ASM 3.0
A Java bytecode engineering library

Eric Bruneton

THIS DOCUMENTATION IS PROVIDED BY THE AUTHOR “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Version 1.0, February 2007

Contents

1 Introduction

1.1 Motivations

Program analysis, generation and transformation are useful techniques that can be used in many situations:

- Program analysis, which can range from a simple syntaxic parsing to a full semantic analysis, can be used to find potential bugs in applications, to detect unused code, to reverse engineer code, etc.
- Program generation is used in compilers. This include traditional compilers, but also stub or skeleton compilers used for distributed programming, Just in Time compilers, etc.
- Program transformation can be used to optimize or obfuscate programs, to insert debugging or performance monitoring code into applications, for aspect oriented programming, etc.

All these techniques can be used for any programming language, but this is more or less easy to do, depending on the language. In the case of Java they can be used on Java source code or on compiled Java classes. One of the advantages of working on compiled classes is that, obviously, the source code is not needed. Program transformations can therefore be used on any applications, including closed source and commercial ones. Another advantage of working on compiled code is that it becomes possible to analyze, generate or transform classes at runtime, just before they are loaded into the Java Virtual Machine (generating and compiling source code at runtime is possible, but this is really slow and requires a full Java compiler). The advantage is that tools such as stub compilers or aspect weavers become transparent to users.

Due to the many possible usages of program analysis, generation and transformation techniques, many tools to analyze, generate and transform programs have been implemented, for many languages, Java included. ASM is one of these tools for the Java language, designed for *runtime* – but also offline – class generation and transformation. The ASM[1] library was therefore designed to work on *compiled* Java classes. It was also designed to be as *fast* and as *small* as possible. Being as fast as possible is important in order not to slow down too much the applications that use ASM at runtime, for dynamic class generation or transformation. And being as small as possible is important in order to be used in memory constrained environments, and to avoid bloating the size of small applications or libraries using ASM.

ASM is not the only tool for generating and transforming compiled Java classes, but it is one of the most recent and efficient. It can be downloaded from `http://asm.objectweb.org`. Its main advantages are the following:

- It has a simple, well designed and modular API that is easy to use.
- It is well documented and has an associated Eclipse plugin.
- It provides support for the latest Java version, Java 6.
- It is small, fast, and very robust.
- Its large user community can provide support for new users.
- Its open source license allows you to use it in almost any way you want.

1.2 Overview

1.2.1 Scope

The goal of the ASM library is to generate, transform and analyze compiled Java classes, represented as byte arrays (as they are stored on disk and loaded in the Java Virtual Machine). For this purpose ASM provides tools to read, write and transform such byte arrays by using higher level concepts than bytes, such as numeric constants, strings, Java identifiers, Java types, Java class structure elements, etc. Note that the scope of the ASM library is strictly limited to reading, writing, transforming and analyzing classes. In particular the class loading process is out of scope.

[1] the ASM name does not mean anything: it is just a reference to the `__asm__` keyword in C, which allows some functions to be implemented in assembly language.

1.2.2 Model

The ASM library provides two APIs for generating and transforming compiled classes: the core API provides an *event based* representation of classes, while the tree API provides an *object based* representation.

With the event based model a class is represented with a sequence of events, each event representing an element of the class, such as its header, a field, a method declaration, an instruction, etc. The event based API defines the set of possible events and the order in which they must occur, and provides a class parser that generates one event per element that is parsed, as well as a class writer that generates compiled classes from sequences of such events.

With the object based model a class is represented with a tree of objects, each object representing a part of the class, such as the class itself, a field, a method, an instruction, etc. and each object having references to the objects that represent its constituents. The object based API provides a way to convert a sequence of events representing a class to the object tree representing the same class and, vice versa, to convert an object tree to the equivalent event sequence. In other words the object based API is built on top of the event based API.

These two APIs can be compared to the Simple API for XML (SAX) and Document Object Model (DOM) APIs for XML documents: the event based API is similar to SAX, while the object based API is similar to DOM. The object based API is built on top of the event based one, like DOM can be provided on top of SAX.

ASM provides both APIs because there is no best API. Indeed each API has its own advantages and drawbacks:

- The event based API is faster and requires less memory than the object based API, since there is no need to create and store in memory a tree of objects representing the class (the same difference also exists between SAX and DOM).
- However implementing class transformations can be more difficult with the event based API, since only one element of the class is available at any given time (the element that corresponds to the current event), while the whole class is available in memory with the object based API.

Note that the two APIs manage only one class at a time, and independently of the others: no information about the class hierarchy is maintained, and if a class transformation affects other classes, it is up to the user to modify these other classes.

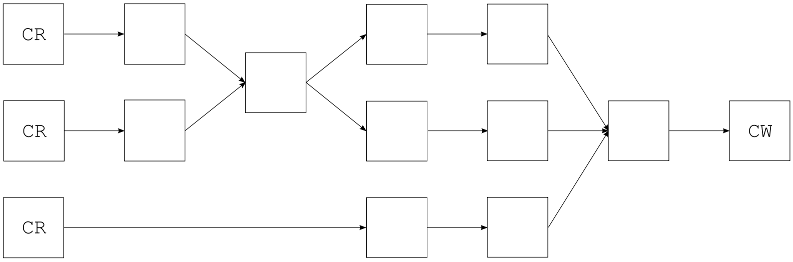
1.2.3 Architecture

ASM applications have a strong architectural aspect. Indeed the event based API is organized around event producers (the class parser), event consumers (the class writer) and various predefined event filters, to which user defined producers, consumers and filters can be added. Using this API is therefore a two step process:

- assembling event producer, filter and consumer components into possibly complex architectures,
- and then starting the event producers to run the generation or transformation process.

The object based API also has an architectural aspect: indeed class generator or transformer components that operate on object trees can be composed, the links between them representing the order of transformations.

Although most component architectures in typical ASM applications are quite simple, it is possible to imagine complex architectures like the following, where arrows represent event based or object based communications between class parsers, writers or transformers, with possible conversions between the event based and object based representations anywhere in the chain:



1.3 Organization

The ASM library is organized in several packages that are distributed in several `jar` files:

- the `org.objectweb.asm` and `org.objectweb.asm.signature` packages define the event based API and provide the class parser and writer components. They are contained in the `asm.jar` archive.
- the `org.objectweb.asm.util` package, in the `asm-util.jar` archive, provides various tools based on the core API that can be used during the development and debuging of ASM applications.
- the `org.objectweb.asm.commons` package provides several useful predefined class transformers, mostly based on the core API. It is contained in the `asm-commons.jar` archive.
- the `org.objectweb.asm.tree` package, in the `asm-tree.jar` archive, defines the object based API, and provides tools to convert between the event based and the object based representations.
- the `org.objectweb.asm.tree.analysis` package provides a class analysis framework and several predefined class analyzers, based on the tree API. It is contained in the `asm-analysis.jar` archive.

This document is organized in two parts. The first part covers the core API, i.e. the `asm`, `asm-util` and `asm-commons` archives. The second part covers the tree API, i.e. the `asm-tree` and `asm-analysis` archives. Each part contains at least one chapter for the API related to classes, one chapter for the API related to methods, and one chapter for the API related to annotations, generic types, etc. Each chapter covers the programming interfaces as well as the related tools and predefined components. The source code of all the examples is available on the ASM web site.

This organization makes it easier to introduce class file features progressively, but sometimes required to spread the presentation of a single ASM class in several sections. It is therefore recommanded to read this document in sequential order. For a reference guide about the ASM API, please use the Javadoc.

Typographic conventions

Italic is used for emphasizing elements in a sentence.

`Constant width` is used for code fragments.

`Bold constant width` is used for emphasizing code elements.

`Italic constant width` is used for variable parts in code and for labels.

1.4 Acknowledgments

I would like to thank François Horn for his valuable remarks during the elaboration of this document, which greatly improved its structure and readability.

Part I

Core API

2 Classes

This chapter explains how to generate and transform compiled Java classes with the core ASM API. It starts with a presentation of compiled classes and then presents the corresponding ASM interfaces, components and tools to generate and transform them, with many illustrative examples. The content of methods, annotations and generics are explained in the next chapters.

2.1 Structure

2.1.1 Overview

The overall structure of a compiled class is quite simple. Indeed, unlike natively compiled applications, a compiled class retains the structural information and almost all the symbols from the source code. In fact a compiled class contains:

- A section describing the modifiers (such as `public` or `private`), the name, the super class, the interfaces and the annotations of the class.
- One section per field declared in this class. Each section describes the modifiers, the name, the type and the annotations of a field.
- One section per method *and constructor* declared in this class. Each section describes the modifiers, the name, the return and parameter types, and the annotations of a method. It also contains the compiled code of the method, in the form of a sequence of Java bytecode instructions.

There are however some differences between source and compiled classes:

- A compiled class describes only one class, while a source file can contain several classes. For instance a source file describing a class with one inner class is compiled in two class files: one for the main class and one for the inner class. However the main class file contains *references* to its inner classes, and inner classes defined inside methods contain a *reference* to their enclosing method.

- A compiled class does not contain comments, of course, but can contain class, field, method and code *attributes* that can be used to associate additional information to these elements. Since the introduction of annotations in Java 5, which can be used for the same purpose, attributes have become mostly useless.
- A compiled class does not contain a `package` and `import` section, so all type names must be fully qualified.

Another very important structural difference is that a compiled class contains a *constant pool* section. This pool is an array containing all the numeric, string and type constants that appear in the class. These constants are defined only once, in the constant pool section, and are referenced by their index in all other sections of the class file. Hopefully ASM hides all the details related to the constant pool, so you will not have to bother about it. Figure 2.1 summarizes the overall structure of a compiled class. The exact structure is described in the Java Virtual Machine Specification, section 4.

Modifiers, name, super class, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code

Figure 2.1: Overall structure of a compiled class (* means zero or more)

Another important difference is that Java types are represented differently in compiled and source classes. The next sections explain their representation in compiled classes.

2.1.2 Internal names

In many situations a type is constrained to be a class or interface type. For instance the super class of a class, the interfaces implemented by a class, or the exceptions thrown by a method cannot be primitive types or array types, and are necessarily class or interface types. These types are represented in compiled classes with *internal names*. The internal name of a class is just the fully qualified name of this class, where dots are replaced with slashes. For example the internal name of `String` is `java/lang/String`.

2.1.3 Type descriptors

Internal names are used only for types that are constrained to be class or interface types. In all other situations, such as field types, Java types are represented in compiled classes with *type descriptors* (see Figure 2.2).

Java type	Type descriptor
`boolean`	`Z`
`char`	`C`
`byte`	`B`
`short`	`S`
`int`	`I`
`float`	`F`
`long`	`J`
`double`	`D`
`Object`	`Ljava/lang/Object;`
`int[]`	`[I`
`Object[][]`	`[[Ljava/lang/Object;`

Figure 2.2: Type descriptors of some Java types

The descriptors of the primitive types are single characters: `Z` for `boolean`, `C` for `char`, `B` for `byte`, `S` for `short`, `I` for `int`, `F` for `float`, `J` for `long` and `D` for `double`. The descriptor of a class type is the internal name of this class, preceded by `L` and followed by a semicolon. For instance the type descriptor of `String` is `Ljava/lang/String;`. Finally the descriptor of an array type is a square bracket followed by the descriptor of the array element type.

2.1.4 Method descriptors

A *method descriptor* is a list of type descriptors that describe the parameter types and the return type of a method, in a single string. A method descriptor starts with a left parenthesis, followed by the type descriptors of each formal parameter, followed by a right parenthesis, followed by the type descriptor of the return type, or `V` if the method returns `void` (a method descriptor does not contain the method's name or the argument names).

Method declaration in source file	Method descriptor
`void m(int i, float f)`	`(IF)V`
`int m(Object o)`	`(Ljava/lang/Object;)I`
`int[] m(int i, String s)`	`(ILjava/lang/String;)[I`
`Object m(int[] i)`	`([I)Ljava/lang/Object;`

Figure 2.3: Sample method descriptors

Once you know how type descriptors work, understanding method descriptors is easy. For instance `(I)I` describes a method that takes one argument of type `int`, and returns an `int`. Figure 2.3 gives several method descriptor examples.

2.2 Interfaces and components

2.2.1 Presentation

The ASM API for generating and transforming compiled classes is based on the `ClassVisitor` interface (see Figure 2.4). Each method in this interface corresponds to the class file structure section of the same name (see Figure 2.1). Simple sections are visited with a single method call whose arguments describe their content, and which returns `void`. Sections whose content can be of arbitrary length and complexity are visited with a initial method call that returns an auxiliary visitor interface. This is the case of the `visitAnnotation`, `visitField` and `visitMethod` methods, which return an `AnnotationVisitor`, a `FieldVisitor` and a `MethodVisitor` respectively.

The same principles are used recursively for these auxiliary interfaces. For example each method in the `FieldVisitor` interface (see Figure 2.5) corresponds to the class file sub structure of the same name, and `visitAnnotation`

```
public interface ClassVisitor {
  void visit(int version, int access, String name, String signature,
      String superName, String[] interfaces);
  void visitSource(String source, String debug);
  void visitOuterClass(String owner, String name, String desc);
  AnnotationVisitor visitAnnotation(String desc, boolean visible);
  void visitAttribute(Attribute attr);
  void visitInnerClass(String name, String outerName, String innerName,
      int access);
  FieldVisitor visitField(int access, String name, String desc,
      String signature, Object value);
  MethodVisitor visitMethod(int access, String name, String desc,
      String signature, String[] exceptions);
  void visitEnd();
}
```

Figure 2.4: The `ClassVisitor` interface

returns an auxiliary `AnnotationVisitor`, as in `ClassVisitor`. The creation and usage of these auxiliary visitors is explained in the next chapters: indeed this chapter is restricted to simple problems that can be solved with the `ClassVisitor` interface alone.

```
public interface FieldVisitor {
  AnnotationVisitor visitAnnotation(String desc, boolean visible);
  void visitAttribute(Attribute attr);
  void visitEnd();
}
```

Figure 2.5: The `FieldVisitor` interface

The methods of the `ClassVisitor` interface must be called in the following order, specified in the Javadoc of this interface:

```
visit visitSource? visitOuterClass? ( visitAnnotation | visitAttribute )*
( visitInnerClass | visitField | visitMethod )*
visitEnd
```

This means that `visit` must be called first, followed by at most one call to `visitSource`, followed by at most one call to `visitOuterClass`, followed by any number of calls in any order to `visitAnnotation` and `visitAttribute`, followed by any number of calls in any order to `visitInnerClass`, `visitField` and `visitMethod`, and terminated by a single call to `visitEnd`.

ASM provides three core components based on the `ClassVisitor` interface to generate and transform classes:

- The `ClassReader` class parses a compiled class given as a byte array, and calls the corresponding `visit`*`Xxx`* methods on the `ClassVisitor` instance passed as argument to its `accept` method. It can be seen as an event producer.
- The `ClassWriter` class is an implementation of the `ClassVisitor` interface that builds compiled classes directly in binary form. It produces as output a byte array containing the compiled class, which can be retrieved with the `toByteArray` method. It can be seen as an event consumer.
- The `ClassAdapter` class is a `ClassVisitor` implementation that delegates all the method calls it receives to another `ClassVisitor` instance. It can be seen as an event filter.

The next sections show with concrete examples how these components can be used to generate and transform classes.

2.2.2 Parsing classes

The only required component to parse an existing class is the `ClassReader` component. Let's take an example to illustrate this. Suppose that we would like to print the content of a class, in a similar way as the `javap` tool. The first step is to write an implementation of the `ClassVisitor` interface that prints information about the classes it visits. Here is a possible, overly simplified implementation:

```
public class ClassPrinter implements ClassVisitor {
  public void visit(int version, int access, String name,
      String signature, String superName, String[] interfaces) {
    System.out.println(name + " extends " + superName + " {");
  }
  public void visitSource(String source, String debug) {
  }
  public void visitOuterClass(String owner, String name, String desc) {
  }
  public AnnotationVisitor visitAnnotation(String desc,
      boolean visible) {
    return null;
  }
  public void visitAttribute(Attribute attr) {
  }
  public void visitInnerClass(String name, String outerName,
      String innerName, int access) {
  }
  public FieldVisitor visitField(int access, String name, String desc,
      String signature, Object value) {
    System.out.println("    " + desc + " " + name);
    return null;
  }
  public MethodVisitor visitMethod(int access, String name,
      String desc, String signature, String[] exceptions) {
    System.out.println("    " + name + desc);
    return null;
  }
  public void visitEnd() {
    System.out.println("}");
  }
}
```

The second step is to combine this `ClassPrinter` with a `ClassReader` component, so that the events produced by the `ClassReader` are consumed by our `ClassPrinter`:

```
ClassPrinter cp = new ClassPrinter();
ClassReader cr = new ClassReader("java.lang.Runnable");
cr.accept(cp, 0);
```

The second line creates a `ClassReader` to parse the `Runnable` class. The `accept` method called at the last line parses the `Runnable` class bytecode and calls the corresponding `ClassVisitor` methods on `cp`. The result is the following output:

```
java/lang/Runnable extends java/lang/Object {
    run()V
}
```

Note that there are several ways to construct a `ClassReader` instance. The class that must be read can be specified by name, as above, or by value, as a byte array or as an `InputStream`. An input stream to read the content of a class can be obtained with the `ClassLoader`'s `getResourceAsStream` method with:

```
cl.getResourceAsStream(classname.replace(’.’, ’/’) + ".class");
```

2.2.3 Generating classes

The only required component to generate a class is the ClassWriter component. Let's take an example to illustrate this. Consider the following interface:

```
package pkg;
public interface Comparable extends Mesurable {
  int LESS = -1;
  int EQUAL = 0;
  int GREATER = 1;
  int compareTo(Object o);
}
```

It can be generated with six method calls to a ClassVisitor:

```
ClassWriter cw = new ClassWriter(0);
cw.visit(V1_5, ACC_PUBLIC + ACC_ABSTRACT + ACC_INTERFACE,
    "pkg/Comparable", null, "java/lang/Object",
    new String[] { "pkg/Mesurable" });
cw.visitField(ACC_PUBLIC + ACC_FINAL + ACC_STATIC, "LESS", "I",
    null, new Integer(-1)).visitEnd();
cw.visitField(ACC_PUBLIC + ACC_FINAL + ACC_STATIC, "EQUAL", "I",
    null, new Integer(0)).visitEnd();
cw.visitField(ACC_PUBLIC + ACC_FINAL + ACC_STATIC, "GREATER", "I",
    null, new Integer(1)).visitEnd();
cw.visitMethod(ACC_PUBLIC + ACC_ABSTRACT, "compareTo",
    "(Ljava/lang/Object;)I", null, null).visitEnd();
cw.visitEnd();
byte[] b = cw.toByteArray();
```

The first line creates a ClassWriter instance that will actually build the byte array representation of the class (the constructor argument is explained in the next chapter).

The call to the visit method defines the class header. The V1_5 argument is a constant defined, like all other ASM constants, in the ASM Opcodes interface. It specifies the class version, Java 1.5. The ACC_*XXX* constants are flags that correspond to Java modifiers. Here we specify that the class is an interface, and that it is public and abstract (because it cannot be instantiated). The next argument specifies the class name, in internal form (see section 2.1.2). Recall that compiled classes do not contain a package or import section, so all class names must be fully qualified. The next argument corresponds to generics (see section 4.1). In our case it is null because the interface is not parameterized by a type variable. The fifth argument is the super class, in internal form (interface classes implicitly inherit from `Object`). The last argument is an array of the interfaces that are extended, specified by their internal names.

The next three calls to `visitField` are similar, and are used to define the three interface fields. The first argument is a set of flags that correspond to Java modifiers. Here we specify that the fields are `public`, `final` and `static`. The second argument is the name of the field, as it appears in source code. The third argument is the type of the field, in type descriptor form. Here the fields are `int` fields, whose descriptor is `I`. The fourth argument corresponds to generics. In our case it is `null` because the field types are not using generics. The last argument is the field's *constant* value: this argument must be used only for truly constant fields, i.e. `final static` fields. For other fields it must be `null`. Since there are no annotations here, we call the `visitEnd` method of the returned `FieldVisitor` immediately, i.e. without any call to its `visitAnnotation` or `visitAttribute` methods.

The `visitMethod` call is used to define the `compareTo` method. Here again the first argument is a set of flags that correspond to Java modifiers. The second argument is the method name, as it appears in source code. The third argument is the descriptor of the method. The fourth argument corresponds to generics. In our case it is `null` because the method is not using generics. The last argument is an array of the exceptions that can be thrown by the method, specified by their internal names. Here it is `null` because the method does not declare any exception. The `visitMethod` method returns a `MethodVisitor` (see Figure 3.4), which can be used to define the method's annotations and attributes, and most importantly the method's code. Here, since there are no annotations and since the method is abstract, we call the `visitEnd` method of the returned `MethodVisitor` immediately.

Finally a last call to `visitEnd` is used to inform `cw` that the class is finished and a call to `toByteArray` is used to retrieve it as a byte array.

