Practical Byte Code Engineering

by Rob Kenworthy MVB · Apr. 07, 18 · Java Zone · Tutorial

Over the past few years, I have written a few blogs about how to use byte code engineering. My first article was a brief overview while others discussed specific case studies. In hindsight, I think I have overlooked covering the basic building blocks of byte code engineering: the Java agent and the Instrumentation API. Additionally, some downloadable and practical byte code engineering example projects might helpful. This article aims to reconcile these issues.

There are two main ways to instrument Java byte code. One way is to modify your target classes prior to run time and then adjust your classpath (and possibly boot classpath) accordingly to point to your instrumented classes, Fortunately, (since Java 1.5), there is a specific Java API for instrumentation (among other things) called JVM TI (Tooling Interface). JVM TI allows you to attach *native* or *Java* agents. This blog will focus only on Java agents (I tell people I prefer them for their platform portability, but the truth is my C programming skills are really rusty).

The Java agent deployment unit is a jar file. The jar file must have a manifest built to support agents. You can refer to the instrumentation package documentation for details for the manifest and other requirements, but here is the condensed version of the manifest attributes:

- **Premain-Class**: Required to support attaching agents *as the JVM is started*. You can think of this as similar to the class containing "public static final void main" that is used to invoke any Java program.
- **Agent-Class**: Required to support attaching agents dynamically *after the JVM is already running*. Again, you can think of this as similar to the class containing "public static final void main" that is used to invoke any Java program.

You will probably want to define both properties and the values will probably point to the same class.

- **Boot-Class-Path**: I never define this property. If I want to manipulate the bootclasspath, I prefer to do it using the Instrumentation API once the agent has loaded.
- **Can-Redefine-Classes**: Indicator that this agent can call Instrumentation.redefineClasses(...). "Redefinition" is applied to classes that have already been loaded.
- **Can-Retransform-Classes**: Indicator that this agent can call Instrumentation.retransformClasses(...) "Retransformation" is applied to classes as they are loaded.
- **Can-Set-Native-Method-Prefix**: This blog is about non-native agents, but if interested, you can get a detailed description of this property in the Instrumentation.setNativeMethodPrefix method documentation.Next, lets take a look at an agent "entry point" class. As specified in the *Command-Line Interfaces*ection of the java.lang.instrumentation package javadocs,the entry point method name must be either "premain" for agents that are attached when the JVM is initially invoked or "agentmain" for agents that are attached after the JVM has started. Here are the valid method signatures:
- public static void premain(String agentArgs, Instrumentation inst);
- public static void premain(String agentArgs);
- public static void agentmain(String agentArgs, Instrumentation inst);
- public static void agentmain(String agentArgs);

The JVM will attempt to invoke the flavor with the Instrumentation parameter first and only invoke the other if the first doesn't exist.

To invoke an agent at JVM initialization time, you specify the following JVM parameter:

```
-javaagent:<path to agent jar>=<agent arguments>
```

Attaching an agent to a running JVM is a little more complicated. The Attach API's VirtualMachine.attach(String pid) method will allow one JVM to attach to another. Once attached, the VirtualMachine.loadAgent(...) methods can be used to load an agent into the JVM that you are attached to. You should note that the "pid" parameter of the attach method is the process ID of the target JVM. If you know the PID of the target JVM you could use the following (oversimplified) code to attach:

```
public static void main(String args[]) throws Exception {
    VirtualMachine.attach(args[0]).loadAgent(args[1]);
}
```

Assuming this method was in a class called ca.discotek.attachexample.Attacher, you would invoke this code like so:

```
java ca.discotek.attachexample.Attacher 1234 C:/agentdir/myagent.jar
```

Please note that the Attach API is part of tools.jar, which is only available in JDKs.

Getting back to building an agent class, if you want to do any class transformations, you will need to implement the java.lang.instrument.ClassFileTransfomer interface, which defines the following method:

```
byte[] transform(ClassLoader loader,
                 String className,
                 Class<?> classBeingRedefined,
                 ProtectionDomain protectionDomain,
                 byte[] classfileBuffer)
                 throws IllegalClassFormatException;
```

This method returns a byte array which contains the byte code definition for a given class. Here are some notes regarding the parameters:

1. **loader**: Will be null if the ClassLoader is the bootstrap ClassLoader, so don't assume it will be non-null!
2. **className**: Believe it or not, this value can be null sometimes, so don't assume it will be non-null! Also, it will always use a forward slash as a package/class name separator (e.g. *java/lang/String*, not *java.lang.String*).
3. **classBeingRedefined**: Will be null if the class is being loaded for the first time, so don't assume it will be non-null!
4. **protectionDomain**: I never use this parameter, so I don't have anything to say about it.
5. **classfileBuffer**: This will never be non-null!

