the JVM. A Java class file is produced by a Java compiler from Java programming language source files (.java files). The following are what we get:

```
public class Test {
  public Test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: invokestatic  #2                  // Method printOne:()V
       3: invokestatic  #2                  // Method printOne:()V
       6: invokestatic  #3                  // Method printTwo:()V
       9: return

  public static void printOne();
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #5                  // String Hello World
       5: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return

  public static void printTwo();
    Code:
       0: invokestatic  #2                  // Method printOne:()V
       3: invokestatic  #2                  // Method printOne:()V
       6: return
}
```

Detail:

- Look at the public Test() constructor. The bytecode for this constructor consists of three opcode instructions. The first opcode, aload_0, pushes the value from index 0 of the local variable table onto the operand stack. The local variable table is used to pass parameters to methods. The next opcode instruction, invokespecial, calls the constructor of this class's superclass. Because all classes that do not explicitly extend any other class implicitly inherit from

java.lang.Object, the compiler provides the necessary bytecode to invoke this base class constructor. During this opcode, the top value from the operand stack, this, is popped. The last instruction, return, just return what should be returned. The index number is not continuous because some of the opcodes have parameters that take up space in the bytecode array.

- The # number is the constant index for looking up the constant in the constant pool. The constant pool is a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures.We can use "javap -c -v" to see the whole constant pool.
- Java programming language provides two basic kinds of methods: instance methods (invokevirtual) and class (or static) methods (invokestatic). When the Java virtual machine invokes a class method, it selects the method to invoke based on the type of the object reference, which is always known at compile-time. On the other hand, when the virtual machine invokes an instance method, it selects the method to invoke based on the actual class of the object, which may only be known at run time.
- A great technical page to learn more about Java Bytecode!

Part 2: Visitor Pattern

In object-oriented programming and software engineering, the visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures.

In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.

The visitor pattern requires a programming language that supports single dispatch. Under this condition, consider two objects, each of some class type; one is called the "element", and the other is called the "visitor". An element has an accept() method that can take the visitor as an argument. The accept() method calls a visit() method of the visitor; the element passes itself as an argument to the visit() method.

Code Example:
In this code example, we will use the visitor pattern similar to how it is used in ASM for bytecode manipulation.
1. Add an accept(Visitor) method to the "element" hierarchy
2. Create a "visitor" base class w/ a visit() method for every "element" type
3. Create a "visitor" derived class for each "operation" to do on "elements"
4. Client creates "visitor" objects and passes each to accept() calls

```
interface Element {
   // 1. accept(Visitor) interface
   public void accept( Visitor v ); // first dispatch
}

class This implements Element {
   // 1. accept(Visitor) implementation
   public void   accept( Visitor v ) {
     v.visit( this );
   }
   public String thiss() {
     return "This";
   }
}

class That implements Element {
   public void   accept( Visitor v ) {
     v.visit( this );
   }
   public String that() {
     return "That";
   }
}

class TheOther implements Element {
   public void   accept( Visitor v ) {
     v.visit( this );
   }
   public String theOther() {
     return "TheOther";
   }
}

// 2. Create a "visitor" base class with a visit() method for every "element" type
interface Visitor {
```

```
   public void visit( This e ); // second dispatch
   public void visit( That e );
   public void visit( TheOther e );
}

// 3. Create a "visitor" derived class for each "operation" to perform on "elements"
class UpVisitor implements Visitor {
   public void visit( This e ) {
      System.out.println( "do Up on " + e.thiss() );
   }
   public void visit( That e ) {
      System.out.println( "do Up on " + e.that() );
   }
   public void visit( TheOther e ) {
      System.out.println( "do Up on " + e.theOther() );
   }
}

class DownVisitor implements Visitor {
   public void visit( This e ) {
      System.out.println( "do Down on " + e.thiss() );
   }
   public void visit( That e ) {
      System.out.println( "do Down on " + e.that() );
   }
   public void visit( TheOther e ) {
      System.out.println( "do Down on " + e.theOther() );
   }
}

class VisitorDemo {
   public static Element[] list = { new This(), new That(), new TheOther() };

   // 4. Client creates "visitor" objects and passes each to accept() calls
   public static void main( String[] args ) {
      UpVisitor    up   = new UpVisitor();
      DownVisitor  down = new DownVisitor();
      for (int i=0; i < list.length; i++) {
         list[i].accept( up );
      }
      for (int i=0; i < list.length; i++) {
         list[i].accept( down );
      }
   }
}
```