Using generated classes

The previous byte array can be stored in a `Comparable.class` file for future use. Alternatively it can be loaded dynamically with a `ClassLoader`. One method is to define a `ClassLoader` subclass whose `defineClass` method is public:

```
class MyClassLoader extends ClassLoader {
  public Class defineClass(String name, byte[] b) {
    return defineClass(name, b, 0, b.length);
  }
}
```

Then the generated class can be loaded directly with:

```
Class c = myClassLoader.defineClass("pkg.Comparable", b);
```

Another method to load a generated class, which is probably cleaner, is to define a `ClassLoader` subclass whose `findClass` method is overridden in order to generate the requested class on the fly:

```
class StubClassLoader extends ClassLoader {
  @Override
  protected Class findClass(String name)
      throws ClassNotFoundException {
    if (name.endsWith("_Stub")) {
      ClassWriter cw = new ClassWriter(0);
      ...
      byte[] b = cw.toByteArray();
      return defineClass(name, b, 0, b.length);
    }
    return super.findClass(name);
  }
}
```

In fact the way of using your generated classes depends on the context, and is out of scope of the ASM API. If you are writing a compiler, the class generation process will be driven by an abstract syntax tree representing the program to be compiled, and the generated classes will be stored on disk. If you are writing a dynamic proxy class generator or aspect weaver you will use, in one way or another, a `ClassLoader`.

2.2.4 Transforming classes

So far the `ClassReader` and `ClassWriter` components were used alone. The events were produced "by hand" and consumed directly by a `ClassWriter` or, symetrically, they were produced by a `ClassReader` and consumed "by hand", i.e. by a custom `ClassVisitor` implementation. Things start to become really interesting when these components are used together. The first step is to direct the events produced by a `ClassReader` to a `ClassWriter`. The result is that the class parsed by the class reader is reconstructed by the class writer:

```
byte[] b1 = ...;
ClassWriter cw = new ClassWriter();
ClassReader cr = new ClassReader(b1);
cr.accept(cw, 0);
byte[] b2 = cw.toByteArray(); // b2 represents the same class as b1
```

This is not really interesting in itself (there are easier ways to copy a byte array!), but wait. The next step is to introduce a `ClassAdapter` between the class reader and the class writer:

```
byte[] b1 = ...;
ClasssWriter cw = new ClassWriter();
ClassAdapter ca = new ClassAdapter(cw); // ca forwards all events to cw
ClassReader cr = new ClassReader(b1);
cr.accept(ca, 0);
byte[] b2 = cw.toByteArray(); // b2 represents the same class as b1
```

The architecture corresponding to the above code is depicted in Figure 2.6, where components are represented with squares, and events with arrows (with a vertical time line as in sequence diagrams).

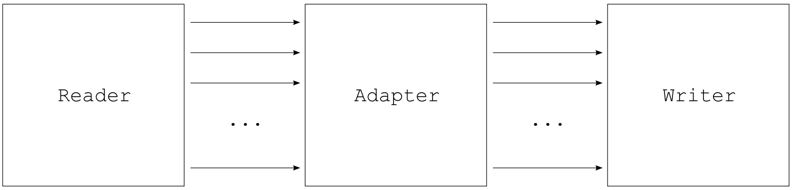


Figure 2.6: A transformation chain

The result does not change, however, because the `ClassAdapter` event filter does not filter anything. But it is now sufficient to filter some events, by overriding this class, in order to be able to transform a class. For example, consider the following `ClassAdapter` subclass:

```
public class ChangeVersionAdapter extends ClassAdapter {
  public ChangeVersionAdapter(ClassVisitor cv) {
    super(cv);
  }
  @Override
  public void visit(int version, int access, String name,
      String signature, String superName, String[] interfaces) {
    cv.visit(V1_5, access, name, signature, superName, interfaces);
  }
}
```

This class overrides only one method of the `ClassAdapter` class. As a consequence all calls are forwarded unchanged to the class visitor `cv` passed to the constructor, except calls to the `visit` method, which are forwarded with a modified class version number. The corresponding sequence diagram is shown in Figure 2.7.

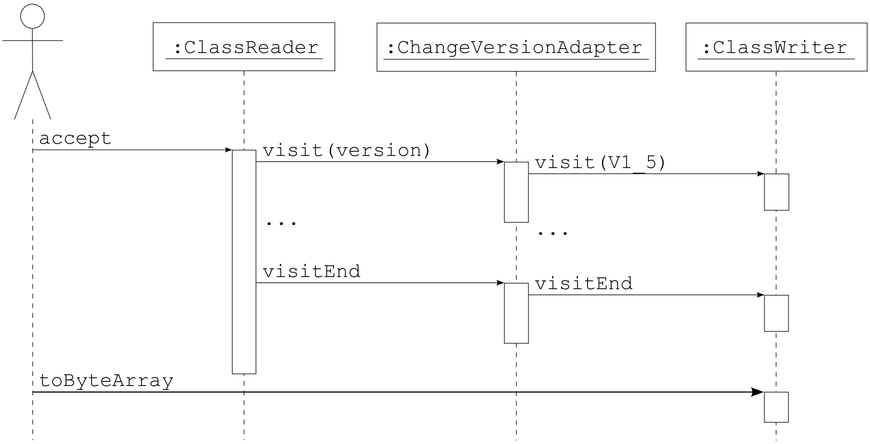


Figure 2.7: Sequence diagram for the `ChangeVersionAdapter`

By modifying other arguments of the `visit` method you can implement other transformations than just changing the class version. For instance you can add an interface to the list of implemented interfaces. It is also possible to change the name of the class, *but this requires much more than just changing the `name` argument in the `visit` method.* Indeed the name of the class can appear in many different places inside a compiled class, and *all* these occurrences must be changed to really rename the class.

Optimization

The previous transformation changes only four bytes in the original class. However, with the above code, `b1` is fully parsed and the corresponding events are used to construct `b2` from scratch, which is not very efficient. It would be much more efficient to copy the parts of `b1` that are not transformed directly into `b2`, without parsing these parts and without generating the corresponding events. ASM automatically performs this optimization for methods:

- If a `ClassReader` component detects that a `MethodVisitor` returned by the `ClassVisitor` passed as argument to its `accept` method comes from a `ClassWriter`, this means that the content of this method will not be transformed, and will in fact not even be *seen* by the application.

- In this case the `ClassReader` component does not parse the content of this method, does not generate the corresponding events, and just copies the byte array representation of this method in the `ClassWriter`.

This optimization is performed by the `ClassReader` and `ClassWriter` components if they have a reference to each other, which can be set like this:

```
byte[] b1 = ...
ClassReader cr = new ClassReader(b1);
ClassWriter cw = new ClassWriter(cr, 0);
ChangeVersionAdapter ca = new ChangeVersionAdapter(cw);
cr.accept(ca, 0);
byte[] b2 = cw.toByteArray();
```

Thanks to this optimization the above code is *two times* faster than the previous one, because `ChangeVersionAdapter` does not transform any method. For common class transformations, which transform some or all methods, the speedup is smaller, but is still noticeable: it is indeed of the order of 10 to 20%. Unfortunately this optimization requires to copy all the constants defined in the original class into the transformed one. This is not a problem for tranformations that *add* fields, methods or instructions, but this leads to bigger class files, compared to the unoptimized case, for transformations that *remove* or *rename* many class elements. It is therefore recommanded to use this optimization only for "additive" transformations.

Using transformed classes

The transformed class `b2` can be stored on disk or loaded with a `ClassLoader`, as described in the previous section. But class transformations done inside a `ClassLoader` can only transform the classes loaded by this class loader. If you want to transform *all* classes you will have to put your transformation inside a `ClassFileTransformer`, as defined in the `java.lang.instrument` package (see the documentation of this package for more details):

```
public static void premain(String agentArgs, Instrumentation inst) {
  inst.addTransformer(new ClassFileTransformer() {
    public byte[] transform(ClassLoader l, String name, Class c,
        ProtectionDomain d, byte[] b)
```

```
          throws IllegalClassFormatException {
        ClassReader cr = new ClassReader(b);
        ClassWriter cw = new ClassWriter(cr, 0);
        ClassVisitor cv = new ChangeVersionAdapter(cw);
        cr.accept(cv, 0);
        return cw.toByteArray();
      }
    });
  }
```

2.2.5 Removing class members

The method used to transform the class version in the previous section can of course be applied to other methods of the `ClassVisitor` interface. For instance, by changing the `access` or `name` argument in the `visitField` and `visitMethod` methods, you can change the modifiers or the name of a field or of a method. Furthermore, instead of forwarding a method call with modified arguments, you can choose to *not* forward this call at all. The effect is that the corresponding class element is *removed*.

For example the following class adapter removes the information about outer and inner classes, as well as the name of the source file from which the class was compiled (the resulting class remains fully functional, because these elements are only used for debugging purposes). This is done by not forwarding anything in the appropriate visit methods:

```
public class RemoveDebugAdapter extends ClassAdapter {
  public RemoveDebugAdapter(ClassVisitor cv) {
    super(cv);
  }
  @Override
  public void visitSource(String source, String debug) {
  }
  @Override
  public void visitOuterClass(String owner, String name, String desc) {
  }
  @Override
  public void visitInnerClass(String name, String outerName,
      String innerName, int access) {
  }
}
```

This strategy does not work for fields and methods, because the `visitField` and `visitMethod` methods must return a result. In order to remove a field or method, you must not forward the method call, and return `null` to the caller. For example the following class adapter removes a single method, specified by its name and by its descriptor (the name is not sufficient to identify a method, because a class can contain several methods of the same name but with different parameters):

```
public class RemoveMethodAdapter extends ClassAdapter {
  private String mName;
  private String mDesc;
  public RemoveMethodAdapter(
      ClassVisitor cv, String mName, String mDesc) {
    super(cv);
    this.mName = mName;
    this.mDesc = mDesc;
  }
  @Override
  public MethodVisitor visitMethod(int access, String name,
      String desc, String signature, String[] exceptions) {
    if (name.equals(mName) && desc.equals(mDesc)) {
      // do not delegate to next visitor -> this removes the method
      return null;
    }
    return cv.visitMethod(access, name, desc, signature, exceptions);
  }
}
```

2.2.6 Adding class members

Instead of forwarding fewer calls than you receive, you can “forward” more, which has the effect of *adding* class elements. The new calls can be inserted at several places between the original method calls, provided that the order in which the various `visit`***`Xxx`*** methods must be called is respected (see section 2.2.1).

For instance, if you want to add a field to a class you have to insert a new call to `visitField` between the original method calls, and you must put this new call in one of the visit method of your class adapter. You cannot do this in the `visit` method, for example, because this may result in a call to `visitField` followed by `visitSource`, `visitOuterClass`, `visitAnnotation` or `visitAttribute`, which is not valid. You cannot put this new call in the `visitSource`, `visitOuterClass`, `visitAnnotation` or `visitAttribute` methods, for the same reason. The only possibilities are the `visitInnerClass`, `visitField`, `visitMethod` or `visitEnd` methods.

If you put the new call in the `visitEnd` method the field will always be added (unless you add explicit conditions), because this method is always called. If you put it in `visitField` or `visitMethod`, *several* fields will be added: one per field or method in the original class. Both solutions can make sense; it depends on what you need. For instance you can add a single counter field to count the invocations on an object, or one counter per method, to count the invocations of each method separately.

Note: in fact the only truly correct solution is to add new members by making additional calls in the `visitEnd` method. Indeed a class must not contain duplicate members, and the only way to be sure that a new member is unique is to compare it with all the existing members, which can only be done once they have all been visited, i.e. in the `visitEnd` method. This is rather constraining. Using generated names that are unlikely to be used by a programmer, such as `_counter$` or `_4B7F_` is sufficient in practice to avoid duplicate members without having to add them in `visitEnd`. Note that, as discussed in the first chapter, the tree API does not have this limitation: it is possible to add new members at any time inside a transformation with this API.

In order to illustrate the above discussion, here is a class adapter that adds a field to a class, unless this field already exists:

```
public class AddFieldAdapter extends ClassAdapter {
  private int fAcc;
  private String fName;
  private String fDesc;
  private boolean isFieldPresent;
  public AddFieldAdapter(ClassVisitor cv, int fAcc, String fName,
      String fDesc) {
    super(cv);
    this.fAcc = fAcc;
    this.fName = fName;
    this.fDesc = fDesc;
  }
  @Override
  public FieldVisitor visitField(int access, String name, String desc,
      String signature, Object value) {
    if (name.equals(fName)) {
      isFieldPresent = true;
    }
    return cv.visitField(access, name, desc, signature, value);
  }
  @Override
```

```
    public void visitEnd() {
      if (!isFieldPresent) {
        FieldVisitor fv = cv.visitField(fAcc, fName, fDesc, null, null);
        if (fv != null) {
          fv.visitEnd();
        }
      }
      cv.visitEnd();
    }
  }
```

The field is added in the `visitEnd` method. The `visitField` method is not overridden to modify existing fields or to remove a field, but just to detect if the field we want to add already exists or not. Note the `fv != null` test in the `visitEnd` method, before calling `fv.visitEnd()`: this is because, as we have seen in the previous section, a class visitor can return `null` in `visitField`.

2.2.7 Transformation chains

So far we have seen simple transformation chains made of a `ClassReader`, a class adapter, and a `ClassWriter`. It is of course possible to use more complex chains, with *several* class adapters chained together. Chaining several adapters allows you to compose several independent class transformations in order to do complex transformations. Note also that a transformation chain is not necessarily linear. You can write a `ClassVisitor` that forwards all the method calls it receives to several `ClassVisitor` at the same time:

```
  public class MultiClassAdapter implements ClassVisitor {
    protected ClassVisitor[] cvs;
    public MultiClassAdapter(ClassVisitor[] cvs) {
      this.cvs = cvs;
    }
    @Override public void visit(int version, int access, String name,
        String signature, String superName, String[] interfaces) {
      for (ClassVisitor cv : cvs) {
        cv.visit(version, access, name, signature, superName, interfaces);
      }
    }
    ...
  }
```

Symmetrically several class adapters can delegate to the same `ClassVisitor` (this requires some precautions to ensure, for example, that the `visit` and visitEnd methods are called exactly once on this ClassVisitor). Thus a transformation chain such as the one shown in Figure 2.8 is perfectly possible.

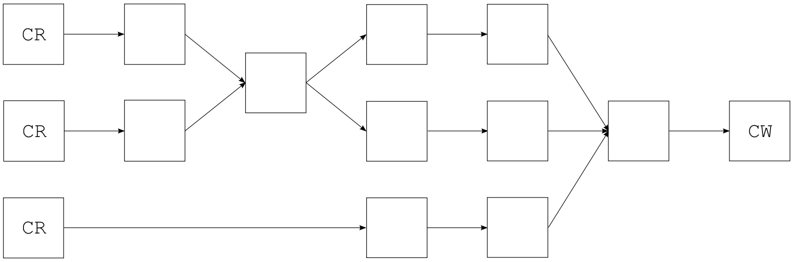


Figure 2.8: A complex transformation chain

2.3 Tools

In addition to the ClassVisitor interface and to the three related components ClassReader, ClassAdapter and ClassWriter, ASM provides, in the org.objectweb.asm.util package, several tools that can be useful during the development of a class generator or adapter, but which are not needed at runtime. ASM also provides a utility class for manipulating internal names, type descriptors and method descriptors at runtime. All these tools are presented below.

2.3.1 Type

As you have seen in the previous sections, the ASM API exposes Java types as they are stored in compiled classes, i.e. as internal names or type descriptors. It would be possible to expose them as they appear in source code, to make code more readable. But this would require systematic conversions between the two representations in ClassReader and ClassWriter, which would degrade performances. This is why ASM does not transparently transform internal names and type descriptors to their equivalent source code form. However it provides the Type class for doing that manually when necessary.

A Type object represents a Java type, and can be constructed either from a type descriptor or from a Class object. The Type class also contains static variables representing the primitive types. For example Type.INT_TYPE is the Type object representing the int type.

The getInternalName method returns the internal name of a Type. For example Type.getType(String.class).getInternalName() gives the internal name of the String class, i.e. "java/lang/String". This method must be used only for class or interface types.

The getDescriptor method returns the descriptor of a Type. So, for example, instead of using "Ljava/lang/String;" in your code you could use Type.getType(String.class).getDescriptor(). Or, instead of using I, you could use Type.INT_TYPE.getDescriptor().

The Type class also provides static methods to get the Type objects corresponding to the argument types and return types of a method, defined by its descriptor or by a java.lang.reflect.Method object. For instance Type.getArgumentTypes("(I)V") returns an array containing the single element Type.INT_TYPE. Similarly Type.getReturnType("(I)V") returns the Type.VOID_TYPE object.

2.3.2 TraceClassVisitor

In order to check that a generated or transformed class is conforming to what you expect, the byte array returned by a ClassWriter is not really helpful because it is unreadable by humans. A textual representation would be much easier to use. This is what the TraceClassVisitor class provides. This class, as its name implies, implements the ClassVisitor interface, and builds a textual representation of the visited class. So, instead of using a ClassWriter to generate your classes, you can use a TraceClassVisitor, in order to get a readable trace of what is actually generated. Or, even better, you can use both at the same time. Indeed the TraceClassVisitor can, in addition to its default behavior, delegate all calls to its methods to another visitor, for instance a ClassWriter:

```
ClassWriter cw = new ClassWriter(0);
TraceClassVisitor cv = new TraceClassVisitor(cw, printWriter);
cv.visit(...);
...
cv.visitEnd();
byte b[] = cw.toByteArray();
```

This code creates a `TraceClassVisitor` that delegates all the calls it receives to `cw`, and that prints a textual representation of these calls to `printWriter`. For example, using a `TraceClassVisitor` in the example of section 2.2.3 would give:

```
// class version 49.0 (49)
// access flags 1537
public abstract interface pkg/Comparable implements pkg/Mesurable  {
  // access flags 25
  public final static I LESS = -1
  // access flags 25
  public final static I EQUAL = 0
  // access flags 25
  public final static I GREATER = 1
  // access flags 1025
  public abstract compareTo(Ljava/lang/Object;)I
}
```

Note that you can use a `TraceClassVisitor` at any point in a generation or transformation chain, and not only just before a `ClassWriter`, in order to see what happens at this point in the chain. Note also that the textual representation of classes generated by this adapter can be used to compare classes easily, with `String.equals()`.

2.3.3 CheckClassAdapter

The `ClassWriter` class does not check that its methods are called in the appropriate order and with valid arguments. It is therefore possible to generate invalid classes that will be rejected by the Java Virtual Machine verifier. In order to detect some of these errors as soon as possible, it is possible to use the `CheckClassAdapter` class. Like `TraceClassVisitor`, this class implements the `ClassVisitor` interface, and delegates all calls to its method to another `ClassVisitor`, for instance a `TraceClassVisitor` or a `ClassWriter`. However, instead of printing a textual representation of the visited class, this class checks that its methods are called in the appropriate order, and with valid arguments, before delegating to the next visitor. In case of errors an `IllegalStateException` or `IllegalArgumentException` is thrown.

In order to check a class, print a textual representation of this class, and finally create a byte array representation, you should use something like:

```
ClassWriter cw = new ClassWriter(0);
```

```
TraceClassVisitor tcv = new TraceClassVisitor(cw, printWriter);
CheckClassAdapter cv = new CheckClassAdapter(tcv);
cv.visit(...);
...
cv.visitEnd();
byte b[] = cw.toByteArray();
```

Note that if you chain these class visitors in a different order, the operations they perform will be done in a different order too. For example, with the following code, the checks will take place *after* the trace:

```
ClassWriter cw = new ClassWriter(0);
CheckClassAdapter cca = new CheckClassAdapter(cw);
TraceClassVisitor cv = new TraceClassVisitor(cca, printWriter);
```

Like with `TraceClassVisitor`, you can use a `CheckClassAdapter` at any point in a generation or transformation chain, and not only just before a `ClassWriter`, in order to check classes at this point in the chain.

2.3.4 `ASMifierClassVisitor`

This class implements the `ClassVisitor` interface in such a way that each method prints the Java code that was used to call it. For instance calling `visitEnd()` prints `cv.visitEnd();`. The result is that, when this visitor visits a class, it prints the source code to generate this class with ASM. This is useful if you use this visitor to visit an already existing class. For instance, if you don't know how to generate some compiled class with ASM, write the corresponding source code, compile it with `javac`, and visit the compiled class with the `ASMifierClassVisitor`. You will get the ASM code to generate this compiled class!

The `ASMifierClassVisitor` class can be used from the command line. For example using:

```
java -classpath asm.jar:asm-util.jar \
     org.objectweb.asm.util.ASMifierClassVisitor \
     java.lang.Runnable
```

produces code that, after indentation, reads:

```
package asm.java.lang;
import org.objectweb.asm.*;
public class RunnableDump implements Opcodes {
  public static byte[] dump() throws Exception {
    ClassWriter cw = new ClassWriter(0);
    FieldVisitor fv;
    MethodVisitor mv;
    AnnotationVisitor av0;
    cw.visit(V1_5, ACC_PUBLIC + ACC_ABSTRACT + ACC_INTERFACE,
        "java/lang/Runnable", null, "java/lang/Object", null);
    {
      mv = cw.visitMethod(ACC_PUBLIC + ACC_ABSTRACT, "run", "()V",
          null, null);
      mv.visitEnd();
    }
    cw.visitEnd();
    return cw.toByteArray();
  }
}
```

3 Methods

This chapter explains how to generate and transform compiled methods with the core ASM API. It starts with a presentation of compiled methods and then presents the corresponding ASM interfaces, components and tools to generate and transform them, with many illustrative examples.

3.1 Structure

Inside compiled classes the code of methods is stored as a sequence of *bytecode* instructions. In order to generate and transform classes it is fundamental to know these instructions and to understand how they work. This section gives an overview of these instructions which should be sufficient to start coding simple class generators and transformers. For a complete definition you should read the Java Virtual Machine Specification.