Your agent class doesn't have to be the class that implements this interface, but your agent class code will probably be responsible for activating any ClassFileTransformer implementation by calling instrumentation.addTransformer(...). This method comes in two flavors:

```
public void addTransformer(ClassFileTransformer transformer)
public void addTransformer(ClassFileTransformer transformer, boolean canRetransform)
```

These methods will register a ClassFileTransformer with the JVM. The second method contains a boolean parameter, which flags the transformer as interested in processing class byte code as it is loaded. A ClassFileTransformer, which is added without this parameter set to true, will not be able to transform byte code. Its transform method will be invoked, but the byte code array returned by this method will simply be discarded.

One of the main obstacles in front of developers wishing to learn more about agents is putting together a project, which does all of the above before getting to the fun part of implementing some agent functionality. I have put together the following projects to help smooth over these bumps. Please note, resources for these projects can be downloaded from the Practical Byte Code Engineering download page.

agent-example-0-basic [Download]

Builds an agent jar which just prints all the class names and their class loaders as they are loaded:

```
    public byte[] transform(ClassLoader loader, String className,
            Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
```

```
            byte[] classfileBuffer) throws IllegalClassFormatException {

        System.out.println("Basic Agent: " + className + " : " + loader);

        return null;
    }
```

To see it in action

1. Run the default task of the build.xml ANT script at the root of the project.
2. Run the BasicTest program (under test in the root of project) with the following JVM parameter: -javaagent:<path to project>/dist/myagent.jar

agent-example-1-attach [Download]

This project doesn't have a lot of code, but it is somewhat complicated. It demonstrates how you can attach an agent to a running JVM. Let's first look at the agent class. We *attaching* to a JVM, so we only need the agentmain method in our *ca.discotek.agent.example.attache.MyAgent* agent class:

```
    public static void agentmain(String agentArgs, Instrumentation inst) {
        initialize(agentArgs, inst, false);
    }
```

Let's now look at the *initialize* method:

```
    public static void initialize(String agentArgs, Instrumentation inst, boolean isPremain) {
        MyAgent.instrumentation = inst;
        inst.addTransformer(new MyClassFileTransformer(), true);

        Runnable r = new Runnable() {
            public void run() {
                while (true) {
                    try {
                        Thread.sleep(1000);
                        Class classes[] = instrumentation.getAllLoadedClasses();
                        for (int i=0; i<classes.length; i++) {
                            if (classes[i].getName().equals("ca.discotek.agent.example.attach.test.AttachTest")) {
                                System.out.println("Reloading: " + classes[i].getName());
                                instrumentation.retransformClasses(classes[i]);
                                break;
                            }
                        }

                    }
                    catch (Exception e) {
                        e.printStackTrace();
                        break;
                    }
                }

            }
        };
        Thread t = new Thread(r);
        t.start();
    }
```

The *initialize* method does two main things:

1. The second line registers a new instance of *MyClassFileTransformer* as a *ClassFileTransformer* with the JVM using the *Instrumentation.addTransformer(...)* method.
2. Creates some looping code in a thread which will continually schedule the *AttachTest* class to be retransformed.

Next we have the *ca.discotek.agent.example.attache.MyClassFileTransformer*class. It implements the *ClassFileTransformer* interface and has the following *transform method*:

```
    public byte[] transform(ClassLoader loader, String className,
            Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
            byte[] classfileBuffer) throws IllegalClassFormatException {

        if (className != null && className.startsWith("ca/discotek/"))
            System.out.println("Attach Agent: " + className);

        return null;
    }
```

This method doesn't do much except print out the name of any classes that are being loaded/transformed that start with *ca/discotek/*. This isn't particularly interesting, but it will demonstrate that the *MyAgent* agent was attached and successfully added the *MyClassFileTransformer*, which continually reloads classes.

We also have an *ca.discotek.agent.example.attach.Attacher* class with a single method:

```
    public static void main(String[] args) throws Exception {
        VirtualMachine.attach(args[0]).loadAgent(args[1]);
    }
```

This class take a PID of a Java process as the first parameter and the path to an agent jar as a second parameter.