The output will look as follows:

```
do Up on This                do Down on This
do Up on That                do Down on That
do Up on TheOther            do Down on TheOther
```

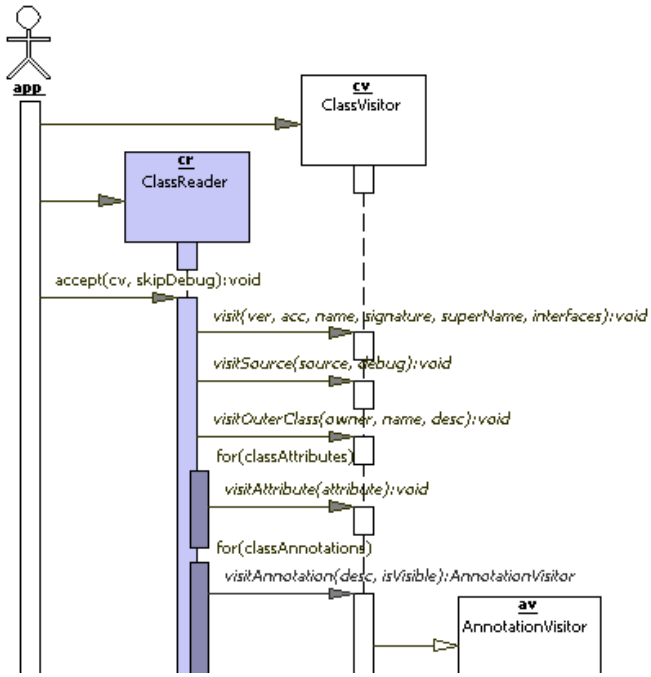
Visitors in ASM:
ASM uses the visitor pattern. "Objects" or "Acceptors" include the ClassReader class and the MethodNode class. Visitor interfaces include ClassVisitor, AnnotationVisitor, FieldVisitor, and MethodVisitor.
The accept() method belongs to the MethodNode class and has the following signatures:

```
void accept(ClassVisitor cv)

void accept(MethodVisitor mv)
```

The visit() method and it's family of other methods such as visitField() etc. are part of the ClassVisitor class and have the following signatures:

```
void visit(int version, int access, String name, String signature, String superName, String[] interfaces)

AnnotationVisitor visitAnnotation(String desc, boolean visible)

void visitAttribute(Attribute attr)

void visitEnd()

FieldVisitor visitField(int access, String name, String desc, String signature, Object value)

void visitInnerClass(String name, String outerName, String innerName, int access)
```

```
MethodVisitor visitMethod(int access, String name, String desc, String signature, String[] exceptions)

void visitOuterClass(String owner, String name, String desc)

void visitSource(String source, String debug)
```



Part 3: Call Trace Instrumentation

In this section we will implement a call trace instrumentation using ASM. The instrumented code will log each method call and return. The output log could easily be parsed into a Calling Context Tree.

Setting Up

For this tutorial, you will need to have the Java Development Kit (JDK) and Apache ant installed. The code for this tutorial is available here. To follow along, first download the ASM 5.0.3 binary distribution and tutorial code. Unzip both, and copy `asm-all-5.0.3.jar` into the tutorial folder:

```
$ unzip ASM-tutorial.zip
$ unzip asm-5.0.3-bin.zip
$ cp asm-5.0.3/lib/all/asm-all-5.0.3.jar ASM-tutorial/
```

Hello ASM: Copying Class Files

Our first ASM program will simply make a copy of a .class file. This will introduce us to the ASM boilerplate, and serve as a template we can reuse later for more interesting instrumentation. Here's what `Copy.java` looks like:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;

import org.objectweb.asm.ClassReader;
import org.objectweb.asm.ClassWriter;

public class Copy {
    public static void main(final String args[]) throws Exception {
        FileInputStream is = new FileInputStream(args[0]);