3.1.1 Execution model

Before presenting the bytecode instructions it is necessary to present the Java Virtual Machine execution model. As you know Java code is executed inside *threads*. Each thread has its own execution stack, which is made of *frames*. Each frame represents a method invocation: each time a method is invoked, a new frame is pushed on the current thread's execution stack. When the method returns, either normally or because of an exception, this frame is popped from the execution stack and execution continues in the calling method (whose frame is now on top of the stack).

Each frame contains two parts: a local variables part and an operand stack part. The *local variables* part contains variables that can be accessed by their index, in random order. The *operand stack* part, as its name implies, is a stack of values that are used as operands by bytecode instructions. This means that the values in this stack can only be accessed in Last In First Out order. Do not confuse the operand stack and the thread's execution stack: each frame in the execution stack contains its *own* operand stack.

The size of the local variables and operand stack parts depends on the method's code. It is computed at compile time and is stored along with the bytecode instructions in compiled classes. As a consequence, all the frames that correspond to the invocation of a given method have the same size, but frames that correspond to different methods can have different sizes for their local variables and operand stack parts.

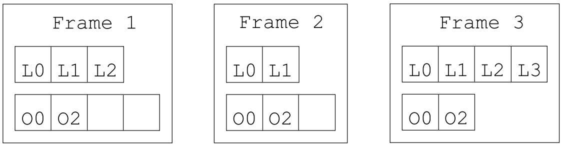


Figure 3.1: An execution stack with 3 frames

Figure 3.1 shows a sample execution stack with 3 frames. The first frame contains 3 local variables, its operand stack has a maximum size of 4, and it contains two values. The second frame contains 2 local variables, and two values in its operand stack. Finally the third frame, on top of the execution stack, contains 4 local variables and two operands.

When it is created, a frame is initialized with an empty stack, and its local variables are initialized with the target object `this` (for non static methods) and with the method's arguments. For instance, calling the method `a.equals(b)` creates a frame with an empty stack and with the first two local variables initialized to `a` and `b` (other local variables are uninitialized).

Each slot in the local variables and operand stack parts can hold any Java value, except `long` and `double` values. These values require two slots. This complicates the management of local variables: for instance the i^{th} method argument is not necessarily stored in local variable i. For example, calling `Math.max(1L, 2L)` creates a frame with the `1L` value in the first two local variable slots, and with the value `2L` in the third and fourth slots.

3.1.2 Bytecode instructions

A bytecode instruction is made of an opcode that identifies this instruction, and of a fixed number of arguments:

- The *opcode* is an unsigned byte value – hence the bytecode name – and is identified by a mnemonic symbol. For example the opcode value 0 is designed by the mnemonic symbol `NOP`, and corresponds to the instruction that does nothing.
- The *arguments* are static values that define the precise instruction behavior. They are given just after the opcode. For instance the `GOTO` *label* instruction, whose opcode value is 167, takes as argument *label*, a label that designates the next instruction to be executed. Instruction arguments must not be confused with instruction operands: argument values are statically known and are stored in the compiled code, while operand values come from the operand stack and are known only at runtime.

The bytecode instructions can be divided in two categories: a small set of instructions is designed to transfer values from the local variables to the operand stack, and vice versa; the other instructions only act on the operand stack: they pop some values from the stack, compute a result based on these values, and push it back on the stack.

The `ILOAD`, `LLOAD`, `FLOAD`, `DLOAD`, and `ALOAD` instructions read a local variable and push its value on the operand stack. They take as argument the index i of the local variable that must be read. `ILOAD` is used to load a `boolean`, `byte`, `char`, `short`, or `int` local variable. `LLOAD`, `FLOAD` and `DLOAD` are used to load a `long`, `float` or `double` value, respectively (`LLOAD` and `DLOAD` actually load the two slots i and $i+1$). Finally `ALOAD` is used to load any non primitive value, i.e. object and array references. Symmetrically the `ISTORE`, `LSTORE`, `FSTORE`, `DSTORE` and `ASTORE` instructions pop a value from the operand stack and store it in a local variable designated by its index `i`.

As you can see the x`LOAD` and x`STORE` instructions are typed (in fact, as you will see below, almost all instructions are typed). This is used to ensure that no illegal conversion is done. Indeed it is illegal to store a value in a local variable and then to load it with a different type. For instance the `ISTORE 1 ALOAD 1` sequence is illegal – it would allow to store an arbitrary memory address in local variable 1, and to convert this address to an object reference! It is however perfectly legal to store in a local variable a value whose type differ from the type of the current value stored in this local variable. This means that the type of a local variable, i.e. the type of the value stored in this local variable, can change during the execution of a method.

As said above, all other bytecode instructions work on the operand stack only. They can be grouped in the following categories (see appendix A.1):

Stack These instructions are used to manipulate values on the stack: `POP` pops the value on top of the stack, `DUP` pushes a copy of the top stack value, `SWAP` pops two values and pushes them in the reverse order, etc.

Constants These instructions push a constant value on the operand stack: `ACONST_NULL` pushes `null`, `ICONST_0` pushes the int value `0`, `FCONST_0` pushes `0f`, `DCONST_0` pushes `0d`, `BIPUSH` *b* pushes the byte value *b*, `SIPUSH` *s* pushes the short value *s*, `LDC` *cst* pushes the arbitrary `int`, `float`, `long`, `double`, `String`, or `class`[1] constant *cst*, etc.

Arithmetic and logic These instructions pop numeric values from the operand stack combine them and push the result on the stack. They do not have any argument. *x*`ADD`, *x*`SUB`, *x*`MUL`, *x*`DIV` and *x*`REM` correspond to the `+`, `-`, `*`, `/` and `%` operations, where *x* is either `I`, `L`, `F` or `D`. Similarly there are other instructions corresponding to `<<`, `>>`, `>>>`, `|`, `&` and `^`, for `int` and `long` values.

Casts These instructions pop a value from the stack, convert it to another type, and push the result back. They correspond to cast expressions in Java. `I2F`, `F2D`, `L2D`, etc. convert numeric values from one numeric type to another. `CHECKCAST` *t* converts a reference value to the type *t*.

Objects These instructions are used to create objects, lock them, test their type, etc. For instance the `NEW` *type* instruction pushes a new object of type *type* on the stack (where *type* is an internal name).

Fields These instructions read or write the value of a field. `GETFIELD` *owner name desc* pops an object reference, and pushes the value of its *name* field. `PUTFIELD` *owner name desc* pops a value and an object reference, and stores this value in its *name* field. In both cases the object must be of type *owner*, and its field must be of type *desc*. `GETSTATIC` and `PUTSTATIC` are similar instructions, but for static fields.

Methods These instructions invoke a method or a constructor. They pop as many values as there are method arguments, plus one value for the target

[1] this corresponds to the *identifier*`.class` Java syntax.

object, and push the result of the method invocation. `INVOKEVIRTUAL` *owner name desc* invokes the *name* method defined in class *owner*, and whose method descriptor is *desc*. `INVOKESTATIC` is used for static methods, `INVOKESPECIAL` for private methods and constructors, and `INVOKEINTERFACE` for methods defined in interfaces.

Arrays These instructions are used to read and write values in arrays. The *x*`ALOAD` instructions pop an index and an array, and push the value of the array element at this index. The *x*`ASTORE` instructions pop a value, an index and an array, and store this value at that index in the array. Here *x* can be `I`, `L`, `F`, `D` or `A`, but also `B`, `C` or `S`.

Jumps These instructions jump to an arbitrary instruction if some condition is true, or unconditionally. They are used to compile `if`, `for`, `do`, `while`, `break` and `continue` instructions. For instance `IFEQ` *label* pops an `int` value from the stack, and jumps to the instruction designed by *label* if this value is 0 (otherwise execution continues normally to the next instruction). Many other jump instructions exist, such as `IFNE` or `IFGE`. Finally `TABLESWITCH` and `LOOKUPSWITCH` correspond to the `switch` Java instruction.

Return Finally the *x*`RETURN` and `RETURN` instructions are used to terminate the execution of a method and to return its result to the caller. `RETURN` is used for methods that return `void`, and *x*`RETURN` for the other methods.

3.1.3 Examples

Lets look at some basic examples to get a more concrete sense of how bytecode instructions work. Consider the following bean class:

```
package pkg;
public class Bean {
  private int f;
  public int getF() {
    return this.f;
  }
  public void setF(int f) {
    this.f = f;
  }
}
```

The bytecode of the getter method is:

```
ALOAD 0
GETFIELD pkg/Bean f I
IRETURN
```

The first instruction reads the local variable 0, which was initialized to **this** during the creation of the frame for this method call, and pushes this value on the operand stack. The second instruction pops this value from the stack, i.e. **this**, and pushes the **f** field of this object, i.e. **this.f**. The last instruction pops this value from the stack, and returns it to the caller. The successive states of the execution frame for this method are shown in Figure 3.2.

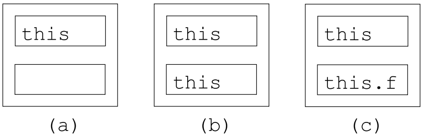


Figure 3.2: Successive frame states for the **getF** method: a) initial state, b) after **ALOAD 0** and c) after **GETFIELD**

The bytecode of the setter method is:

```
ALOAD 0
ILOAD 1
PUTFIELD pkg/Bean f I
RETURN
```

The first instruction pushes **this** on the operand stack, as before. The second instruction pushes the local variable 1, which was initialized with the **f** argument value during the creation of the frame for this method call. The third instruction pops these two values and stores the **int** value in the **f** field of the referenced object, i.e. in **this.f**. The last instruction, which is implicit in the source code but which is mandatory in the compiled code, destroys the current execution frame and returns to the caller. The successive states of the execution frame for this method are shown in Figure 3.3.

The **Bean** class also has a default public constructor which is generated by the compiler, since no explicit constructor was defined by the programmer. This default public constructor is generated as **Bean() { super(); }**. The bytecode of this constructor is the following:

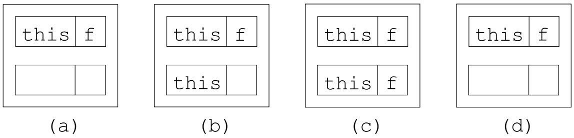


Figure 3.3: Successive frame states for the `setF` method: a) initial state, b) after `ALOAD 0`, c) after `ILOAD 1` and d) after `PUTFIELD`

```
ALOAD 0
INVOKESPECIAL java/lang/Object <init> ()V
RETURN
```

The first instruction pushes `this` on the operand stack. The second instruction pops this value from the stack, and calls the `<init>` method defined in the `Object` class. This corresponds to the `super()` call, i.e. a call to the constructor of the super class, `Object`. You can see here that constructors are named differently in compiled and source classes: in compiled classes they are always named `<init>`, while in source classes they have the name of the class in which they are defined. Finally the last instruction returns to the caller.

Now let us consider a slightly more complex setter method:

```
public void checkAndSetF(int f) {
  if (f >= 0) {
    this.f = f;
  } else {
    throw new IllegalArgumentException();
  }
}
```

The bytecode for this new setter method is the following:

```
  ILOAD 1
  IFLT label
  ALOAD 0
  ILOAD 1
  PUTFIELD pkg/Bean f I
  GOTO end
label:
  NEW java/lang/IllegalArgumentException
  DUP
  INVOKESPECIAL java/lang/IllegalArgumentException <init> ()V
  ATHROW
end:
  RETURN
```

The first instruction pushes the local variable 1, initialized to `f`, on the operand stack. The `IFLT` instruction pops this value from the stack, and compares it to 0. If it is Less Than (`LT`) 0, it jumps to the instruction designated by the *`label`* label, otherwise it does nothing and the execution continues to the next instruction. The next three instructions are the same instructions as in the `setF` method. The `GOTO` instruction unconditionally jumps to the instruction designated by the *`end`* label, which is the `RETURN` instruction. The instructions between the *`label`* and *`end`* labels create and throw an exception: the `NEW` instruction creates an exception object and pushes it on the operand stack. The `DUP` instruction duplicates this value on the stack. The `INVOKESPECIAL` instruction pops one of these two copies and calls the exception constructor on it. Finally the `ATHROW` instruction pops the remaining copy and throws it as an exception (so the execution does not continue to the next instruction).

3.1.4 Exception handlers

There is no bytecode instruction to catch exceptions: instead the bytecode of a method is associated with a list of *exception handlers* that specify the code that must be executed when an exception is thrown in a given part of a method. An exception handler is similar to a `try catch` block: it has a range, which is a sequence of instructions that corresponds to the content of the `try` block, and a handler, which corresponds to the content of the `catch` block. The range is specified by a start and end labels, and the handler with a start label. For example the source code below:

```
public static void sleep(long d) {
  try {
    Thread.sleep(d);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

can be compiled into:

```
TRYCATCHBLOCK try catch catch java/lang/InterruptedException
try:
  LLOAD 0
  INVOKESTATIC java/lang/Thread sleep (J)V
  RETURN
catch:
  INVOKEVIRTUAL java/lang/InterruptedException printStackTrace ()V
  RETURN
```

The code between the *try* and *catch* labels corresponds to the try block, while the code after the *catch* label corresponds to the catch block. The TRYCATCHBLOCK line specifies an exception handler that covers the range between the *try* and *catch* labels, with a handler starting at the *catch* label, and for exceptions whose class is a subclass of InterruptedException. This means that if such an exception is thrown anywhere between *try* and *catch* the stack is cleared, the exception is pushed on this empty stack, and execution continues at *catch*.

3.1.5 Frames

Classes compiled with Java 6 or higher contain, in addition to bytecode instructions, a set of *stack map frames* that are used to speed up the class verification process inside the Java Virtual Machine. A stack map frame gives the state of the execution frame of a method at some point during its execution. More precisely it gives the *type* of the values that are contained in each local variable slot and in each operand stack slot just before some specific bytecode instruction is executed.

For example, if we consider the getF method of the previous section, we can define three stack map frames giving the state of the execution frame just before ALOAD, just before GETFIELD, and just before IRETURN. These three stack map frames correspond to the three cases shown in Figure 3.2 and can be described as follows, where the types between the first square brackets correspond to the local variables, and the others to the operand stack:

State of the execution frame *before*	Instruction
[pkg/Bean] []	ALOAD 0
[pkg/Bean] [pkg/Bean]	GETFIELD
[pkg/Bean] [I]	IRETURN

We can do the same for the checkAndSetF method:

State of the execution frame *before*	Instruction
[pkg/Bean I] []	ILOAD 1
[pkg/Bean I] [I]	IFLT *label*
[pkg/Bean I] []	ALOAD 0
[pkg/Bean I] [pkg/Bean]	ILOAD 1
[pkg/Bean I] [pkg/Bean I]	PUTFIELD
[pkg/Bean I] []	GOTO *end*
[pkg/Bean I] []	*label*:
[pkg/Bean I] []	NEW
[pkg/Bean I] [Uninitialized(label)]	DUP
[pkg/Bean I] [Uninitialized(label) Uninitialized(label)]	INVOKESPECIAL
[pkg/Bean I] [java/lang/IllegalArgumentException]	ATHROW
[pkg/Bean I] []	*end*:
[pkg/Bean I] []	RETURN

This is similar to the previous method, except for the Uninitialized(label) type. This is a special type that is used only in stack map frames, and that designates an object whose memory has been allocated but whose constructor has not been called yet. The argument designates the instruction that created this object. The only possible method that can be called on a value of this type is a constructor. When it is called, *all the occurrences* of this type in the frame are replaced with the real type, here `IllegalArgumentException`. Stack map frames can use three other special types: `UNINITIALIZED_THIS` is the initial type of local variable 0 in constructors, `TOP` corresponds to an undefined value, and `NULL` corresponds to `null`.

As said above, starting from Java 6, compiled classes contain, in addition to bytecode, a set of stack map frames. In order to save space, a compiled method does not contain one frame per instruction: in fact it contains only the frames for the instructions that correspond to jump targets or exception handlers, or that follow unconditional jump instructions. Indeed the other frames can be easily and quickly inferred from these ones.

In the case of the `checkAndSetF` method, this means that only two frames are stored: one for the `NEW` instruction, because it is the target of the `IFLT` instruction, but also because it follows the unconditional jump `GOTO` instruction, and one for the `RETURN` instruction, because it is the target of the `GOTO` instruction, and also because it follows the "unconditional jump" `ATHROW` instruction.

In order to save even more space, each frame is compressed by storing only its difference compared to the previous frame, and the initial frame is not stored at all, because it can easily be deduced from the method parameter types. In the case of the checkAndSetF method the two frames that must be stored are equal and are equal to the initial frame, so they are stored as the single byte value designated by the F_SAME mnemonic. These frames can be represented just before their associated bytecode instruction. This gives the final bytecode for the checkAndSetF method:

```
  ILOAD 1
  IFLT label
  ALOAD 0
  ILOAD 1
  PUTFIELD pkg/Bean f I
  GOTO end
label:
F_SAME
  NEW java/lang/IllegalArgumentException
  DUP
  INVOKESPECIAL java/lang/IllegalArgumentException <init> ()V
  ATHROW
end:
F_SAME
  RETURN
```

3.2 Interfaces and components

3.2.1 Presentation

The ASM API for generating and transforming compiled methods is based on the MethodVisitor interface (see Figure 3.4), which is returned by the ClassVisitor's visitMethod method. In addition to some methods related to annotations and debug information, which are explained in the next chapter, this interface defines one method per bytecode instruction category, based on the number and type of arguments of these instructions (these categories do *not* correspond to the ones presented in section 3.1.2). These methods must be called in the following order (with some additional constraints specified in the Javadoc of the MethodVisitor interface):

```
visitAnnotationDefault?
( visitAnnotation | visitParameterAnnotation | visitAttribute )*
( visitCode
  ( visitTryCatchBlock | visitLabel | visitFrame | visitXxxInsn |
    visitLocalVariable | visitLineNumber )*
  visitMaxs )?
visitEnd
```

This means that annotations and attributes, if any, must be visited first, followed by the method's bytecode, for non abstract methods. For these methods the code must be visited in *sequential* order, between exactly one call to `visitCode` and exactly one call to `visitMaxs`.

```
public interface MethodVisitor {
  AnnotationVisitor visitAnnotationDefault();
  AnnotationVisitor visitAnnotation(String desc, boolean visible);
  AnnotationVisitor visitParameterAnnotation(int parameter,
      String desc, boolean visible);
  void visitAttribute(Attribute attr);
  void visitCode();
  void visitFrame(int type, int nLocal, Object[] local, int nStack,
      Object[] stack);
  void visitInsn(int opcode);
  void visitIntInsn(int opcode, int operand);
  void visitVarInsn(int opcode, int var);
  void visitTypeInsn(int opcode, String desc);
  void visitFieldInsn(int opc, String owner, String name, String desc);
  void visitMethodInsn(int opc, String owner, String name, String desc);
  void visitJumpInsn(int opcode, Label label);
  void visitLabel(Label label);
  void visitLdcInsn(Object cst);
  void visitIincInsn(int var, int increment);
  void visitTableSwitchInsn(int min, int max, Label dflt,
      Label labels[]);
  void visitLookupSwitchInsn(Label dflt, int keys[], Label labels[]);
  void visitMultiANewArrayInsn(String desc, int dims);
  void visitTryCatchBlock(Label start, Label end, Label handler,
      String type);
  void visitLocalVariable(String name, String desc, String signature,
      Label start, Label end, int index);
  void visitLineNumber(int line, Label start);
  void visitMaxs(int maxStack, int maxLocals);
  void visitEnd();
}
```

Figure 3.4: The `MethodVisitor` interface

The `visitCode` and `visitMaxs` methods can therefore be used to detect the start and end of the method's bytecode in a sequence of events. Like for classes, the `visitEnd` method must be called last, and is used to detect the end of a method in a sequence of events.

The `ClassVisitor` and `MethodVisitor` interfaces can be combined in order to generate complete classes:

```
ClassVisitor cv = ...;
cv.visit(...);
MethodVisitor mv1 = cv.visitMethod(..., "m1", ...);
mv1.visitCode();
mv1.visitInsn(...);
...
mv1.visitMaxs(...);
mv1.visitEnd();
MethodVisitor mv2 = cv.visitMethod(..., "m2", ...);
mv2.visitCode();
mv2.visitInsn(...);
...
mv2.visitMaxs(...);
mv2.visitEnd();
cv.visitEnd();
```

Note that it is not necessary to finish one method in order to start visiting another one. In fact `MethodVisitor` instances are completely independent and can be used in any order (as long as `cv.visitEnd()` has not been called):