Lastl, we have *ca.discotek.agent.example.attach.test.AttachTest*, which is just used for creating a JVM to demonstrate the attach/agent functionality:

```
    public static void main(String[] args) throws Exception {
        while (true) {
            System.out.println("Sleeping...");
            Thread.sleep(1000);
        }
    }
```

To see this agent in action

1. Run the default task of the build.xml ANT script at the root of the project.
2. Run the AttachTest program. You can run this from your IDE or the command line.
3. Run the Attacher program (same package as MyAgent) with the following parameters <pid> <path to project>/dist/myagent.jar, where you will need to determine the pid using tools like jps, jconsole, or your operating systems process manager. Remember, you will need to have the JDK's tools.jar in your classpath. The command line will look something like this:

```
java -classpath .../discotek-agent-example-1-attach/bin;/java/jdkx.y.z/lib/tools.jar ca.discotek.agent.example.attach.Attacher
```

You can confirm that you have correctly attached to the running JVM when the *MyClassFileTransformer* class is printing the following repeatedly:

```
Sleeping...
Reloading: ca.discotek.agent.example.attach.test.AttachTest
Attach Agent: ca/discotek/agent/example/attach/test/AttachTest
```

agent-example-2-access-javassist [Download]

Builds an agent jar that will using Javassist to change the access modifiers of a test class from *private* to *public*. It has *ClassFileTransformer.transform()*method implementation as follows:

```
    public byte[] transform(ClassLoader loader, String className,
            Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
            byte[] classfileBuffer) throws IllegalClassFormatException {

        String dotName = className.replace('/', '.');
        if (className != null && transformPattern.matcher(dotName).matches()) {

            try {
                ClassPool pool = ClassPool.getDefault();
                pool.appendClassPath(new ByteArrayClassPath(dotName, classfileBuffer));
```

```
                CtClass ctClass = pool.get(dotName);
                int modifiers = ctClass.getModifiers();
                ctClass.setModifiers(Modifier.setPublic(modifiers));

                CtField ctField = ctClass.getDeclaredField("privateMessage");
                modifiers = ctField.getModifiers();
                ctField.setModifiers(Modifier.setPublic(modifiers));

                CtConstructor ctConstructor = ctClass.getDeclaredConstructor(new CtClass[]{});
                modifiers = ctConstructor.getModifiers();
                ctConstructor.setModifiers(Modifier.setPublic(modifiers));

                CtMethod ctMethod = ctClass.getDeclaredMethod("printMessage");
                modifiers = ctMethod.getModifiers();
                ctMethod.setModifiers(Modifier.setPublic(modifiers));

                return ctClass.toBytecode();
            }
            catch (Exception e) {
                throw new RuntimeException("Bug", e);
            }

        }

        return null;
    }
```

The project also a *test* source folder which contains classes for testing the agent. These classes are:

ca.discotek.agent.example.access.javassist.testee.PrivateTest, which has methods:

```
    private String privateMessage = "This is a private message";

    private PrivateTest() {
        System.out.println("Private Constructor");
    }

    private void printMessage() {
        System.out.println("This is a private method");
    }
```

This class has a private field, private constructor, and a private method. The class itself is also private.

The test source folder also has
class *ca.discotek.agent.example.access.javassist.tester.AccessJavassistTest* with *main*method:

```
    public static void main(String[] args) throws Exception {

        Class c = null;
        try {
            c = Class.forName("ca.discotek.agent.example.access.javassist.testee.PrivateTest");
            System.out.println("Class is public? " + Modifier.isPublic(c.getModifiers()) );
        }
        catch (Throwable t) {
            t.printStackTrace();
        }

        Object o = null;
        try {
            Constructor constructor = c.getDeclaredConstructor(new Class[]{});
            System.out.println("Constructor is public? " + Modifier.isPublic(constructor.getModifiers()) );
            o = constructor.newInstance(new Object[]{});
        }
        catch (Throwable t) {
            t.printStackTrace();
        }
```

```
        try {
            Field field = c.getField("privateMessage");
            System.out.println("Field is public? " + Modifier.isPublic(field.getModifiers()) );
            Object value  = field.get(o);
            System.out.println("Field value: " +  value);
        }
        catch (Throwable t) {
            t.printStackTrace();
        }

        try {
            Method method = c.getMethod("printMessage", new Class[]{});
            System.out.println("Method is public? " + Modifier.isPublic(method.getModifiers()) );
            method.invoke(o, new Object[]{});
        }
        catch (Throwable t) {
            t.printStackTrace();
        }
    }
```

This method will test if any of the private entities in the *PrivateTest* class are accessible.