        ClassReader cr = new ClassReader(is);
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        cr.accept(cw, 0);

        FileOutputStream fos = new FileOutputStream(args[1]);
        fos.write(cw.toByteArray());
        fos.close();
```

```
    }
}
```

The Copy program takes two command-line arguments: `args[0]` is the name of the .class file to copy *from*, `args[1]` is the name of the .class file to copy *to*.

We use two ASM classes in this example: ClassReader reads Java Bytecode from a file, and ClassWriter writes Bytecode to a file. ASM is using the Visitor Pattern: ClassWriter implements the of ClassVisitor interface, and the call to `cr.accept(cw, 0)` causes ClassReader to traverse the Bytecode input, making a series of calls to `cw` that generate the same Bytecode as output.

The `ClassWriter.COMPUTE_FRAMES` argument to the ClassWriter constructor is an option flag that indicates we would like the ClassWriter to compute the size of stack frames for us. The second argument to `cr.accept` is also for option flags. Passing `0` indicates we want the default behavior. See the JavaDocs for ClassReader and ClassWriter for more information.

```
# Compile Copy
$ javac -cp asm-all-5.0.3.jar Copy.java

# Use Copy to copy itself
$ java -cp .:asm-all-5.0.3.jar Copy Copy.class Copy2.class
```

Generating Call Traces

Now that we're familiar with the basics of ASM, let's turn to our instrumentation for generating call traces. We will log method calls and returns to stderr, with each call and return on a separate line. For example, let's consider the following simple Java program:

```
public class Test
{
    public static void main(String[] args) {
        printOne();
        printOne();
        printTwo();
    }

    public static void printOne() {
        System.out.println("Hello World");
    }

    public static void printTwo() {
        printOne();
        printOne();
    }
}
```

We will instrument *call sites* to print to stderr before and after the call. For example, the result of instrumenting Test.class should be equivalent to the following:

```
public class TestInstrumented
{
    public static void main(String[] args) {
        System.err.println("CALL printOne");
        printOne();
        System.err.println("RETURN printOne");

        System.err.println("CALL printOne");
        printOne();
        System.err.println("RETURN printOne");

        System.err.println("CALL printTwo");
        printTwo();
        System.err.println("RETURN printTwo");
    }

    public static void printOne() {
        System.err.println("CALL println");
        System.out.println("Hello World");
        System.err.println("RETURN println");
    }

    public static void printTwo() {
        System.err.println("CALL printOne");
```

```
        printOne();
        System.err.println("RETURN printOne");

        System.err.println("CALL printOne");
        printOne();
        System.err.println("RETURN printOne");
    }
}
```

We will implement our instrumentation by modifying the `Copy` example above. In order to modify the class the class file, we need to insert some code in between `ClassReader` and `ClassWriter`. We will do this using the Adapter Pattern. Adapters wrap an object, overriding some of its methods, and delegating to the others. This gives us an easy way to modify the behavior of the wrapped object. Here we will adapt the `ClassWriter`, so that when emit the Bytecode for a call site, we also emit code to produce trace log before and after the call.

Since method call sites occur inside methods, our main instrumentation work will take place inside method declarations. There is a slight complication: method declarations occur within classes, so we need to traverse a class to instrument its methods.

Our first step will be to adapt the `ClassWriter` object using our `ClassAdapter` class defined below. By default, all `ClassAdapter` methods inherited from `ClassVisitor` will simply call the same method on the adapted `ClassWriter`. Most of the time we want the default behavior. We will `@override` the behavior of only the `ClassWriter.visitMethod` method, which is called once for each method declaration in a class. The return value of visitMethod is a a `MethodVisitor` object, which is used to process the method body. `ClassWriter.visitMethod` returns a MethodVisitor that produces the Bytecode for a method. We will adapt the `MethodVisitor` returned by `ClassWriter.visitMethod`. and insert additional instructions that print the call trace.