```
ClassVisitor cv = ...;
cv.visit(...);
MethodVisitor mv1 = cv.visitMethod(..., "m1", ...);
mv1.visitCode();
mv1.visitInsn(...);
...
MethodVisitor mv2 = cv.visitMethod(..., "m2", ...);
mv2.visitCode();
mv2.visitInsn(...);
...
mv1.visitMaxs(...);
mv1.visitEnd();
...
mv2.visitMaxs(...);
mv2.visitEnd();
cv.visitEnd();
```

ASM provides three core components based on the `MethodVisitor` interface to generate and transform methods:

- The `ClassReader` class parses the content of compiled methods and calls the corresponding methods on the `MethodVisitor` objects returned by the `ClassVisitor` passed as argument to its `accept` method.

- The ClassWriter's visitMethod method returns an implementation of the MethodVisitor interface that builds compiled methods directly in binary form.
- The MethodAdapter class is a MethodVisitor implementation that delegates all the method calls it receives to another MethodVisitor instance.

ClassWriter options

As we have seen in section 3.1.5, computing the stack map frames for a method is not very easy: you have to compute all the frames, find the frames that correspond to jump targets or that follow unconditional jumps, and finally compress these remaining frames. Likewise, computing the size of the local variables and operand stack parts for a method is easier but still not very easy.

Hopefully ASM can compute this for you. When you create a ClassWriter you can specify what must be automatically computed:

- with new ClassWriter(0) nothing is automatically computed. You have to compute yourself the frames and the local variables and operand stack sizes.
- with new ClassWriter(ClassWriter.COMPUTE_MAXS) the sizes of the local variables and operand stack parts are computed for you. You must still call visitMaxs, but you can use any arguments: they will be ignored and recomputed. With this option you still have to compute the frames yourself.
- with new ClassWriter(ClassWriter.COMPUTE_FRAMES) everything is computed automatically. You don't have to call visitFrame, but you must still call visitMaxs (arguments will be ignored and recomputed).

Using these options is convenient but this has a cost: the COMPUTE_MAXS option makes a ClassWriter 10% slower, and using the COMPUTE_FRAMES option makes it *two times* slower. This must be compared to the time it would take to compute this yourself: in specific situations there are often easier and faster algorithms for computing this, compared to the algorithm used in ASM, which must handle all cases.

Note that if you choose to compute the frames yourself, you can let the ClassWriter class do the compression step for you. For this you just have to visit your uncompressed frames with visitFrame(F_NEW, nLocals, locals, nStack, stack), where nLocals and nStack are the number of locals and the operand stack size, and locals and stack are arrays containing the corresponding types (see the Javadoc for more details).

Note also that, in order to compute frames automatically, it is sometimes necessary to compute the common super class of two given classes. By default the ClassWriter class computes this, in the getCommonSuperClass method, by loading the two classes into the JVM and by using the reflection API. This can be a problem if you are generating several classes that reference each other, because the referenced classes may not yet exist. In this case you can override the getCommonSuperClass method to solve this problem.

3.2.2 Generating methods

The bytecode of the getF method defined in section 3.1.3 can be generated with the following method calls, if mv is a MethodVisitor:

```
mv.visitCode();
mv.visitVarInsn(ALOAD, 0);
mv.visitFieldInsn(GETFIELD, "pkg/Bean", "f", "I");
mv.visitInsn(IRETURN);
mv.visitMaxs(1, 1);
mv.visitEnd();
```

The first call starts the bytecode generation. It is followed by three calls that generate the three instructions of this method (as you can see the mapping between the bytecode and the ASM API is quite simple). The call to visitMaxs must be done after all the instructions have been visited. It is used to define the sizes of the local variables and operand stack parts for the execution frame of this method. As we saw in section 3.1.3, these sizes are 1 slot for each part. Finally the last call is used to end the generation of the method.

The bytecode of the setF method and of the constructor can be generated in a similar way. A more interesting example is the checkAndSetF method:

```
mv.visitCode();
mv.visitVarInsn(ILOAD, 1);
Label label = new Label();
mv.visitJumpInsn(IFLT, label);
mv.visitVarInsn(ALOAD, 0);
mv.visitVarInsn(ILOAD, 1);
mv.visitFieldInsn(PUTFIELD, "pkg/Bean", "f", "I");
Label end = new Label();
```

```
mv.visitJumpInsn(GOTO, end);
mv.visitLabel(label);
mv.visitFrame(F_SAME, 0, null, 0, null);
mv.visitTypeInsn(NEW, "java/lang/IllegalArgumentException");
mv.visitInsn(DUP);
mv.visitMethodInsn(INVOKESPECIAL,
    "java/lang/IllegalArgumentException", "<init>", "()V");
mv.visitInsn(ATHROW);
mv.visitLabel(end);
mv.visitFrame(F_SAME, 0, null, 0, null);
mv.visitInsn(RETURN);
mv.visitMaxs(2, 2);
mv.visitEnd();
```

Between the `visitCode` and `visitEnd` calls you can see method calls that map exactly to the bytecode shown at the end of section 3.1.5: one call per instruction, label or frame (the only exceptions are the declaration and construction of the `label` and `end` `Label` objects).

Note: a `Label` object designates the *instruction* that follows the `visitLabel` call for this label. For example `end` designates the `RETURN` instruction, and not the frame that is visited just after, since this is not an instruction. It is perfectly legal to have several labels designating the same instruction, but a label must designate exactly one instruction. In other words it is possible to have successive calls to `visitLabel` with different labels, but a label used in an instruction must be visited exactly once with `visitLabel`. A last constraint is that labels can not be shared: each method must have its own labels.

3.2.3 Transforming methods

You should now have guessed that methods can be transformed like classes, i.e. by using a method adapter that forwards the method calls it receives with some modifications: changing arguments can be used to change individual instructions, not forwarding a received call removes an instruction, and inserting calls between the received ones adds new instructions. The `MethodAdapter` class provides a basic implementation of such a method visitor, which does nothing else than just forwarding all the method calls it receives.

In order to understand how method adapters can be used, let's consider a very simple adapter that removes the `NOP` instructions inside methods (they can be removed without problems since they do nothing):

```
public class RemoveNopAdapter extends MethodAdapter {
  public RemoveNopAdapter(MethodVisitor mv) {
    super(mv);
  }
  @Override
  public void visitInsn(int opcode) {
    if (opcode != NOP) {
      mv.visitInsn(opcode);
    }
  }
}
```

This adapter can be used inside a class adapter as follows:

```
public class RemoveNopClassAdapter extends ClassAdapter {
  public RemoveNopClassAdapter(ClassVisitor cv) {
    super(cv);
  }
  @Override
  public MethodVisitor visitMethod(int access, String name,
      String desc, String signature, String[] exceptions) {
    MethodVisitor mv;
    mv = cv.visitMethod(access, name, desc, signature, exceptions);
    if (mv != null) {
      mv = new RemoveNopAdapter(mv);
    }
    return mv;
  }
}
```

In other words the class adapter just builds a method adapter encapsulating the method visitor returned by the next class visitor in the chain, and returns this adapter. The effect is the construction of a method adapter chain that is similar to the class adapter chain (see Figure 3.5).

Note however that this is not mandatory: it is perfectly possible to build a method adapter chain that is not similar to the class adapter chain. Each method can even have a different method adapter chain. For instance the class adapter could choose to remove `NOP`s only in methods and not in constructors. This can be done as follows:

```
...
mv = cv.visitMethod(access, name, desc, signature, exceptions);
if (mv != null && !name.equals("<init>")) {
  mv = new RemoveNopAdapter(mv);
}
...
```

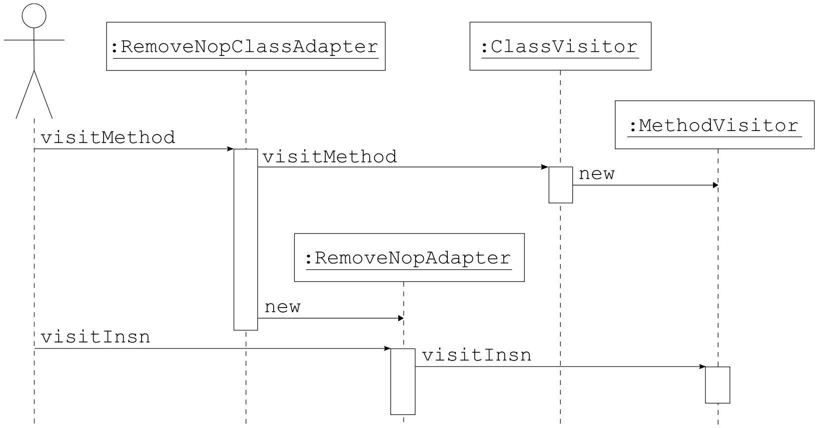


Figure 3.5: Sequence diagram for the RemoveNopAdapter

In this case the adapter chain is shorter for constructors. On the contrary, the adapter chain for constructors could have been longer, with several method adapters chained together created inside `visitMethod`. The method adapter chain can even have a different topology than the class adapter chain. For instance the class adapter chain could be linear, while the method adapter chain has branches:

```
public MethodVisitor visitMethod(int access, String name,
    String desc, String signature, String[] exceptions) {
  MethodVisitor mv1, mv2;
  mv1 = cv.visitMethod(access, name, desc, signature, exceptions);
  mv2 = cv.visitMethod(access, "_" + name, desc, signature, exceptions);
  return new MultiMethodAdapter(mv1, mv2);
}
```

Now that we have seen how method adapters can be used and combined inside a class adapter, let's see how to implement more interesting adapters than `RemoveNopAdapter`.

3.2.4 Stateless transformations

Let's suppose we want to measure the time spent in each class of a program. We need to add a static timer field in each class, and we need to add the execution time of each method of this class to this timer field. In other words we want to transform a class such as `C`:

```
public class C {
  public void m() throws Exception {
    Thread.sleep(100);
  }
}
```

into this:

```
public class C {
  public static long timer;
  public void m() throws Exception {
    timer -= System.currentTimeMillis();
    Thread.sleep(100);
    timer += System.currentTimeMillis();
  }
}
```

In order to have an idea of how this can be implemented in ASM, we can compile these two classes and compare the output of `TraceClassVisitor` or `ASMifierClassVisitor` on these two versions. With `TraceClassVisitor` we get the following differences (in bold):

```
GETSTATIC C.timer : J
INVOKESTATIC java/lang/System.currentTimeMillis()J
LSUB
PUTSTATIC C.timer : J
LDC 100
INVOKESTATIC java/lang/Thread.sleep(J)V
GETSTATIC C.timer : J
INVOKESTATIC java/lang/System.currentTimeMillis()J
LADD
PUTSTATIC C.timer : J
RETURN
MAXSTACK = 4
MAXLOCALS = 1
```

We see that we must add four instructions at the beginning of the method, and four other instructions before the return instruction. We also need to update the maximum operand stack size. The beginning of the method's code is visited with the `visitCode` method. We can therefore add the first four instructions by overriding this method in our method adapter:

```
public void visitCode() {
  mv.visitCode();
  mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
  mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
      "currentTimeMillis", "()J");
```

```
    mv.visitInsn(LSUB);
    mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
  }
```

where `owner` must be set to the name of the class that is being transformed. We must now add four other instructions before any `RETURN`, but also before any x`RETURN` or before `ATHROW`, which are all the instructions that terminate the method's execution. These instructions do not have any argument, and are therefore visited in the `visitInsn` method. We can then override this method in order to add our instructions:

```
  public void visitInsn(int opcode) {
    if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
      mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
      mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
          "currentTimeMillis", "()J");
      mv.visitInsn(LADD);
      mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
    }
    mv.visitInsn(opcode);
  }
```

Finally we must update the maximum operand stack size. The instructions that we add push two long values, and therefore require four slots on the operand stack. At the beginning of the method the operand stack is initially empty, so we know that the four instructions added at the beginning require a stack of size 4. We also know that our inserted code leaves the stack state unchanged (because it pops as many values as it pushes). As a consequence, if the original code requires a stack of size s, the maximum stack size needed by the transformed method is $max(4, s)$. Unfortunately we also add four instructions before the return instructions, and here we do not know the size of the operand stack just before these instructions. We just know that it is less than or equal to s. As a consequence we can just say that the code added before the return instructions may require an operand stack of size up to $s+4$. This worst case scenario rarely happens in practice: with common compilers the operand stack before a `RETURN` contains only the return value, i.e. it has a size of 0, 1 or 2 at most. But if we want to handle all possible cases, we need to use the worst case scenario[2]. We must then override the `visitMaxs` method as follows:

[2] hopefully it is not necessary to give the optimal operand stack size. Giving any value greater than or equal to this optimal value is possible, although it may vaste memory on the thread's execution stack.

```
public void visitMaxs(int maxStack, int maxLocals) {
  mv.visitMaxs(maxStack + 4, maxLocals);
}
```

Of course it is possible to not bother about the maximum stack size and rely on the COMPUTE_MAXS option that, in addition, will compute the optimal value and not a worst case value. But for such simple transformations it does not cost much effort to update maxStack manually.

Now an interesting question is: what about stack map frames? The original code did not contain any frame, nor the transformed one, but is this due to the specific code we used as example? are there some situations where frames must be updated? The answer is no because 1) the inserted code leaves the operand stack unchanged, 2) the inserted code does not contain jump instructions and 3) the jump instructions – or, more formally, the control flow graph – of the original code is not modified. This means that the original frames do not change, and since no new frames must be stored for the inserted code, the compressed original frames do not change either.

We can now put all the elements together in associated ClassAdapter and MethodAdapter subclasses:

```
public class AddTimerAdapter extends ClassAdapter {
  private String owner;
  private boolean isInterface;
  public AddTimerAdapter(ClassVisitor cv) {
    super(cv);
  }
  @Override public void visit(int version, int access, String name,
      String signature, String superName, String[] interfaces) {
    cv.visit(version, access, name, signature, superName, interfaces);
    owner = name;
    isInterface = (access & ACC_INTERFACE) != 0;
  }
  @Override public MethodVisitor visitMethod(int access, String name,
      String desc, String signature, String[] exceptions) {
    MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
        exceptions);
    if (!isInterface && mv != null && !name.equals("<init>")) {
      mv = new AddTimerMethodAdapter(mv);
    }
    return mv;
  }
  @Override public void visitEnd() {
    if (!isInterface) {
      FieldVisitor fv = cv.visitField(ACC_PUBLIC + ACC_STATIC, "timer",
          "J", null, null);
      if (fv != null) {
        fv.visitEnd();
      }
    }
    cv.visitEnd();
  }
  class AddTimerMethodAdapter extends MethodAdapter {
    public AddTimerMethodAdapter(MethodVisitor mv) {
      super(mv);
    }
    @Override public void visitCode() {
      mv.visitCode();
      mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
      mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
          "currentTimeMillis", "()J");
      mv.visitInsn(LSUB);
      mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
    }
    @Override public void visitInsn(int opcode) {
      if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
        mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
        mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
            "currentTimeMillis", "()J");
        mv.visitInsn(LADD);
        mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
      }
      mv.visitInsn(opcode);
    }
    @Override public void visitMaxs(int maxStack, int maxLocals) {
      mv.visitMaxs(maxStack + 4, maxLocals);
    }
  }
}
```

The class adapter is used to instantiate the method adapter (except for constructors), but also to add the timer field and to store the name of the class that is being transformed in a field that can be accessed from the method adapter.

3.2.5 Statefull transformations

The transformation seen in the previous section is local and does not depend on the instructions that have been visited before the current one: the code added at the beginning is always the same and is always added, and likewise for the code inserted before each RETURN instruction. Such transformations are called *stateless* transformations. They are simple to implement but only the simplest transformations verify this property.

More complex transformations require memorizing some state about the instructions that have been visited before the current one. Consider for example a transformation that removes all occurrences of the ICONST_0 IADD sequence, whose empty effect is to add 0. It is clear that when an IADD instruction is visited, it must be removed only if the last visited instruction was an ICONST_0. This requires storing state inside the method adapter. For this reason such transformations are called *statefull* transformations.

Let's look in more details at this example. When an ICONST_0 is visited, it must be removed only if the *next* instruction is an IADD. The problem is that the next instruction is not yet known. The solution is to postpone this decision to the next instruction: if it is an IADD then remove both instructions, otherwise emit the ICONST_0 and the current instruction.

In order to implement transformations that remove or replace some instruction sequence, it is convenient to introduce a **MethodAdapter** subclass whose **visit*Xxx*Insn** methods call a common **visitInsn()** method:

```
public abstract class PatternMethodAdapter extends MethodAdapter {
  protected final static int SEEN_NOTHING = 0;
  protected int state;
  public PatternMethodAdapter(MethodVisitor mv) {
    super(mv);
  }
  @Overrid public void visitInsn(int opcode) {
    visitInsn();
    mv.visitInsn(opcode);
  }
  @Override public void visitIntInsn(int opcode, int operand) {
    visitInsn();
    mv.visitIntInsn(opcode, operand);
  }
  ...
  protected abstract void visitInsn();
}
```

Then the above transformation can be implemented like this:

```
public class RemoveAddZeroAdapter extends PatternMethodAdapter {
  private static int SEEN_ICONST_0 = 1;
  public RemoveAddZeroAdapter(MethodVisitor mv) {
    super(mv);
  }
  @Override public void visitInsn(int opcode) {
    if (state == SEEN_ICONST_0) {
      if (opcode == IADD) {
        state = SEEN_NOTHING;
        return;
      }
    }
    visitInsn();
    if (opcode == ICONST_0) {
      state = SEEN_ICONST_0;
      return;
    }
    mv.visitInsn(opcode);
  }
  @Override protected void visitInsn() {
    if (state == SEEN_ICONST_0) {
      mv.visitInsn(ICONST_0);
    }
    state = SEEN_NOTHING;
  }
}
```

The `visitInsn(int)` method first tests if the sequence has been detected. In this case it reinitializes `state` and returns immediately, which has the effect of removing the sequence. In the other cases it calls the common `visitInsn` method, which emits an `ICONST_0` if this was the *last* visited instruction. Then, if the *current* instruction is an `ICONST_0`, it memorizes this fact and returns, in order to postpone the decision about this instruction. In all other cases the current instruction is forwarded to the next visitor.

Labels and frames

As we have seen in the previous sections, labels and frames are visited just before their associated instruction. In other words they are visited at the same time as instructions, although they are not instructions themselves. This has an impact on transformations that detect *instruction* sequences, but this impact is in fact an advantage. Indeed, what happens if one of the instructions we remove is the target of a jump instruction? If some instruction may jump to the `ICONST_0`, this means that there is a label designating this instruction. After the removal of the two instructions this label will designate the instruction that follows the removed `IADD`, which is what we want. But if some instruction may jump to the `IADD`, we can not remove the instruction sequence (we can not be sure that before this jump a 0 was pushed on the stack). Hopefully, in this case, there must be a label between the ICONST_0 and the IADD, which can easily be detected.

The reasoning is the same for stack map frames: if a stack map frame is visited between the two instructions, we can not remove them. Both cases can be handled by considering labels and frames as instructions in the pattern matching algorithm. This can be done in PatternMethodAdapter (note that visitMaxs also calls the common visitInsn method; this is used to handle the case where the end of the method is a prefix of the sequence that must be detected):

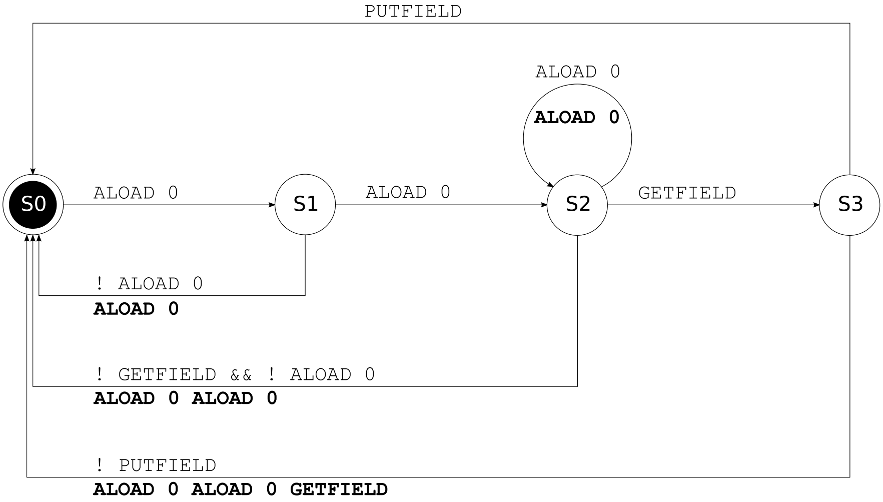
```
public abstract class PatternMethodAdapter extends MethodAdapter {
  ...
  @Override public void visitFrame(int type, int nLocal, Object[] local,
      int nStack, Object[] stack) {
    visitInsn();
    mv.visitFrame(type, nLocal, local, nStack, stack);
  }
  @Override public void visitLabel(Label label) {
    visitInsn();
    mv.visitLabel(label);
  }
  @Override public void visitMaxs(int maxStack, int maxLocals) {
    visitInsn();
    mv.visitMaxs(maxStack, maxLocals);
  }
}
```

As we will see in the next chapter, a compiled method may contain information about source file line numbers, used for instance in exception stack traces. This information is visited with the visitLineNumber method, which is also called at the same time as instructions. Here however the presence of line numbers in the middle of an instruction sequence does not have any impact on the possibility to transform or remove it. The solution is therefore to ignore them completely in the pattern matching algorithm.

A more complex example

The previous example can be easily generalized to more complex instruction sequences. Consider for example a transformation that removes self field assignments, generally due to typos, such as f = f; or, in bytecode, ALOAD 0

ALOAD 0 GETFIELD f PUTFIELD f. Before implementing this transformation, it is preferable to design the state machine to recognize this sequence:



Each transition is labeled with a condition (the value of the current instruction) and an action (an instruction sequence that must be emitted, in bold). For instance the transition from S1 to S0 happens if the current instruction is not an ALOAD 0. In this case the ALOAD 0 that was visited to arrive at this state is emitted. Note the transition from S2 to itself: this happens when three or more consecutive ALOAD 0 are found. In this case we stay in the state where two ALOAD 0 have been visited, and we emit the third one. Once the state machine has been found, writing the corresponding method adapter is straightforward (the 8 switch cases correspond to the 8 transitions in the diagram):