To see this agent in action:

1. Run the default task of the build.xml ANT script at the root of the project.
2. Run the *ca.discotek.agent.example.access.javassist.tester.AccessJavassistTest* program. You can run this from your IDE or the command line. You will need to add the *-javaagent* parameter. Here is what it might look like:

```
java -javaagent:.../discotek-agent-example-2-access-javassist/dist/agent-access-javassist.jar=.*discotek.*test.* -classpath ..
```

This assumes your IDE compiles classes in to *bin* directory at the root of your project (otherwise the *-classpath* in the example above would be incorrect).

agent-example-3-access-asm [Download]

Builds an agent jar that will using ASM to change the access modifiers of a test class from *private* to *public*. It has *ClassFileTransformer.transform()* method implementation as follows:

```
    public byte[] transform(ClassLoader loader, String className,
            Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
            byte[] classfileBuffer) throws IllegalClassFormatException {

        if (className != null && transformPattern.matcher(className.replace('/', '.')).matches()) {
            ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
            AccessClassVisitor accessClassVisitor = new AccessClassVisitor(cw);
            ClassReader cr = new ClassReader(classfileBuffer);
            cr.accept(accessClassVisitor, ClassReader.SKIP_FRAMES);

            return cw.toByteArray();
        }

        return null;
    }
```

We also have *AccessClassVisitor* which is used to perform the byte code modifications:

```
public class AccessClassVisitor extends ClassVisitor {

    static int convertToPublicAccess(int access) {
        access &= ~Opcodes.ACC_PRIVATE;
        access &= ~Opcodes.ACC_PROTECTED;
        access |= Opcodes.ACC_PUBLIC;
        return access;
    }
```

```
    }

    public AccessClassVisitor(ClassVisitor cv) {
        super(Opcodes.ASM5, cv);
    }

    @Override
    public void visit(int version, int access, String name, String signature, String superName, String[] interfaces) {
        super.visit(version, convertToPublicAccess(access), name, signature, superName, interfaces);
    }

    @Override
    public MethodVisitor visitMethod(int access,
            String name,
            String desc,
            String signature,
            String[] exceptions) {

        return super.visitMethod(convertToPublicAccess(access), name, desc, signature, exceptions);
    }

    @Override
    public FieldVisitor visitField(int access,
            String name,
            String desc,
            String signature,
            Object value) {
        return super.visitField(convertToPublicAccess(access), name, desc, signature, value);
    }
}
```

To see this agent in action:

1. Run the default task of the build.xml ANT script at the root of the project.
2. Run the *ca.discotek.agent.example.access.asm.tester.AccessAsmTest* program. You can run this from your IDE or the command line. You will need to add the *-javaagent* parameter. Here is what it might look like:

```
java -noverify -javaagent:.../discotek-agent-example-3-access-asm/dist/agent-access-asm.jar=.*discotek.*test.* -classpath .../
```

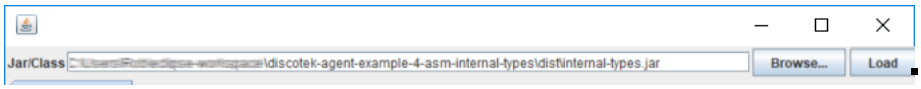
This assumes your IDE compiles classes in to *bin* directory at the root of your project (otherwise the *-classpath* in the example above would be incorrect).

Please note the use of the *-noverify* JVM parameter. This is the first time this parameter has been introduced in these projects. If you are using a modern JVM it is most likely required to to avoid the JVM rules regarding stack frames (explained here). Adding -noverify simply bypasses the byte code verifier, which avoids these rules. You should be careful about using -noverify in a production environment.

agent-example-4-asm-internal-types [Download]