```
class ClassAdapter extends ClassVisitor implements Opcodes {

    public ClassAdapter(final ClassVisitor cv) {
        super(ASM5, cv);
    }

    @Override
    public MethodVisitor visitMethod(final int access, final String name,
            final String desc, final String signature, final String[] exceptions) {
        MethodVisitor mv = cv.visitMethod(access, name, desc, signature, exceptions);
        return mv == null ? null : new MethodAdapter(mv);
    }
}

class MethodAdapter extends MethodVisitor implements Opcodes {

    public MethodAdapter(final MethodVisitor mv) {
        super(ASM5, mv);
    }

    @Override
    public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
        /* TODO: System.err.println("CALL" + name); */

        /* do call */
        mv.visitMethodInsn(opcode, owner, name, desc, itf);

        /* TODO: System.err.println("RETURN" + name);  */
    }
}
```

So far, our MethodAdapter class doesn't add any instrumentation -- it simply delegates to the wrapped MethodVisitor `mv`. We have a good idea how to write the instrumentation in Java syntax, but we don't know how to express it using ASM's API. For this, we will use the `ASMifier` tool, which is distributed with ASM.

We can use ASMifier to translate TestInstrumented into ASM API calls. For brevity, we elide some of the calls below.

```
$ javac TestInstrumented.java
$ java -cp .:asm-all-5.0.3.jar org.objectweb.asm.util.ASMifier TestInstrumented
/** WARNING: THINGS ARE ELIDED **/
{
mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "printOne", "()V", null, null);
mv.visitCode();

mv.visitFieldInsn(GETSTATIC, "java/lang/System", "err", "Ljava/io/PrintStream;");
mv.visitLdcInsn("CALL println");
```

```
mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V", false);

mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
mv.visitLdcInsn("Hello World");
mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V", false);

mv.visitFieldInsn(GETSTATIC, "java/lang/System", "err", "Ljava/io/PrintStream;");
mv.visitLdcInsn("RETURN println");
mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V", false);

mv.visitInsn(RETURN);
mv.visitMaxs(2, 0);
mv.visitEnd();
}
/** WARNING: MORE THINGS ARE ELIDED **/
```

The output of ASMifier is an ASM program that generates `TestInstrumented.class`. The important parts for our needs are the calls to `System.err.println`:

```
mv.visitFieldInsn(GETSTATIC, "java/lang/System", "err", "Ljava/io/PrintStream;");
mv.visitLdcInsn("CALL println");
mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V", false);
```

Now that we know how to call `System.err.println`, we can complete our implementation of MethodAdapter:

```
class MethodAdapter extends MethodVisitor implements Opcodes {

    public MethodAdapter(final MethodVisitor mv) {
        super(ASM5, mv);
    }

    @Override
    public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
        /* System.err.println("CALL" + name); */
        mv.visitFieldInsn(GETSTATIC, "java/lang/System", "err", "Ljava/io/PrintStream;");
        mv.visitLdcInsn("CALL " + name);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V", false);

        /* do call */
        mv.visitMethodInsn(opcode, owner, name, desc, itf);

        /* System.err.println("RETURN" + name);  */
        mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "err", "Ljava/io/PrintStream;");
        mv.visitLdcInsn("RETURN " + name);
        mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V", f
    }
}
```

That's it! Let's give it a try on our `Test` example.

```
# Build Instrumenter
$ javac -cp asm-all-5.0.3.jar Instrumenter.java

# Build Example
$ javac Test.java

# Move Test.class out of the way
$ cp Test.class Test.class.bak

# Instrument Test
$ java -cp .:asm-all-5.0.3.jar Instrumenter Test.class.bak Test.class

# Run!
$ java Test
CALL printOne
CALL println
Hello World
RETURN println
RETURN printOne
CALL printOne
CALL println
Hello World
RETURN println
RETURN printOne
CALL printTwo
CALL printOne
```

```
CALL println
Hello World
RETURN println
RETURN printOne
CALL printOne
CALL println
Hello World
RETURN println
RETURN printOne
RETURN printTwo
```

The output is exactly what we expected, plus the four "Hello World" lines printed to stdout (which is interleaved with stderr).

More Information

Now you know enough of ASM to get started writing your own instrumentation. You can learn more about it from the following links:

- ASM Homepage
- ASM 5.0.3 Binary Distribution
- User Guide
- ASM JavaDoc