```
class RemoveGetFieldPutFieldAdapter extends PatternMethodAdapter {
  private final static int SEEN_ALOAD_0 = 1;
  private final static int SEEN_ALOAD_0ALOAD_0 = 2;
  private final static int SEEN_ALOAD_0ALOAD_0GETFIELD = 3;
  private String fieldOwner;
  private String fieldName;
  private String fieldDesc;
  public RemoveGetFieldPutFieldAdapter(MethodVisitor mv) {
    super(mv);
  }
  @Override
  public void visitVarInsn(int opcode, int var) {
```

```
  switch (state) {
  case SEEN_NOTHING: // S0 -> S1
    if (opcode == ALOAD && var == 0) {
      state = SEEN_ALOAD_0;
      return;
    }
    break;
  case SEEN_ALOAD_0: // S1 -> S2
    if (opcode == ALOAD && var == 0) {
      state = SEEN_ALOAD_0ALOAD_0;
      return;
    }
    break;
  case SEEN_ALOAD_0ALOAD_0: // S2 -> S2
    if (opcode == ALOAD && var == 0) {
      mv.visitVarInsn(ALOAD, 0);
      return;
    }
    break;
  }
  visitInsn();
  mv.visitVarInsn(opcode, var);
}
@Override
public void visitFieldInsn(int opcode, String owner, String name,
    String desc) {
  switch (state) {
  case SEEN_ALOAD_0ALOAD_0: // S2 -> S3
    if (opcode == GETFIELD) {
      state = SEEN_ALOAD_0ALOAD_0GETFIELD;
      fieldOwner = owner;
      fieldName = name;
      fieldDesc = desc;
      return;
    }
    break;
  case SEEN_ALOAD_0ALOAD_0GETFIELD: // S3 -> S0
    if (opcode == PUTFIELD && name.equals(fieldName)) {
      state = SEEN_NOTHING;
      return;
    }
    break;
  }
  visitInsn();
  mv.visitFieldInsn(opcode, owner, name, desc);
}
@Override protected void visitInsn() {
        switch (state) {
        case SEEN_ALOAD_0: // S1 -> S0
          mv.visitVarInsn(ALOAD, 0);
          break;
        case SEEN_ALOAD_0ALOAD_0: // S2 -> S0
          mv.visitVarInsn(ALOAD, 0);
          mv.visitVarInsn(ALOAD, 0);
          break;
        case SEEN_ALOAD_0ALOAD_0GETFIELD: // S3 -> S0
          mv.visitVarInsn(ALOAD, 0);
          mv.visitVarInsn(ALOAD, 0);
          mv.visitFieldInsn(GETFIELD, fieldOwner, fieldName, fieldDesc);
          break;
        }
        state = SEEN_NOTHING;
      }
    }
```

Note that, for the same reasons as in the AddTimerAdapter case in section 3.2.4, the statefull transformations presented in this section do not need to transform stack map frames: the original frames stay valid after the transformation. They don't even need to transform the local variables and operand stack size. Finally it must be noted that statefull transformations are not limited to transformations that detect and transform instruction sequences. Many other types of transformation are also statefull. This is the case, for instance, of the method adapters presented in the next section.

3.3 Tools

The org.objectweb.asm.commons package contains some predefined method adapters that can be useful to define your own adapters. This section presents three of them and shows how they can be used with the AddTimerAdapter example of section 3.2.4. It also shows how the tools seen in the previous chapter can be used to ease method generation or transformation.

3.3.1 Basic tools

The tools presented in section 2.3 can also be used for methods.

Type

Many bytecode instructions, such as *x*`LOAD`, *x*`ADD` or *x*`RETURN`, depend on the type to which they are applied. The `Type` class provides a `getOpcode` method that can be used to get, for these instructions, the opcode corresponding to a given type. This method takes as parameter an opcode for the `int` type, and returns the opcode for the type on which it is called. For instance `t.getOpcode(IMUL)` returns `FMUL` if `t` is equal to `Type.FLOAT_TYPE`.

TraceClassVisitor

This class, which has already been presented in the previous chapter, prints a textual representation of the classes it visits, *including a textual representation of their methods*, in a form very similar to the one used in this chapter. It can therefore be used to trace the content of generated or transformed methods at any point in a transformation chain. For example:

```
java -classpath asm.jar:asm-util.jar \
     org.objectweb.asm.util.TraceClassVisitor \
     java.lang.Void
```

prints:

```
// class version 49.0 (49)
// access flags 49
public final class java/lang/Void {
  // access flags 25
  // signature Ljava/lang/Class<Ljava/lang/Void;>;
  // declaration: java.lang.Class<java.lang.Void>
  public final static Ljava/lang/Class; TYPE
  // access flags 2
  private <init>()V
    ALOAD 0
    INVOKESPECIAL java/lang/Object.<init> ()V
    RETURN
    MAXSTACK = 1
    MAXLOCALS = 1
  // access flags 8
  static <clinit>()V
    LDC "void"
    INVOKESTATIC java/lang/Class.getPrimitiveClass (...)...
    PUTSTATIC java/lang/Void.TYPE : Ljava/lang/Class;
    RETURN
    MAXSTACK = 1
    MAXLOCALS = 0
}
```

This shows how to generate a static block `static { ... }`, namely with a `<clinit>` method (for CLass INITializer). Note that if you want to trace the content of a single method at some point in a chain, instead of tracing all the content of its class, you can use `TraceMethodVisitor` instead of `TraceClassVisitor`:

```
public MethodVisitor visitMethod(int access, String name,
    String desc, String signature, String[] exceptions) {
  MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
        exceptions);
  if (debug && mv != null && ...) { // if this method must be traced
    mv = new TraceMethodVisitor(mv) {
      @Override public void visitEnd() {
        print(aPrintWriter); // print it after it has been visited
      }
    };
  }
  return new MyMethodAdapter(mv);
}
```

This code prints the method after transformation by `MyMethodAdapter`.

CheckClassAdapter

This class, which has already been presented in the previous chapter, checks that the `ClassVisitor` methods are called in the appropriate order, and with valid arguments, and it does the same for the `MethodVisitor` methods. It can therefore be used to check that the `MethodVisitor` interface is correctly used at any point in a transformation chain. Like with `TraceMethodVisitor`, you can use the `CheckMethodAdapter` class to check a single method instead of checking all its class:

```
public MethodVisitor visitMethod(int access, String name,
    String desc, String signature, String[] exceptions) {
  MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
        exceptions);
  if (debug && mv != null && ...) { // if this method must be checked
    mv = new CheckMethodAdapter(mv);
  }
  return new MyMethodAdapter(mv);
}
```

This code checks that MyMethodAdapter uses the MethodVisitor interface correctly. Note however that this adapter can *not* check that the bytecode is correct: for instance it does not detect that ISTORE 1 ALOAD 1 is invalid.

ASMifierClassVisitor

This class, which has already been presented in the previous chapter, also works with the content of methods. It can be used to know how to generate some compiled code with ASM: just write the corresponding source code in Java, compile it with javac, and use the ASMifierClassVisitor to visit this class. You will get the ASM code to generate the bytecode corresponding to your source code.

3.3.2 AnalyzerAdapter

This method adapter computes the stack map frames before each instruction, based on the frames visited in visitFrame. Indeed, as explained in section 3.1.5, visitFrame is only called before some specific instructions in a method, in order to save space, and because "the other frames can be easily and quickly inferred from these ones". This is what this adapter does. Of course it only works on classes that contain precomputed stack map frames, i.e. compiled with Java 6 or higher (or previously upgraded to Java 6 with an ASM adapter using the COMPUTE_FRAMES option).

In the case of our AddTimerAdapter example, this adapter could be used to get the size of the operand stack just before the RETURN instructions, thereby allowing to compute an optimal transformed value for maxStack in visitMaxs (in fact this method is not recommended in practice, because it is much less efficient than using COMPUTE_MAXS):

```
class AddTimerMethodAdapter2 extends AnalyzerAdapter {
  private int maxStack;
  public AddTimerMethodAdapter2(String owner, int access,
      String name, String desc, MethodVisitor mv) {
    super(owner, access, name, desc, mv);
  }
  @Override public void visitCode() {
    super.visitCode();
    mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
        "currentTimeMillis", "()J");
    mv.visitInsn(LSUB);
    mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
    maxStack = 4;
  }
  @Override public void visitInsn(int opcode) {
    if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
      mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
      mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
          "currentTimeMillis", "()J");
      mv.visitInsn(LADD);
      mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
      maxStack = Math.max(maxStack, stack.size() + 4);
    }
    super.visitInsn(opcode);
  }
  @Override public void visitMaxs(int maxStack, int maxLocals) {
    super.visitMaxs(Math.max(this.maxStack, maxStack), maxLocals);
  }
}
```

The `stack` field is defined in the `AnalyzerAdapter` class, and contains the types in operand stack. More precisely, in a `visit`*Xxx*`Insn`, and before the overridden method is called, this list contains the state of the operand stack just before this instruction. Note that the overridden methods *must* be called so that the `stack` field is correctly updated (hence the use of `super` instead of `mv` in the original code).

Alternatively the new instructions can be inserted by calling the methods of the super class: the effect is that the frames for these instructions will be computed by `AnalyzerAdapter`. Since, in addition, this adapter updates the arguments of `visitMaxs` based on the frames it computes, we do not need to update them ourselves:

```
class AddTimerMethodAdapter3 extends AnalyzerAdapter {
  public AddTimerMethodAdapter3(String owner, int access,
      String name, String desc, MethodVisitor mv) {
    super(owner, access, name, desc, mv);
  }
  @Override public void visitCode() {
    super.visitCode();
    super.visitFieldInsn(GETSTATIC, owner, "timer", "J");
    super.visitMethodInsn(INVOKESTATIC, "java/lang/System",
        "currentTimeMillis", "()J");
    super.visitInsn(LSUB);
    super.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
  }
```

```
    @Override public void visitInsn(int opcode) {
      if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
        super.visitFieldInsn(GETSTATIC, owner, "timer", "J");
        super.visitMethodInsn(INVOKESTATIC, "java/lang/System",
            "currentTimeMillis", "()J");
        super.visitInsn(LADD);
        super.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
      }
      super.visitInsn(opcode);
    }
  }
```

3.3.3 LocalVariablesSorter

This method adapter renumbers the local variables used in a method in the order they appear in this method. For instance in a method with two parameters, the first local variable read or written whose index is greater than or equal to 3 – the first three local variables correspond to this and to the two method parameters, and can therefore not be changed – is assigned index 3, the second one is assigned index 4, and so on. This adapter is useful to insert new local variables in a method. Without this adapter it would be necessary to add new local variables after all the existing ones, but unfortunately their number is not known until the end of the method, in `visitMaxs`.

In order to show how this adapter can be used, let's suppose that we want to use a local variable to implement `AddTimerAdapter`:

```
public class C {
  public static long timer;
  public void m() throws Exception {
    long t = System.currentTimeMillis();
    Thread.sleep(100);
    timer += System.currentTimeMillis() - t;
  }
}
```

This can be done easily by extending `LocalVariablesSorter`, and by using the `newLocal` method defined in this class:

```
class AddTimerMethodAdapter4 extends LocalVariablesSorter {
  private int time;
  public AddTimerMethodAdapter4(int access, String desc,
      MethodVisitor mv) {
    super(access, desc, mv);
  }
  @Override public void visitCode() {
    super.visitCode();
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
        "currentTimeMillis", "()J");
    time = newLocal(Type.LONG_TYPE);
    mv.visitVarInsn(LSTORE, time);
  }
  @Override public void visitInsn(int opcode) {
    if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
      mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
          "currentTimeMillis", "()J");
      mv.visitVarInsn(LLOAD, time);
      mv.visitInsn(LSUB);
      mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
      mv.visitInsn(LADD);
      mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
    }
    super.visitInsn(opcode);
  }
  @Override public void visitMaxs(int maxStack, int maxLocals) {
    super.visitMaxs(maxStack + 4, maxLocals);
  }
}
```

Note that the original frames associated to the method become invalid when the local variables are renumbered, and a fortiori when new local variables are inserted. Hopefully it is possible to avoid recomputing these frames from scratch: indeed no frames must be added or removed, and it “suffices” to reorder the content of local variables in the original frames to get the frames of the transformed method. `LocalVariablesSorter` takes care of that automatically. If you also need to do incremental stack map frame updates for one of your method adapters, you can inspire yourself from the source of this class.

As you can see above using a local variable does not solve the problem we had in the original version of this class, concerning the worst case value for `maxStack`. If you want to use an `AnalyzerAdapter` to solve that, in addition to a `LocalVariablesSorter`, you must use these adapters through delegation instead of via inheritance (since multiple inheritance is not possible):

```
  class AddTimerMethodAdapter5 extends MethodAdapter {
    public LocalVariablesSorter lvs;
    public AnalyzerAdapter aa;
    private int time;
    private int maxStack;
    public AddTimerMethodAdapter5(MethodVisitor mv) {
```

```
    super(mv);
  }
  @Override public void visitCode() {
    mv.visitCode();
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
        "currentTimeMillis", "()J");
    time = lvs.newLocal(Type.LONG_TYPE);
    mv.visitVarInsn(LSTORE, time);
    maxStack = 4;
  }
  @Override public void visitInsn(int opcode) {
    if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
      mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
          "currentTimeMillis", "()J");
      mv.visitVarInsn(LLOAD, time);
      mv.visitInsn(LSUB);
      mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
      mv.visitInsn(LADD);
      mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
      maxStack = Math.max(aa.stack.size() + 4, maxStack);
    }
    mv.visitInsn(opcode);
  }
  @Override public void visitMaxs(int maxStack, int maxLocals) {
    mv.visitMaxs(Math.max(this.maxStack, maxStack), maxLocals);
  }
}
```

In order to use this adapter you must chain a `LocalVariablesSorter` to an `AnalyzerAdapter`, itself chained to your adapter: the first adapter will sort local variables and update frames accordingly, the analyzer adapter will compute intermediate frames taking into account the renumbering done in the previous adapter, and your adapter will have access to these renumbered intermediate frames. This chain can be constructed as follows in `visitMethod`:

```
mv = cv.visitMethod(access, name, desc, signature, exceptions);
if (!isInterface && mv != null && !name.equals("<init>")) {
   AddTimerMethodAdapter5 at = new AddTimerMethodAdapter5(mv);
   at.aa = new AnalyzerAdapter(owner, access, name, desc, at);
   at.lvs = new LocalVariablesSorter(access, desc, at.aa);
   return at.lvs;
}
```

3.3.4 AdviceAdapter

This method adapter is an abstract class that can be used to insert code at the beginning of a method and just before any RETURN or ATHROW instruction. Its main advantage is that it also works for constructors, where code must not be inserted just at the beginning of the constructor, but after the call to the super constructor. In fact most of the code of this adapter is dedicated to the detection of this super constructor call.

If you look carefully at the AddTimerAdapter class in section 3.2.4, you will see that the AddTimerMethodAdapter is not used for constructors, because of this problem. By inheriting from AdviceAdapter this method adapter can be improved to work on constructors too (note that AdviceAdapter inherits from LocalVariablesSorter, so we could also easily use a local variable):

```
class AddTimerMethodAdapter6 extends AdviceAdapter {
  public AddTimerMethodAdapter6(int access, String name, String desc,
      MethodVisitor mv) {
    super(mv, access, name, desc);
  }
  @Override protected void onMethodEnter() {
    mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
        "currentTimeMillis", "()J");
    mv.visitInsn(LSUB);
    mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
  }
  @Override protected void onMethodExit(int opcode) {
    mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
        "currentTimeMillis", "()J");
    mv.visitInsn(LADD);
    mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
  }
  @Override public void visitMaxs(int maxStack, int maxLocals) {
    super.visitMaxs(maxStack + 4, maxLocals);
  }
}
```

4 Metadata

This chapter explains how to generate and transform compiled Java classes metadata, such as annotations, with the core API. Each section starts with a presentation of one type of metadata, and then presents the corresponding ASM interfaces, components and tools to generate and transform these metadata, with some illustrative examples.

4.1 Generics

Generic classes such as `List<E>`, and classes using them, contain information about the generic types they declare or use. This information is not used at runtime by the bytecode instructions, but it can be accessed via the reflection API. It is also used by compilers, for separate compilation.

4.1.1 Structure

For backward compatibility reasons the information about generic types is not stored in type or method descriptors (which were defined long before the introduction of generics in Java 5), but in similar constructs called type, method and class *signatures*. These signatures are stored in addition to descriptors in class, field and method *declarations* when a generic type is involved (generic types do not affect the bytecode of methods: the compiler uses them to perform static type checks, but then compiles methods as if they were not used, by reintroducing type casts where necessary).

Unlike type and method descriptors, and due to the recursive nature of generic types (a generic type can be parameterized by a generic type – consider for example `List<List<E> >`) the grammar of type signatures is quite complex. It is given by the following rules (see the Java Virtual Machine Specification for a complete description of these rules):

TypeSignature: `Z` | `C` | `B` | `S` | `I` | `F` | `J` | `D` | *FieldTypeSignature*

FieldTypeSignature: *ClassTypeSignature* | `[` *TypeSignature* | *TypeVar*

ClassTypeSignature: `L` *Id* (`/` *Id*)* *TypeArgs*? (`.` *Id* *TypeArgs*?)* `;`

TypeArgs: `<` *TypeArg*+ `>`

TypeArg: `*` | (`+` | `-`)? *FieldTypeSignature*

TypeVar: `T` *Id* `;`

The first rule says that a *type signature* is either a primitive type descriptor or a field type signature. The second rule defines a field type signature as a class type signature, an array type signature, or a type variable. The third rule defines class type signatures: they are class type descriptors with possible type arguments, between angle brackets, after the main class name or after the inner class names (prefixed with dots). The remaining rules define type arguments and type variables. Note that a type argument can be a complete field type signature, with its own type arguments: type signatures can therefore be very complex (see Figure 4.1).

Java type and corresponding type signature

`List<E>`
`  Ljava/util/List<TE;>;`

`List<?>`
`  Ljava/util/List<*>;`

`List<? extends Number>`
`  Ljava/util/List<+Ljava/lang/Number;>;`

`List<? super Integer>`
`  Ljava/util/List<-Ljava/lang/Integer;>;`

`List<List<String>[]>`
`  Ljava/util/List<[Ljava/util/List<Ljava/lang/String;>;>;`

`HashMap<K, V>.HashIterator<K>`
`  Ljava/util/HashMap<TK;TV;>.HashIterator<TK;>;`

Figure 4.1: Sample type signatures

Method signatures extend method descriptors like type signatures extend type descriptors. A *method signature* describes the type signatures of the method parameters and the signature of its return type. Unlike method descriptors, it also contains the signatures of the exceptions thrown by the method, preceded by `^`, and can also contain optional formal type parameters between angle brackets:

MethodTypeSignature:
  TypeParams? `(` *TypeSignature** `)` (*TypeSignature* | `V`) *Exception**

Exception: `^`*ClassTypeSignature* | `^`*TypeVar*

TypeParams: `<` *TypeParam*+ `>`

TypeParam: *Id* `:` *FieldTypeSignature*? (`:` *FieldTypeSignature*)*

For example the method signature of the following generic static method, parameterized by the type variable `T`:

```
static <T> Class<?  extends T> m (int n)
```

is the following method signature:

```
<T:Ljava/lang/Object;>(I)Ljava/lang/Class<+TT;>;
```

Finally a *class signature*, which must not be confused with a class type signature, is defined as the type signature of its super class, followed by the type signatures of the implemented interfaces, and with optional formal type parameters:

ClassSignature: *TypeParams*? *ClassTypeSignature* *ClassTypeSignature**

For example the class signature of a class declared as `C<E> extends List<E>` is `<E:Ljava/lang/Object;>Ljava/util/List<TE;>;`.

4.1.2 Interfaces and components

Like for descriptors, and for the same efficiency reasons (see section 2.3.1), the ASM API exposes signatures as they as stored in compiled classes (the main occurences of signatures are in the `visit`, `visitField` and `visitMethod` methods of the `ClassVisitor` interface, as an optional class, type or method signature argument respectively). Hopefully it also provides some tools to generate and transform signatures, in the `org.objectweb.asm.signature` package, based on the `SignatureVisitor` interface (see Figure 4.2).