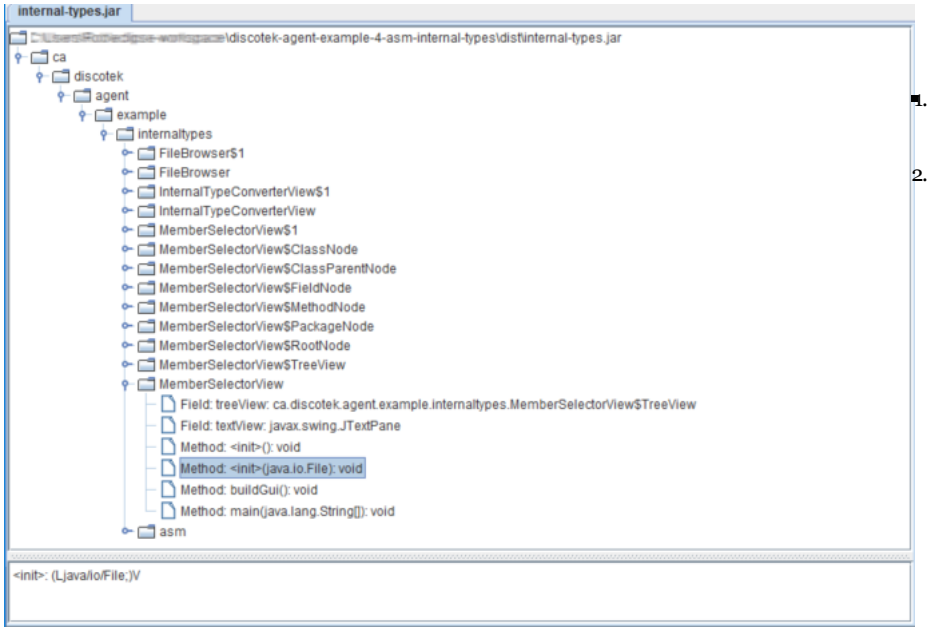
This next project does not a build an agent. It builds a program that can help anyone struggling with specifying JVM internal descriptors correctly. Undertanding JVM internal descriptor correctly is very important for byte code engineering and
are used very frequently with ASM. They are used by Javassist as well, but not as frequently. The main Javassist example I can think of is the CtClass's getConstructor(String descriptor) and getMethod(String name, String descriptor) methods.

I won't paste the relevant code here, but here is a screen shot of it in action. Here it has loaded the jar built by running the default build of the ANT script in the root of this project.



To see this code in action either:

Run the code directly from your IDE using entry point



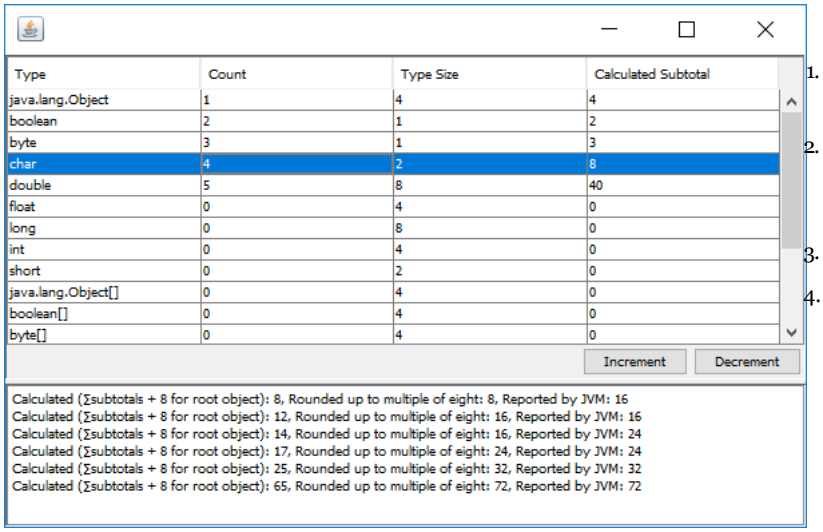
class *ca.discotek.agent.example.internaltypes.InternalTypeConverterView*.

1. Run the default task of the build.xml ANT script at the root of the project.
2. Then invoke the program using a command like: *java -classpath .../discotek-agent-example-4-asm-internal-types/dist/internal-types.jar ca.discotek.agent.example.internaltypes.InternalTypeConverterView*

agent-example-5-objectsize-asm [Download]

Instead of byte code transformations, this next project uses Instrumentation's getObjectSize(Object o) method to calculate the the size of an arbitrary object in memory.

Here is a screen shot of the final product:



In the top half of the GUI there is a table with four columns:

1. **Type**: There is a row for every basic Java type. Every class that you define will contain some combination of these types (possibly none)
2. **Count**: This column represents the number of fields of the of the *Type* specified for a given row that would appear in an arbitrary class
3. **Type Size**: This column is the size in bytes for *Type* in each row
4. **Calculated Subtotal**: This column is the calculated size that the types for a given row would consume (i.e. *Count* * *Type Size*)

Just below the table, there are *Increment* and *Decrement* buttons. These buttons will increase or decrease the number in the *Count* column for a given row.

At the bottom of the table, there is a text field, which shows the calculated total object size and the size as return by Instrumentation's getObjectSize(Object o)method.

The code in the agent class is very simple:

```
1       public static void premain(String agentArgs, Instrumentation inst) {
2           initialize(agentArgs, inst, true);
3       }
4
5       public static void agentmain(String agentArgs, Instrumentation inst) {
6           initialize(agentArgs, inst, false);
7       }
8
9       public static void initialize(String agentArgs, Instrumentation inst, boolean isPremain) {
10          try {  ObjectSizeAnalyzerView.showObjectAnalyzerView(inst); }
11          catch (Exception e) { e.printStackTrace(); }
12      }
```

Most of the *ObjectSizeAnalyzeView* code isn't very interesting either. However, ASM is used to generate an object with the fields as specified in the table, which is worth noting. A single *updateSize()* method is used to generate a new class with the specified fields and then instantiate an instance of that class. It then passes that object as a parameter to the Instrumentation.getObjectSize(Object o)method

```
    void updateSize() {

        String className = "MyClass" + classIndex++;

        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
        cw.visit(Opcodes.V1_6, Opcodes.ACC_PUBLIC, className, null, "java/lang/Object", null);

        Method m = Method.getMethod("void()");
        GeneratorAdapter mg = new GeneratorAdapter(Opcodes.ACC_PUBLIC, m, null, null, cw);
        mg.loadThis();
        mg.invokeConstructor(Type.getType(Object.class), m);
        mg.returnValue();
        mg.endMethod();

        int fieldIndex = 0;
        Type type;
        int count;
        String desc;
        for (int i=0; i<TYPES.length; i++) {
            count = model.countList.get(i);
            for (int j=0; j<count; j++) {
                type = Type.getType(TYPE_CLASSES[i]);
                desc = getDescriptor(type);
                cw.visitField(Opcodes.ACC_PUBLIC, "field" + fieldIndex++, desc, null, null);
            }
        }

        cw.visitEnd();

        MyClassLoader loader = new MyClassLoader(className, cw.toByteArray());
        try {
            Class c = loader.loadClass(className);
            currentObject = c.newInstance();
            long objectSize = instrumentation.getObjectSize(currentObject);
            StringBuilder buffer = new StringBuilder();
            long calculated = 8;
            Integer ints[] = model.countList.toArray(new Integer[model.countList.size()]);
            for (int i=0; i
```

To see this agent in action:

1. Run the default task of the build.xml ANT script at the root of the project.
2. Run the *ca.discotek.agent.example.objectsize.asm.test.ObjectSizeAsmTest* program. You can run this from your IDE or the command line. You will need to add the *-javaagent* parameter. Here is what it might look like:

```
java -javaagent:.../discotek-agent-example-5-objectsize-asm/dist/agent-objectsize-asm.jar -classpath .../discotek-agent-exampl
```

agent-example-6-profile-javassist [Download]

This project produces an agent jar that demonstrates one of Javassist's shortcomings. Specifically, if you add a local variable using CtBehaviour's *insertBefore* method, you cannot later reference it in code added using CtBehaviour's *insertAfter* method.
Javassist doesn't know anything about the byte code you previously inserted. Let's see what happens when we use Javassist to create an agent to to instrument methods in order to profile execution time. The agent will instrument all methods in classes whose name match a regular expression provided in the agent arguments. Here is how the agent is initialized:

```
    public static void premain(String agentArgs, Instrumentation inst) {
        initialize(agentArgs, inst, false);
    }

    public static void agentmain(String agentArgs, Instrumentation inst) {
        initialize(agentArgs, inst, true);
    }

    static void initialize(String agentArgs, Instrumentation inst, boolean isPremain) {
        MyAgent.instrumentation = inst;
        inst.addTransformer(new MyAgent(Pattern.compile(agentArgs)), true);
    }

    public MyAgent(Pattern transformPattern) {
        this.transformPattern = transformPattern;
    }
```

The *transform* method filters out any unwanted classes and passes the byte code to an *instrument(...)* method for processing:

```
    void instrument(CtBehavior behaviour) throws CannotCompileException {
        String beforeCode = "long __start_time__ = System.currentTimeMillis();";
        behaviour.insertBefore(beforeCode);

        StringBuilder buffer = new StringBuilder();
        buffer.append("long __end_time__ = System.currentTimeMillis();");
        String method = "$class" + '.' + behaviour.getName() + behaviour.getSignature();
        buffer.append("System.out.println(\"Ellapsed " + method + ": \" + (__end_time__ - __start_time__));");

        String afterCode = buffer.toString();
        behaviour.insertAfter(afterCode, true);

        behaviour.addLocalVariable("__start_time__", CtClass.longType);
        behaviour.addLocalVariable("__end_time__", CtClass.longType);
    }
```

Despite the fact this code will produce a run-time Javassist error, here are some points worth noting:

- The *instrument(...)* method takes a *CtBehaviour* object as a parameter. CtBehaviour is the super class of both CtConstructor and CtMethod, so it can process either.
- A constructor is a special type of methods used to initialize an object. As such, it has rule that there should be no byte code instructions to invoke any methods in the constructor preceding the constructors call to *super*. Fortunately, Javassist's CtBehaviour.insertBefore is aware and will insert your code after the call to *super*.
- You will notice that the names of the local variables inserted into each method (e.g. __start_time__, __end_time__) are a little strange looking. If you are inserting any construct into byte code that you are unfamiliar with, you need to make sure you don't break rules about uniqueness. In this case, you can't have two local variables with the same name. If we had used "start" as the variable name instead of "__start_time__", it is more likely a clash might occur. I usually include "discotek" in the name of a variable to ensure uniqueness.

Lastly, here is our simple test client code:

```
public class ProfileJavassistTest {

    public static final void main(String[] args) throws Exception {
        System.out.println("Jon Snow, Ned Stark's bastard, likes to say \"Winter is coming.\"");
    }
}
```

To see this agent in action:

1. Run the default task of the build.xml ANT script at the root of the project.
2. Run the *ca.discotek.agent.example.profile.javassist.test.ProfileJavassistTest* program. You can run this from your IDE or the command line. You will need to add the *-javaagent* parameter. Here is what it might look like:

```
java -noverify -javaagent:.../discotek-agent-example-6-profile-javassist/dist/agent-profile-javassist.jar=.*discotek.*test.* -
```

Here is what the output will look like:

```
javassist.CannotCompileException: [source error] no such field: __start_time__
    at javassist.CtBehavior.insertAfter(CtBehavior.java:815)
    at ca.discotek.agent.example.profile.javassist.MyAgent.instrument(MyAgent.java:79)
    at ca.discotek.agent.example.profile.javassist.MyAgent.transform(MyAgent.java:54)
    at sun.instrument.TransformerManager.transform(Unknown Source)
    at sun.instrument.InstrumentationImpl.transform(Unknown Source)
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(Unknown Source)
    at java.security.SecureClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.access$100(Unknown Source)
    at java.net.URLClassLoader$1.run(Unknown Source)
    at java.net.URLClassLoader$1.run(Unknown Source)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at sun.launcher.LauncherHelper.checkAndLoadMain(Unknown Source)
Caused by: compile error: no such field: __start_time__
    at javassist.compiler.TypeChecker.fieldAccess(TypeChecker.java:812)
    at javassist.compiler.TypeChecker.atFieldRead(TypeChecker.java:770)
    at javassist.compiler.TypeChecker.atMember(TypeChecker.java:952)
    at javassist.compiler.JvstTypeChecker.atMember(JvstTypeChecker.java:65)
    at javassist.compiler.ast.Member.accept(Member.java:38)
    at javassist.compiler.TypeChecker.atBinExpr(TypeChecker.java:328)
    at javassist.compiler.ast.BinExpr.accept(BinExpr.java:40)
    at javassist.compiler.TypeChecker.atPlusExpr(TypeChecker.java:370)
    at javassist.compiler.TypeChecker.atBinExpr(TypeChecker.java:311)
    at javassist.compiler.ast.BinExpr.accept(BinExpr.java:40)
    at javassist.compiler.JvstTypeChecker.atMethodArgs(JvstTypeChecker.java:220)
    at javassist.compiler.TypeChecker.atMethodCallCore(TypeChecker.java:702)
    at javassist.compiler.TypeChecker.atCallExpr(TypeChecker.java:681)
    at javassist.compiler.JvstTypeChecker.atCallExpr(JvstTypeChecker.java:156)
    at javassist.compiler.ast.CallExpr.accept(CallExpr.java:45)
    at javassist.compiler.CodeGen.doTypeCheck(CodeGen.java:241)
    at javassist.compiler.CodeGen.atStmnt(CodeGen.java:329)
    at javassist.compiler.ast.Stmnt.accept(Stmnt.java:49)
    at javassist.compiler.Javac.compileStmnt(Javac.java:568)
    at javassist.CtBehavior.insertAfterHandler(CtBehavior.java:914)
    at javassist.CtBehavior.insertAfter(CtBehavior.java:778)
    ... 17 more
Jon Snow, Ned Stark's bastard, likes to say "Winter is coming."
```

Here we see that when we tried to add the code using the *insertAfter* method, it couldn't find the *__start_time__* local variable added in the *insertBefore*method. This is a colossal pain in the neck. The beauty of Javassist is that you can insert real Java code without understanding much about byte code.

agent-example-7-profile-asm [Download]

This project produces an agent jar that demonstrates how you can add execution profiling code to byte code methods. The agent will instrument all methods in classes whose name match a regular expression provided in the agent arguments. Here is how the agent is initialized (exactly the same as the last project example):

```
    public static void premain(String agentArgs, Instrumentation inst) {
        initialize(agentArgs, inst, true);
    }

    public static void agentmain(String agentArgs, Instrumentation inst) {
        initialize(agentArgs, inst, false);
    }
```

```
    }

    static void initialize(String agentArgs, Instrumentation inst, boolean isPremain) {
        MyAgent.instrumentation = inst;
        inst.addTransformer(new MyAgent(Pattern.compile(agentArgs)), true);
    }

    public MyAgent(Pattern transformPattern) {
        this.transformPattern = transformPattern;
    }
```

And here is the *transform* method:

```
    public byte[] transform(ClassLoader loader, String className,
            Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
            byte[] classfileBuffer) throws IllegalClassFormatException {

        if (className != null && transformPattern.matcher(className.replace('/', '.')).matches()) {
            ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
            ProfileClassVisitor accessClassVisitor = new ProfileClassVisitor(cw);
            ClassReader cr = new ClassReader(classfileBuffer);
            cr.accept(accessClassVisitor, ClassReader.SKIP_FRAMES);

            return cw.toByteArray();
        }

        return null;
    }
```

The *transform* method hands off the byte code processing to the ProfileClassVisitor class:

```
public class ProfileClassVisitor extends ClassVisitor {

    String classDotName;

    public ProfileClassVisitor(ClassVisitor cv) {
        super(Opcodes.ASM5, cv);
    }

    @Override
    public void visit(int version, int access, String name, String signature, String superName, String[] interfaces) {
        super.visit(version, access, name, signature, superName, interfaces);
        this.classDotName = name.replace('/', '.');
    }

    @Override
    public MethodVisitor visitMethod(int access,
            String name,
            String desc,
            String signature,
            String[] exceptions) {

        MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);
        return new ProfileMethodVisitor(mv, access, name, desc);
    }

    class ProfileMethodVisitor extends AdviceAdapter {

        String methodName = null;
        String desc = null;

        int startTimeVar = -1;

        Label timeStart = new Label();
        Label timeEnd = new Label();

        Label finallyStart = new Label();
```

```
        Label finallyStart = new Label();
        Label finallyEnd = new Label();

        String signature = null;

        public ProfileMethodVisitor(MethodVisitor mv, int access, String name, String desc) {
            super(Opcodes.ASM5, mv, access, name, desc);
            this.methodName = name;
            this.desc = desc;

            signature = classDotName + '.' + methodName + toParameterString(desc);
        }

        String toParameterString(String desc) {
            Type methodType = Type.getMethodType(desc);
            StringBuilder buffer = new StringBuilder();

            buffer.append('(');

            Type argTypes[] = methodType.getArgumentTypes();
            for (int i=0; i<argTypes.length; i++) {
                buffer.append(argTypes[i].getClassName());

                if (i<argTypes.length-1)
                    buffer.append(", ");
            }

            buffer.append(')');

            return buffer.toString();
        }

        public void visitCode() {
            super.visitCode();

            visitLabel(timeStart);
            startTimeVar = newLocal(Type.LONG_TYPE);
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/System", "currentTimeMillis", "()J", false);
            mv.visitVarInsn(LSTORE, startTimeVar);

            visitLabel(finallyStart);
        }

        protected void onMethodExit(int opcode) {
            if (opcode != ATHROW)
                onFinally(opcode);
        }

        private void onFinally(int opcode) {
            if (opcode == ATHROW)
                mv.visitInsn(Opcodes.DUP);
            else
                mv.visitInsn(Opcodes.ACONST_NULL);

            int throwableVarIndex = newLocal(Type.getType(Throwable.class));
            mv.visitVarInsn(Opcodes.ASTORE, throwableVarIndex);

            mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
            mv.visitLdcInsn(signature + ": ");
            mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/PrintStream", "print", "(Ljava/lang/String;)V", false);

            mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/System", "currentTimeMillis", "()J", false);
            mv.visitVarInsn(Opcodes.LLOAD, startTimeVar);
```