```
public interface SignatureVisitor {
  char EXTENDS = ’+’;
  char SUPER = ’-’;
  char INSTANCEOF = ’=’;
  void visitFormalTypeParameter(String name);
  SignatureVisitor visitClassBound();
  SignatureVisitor visitInterfaceBound();
  SignatureVisitor visitSuperclass();
  SignatureVisitor visitInterface();
  SignatureVisitor visitParameterType();
  SignatureVisitor visitReturnType();
  SignatureVisitor visitExceptionType();
  void visitBaseType(char descriptor);
  void visitTypeVariable(String name);
  SignatureVisitor visitArrayType();
  void visitClassType(String name);
  void visitInnerClassType(String name);
  void visitTypeArgument();
  SignatureVisitor visitTypeArgument(char wildcard);
  void visitEnd();
}
```

Figure 4.2: The `SignatureVisitor` interface

This interface is used to visit type signatures, method signatures and class signatures. The methods used to visit type signatures are in bold, and must be called in the following order, which reflects the previous grammar rules (note that two of them return a `SignatureVisitor`: this is due to the recursive definition of type signatures):

```
visitBaseType | visitArrayType | visitTypeVariable |
( visitClassType visitTypeArgument*
  ( visitInnerClassType visitTypeArgument* )* visitEnd ) )
```

The methods used to visit method signatures are the following:

```
( visitFormalTypeParameter visitClassBound? visitInterfaceBound* )*
visitParameterType* visitReturnType visitExceptionType*
```

Finally the methods used to visit class signatures are:

```
( visitFormalTypeParameter visitClassBound? visitInterfaceBound* )*
visitSuperClass visitInterface*
```

Most of these methods return a `SignatureVisitor`: it is intended to visit a *type* signature. Note that, unlike with the `MethodVisitor`s returned by a `ClassVisitor`, the `SignatureVisitor`s returned by a `SignatureVisitor` must not be `null`, and must be used sequentially: in fact no method of the parent visitor must be called before a nested signature is fully visited.

Like for classes, the ASM API provides two components based on this interface: the `SignatureReader` component parses a signature and calls the appropriate visit methods on a given signature visitor, and the `SignatureWriter` component builds a signature based on the method calls it received.

These two classes can be used to generate and transform signatures by using the same principles as with classes and methods. For example, let's suppose you want to rename the class names that appear in some signatures. This can be done with the following signature adapter, which forwards all the method calls it receives unchanged, except for the `visitClassType` and `visitInnerClassType` methods (we suppose here that `sv` methods always return `this`, which is the case of `SignatureWriter`):

```
public class RenameSignatureAdapter implements SignatureVisitor {
  private SignatureVisitor sv;
  private Map<String, String> renaming;
  private String oldName;
  public RenameSignatureAdapter(SignatureVisitor sv,
      Map<String, String> renaming) {
    this.sv = sv;
    this.renaming = renaming;
  }
  public void visitFormalTypeParameter(String name) {
    sv.visitFormalTypeParameter(name);
  }
  public SignatureVisitor visitClassBound() {
    sv.visitClassBound();
    return this;
  }
  public SignatureVisitor visitInterfaceBound() {
    sv.visitInterfaceBound();
    return this;
  }
  ...
  public void visitClassType(String name) {
    oldName = name;
    String newName = renaming.get(oldName);
    sv.visitClassType(newName == null ? name : newName);
  }
  public void visitInnerClassType(String name) {
    oldName = oldName + "." + name;
    String newName = renaming.get(oldName);
```

```
      sv.visitInnerClassType(newName == null ? name : newName);
    }
    public void visitTypeArgument() {
      sv.visitTypeArgument();
    }
    public SignatureVisitor visitTypeArgument(char wildcard) {
      sv.visitTypeArgument(wildcard);
      return this;
    }
    public void visitEnd() {
      sv.visitEnd();
    }
  }
```

Then the result of the following code is `"LA<TK;TV;>.B<TK;>;"`:

```
String s = "Ljava/util/HashMap<TK;TV;>.HashIterator<TK;>;";
Map<String, String> renaming = new HashMap<String, String>();
renaming.put("java/util/HashMap", "A");
renaming.put("java/util/HashMap.HashIterator", "B");
SignatureWriter sw = new SignatureWriter();
SignatureVisitor sa = new RenameSignatureAdapter(sw, renaming);
SignatureReader sr = new SignatureReader(s);
sr.acceptType(sa);
sw.toString();
```

4.1.3 Tools

The `TraceClassVisitor` and `ASMifierClassVisitor` classes, presented in section 2.3, print the signatures contained in class files in their internal form. They can be used to find the signature corresponding to a given generic type in the following way: write a Java class with some generic types, compile it, and use these command line tools to find the corresponding signatures.

4.2 Annotations

Class, field, method and method parameter annotations, such as `@Deprecated` or `@Override`, are stored in compiled classes if their retention policy is not `RetentionPolicy.SOURCE`. This information is not used at runtime by the bytecode instructions, but it can be accessed via the reflection API if the retention policy is `RetentionPolicy.RUNTIME`. It can also be used by compilers.

4.2.1 Structure

Annotations in source code can have various forms, such as `@Deprecated`, `@Retention(RetentionPolicy.CLASS)` or `@Task(desc="refactor", id=1)`. Internally, however, all annotations have the same form and are specified by an annotation type and by a set of name value pairs, where values are restricted to:

- primitive, `String` or `Class` values,
- enum values,
- annotation values,
- arrays of the above values.

Note that an annotation can contain other annotations, or even annotation arrays. Annotations can therefore be quite complex.

4.2.2 Interfaces and components

The ASM API for generating and transforming annotations is based on the `AnnotationVisitor` interface (see Figure 4.3).

```
public interface AnnotationVisitor {
  void visit(String name, Object value);
  void visitEnum(String name, String desc, String value);
  AnnotationVisitor visitAnnotation(String name, String desc);
  AnnotationVisitor visitArray(String name);
  void visitEnd();
}
```

Figure 4.3: The `AnnotationVisitor` interface

The methods of this interface are used to visit the name value pairs of an annotation (the annnotation type is visited in the methods that return this interface, i.e. the `visitAnnotation` methods). The first method is used for primitive, `String` and `Class` values (the later being represented by `Type` objects), and the others are used for enum, annotation and array values. They can be called in any order, except `visitEnd`:

```
( visit | visitEnum | visitAnnotation | visitArray )* visitEnd
```

Note that two methods return an `AnnotationVisitor`: this is because annotations can contain other annotations. Also unlike with the `MethodVisitor`s returned by a `ClassVisitor`, the `AnnotationVisitor`s returned by these two methods must be used sequentially: in fact no method of the parent visitor must be called before a nested annotation is fully visited.

Note also that the `visitArray` method returns an `AnnotationVisitor` to visit the elements of an array. However, since the elements of an array are not named, the `name` arguments are ignored by the methods of the visitor returned by `visitArray`, and can be set to `null`.

Adding, removing and detecting annotations

Like for fields and methods, an annotation can be removed by returning `null` in the `visitAnnotation` methods:

```
public class RemoveAnnotationAdapter extends ClassAdapter {
  private String annDesc;
  public RemoveAnnotationAdapter(ClassVisitor cv, String annDesc) {
    super(cv);
    this.annDesc = annDesc;
  }
  @Override
  public AnnotationVisitor visitAnnotation(String desc, boolean vis) {
    if (desc.equals(annDesc)) {
      return null;
    }
    return cv.visitAnnotation(desc, vis);
  }
}
```

Adding a class annotation is more difficult because of the constraints in which the methods of the `ClassVisitor` interface must be called. Indeed all the methods that may follow a `visitAnnotation` must be overridden to detect when all annotations have been visited (method annotations are easier to add, thanks to the `visitCode` method):

```
public class AddAnnotationAdapter extends ClassAdapter {
  private String annotationDesc;
  private boolean isAnnotationPresent;
  public AddAnnotationAdapter(ClassVisitor cv, String annotationDesc) {
    super(cv);
    this.annotationDesc = annotationDesc;
  }
```

```
    @Override public void visit(int version, int access, String name,
        String signature, String superName, String[] interfaces) {
      int v = (version & 0xFF) < V1_5 ? V1_5 : version;
      cv.visit(v, access, name, signature, superName, interfaces);
    }
    @Override public AnnotationVisitor visitAnnotation(String desc,
        boolean visible) {
      if (visible && desc.equals(annotationDesc)) {
        isAnnotationPresent = true;
      }
      return cv.visitAnnotation(desc, visible);
    }
    @Override public void visitInnerClass(String name, String outerName,
        String innerName, int access) {
      addAnnotation();
      cv.visitInnerClass(name, outerName, innerName, access);
    }
    @Override
    public FieldVisitor visitField(int access, String name, String desc,
        String signature, Object value) {
      addAnnotation();
      return cv.visitField(access, name, desc, signature, value);
    }
    @Override
    public MethodVisitor visitMethod(int access, String name,
        String desc, String signature, String[] exceptions) {
      addAnnotation();
      return cv.visitMethod(access, name, desc, signature, exceptions);
    }
    @Override public void visitEnd() {
      addAnnotation();
      cv.visitEnd();
    }
    private void addAnnotation() {
      if (!isAnnotationPresent) {
        AnnotationVisitor av = cv.visitAnnotation(annotationDesc, true);
        if (av != null) {
          av.visitEnd();
        }
        isAnnotationPresent = true;
      }
    }
  }
```

Note that this adapter upgrades the class version to 1.5 if it was less than that. This is necessary because the JVM ignores annotations in classes whose version is less than 1.5.

The last and probably most frequent use case of annotations in class and method adapters is to use annotations in order to parameterize a transformation. For instance you could transform field accesses only for fields that have a `@Persistent` annotation, add logging code only to methods that have a `@Log` annotation, and so on. All these use cases can easily be implemented because annotations must be visited first: class annotations must be visited before fields and methods, and method and parameter annotations must be visited before the code. It is therefore sufficient to set a flag when the desired annotation is detected, and to use it later on in the transformation, as is done in the above example with the `isAnnotationPresent` flag.

4.2.3 Tools

The `TraceClassVisitor`, `CheckClassAdapter` and `ASMifierClassVisitor` classes, presented in section 2.3, also support annotations (like for methods, it is also possible to use `TraceAnnotationVisitor` or `CheckAnnotationAdapter` to work at the level of individual annotations instead of at the class level). They can be used to see how to generate some specific annotation. For example using:

```
java -classpath asm.jar:asm-util.jar \
     org.objectweb.asm.util.ASMifierClassVisitor \
     java.lang.Deprecated
```

prints code that, after minor refactoring, reads:

```
package asm.java.lang;
import org.objectweb.asm.*;
public class DeprecatedDump implements Opcodes {
  public static byte[] dump() throws Exception {
    ClassWriter cw = new ClassWriter(0);
    AnnotationVisitor av;
    cw.visit(V1_5, ACC_PUBLIC + ACC_ANNOTATION + ACC_ABSTRACT
        + ACC_INTERFACE, "java/lang/Deprecated", null,
        "java/lang/Object",
        new String[] { "java/lang/annotation/Annotation" });
    {
      av = cw.visitAnnotation("Ljava/lang/annotation/Documented;",
          true);
      av.visitEnd();
    }
    {
      av = cw.visitAnnotation("Ljava/lang/annotation/Retention;", true);
      av.visitEnum("value", "Ljava/lang/annotation/RetentionPolicy;",
          "RUNTIME");
      av.visitEnd();
    }
    cw.visitEnd();
    return cw.toByteArray();
  }
}
```

This code shows how two create an annotation class, with the `ACC_ANNOTATION` flag, and shows how to create two class annotations, one without value, and one with an enum value. Method and parameter annotations can be created in a similar way, with the `visitAnnotation` and `visitParameterAnnotation` methods defined in the `MethodVisitor` interface.

4.3 Debug

Classes compiled with `javac -g` contain the name of their source file, a mapping between source line numbers and bytecode instructions, and a mapping betwen local variable names in source code and local variable slots in bytecode. This optional information is used in debuggers and in exception stack traces when it is available.

4.3.1 Structure

The source file name of a class is stored in a dedicated class file structure section (see Figure 2.1).

The mapping between source line numbers and bytecode instructions is stored as a list of (*line number*, *label*) pairs in the compiled code section of methods. For example if *l1*, *l2* and *l3* are three labels that appear in this order, then the following pairs:

```
(n1, l1)
(n2, l2)
(n3, l3)
```

mean that instructions between *l1* and *l2* come from line `n1`, that instructions between *l2* and *l3* come from line `n2`, and that instructions after *l3* come from line `n3`. Note that a given line number can appear in several pairs. This is because the instructions corresponding to expressions that appear on a single source line may not be contiguous in the bytecode. For example `for (`*init*`;` *cond*`;` *incr*`)` *statement*`;` is generaly compiled in the following order: *init statement incr cond*.

The mapping between local variable names in source code and local variable slots in bytecode is stored as a list of (*name*, *type descriptor*, *type signature*, *start*, *end*, *index*) tuples in the compiled code section of methods. Such a tuple means that, between the two labels *start* and *end*, the local variable in slot *index* corresponds to the local variable whose name and type in source code are given by the first three tuple elements. Note that the compiler may use the same local variable slot to store distinct source local variables with different scopes. Conversely a unique source local variable may be compiled into a local variable slot with a non contiguous scope. For instance it is possible to have a situation like this:

```
l1:
  ... // here slot 1 contains local variable i
l2:
  ... // here slot 1 contains local variable j
l3:
  ... // here slot 1 contains local variable i again
end:
```

The corresponding tuples are:

```
("i", "I", null, l1, l2, 1)
("j", "I", null, l2, l3, 1)
("i", "I", null, l3, end, 1)
```

4.3.2 Interfaces and components

The debug information is visited with three methods of the `ClassVisitor` and `MethodVisitor` interfaces:

- the source file name is visited with the `visitSource` method of the `ClassVisitor` interface;
- the mapping between source line numbers and bytecode instructions is visited with the `visitLineNumber` method of the `MethodVisitor` interface, one pair at a time;
- the mapping between local variable names in source code and local variable slots in bytecode is visited with the `visitLocalVariable` method of the `MethodVisitor` interface, one tuple at a time.

The `visitLineNumber` method must be called after the label passed as argument has been visited. In practice it is called just after this label, which makes it very easy to know the source line of the current instruction in a method visitor:

```
public class MyAdapter extends MethodAdapter {
  int currentLine;
  public MyAdapter(MethodVisitor mv) {
    super(mv);
  }
  @Override
  public void visitLineNumber(int line, Label start) {
    mv.visitLineNumber(line, start);
    currentLine = line;
  }
  ...
}
```

Similarly the `visitLocalVariable` method must be called after the labels passed as argument have been visited. Here are example method calls that correspond to the pairs and tuples presented in the previous section:

```
visitLineNumber(n1, l1);
visitLineNumber(n2, l2);
visitLineNumber(n3, l3);
visitLocalVariable("i", "I", null, l1, l2, 1);
visitLocalVariable("j", "I", null, l2, l3, 1);
visitLocalVariable("i", "I", null, l3, end, 1);
```

Ignoring debug information

In order to visit line numbers and local variable names, the `ClassReader` class may need to introduce "artificial" `Label` objects, in the sense that they are not needed by jump instructions, but only to represent the debug information. This can introduce false positives in situations such as the one explained in section 3.2.5, where a `Label` in the middle of an instruction sequence was considered to be a jump target, and therefore prevented this sequence from being removed.

In order to avoid these false positives it is possible to use the `SKIP_DEBUG` option in the `ClassReader.accept` method. With this option the class reader does not visit the debug information, and does not create artificial labels for it. Of course the debug information will be removed from the class, so this option can be used only if this is not a problem for your application.

Note: the ClassReader class provides other options such as SKIP_CODE to skip the visit of compiled code (this can be useful if you just need the class structure), SKIP_FRAMES to skip the stack map frames, and EXPAND_FRAMES to uncompress these frames.

4.3.3 Tools

Like for generic types and annotations, you can use the TraceClassVisitor, CheckClassAdapter and ASMifierClassVisitor classes to find how to work with debug information.

Part II

Tree API

5 Classes

This chapter explains how to generate and transform classes with the ASM tree API. It starts with a presentation of the tree API alone, and then explains how to compose it with the core API. The tree API for the content of methods, annotations and generics is explained in the next chapters.

5.1 Interfaces and components

5.1.1 Presentation

The ASM tree API for generating and transforming compiled Java classes is based on the `ClassNode` class (see Figure 5.1).

```
public class ClassNode ... {
  public int version;
  public int access;
  public String name;
  public String signature;
  public String superName;
  public List<String> interfaces;
  public String sourceFile;
  public String sourceDebug;
  public String outerClass;
  public String outerMethod;
  public String outerMethodDesc;
  public List<AnnotationNode> visibleAnnotations;
  public List<AnnotationNode> invisibleAnnotations;
  public List<Attribute> attrs;
  public List<InnerClassNode> innerClasses;
  public List<FieldNode> fields;
  public List<MethodNode> methods;
}
```

Figure 5.1: The `ClassNode` class (only fields and inherited fields are shown)

As you can see the public fields of this class correspond to the class file structure sections presented in Figure 2.1. The content of these fields is the same as in the core API. For instance **`name`** is an internal name and **`signature`** is a class signature (see sections 2.1.2 and 4.1). Some fields contain other *Xxx*`Node` classes: these classes, presented in details in the next chapters, have a similar structure, i.e. have fields that correspond to sub sections of the class file structure. For instance the **`FieldNode`** class looks like this:

```
public class FieldNode ... {
  public int access;
  public String name;
  public String desc;
  public String signature;
  public Object value;
  public FieldNode(int access, String name, String desc,
      String signature, Object value) {
    ...
  }
  ...
}
```

The **`MethodNode`** class is similar:

```
public class MethodNode ... {
  public int access;
  public String name;
  public String desc;
  public String signature;
  public List<String> exceptions;
  ...
  public MethodNode(int access, String name, String desc,
      String signature, String[] exceptions)
  {
    ...
  }
}
```

5.1.2 Generating classes

Generating a class with the tree API simply consists in creating a **`ClassNode`** object and in initializing its fields. For instance the **`Comparable`** interface in section 2.2.3 can be built as follows, with approximatively the same amount of code as in section 2.2.3:

```
ClassNode cn = new ClassNode();
```

```
cn.version = V1_5;
cn.access = ACC_PUBLIC + ACC_ABSTRACT + ACC_INTERFACE;
cn.name = "pkg/Comparable";
cn.superName = "java/lang/Object";
cn.interfaces.add("pkg/Mesurable");
cn.fields.add(new FieldNode(ACC_PUBLIC + ACC_FINAL + ACC_STATIC,
    "LESS", "I", null, new Integer(-1)));
cn.fields.add(new FieldNode(ACC_PUBLIC + ACC_FINAL + ACC_STATIC,
    "EQUAL", "I", null, new Integer(0)));
cn.fields.add(new FieldNode(ACC_PUBLIC + ACC_FINAL + ACC_STATIC,
    "GREATER", "I", null, new Integer(1)));
cn.methods.add(new MethodNode(ACC_PUBLIC + ACC_ABSTRACT,
    "compareTo", "(Ljava/lang/Object;)I", null, null));
```

Using the tree API to generate a class takes about 30% more time (see Appendix A.1) and consumes more memory than using the core API. But it makes it possible to generate the class elements in any order, which can be convenient in some cases.

5.1.3 Adding and removing class members

Adding and removing class members simply consists in adding or removing elements in the `fields` or `methods` lists of a `ClassNode` object. For example, if we define the `ClassTransformer` class as follows, in order to be able to compose class transformers easily:

```
public class ClassTransformer {
  protected ClassTransformer ct;
  public ClassTransformer(ClassTransformer ct) {
    this.ct = ct;
  }
  public void transform(ClassNode cn) {
    if (ct != null) {
      ct.transform(cn);
    }
  }
}
```

then the `RemoveMethodAdapter` in section 2.2.5 can be implemented as follows:

```
public class RemoveMethodTransformer extends ClassTransformer {
  private String methodName;
  private String methodDesc;
  public RemoveMethodTransformer(ClassTransformer ct,
      String methodName, String methodDesc) {
```

```
    super(ct);
    this.methodName = methodName;
    this.methodDesc = methodDesc;
  }
  @Override public void transform(ClassNode cn) {
    Iterator i = cn.methods.iterator();
    while (i.hasNext()) {
      MethodNode mn = (MethodNode) i.next();
      if (methodName.equals(mn.name) && methodDesc.equals(mn.desc)) {
        i.remove();
      }
    }
    super.transform(cn);
  }
}
```

As can be seen the main difference with the core API is that you need to iterate over all methods, while you don't need to do so with the core API (this is done for you in `ClassReader`). In fact this difference is valid for almost all tree based transformations. For instance the `AddFieldAdapter` of section 2.2.6 also needs an iterator when implemented with the tree API:

```
public class AddFieldTransformer extends ClassTransformer {
  private int fieldAccess;
  private String fieldName;
  private String fieldDesc;
  public AddFieldTransformer(ClassTransformer ct, int fieldAccess,
      String fieldName, String fieldDesc) {
    super(ct);
    this.fieldAccess = fieldAccess;
    this.fieldName = fieldName;
    this.fieldDesc = fieldDesc;
  }
  @Override public void transform(ClassNode cn) {
    boolean isPresent = false;
    for (FieldNode fn : (List<FieldNode>) cn.fields) {
      if (fieldName.equals(fn.name)) {
        isPresent = true;
        break;
      }
    }
    if (!isPresent) {
      cn.fields.add(new FieldNode(fieldAccess, fieldName, fieldDesc,
          null, null));
    }
    super.transform(cn);
  }
```

```
    }
```

Like for class generation, using the tree API to transform classes takes more time and consumes more memory than using the core API. But it makes it possible to implement some transformations more easily. This is the case, for example, of a transformation that adds to a class an annotation containing a digital signature of its content. With the core API the digital signature can be computed only when all the class has been visited, but then it is too late to add an annotation containing it, because annotations must be visited before class members. With the tree API this problem disappears because there is no such constraint in this case.

In fact it *is* possible to implement the `AddDigitialSignature` example with the core API, but then the class must be transformed in two *passes*. During the first pass the class is visited with a `ClassReader` (and no `ClassWriter`), in order to compute the digital signature based on the class content. During the second pass the same `ClassReader` is reused to do a second visit of the class, this time with an `AddAnnotationAdapter` chained to a `ClassWriter`. By generalizing this argument we see that, in fact, any transformation can be implemented with the core API alone, by using several passes if necessary. But this increases the transformation code complexity, this requires to store state between passes (which can be as complex as a full tree representation!), and parsing the class several times has a cost, which must be compared to the cost of constructing the corresponding `ClassNode`.

The conclusion is that *the tree API is generally used for transformations that cannot be implemented in one pass with the core API*. But there are of course exceptions. For example an obfuscator cannot be implemented in one pass, because you cannot transform classes before the mapping from original to obfuscated names is fully constructed, which requires to parse all classes. But the tree API is not a good solution either, because it would require keeping in memory the object representation of *all* the classes to obfuscate. In this case it is better to use the core API with two passes: one to compute the mapping between original and obfuscated names (a simple hash table that requires much less memory than a full object representation of all the classes), and one to transform the classes based on this mapping.

5.2 Components composition

So far we have only seen how to create and transform ClassNode objects, but we haven't seen how to construct a ClassNode from the byte array representation of a class or, vice versa, to construct this byte array from a ClassNode. In fact this is done by composing the core API and tree API components, as explained in this section.

5.2.1 Presentation

In addition to the fields shown in Figure 5.1, the ClassNode class implements the ClassVisitor interface, and also provides an accept method that takes a ClassVisitor as parameter. The accept method generates events based on the ClassNode field values, while the ClassVisitor methods perform the inverse operation, i.e. set the ClassNode fields based on the received events:

```
public class ClassNode ... implements ClassVisitor {
  ...
  public void visit(int version, int access, String name,
      String signature, String superName, String[] interfaces[]) {
    this.version = version;
    this.access = access;
    this.name = name;
    this.signature = signature;
    ...
  }
  ...
  public void accept(ClassVisitor cv) {
    cv.visit(version, access, name, signature, ...);
    ...
  }
}
```

Constructing a ClassNode from a byte array can therefore be done by composing it with a ClassReader, so that the events generated by the ClassReader are consumed by the ClassNode component, resulting in the initialization of its fields (as can be seen from the above code):

```
ClassNode cn = new ClassNode();
ClassReader cr = new ClassReader(...);
cr.accept(cn, 0);
```

Symetrically a `ClassNode` can be converted to its byte array representation by composing it with a `ClassWriter`, so that the events generated by the `ClassNode`'s `accept` method are consumed by the `ClassWriter`:

```
ClassWriter cw = new ClassWriter(0);
cn.accept(cw);
byte[] b = cw.toByteArray();
```

5.2.2 Patterns

Transforming a class with the tree API can be done by putting these elements together:

```
ClassNode cn = new ClassNode();
ClassReader cr = new ClassReader(...);
cr.accept(cn, 0);
... // here transform cn as you want
ClassWriter cw = new ClassWriter(0);
cn.accept(cw);
byte[] b = cw.toByteArray();
```

It is also possible to use a tree based class transformer like a class adapter with the core API. Two common patterns are used for that. The first one uses inheritance:

```
public class MyClassAdapter extends ClassNode {
  ClassVisitor cv;
  public MyClassAdapter(ClassVisitor cv) {
    this.cv = cv;
  }
  @Override public void visitEnd() {
    // put your transformation code here
    accept(cv);
  }
}
```

When this class adapter is used in a classical transformation chain:

```
ClassWriter cw = new ClassWriter(0);
ClassVisitor ca = new MyClassAdapter(cw);
ClassReader cr = new ClassReader(...);
cr.accept(ca, 0);
byte[] b = cw.toByteArray();
```

the events generated by `cr` are consumed by the `ClassNode ca`, which results in the initialization of the fields of this object. At the end, when the `visitEnd` event is consumed, `ca` performs the transformation and, by calling its `accept` method, generates new events corresponding to the transformed class, which are consumed by `cw`. The corresponding sequence diagram is show in Figure 5.2, if we suppose that `ca` changes the class version.

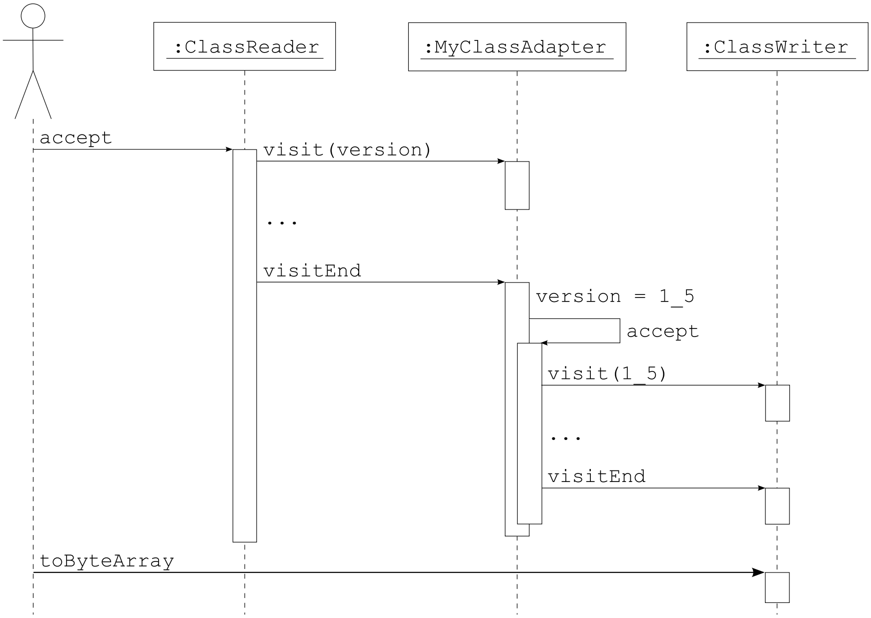


Figure 5.2: Sequence diagram for `MyClassAdapter`

When compared to the sequence diagram for `ChangeVersionAdapter` in Figure 2.7, we can see that the events between `ca` and `cw` occur after the events between `cr` and `ca`, instead of simultaneously with a normal class adapter. In fact this happens with all tree based transformations, and explains why they are less constrained than event based ones.

The second pattern that can be used to acheive the same result, with a similar sequence diagram, uses delegation instead of inheritance:

```
public class MyClassAdapter extends ClassAdapter {
  ClassVisitor next;
  public MyClassAdapter(ClassVisitor cv) {
    super(new ClassNode());
    next = cv;
  }
  @Override public void visitEnd() {
    ClassNode cn = (ClassNode) cv;
    // put your transformation code here
    cn.accept(next);
  }
}
```

This pattern uses two objects instead of one, but works exactly in the same way as the first pattern: the received events are used to construct a `ClassNode`, which is transformed and converted back to an event based representation when the last event is received.

Both patterns allow you to compose your tree based class adapters with event based adapters. They can also be used to compose tree based adapters together, but if you only need to compose tree based adapters this is not the best solution: in this case using classes such as `ClassTransformer` will avoid unnecessary conversions between the two representations.

6 Methods

This chapter explains how to generate and transform methods with the ASM tree API. It starts with a presentation of the tree API alone, with some illustrative examples, and then presents how to compose it with the core API. The tree API for generics and annotations is presented in the next chapter.

6.1 Interfaces and components

6.1.1 Presentation

The ASM tree API for generating and transforming methods is based on the `MethodNode` class (see Figure 6.1).

```
public class MethodNode ... {
  public int access;
  public String name;
  public String desc;
  public String signature;
  public List<String> exceptions;
  public List<AnnotationNode> visibleAnnotations;
  public List<AnnotationNode> invisibleAnnotations;
  public List<Attribute> attrs;
  public Object annotationDefault;
  public List<AnnotationNode>[] visibleParameterAnnotations;
  public List<AnnotationNode>[] invisibleParameterAnnotations;
  public InsnList instructions;
  public List<TryCatchBlockNode> tryCatchBlocks;
  public List<LocalVariableNode> localVariables;
  public int maxStack;
  public int maxLocals;
}
```

Figure 6.1: The `MethodNode` class (only fields and inherited fields are shown)

Most of the fields of this class are similar to the corresponding fields in `ClassNode`. The most important ones are the last ones, starting from the `instructions` field. This field is a list of instructions, managed with an `InsnList` object, whose public API is the following:

```
public class InsnList {
  int size();
  AbstractInsnNode getFirst();
  AbstractInsnNode getLast();
  AbstractInsnNode get(int index);
  boolean contains(AbstractInsnNode insn);
  int indexOf(AbstractInsnNode insn);
  void accept(MethodVisitor mv);
  ListIterator iterator();
  ListIterator iterator(int index);
  AbstractInsnNode[] toArray();
  void set(AbstractInsnNode location, AbstractInsnNode insn);
  void add(AbstractInsnNode insn);
  void add(InsnList insns);
  void insert(AbstractInsnNode insn);
  void insert(InsnList insns);
  void insert(AbstractInsnNode location, AbstractInsnNode insn);
  void insert(AbstractInsnNode location, InsnList insns);
  void insertBefore(AbstractInsnNode location, AbstractInsnNode insn);
  void insertBefore(AbstractInsnNode location, InsnList insns);
  void remove(AbstractInsnNode insn);
  void clear();
}
```

An `InsnList` is a doubly linked list of instructions, *whose links are stored in the* `AbstractInsnNode` *objects themselves.* This point is extremely important because it has many consequences on the way instruction objects and instruction lists must be used:

- An `AbstractInsnNode` object cannot appear more than once in an instruction list.
- An `AbstractInsnNode` object cannot belong to several instruction lists at the same time.
- As a consequence, adding an `AbstractInsnNode` to a list requires removing it from the list to which it was belonging, if any.
- As another consequence, adding all the elements of a list into another one clears the first list.

The AbstractInsnNode class is the super class of the classes that represent bytecode instructions. Its public API is the following:

```
public abstract class AbstractInsnNode {
  int getOpcode();
  int getType();
  AbstractInsnNode getPrevious();
  AbstractInsnNode getNext();
  void accept(MethodVisitor cv);
  AbstractInsnNode clone(Map labels);
}
```

Its sub classes are *Xxx*InsnNode classes corresponding to the visit*Xxx*Insn methods of the MethodVisitor interface, and are all build in the same way. For instance the VarInsnNode class corresponds to the visitVarInsn method and has the following structure:

```
public class VarInsnNode extends AbstractInsnNode {
  public int var;
  public VarInsnNode(int opcode, int var) {
    super(opcode);
    this.var = var;
  }
  ...
}
```

Labels and frames, as well as line numbers, although they are not instructions, are also represented by sub classes of the AbstractInsnNode classes, namely the LabelNode, FrameNode and LineNumberNode classes. This allows them to be inserted just before the corresponding real instructions in the list, as in the core API (where labels and frames are visited just before their corresponding instruction). It is therefore easy to find the target of a jump instruction, with the getNext method provided by the AbstractInsnNode class: this is the first AbstractInsnNode after the target label that is a real instruction. Another consequence is that, like with the core API, removing an instruction does not break jump instructions, as long as labels remain unchanged.

6.1.2 Generating methods

Generating a method with the tree API consists in creating a MethodNode and in initializing its fields. The most interesting part is the generation of the method's code. As an example, the checkAndSetF method of section 3.1.5 can be generated as follows:

```
MethodNode mn = new MethodNode(...);
InsnList il = mn.instructions;
il.add(new VarInsnNode(ILOAD, 1));
LabelNode label = new LabelNode();
il.add(new JumpInsnNode(IFLT, label));
il.add(new VarInsnNode(ALOAD, 0));
il.add(new VarInsnNode(ILOAD, 1));
il.add(new FieldInsnNode(PUTFIELD, "pkg/Bean", "f", "I"));
LabelNode end = new LabelNode();
il.add(new JumpInsnNode(GOTO, end));
il.add(label);
il.add(new FrameNode(F_SAME, 0, null, 0, null));
il.add(new TypeInsnNode(NEW, "java/lang/IllegalArgumentException"));
il.add(new InsnNode(DUP));
il.add(new MethodInsnNode(INVOKESPECIAL,
    "java/lang/IllegalArgumentException", "<init>", "()V"));
il.add(new InsnNode(ATHROW));
il.add(end);
il.add(new FrameNode(F_SAME, 0, null, 0, null));
il.add(new InsnNode(RETURN));
mn.maxStack = 2;
mn.maxLocals = 2;
```

Like with classes, using the tree API to generate methods takes more time and consumes more memory than using the core API. But it makes it possible to generate their content in any order. In particular the instructions can be generated in a different order than the sequential one, which can be useful in some cases.

Consider for example an expression compiler. Normaly an expression e_1+e_2 is compiled by emitting code for e_1, then emitting code for e_2, and then emitting code for adding the two values. But if e_1 and e_2 are not of the same primitive type, a cast must be inserted just after the code for e_1, and another one just after the code for e_2. However the exact casts that must be emitted depend on e_1 *and* e_2 types.

Now, if the type of an expression is returned by the method that emits the compiled code, we have a problem if we are using the core API: the cast that must be inserted after e_1 is known only after e_2 has been compiled, but this is too late because we cannot insert an instruction between previously visited instructions[1]. With the tree API this problem does not exist. For example, one possibility is to use a `compile` method such as:

[1] the solution is to compile expressions in two passes: one to compute the expression types and the casts that must be inserted, and one to emit the compiled code.

```
public Type compile(InsnList output) {
  InsnList il1 = new InsnList();
  InsnList il2 = new InsnList();
  Type t1 = e1.compile(il1);
  Type t2 = e2.compile(il2);
  Type t = ...; // compute common super type of t1 and t2
  output.addAll(il1); // done in constant time
  output.add(...); // cast instruction from t1 to t
  output.addAll(il2); // done in constant time
  output.add(...); // cast instruction from t2 to t
  output.add(new InsnNode(t.getOpcode(IADD)));
  return t;
}
```

6.1.3 Transforming methods

Transforming a method with the tree API simply consists in modifying the fields of a `MethodNode` object, and in particular the `instructions` list. Although this list can be modified in arbitray ways, a common pattern is to modify it while iterating over it. Indeed, unlike with the general `ListIterator` contract, the `ListIterator` returned by an `InsnList` supports many conccurrent[2] list modifications. In fact you can use the `InsnList` methods to remove one or more elements before and including the current one, to remove one or more elements after the *next* element (i.e. not just after the current element, but after its successor), or to insert one or more elements before the current one or after its successor. These changes will be reflected in the iterator, i.e. the elements inserted (resp. removed) after the next element will be seen (resp. not seen) in the iterator.

Another common pattern to modify an instruction list, when you need to insert several instructions after an instruction *i* inside a list, is to add these new instructions in a temporary instruction list, and to insert this temporary list inside the main one in one step:

```
InsnList il = new InsnList();
il.add(...);
...
il.add(...);
mn.instructions.insert(i, il);
```

[2] i.e. modifications interleaved with calls to `Iterator.next`. True, multi-threaded concurrent modifications are not supported.

Inserting the instructions one by one is also possible but more cumbersome, because the insertion point must be updated after each insertion.

6.1.4 Stateless and statefull transformations

Let's take some examples to see concretely how methods can be transformed with the tree API. In order to see the differences between the core and the tree API, it is interesting to reimplement the `AddTimerAdapter` example of section 3.2.4 and the `RemoveGetFieldPutFieldAdapter` of section 3.2.5. The timer example can be implemented as follows:

```
public class AddTimerTransformer extends ClassTransformer {
  public AddTimerTransformer(ClassTransformer ct) {
    super(ct);
  }
  @Override public void transform(ClassNode cn) {
    for (MethodNode mn : (List<MethodNode>) cn.methods) {
      if ("<init>".equals(mn.name) || "<clinit>".equals(mn.name)) {
        continue;
      }
      InsnList insns = mn.instructions;
      if (insns.size() == 0) {
        continue;
      }
      Iterator j = insns.iterator();
      while (j.hasNext()) {
        AbstractInsnNode in = (AbstractInsnNode) j.next();
        int op = in.getOpcode();
        if ((op >= IRETURN && op <= RETURN) || op == ATHROW) {
          InsnList il = new InsnList();
          il.add(new FieldInsnNode(GETSTATIC, cn.name, "timer", "J"));
          il.add(new MethodInsnNode(INVOKESTATIC, "java/lang/System",
              "currentTimeMillis", "()J"));
          il.add(new InsnNode(LADD));
          il.add(new FieldInsnNode(PUTSTATIC, cn.name, "timer", "J"));
          insns.insert(in.getPrevious(), il);
        }
      }
      InsnList il = new InsnList();
      il.add(new FieldInsnNode(GETSTATIC, cn.name, "timer", "J"));
      il.add(new MethodInsnNode(INVOKESTATIC, "java/lang/System",
          "currentTimeMillis", "()J"));
      il.add(new InsnNode(LSUB));
      il.add(new FieldInsnNode(PUTSTATIC, cn.name, "timer", "J"));
      insns.insert(il);
      mn.maxStack += 4;
    }
    int acc = ACC_PUBLIC + ACC_STATIC;
    cn.fields.add(new FieldNode(acc, "timer", "J", null, null));
    super.transform(cn);
  }
}
```

You can see here the pattern discussed in the previous section for inserting several instructions in the instruction list, which consists in using a temporary instruction list. This example also shows that it is possible to insert instructions before the current one while iterating over an instruction list. Note that the amount of code that is necessary to implement this adapter is approximatively the same with the core and tree APIs.

The method adapter that removes field self assignments (see section 3.2.5) can be implemented as follows (if we suppose that `MethodTransformer` is similar to the `ClassTransformer` class of the previous chapter):

```
public class RemoveGetFieldPutFieldTransformer extends
    MethodTransformer {
  public RemoveGetFieldPutFieldTransformer(MethodTransformer mt) {
    super(mt);
  }
  @Override public void transform(MethodNode mn) {
    InsnList insns = mn.instructions;
    Iterator i = insns.iterator();
    while (i.hasNext()) {
      AbstractInsnNode i1 = (AbstractInsnNode) i.next();
      if (isALOAD0(i1)) {
        AbstractInsnNode i2 = getNext(i1);
        if (i2 != null && isALOAD0(i2)) {
          AbstractInsnNode i3 = getNext(i2);
          if (i3 != null && i3.getOpcode() == GETFIELD) {
            AbstractInsnNode i4 = getNext(i3);
            if (i4 != null && i4.getOpcode() == PUTFIELD) {
              if (sameField(i3, i4)) {
                while (i.next() != i4) {
                }
                insns.remove(i1);
                insns.remove(i2);
                insns.remove(i3);
                insns.remove(i4);
              }
            }
          }
        }
      }
    }
    super.transform(mn);
  }
  private static AbstractInsnNode getNext(AbstractInsnNode insn) {
    do {
      insn = insn.getNext();
      if (insn != null && !(insn instanceof LineNumberNode)) {
        break;
      }
    } while (insn != null);
    return insn;
  }
  private static boolean isALOAD0(AbstractInsnNode i) {
    return i.getOpcode() == ALOAD && ((VarInsnNode) i).var == 0;
  }
  private static boolean sameField(AbstractInsnNode i,
      AbstractInsnNode j) {
    return ((FieldInsnNode) i).name.equals(((FieldInsnNode) j).name);
  }
}
```

Here again we can see that it is possible to remove instructions in an instruction list while iterating over it. Note however the `while (i.next() != i4)` loop: this is necessary to place the iterator *after* the instructions that must be removed (since it is not possible to remove the instruction just after the current one). Both the visitor and tree based implementations can detect labels and frames in the middle of the sequence to be detected, and in this case do not remove it. But ignoring the line numbers inside the sequence requires more code with the tree based API (see the `getNext` method) than with the core API. The major difference between the two implementations, however, is that no state machine is needed with the tree API. In particular the special case of three or more successive `ALOAD 0` instructions, which can easily be overlooked, is no longer a problem.

With the above implementation, a given instruction may be examined more than once since, at each step of the `while` loop, `i2`, `i3` and `i4`, which will be examined in future iterations, may also be examined at this iteration. It is in fact possible to use a more efficient implementation, where each instruction is examined at most once:

```
  public class RemoveGetFieldPutFieldTransformer2 extends
      MethodTransformer {
    ...
    @Override public void transform(MethodNode mn) {
```

```
    InsnList insns = mn.instructions;
    Iterator i = insns.iterator();
    while (i.hasNext()) {
      AbstractInsnNode i1 = (AbstractInsnNode) i.next();
      if (isALOAD0(i1)) {
        AbstractInsnNode i2 = getNext(i);
        if (i2 != null && isALOAD0(i2)) {
          AbstractInsnNode i3 = getNext(i);
          while (i3 != null && isALOAD0(i3)) {
            i1 = i2;
            i2 = i3;
            i3 = getNext(i);
          }
          if (i3 != null && i3.getOpcode() == GETFIELD) {
            AbstractInsnNode i4 = getNext(i);
            if (i4 != null && i4.getOpcode() == PUTFIELD) {
              if (sameField(i3, i4)) {
                insns.remove(i1);
                insns.remove(i2);
                insns.remove(i3);
                insns.remove(i4);
              }
            }
          }
        }
      }
    }
    super.transform(mn);
  }
  private static AbstractInsnNode getNext(Iterator i) {
    while (i.hasNext()) {
      AbstractInsnNode in = (AbstractInsnNode) i.next();
      if (!(in instanceof LineNumberNode)) {
        return in;
      }
    }
    return null;
  }
  ...
}
```

The difference with the previous implementation is the `getNext` method, which now acts on the list iterator. When the sequence is recognized the iterator is just after it, so the `while (i.next() != i4)` loop is no longer necessary. But here the special case of three or more successive `ALOAD 0` instructions shows up again (see the `while (i3 != null)` loop).

6.1.5 Global transformations

All the method transformations that we have seen so far were *local*, even the statefull ones, in the sense that the transformation of an instruction *i* only depended on instructions at a fixed distance from *i*. There are however *global* transformations, where the transformation of an instruction *i* may depend on instructions that can be at an arbitrary distance of *i*. For these transformations the tree API is really helpful, i.e., using the core API to implement them would be really complicated.

One example is a transformation that replaces a jump to a `GOTO` *`label`* instruction with a jump to *`label`*, and that replaces a `GOTO` to a `RETURN` instruction with this `RETURN` instruction. Indeed the target of a jump instruction can be at an arbitrary distance of this instruction, before or after it. Such a transformation can be implemented as follows:

```
public class OptimizeJumpTransformer extends MethodTransformer {
  public OptimizeJumpTransformer(MethodTransformer mt) {
    super(mt);
  }
  @Override public void transform(MethodNode mn) {
    InsnList insns = mn.instructions;
    Iterator i = insns.iterator();
    while (i.hasNext()) {
      AbstractInsnNode in = (AbstractInsnNode) i.next();
      if (in instanceof JumpInsnNode) {
        LabelNode label = ((JumpInsnNode) in).label;
        AbstractInsnNode target;
        // while target == goto l, replace label with l
        while (true) {
          target = label;
          while (target != null && target.getOpcode() < 0) {
            target = target.getNext();
          }
          if (target != null && target.getOpcode() == GOTO) {
            label = ((JumpInsnNode) target).label;
          } else {
            break;
          }
        }
        // update target
        ((JumpInsnNode) in).label = label;
        // if possible, replace jump with target instruction
        if (in.getOpcode() == GOTO && target != null) {
          int op = target.getOpcode();
          if ((op >= IRETURN && op <= RETURN) || op == ATHROW) {
            // replace 'in' with clone of 'target'
            insns.set(in, target.clone(null));
          }
        }
      }
    }
    super.transform(mn);
  }
}
```

This code works as follows: when a jump instruction `in` is found, its target is stored in `label`. Then the instruction that comes just after this label is searched for with the innermost while loop (`AbstractInsnNode` objects that do not represent real instructions, such as `FrameNode` or `LabelNode`, have a negative "opcode"). As long as this instruction is a `GOTO`, `label` is replaced with the target of this instruction, and the previous steps are repeated. Finally the target label of `in` is replaced with this updated `label` value and, if `in` is itself a `GOTO` and if its updated target is a `RETURN` instruction, `in` is replaced with a *clone* of this return instruction (recall that an instruction object cannot appear more than once in an instruction list).

The effect of this transformation on the `checkAndSetF` method defined in section 3.1.5 is shown below:

```
// before                  // after
  ILOAD 1                    ILOAD 1
  IFLT label                 IFLT label
  ALOAD 0                    ALOAD 0
  ILOAD 1                    ILOAD 1
  PUTFIELD ...               PUTFIELD ...
  GOTO end                   RETURN
label:                     label:
F_SAME                     F_SAME
  NEW ...                    NEW ...
  DUP                        DUP
  INVOKESPECIAL ...          INVOKESPECIAL ...
  ATHROW                     ATHROW
end:                       end:
F_SAME                     F_SAME
  RETURN                     RETURN
```

Note that, although this transformation changes the jump instructions (more formally the control flow graph), it does *not* need to update the method's frames. Indeed the state of the execution frame remains the same at each instruction and, since no new jump target is introduced, no new frame must be visited. It may happen, however, that a frame is no longer needed. For instance, in the above example, the ***end*** label is no longer used after transformation, as well as the F_SAME frame and the RETURN instruction after it. Hopefully it is perfectly legal to visit more frames than is strictly necessary, as well as to include unused code – called *dead* or *unreachable* code – in a method. The above method adapter is therefore correct, even if it could be improved to remove dead code and frames.

6.2 Components composition

So far we have only seen how to create and transform MethodNode objects, but we haven't seen the link with the byte array representation of classes. Like for classes, this link is done by composing the core API and tree API components, as explained in this section.

6.2.1 Presentation

In addition to the fields shown in Figure 6.1 the MethodNode class implements the MethodVisitor interface, and also provides two accept methods that take a MethodVisitor or a ClassVisitor as parameter. The accept methods generate events based on the MethodNode field values, while the MethodVisitor methods perform the inverse operation, i.e. set the MethodNode fields based on the received events.

6.2.2 Patterns

Like for classes, it is possible to use a tree based method transformer like a method adapter with the core API. The two patterns that can be used for classes are indeed also valid for methods, and work exactly in the same way. The pattern based on inheritance is the following:

```
public class MyMethodAdapter extends MethodNode {
  MethodVisitor mv;
  public MyMethodAdapter(int access, String name, String desc,
      String signature, String[] exceptions, MethodVisitor mv) {
    super(access, name, desc, signature, exceptions);
    this.mv = mv;
  }
  @Override public void visitEnd() {
    // put your transformation code here
    accept(mv);
  }
}
```

While the pattern based on delegation is:

```
  public class MyMethodAdapter extends MethodAdapter {
    MethodVisitor next;
    public MyMethodAdapter(int access, String name, String desc,
        String signature, String[] exceptions, MethodVisitor mv) {
      super(new MethodNode(access, name, desc, signature, exceptions));
      next = mv;
    }
    @Override public void visitEnd() {
      MethodNode mn = (MethodNode) mv;
      // put your transformation code here
      mn.accept(next);
    }
  }
```

A variant of the first pattern is to use it with an anonymous inner class directly in the `visitMethod` of a `ClassAdapter`:

```
  public MethodVisitor visitMethod(int access, String name,
      String desc, String signature, String[] exceptions) {
    return new MethodNode(access, name, desc, signature, exceptions) {
      @Override public void visitEnd() {
        // put your transformation code here
        accept(cv);
      }
    };
  }
```

These patterns show that it is possible to use the tree API only for methods, and the core API for classes. *In practice this strategy is often used.*

7 Method Analysis

This chapter presents the ASM API for analyzing the code of methods, which is based on the tree API. It starts with a presentation of code analysis algorithms, and then presents the corresponding ASM API, with some examples.

7.1 Presentation

Code analysis is a very large topic, and many algorithms exist for analyzing code. It would be impossible and out of the scope of this document to present them all here. In fact the goal of this section is just to give an overview of the algorithms that are used in ASM. A better presentation of this topic can be found in books about compilers. The next sections present two important types of code analysis techniques, namely data flow and control flow analysis:

- A *data flow* analysis consists in computing the state of the execution frames of a method, for each instruction of this method. This state can be represented in a more or less abstract way. For example reference values can be represented by a single value, by one value per class, by three possible values in the { null, not null, may be null } set, etc.
- A *control flow* analysis consists in computing the control flow graph of a method, and in performing analyses on this graph. The *control flow graph* is a graph whose nodes are instructions, and whose oriented edges connect two instructions $i \rightarrow j$ if j can be executed just after i.

7.1.1 Data flow analyses

Two types of data flow analyses can be performed:

- a *forward* analysis computes, for each instruction, the state of the execution frame after this instruction, from the state before its execution.

- a *backward* analysis computes, for each instruction, the state of the execution frame *before* this instruction, from the state after its execution.

A forward data flow analysis is performed by simulating the execution of each bytecode instruction of a method on its execution frame, which generally consists in:

- popping values from the stack,
- combining them,
- and pushing the result on the stack.

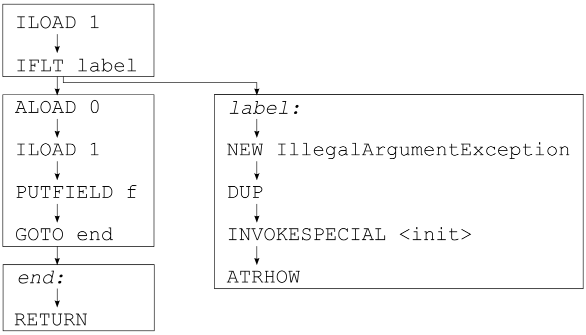
This looks like what an interpreter or the Java Virtual Machine does, but in fact it is completely different because the goal is to simulate *all potential execution paths* in a method, *for all possible argument values*, instead of the single execution path determined by some specific method argument values. One consequence is that, for branch instructions, *both* branches are simulated (while a real interpreter follows only one branch, depending on the actual condition value).

Another consequence is that the manipulated values are in fact *sets* of possible values. These sets can be very large, such as "all possible values", "all the integers", "all possible objects" or "all possible `String` objects", in which case they can also be called *types*. They can also be more precise, such as "all positive integers", "all integers between 0 and 10" or "all possible non `null` objects". Simulating the execution of an instruction *i* consists in finding the set of all possible results of *i*, for all combinations of values in its operand value sets. For instance, if integers are represented by three sets P = "positive or null", N = "negative or null", and A = "all integers", simulating the `IADD` instruction means returning P if both operands are P, N if both operands are N, and A in all other cases.

A last consequence is the need to compute *unions* of sets of values: for example the set of possible values corresponding to `(b ? e1 : e2)` is the union of the possible values of `e1` and of the possible values of `e2`. More generally this operation is needed each time the control flow graph contains two or more edges with a common destination. In the previous example, where integers are represented by the three sets P, N, and A, computing the union of two of these sets is easy: it is always A, unless the two sets are equal.

7.1.2 Control flow analyses

A control flow analysis is an analysis based on the control flow graph of a method. As an example, the control flow graph of the `checkAndSetF` method in section 3.1.3 is given below (with labels included in the graph like real instructions):



This graph can be decomposed in four *basic blocks* (shown above with rectangles), a basic block being a sequence of instructions such that each instruction except the last one has exactly one successor, and such that no instruction except the first one can be the target of a jump.

7.2 Interfaces and components

The ASM API for code analysis is in the `org.objectweb.asm.tree.analysis` package. As the package name implies, it is based on the tree API. In fact this package provides a framework for doing forward data flow analyses.

In order to be able to perform various data flow analyses, with more or less precise sets of values, the data flow analysis algorithm is split in two parts: one is fixed and is provided by the framework, the other is variable and provided by users. More precisely:

- The overall data flow analysis algorithm, and the task of popping from the stack, and pushing back to the stack, the appropriate number of values, is implemented once and for all in the `Analyzer` and `Frame` classes.

- The task of combining values and of computing unions of value sets is performed by user defined implementations of the `Interpreter` and `Value` *interfaces*. Several predefined implementations are provided, and are explained in the next sections.

Although the primary goal of the framework is to perform data flow analyses, the `Analyzer` class can also construct the control flow graph of the analyzed method. This can be done by overriding the `newControlFlowEdge` and `newControlFlowExceptionEdge` methods of this class, which by default do nothing. The result can be used for doing control flow analyses.

7.2.1 Basic data flow analysis

The `BasicInterpreter` class is one of the predefined implementations of the `Interpreter` interface. It simulates the effect of bytecode instructions by using seven sets of values, defined in the `BasicValue` class:

- `UNINITIALIZED_VALUE` means "all possible values".
- `INT_VALUE` means "all `int`, `short`, `byte`, `boolean` or `char` values".
- `FLOAT_VALUE` means "all `float` values".
- `LONG_VALUE` means "all `long` values".
- `DOUBLE_VALUE` means "all `double` values".
- `REFERENCE_VALUE` means "all object and array values".
- `RETURNADDRESS_VALUE` is used for subroutines (see Appendix A.2).

This interpreter is not very useful in itself (the method frames already provide such an information, with more details – see section 3.1.5), but it can be used as an "empty" `Interpreter` implementation in order to construct an `Analyzer`. This analyzer can then be used to detect unreachable code in a method. Indeed, even by following both branches in jumps instructions, it is not possible to reach code that cannot be reached from the first instruction. The consequence is that, after an analysis, and whatever the `Interpreter` implementation, the computed frames – returned by the `Analyzer.getFrames` method – are `null` for instructions that cannot be reached. This property can be used to implement a `RemoveDeadCodeAdapter` class very easily (there are more efficient ways, but they require to write more code):

```
public class RemoveDeadCodeAdapter extends MethodAdapter {
  String owner;
  MethodVisitor next;
  public RemoveDeadCodeAdapter(String owner, int access, String name,
      String desc, MethodVisitor mv) {
    super(new MethodNode(access, name, desc, null, null));
    this.owner = owner;
    next = mv;
  }
  @Override public void visitEnd() {
    MethodNode mn = (MethodNode) mv;
    Analyzer a = new Analyzer(new BasicInterpreter());
    try {
      a.analyze(owner, mn);
      Frame[] frames = a.getFrames();
      AbstractInsnNode[] insns = mn.instructions.toArray();
      for (int i = 0; i < frames.length; ++i) {
        if (frames[i] == null && !(insns[i] instanceof LabelNode)) {
          mn.instructions.remove(insns[i]);
        }
      }
    } catch (AnalyzerException ignored) {
    }
    mn.accept(next);
  }
}
```

When combined with the `OptimizeJumpAdapter` of section 6.1.5, the dead code introduced by the jump optimizer is removed. For example, using this adapter chain on the `checkAndSetF` method gives:

```
// after OptimizeJump        // after RemoveDeadCode
  ILOAD 1                      ILOAD 1
  IFLT label                   IFLT label
  ALOAD 0                      ALOAD 0
  ILOAD 1                      ILOAD 1
  PUTFIELD ...                 PUTFIELD ...
  RETURN                       RETURN
label:                       label:
F_SAME                       F_SAME
  NEW ...                      NEW ...
  DUP                          DUP
  INVOKESPECIAL ...            INVOKESPECIAL ...
  ATHROW                       ATHROW
end:                         end:
F_SAME
  RETURN
```

Note that dead labels are not removed. This is done on purpose: indeed it does not change the resulting code, but avoids removing a label that, although not reachable, might be referenced in a LocalVariableNode, for instance.

7.2.2 Basic data flow verifier

The BasicVerifier class extends the BasicInterpreter class. It uses the same seven sets but, unlike BasicInterpreter, checks that instructions are used correctly. For instance it checks that the operands of an IADD instruction are INTEGER_VALUE values (while BasicInterpreter just returns the result, i.e. INTEGER_VALUE). This class can be used for debugging purposes during the development of a class generator or adapter, as explained in section 3.3, in order to perform more verifications than those done in CheckMethodAdapter. For instance this class can detect that the ISTORE 1 ALOAD 1 sequence is invalid, which cannot be done with CheckMethodAdapter. It can be included in a utility method adapter like this:

```
public class BasicVerifierAdapter extends MethodAdapter {
  String owner;
  MethodVisitor next;
  public BasicVerifierAdapter(String owner, int access, String name,
      String desc, MethodVisitor mv) {
    super(new MethodNode(access, name, desc, null, null));
    this.owner = owner;
    next = mv;
  }
  @Override public void visitEnd() {
    MethodNode mn = (MethodNode) mv;
    Analyzer a = new Analyzer(new BasicVerifier());
    try {
      a.analyze(owner, mn);
    } catch (AnalyzerException e) {
      throw new RuntimeException(e.getMessage());
    }
    mn.accept(next);
  }
}
```

7.2.3 Simple data flow verifier

The SimpleVerifier class extends the BasicVerifier class. It uses more sets to simulate the execution of bytecode instructions: indeed each class is represented by its own set, representing all possible objects of this class. It can therefore detect more errors, such as the fact of calling a method defined in the String class on an object whose possible values are “all objects of type Thread”.

This class uses the Java reflection API in order to perform verifications and computations related to the class hierarchy. It therefore loads the classes referenced by a method into the JVM. This default behavior can be changed by overriding the protected methods of this class.

Like BasicVerifier, this class could be used during the development of a class generator or adapter in order find bugs more easily. But it can also be used for other purposes. One example is a transformation that removes unnecessary casts in methods: if this analyzer finds that the operand of a CHECKCAST to instruction is the set of values “all objects of type from”, and if to is a super class of from, then the CHECKCAST instruction is unnecessary and can be removed. The implementation of this transformation is the following:

```
public class RemoveUnusedCastTransformer extends MethodTransformer {
  String owner;
  public RemoveUnusedCastTransformer(String owner,
      MethodTransformer mt) {
    super(mt);
    this.owner = owner;
  }
  @Override public MethodNode transform(MethodNode mn) {
    Analyzer a = new Analyzer(new SimpleVerifier());
    try {
      a.analyze(owner, mn);
      Frame[] frames = a.getFrames();
      AbstractInsnNode[] insns = mn.instructions.toArray();
      for (int i = 0; i < insns.length; ++i) {
        AbstractInsnNode insn = insns[i];
        if (insn.getOpcode() == CHECKCAST) {
          Frame f = frames[i];
          if (f != null && f.getStackSize() > 0) {
            Object operand = f.getStack(f.getStackSize() - 1);
            Class to = getClass(((TypeInsnNode) insn).desc);
            Class from = getClass(((BasicValue) operand).getType());
            if (to.isAssignableFrom(from)) {
              mn.instructions.remove(insn);
            }
          }
        }
      }
    } catch (AnalyzerException ignored) {
    }
    return mt == null ? mn : mt.transform(mn);
  }
  private static Class getClass(String desc) {
    try {
      return Class.forName(desc.replace(’/’, ’.’));
    } catch (ClassNotFoundException e) {
      throw new RuntimeException(e.toString());
    }
  }
  private static Class getClass(Type t) {
    if (t.getSort() == Type.OBJECT) {
      return getClass(t.getInternalName());
    }
    return getClass(t.getDescriptor());
  }
}
```

For Java 6 classes (or classes upgraded to Java 6 with `COMPUTE_FRAMES`), however, it is simpler and much more efficient to use an `AnalyzerAdapter` for doing this with the core API:

```
public class RemoveUnusedCastAdapter extends MethodAdapter {
  public AnalyzerAdapter aa;
  public RemoveUnusedCastAdapter(MethodVisitor mv) {
    super(mv);
  }
  @Override public void visitTypeInsn(int opcode, String desc) {
    if (opcode == CHECKCAST) {
      Class to = getClass(desc);
      if (aa.stack != null && aa.stack.size() > 0) {
        Object operand = aa.stack.get(aa.stack.size() - 1);
        if (operand instanceof String) {
          Class from = getClass((String) operand);
          if (to.isAssignableFrom(from)) {
            return;
          }
        }
      }
    }
    mv.visitTypeInsn(opcode, desc);
  }
  private static Class getClass(String desc) {
    try {
      return Class.forName(desc.replace(’/’, ’.’));
    } catch (ClassNotFoundException e) {
      throw new RuntimeException(e.toString());
    }
    }
  }
```

7.2.4 User defined data flow analysis

Let's suppose that we would like to detect field accesses and method calls on potentially `null` objects, such as in the following source code fragment (where the first line prevents some compilers from detecting the bug, which would otherwise be detected as an "o may not have been initialized" error):

```
Object o = null;
while (...) {
  o = ...;
}
o.m(...); // potential NullPointerException!
```

Then we need a data flow analysis that can tell us that, at the `INVOKEVIRTUAL` instruction corresponding to the last line, the bottom stack value, corresponding to `o`, may be `null`. In order to do that we need to distinguish three sets for reference values: the `NULL` set containing the `null` value, the `NONNULL` set containing all non null reference values, and the `MAYBENULL` set containing all the reference values. Then we just need to consider that `ACONST_NULL` pushes the `NULL` set on the operand stack, while all other instructions that push a reference value on the stack push the `NONNULL` set (in other words we consider that the result of any field access or method call is not `null` – we cannot do better without a global analysis of all the classes of the program). The `MAYBENULL` set is necessary to represent the union of the `NULL` and `NONNULL` sets.

The above rules must be implemented in a custom `Interpreter` implementation. It would be possible to implement it from scratch, but it is also possible, and much easier, to implement it by extending the `BasicInterpreter` class. Indeed, if we consider that `BasicValue.REFERENCE_VALUE` corresponds to the `NONNULL` set, then we just need to override the method that simulates the execution of `ACONST_NULL`, so that it returns `NULL`, as well as the method that computes set unions:

```
class IsNullInterpreter extends BasicInterpreter {
  public final static BasicValue NULL = new BasicValue(null);
  public final static BasicValue MAYBENULL = new BasicValue(null);
  @Override public Value newOperation(AbstractInsnNode insn) {
    if (insn.getOpcode() == ACONST_NULL) {
      return NULL;
    }
    return super.newOperation(insn);
  }
  @Override public Value merge(Value v, Value w) {
    if (isRef(v) && isRef(w) && v != w) {
      return MAYBENULL;
    }
    return super.merge(v, w);
  }
  private boolean isRef(Value v) {
    return v == REFERENCE_VALUE || v == NULL || v == MAYBENULL;
  }
}
```

It is then easy to use this `IsNullInterpreter` in order to detect instructions that can lead to potential null pointer exceptions:

```
public class NullDereferenceAnalyzer {
  public List<AbstractInsnNode> findNullDereferences(String owner,
      MethodNode mn) throws AnalyzerException {
    List<AbstractInsnNode> result = new ArrayList<AbstractInsnNode>();
    Analyzer a = new Analyzer(new IsNullInterpreter());
    a.analyze(owner, mn);
    Frame[] frames = a.getFrames();
    AbstractInsnNode[] insns = mn.instructions.toArray();
    for (int i = 0; i < insns.length; ++i) {
      AbstractInsnNode insn = insns[i];
      if (frames[i] != null) {
        Value v = getTarget(insn, frames[i]);
        if (v == NULL || v == MAYBENULL) {
          result.add(insn);
        }
      }
    }
    return result;
  }
  private static Value getTarget(AbstractInsnNode insn, Frame f) {
    switch (insn.getOpcode()) {
    case GETFIELD:
    case ARRAYLENGTH:
    case MONITORENTER:
    case MONITOREXIT:
      return getStackValue(f, 0);
    case PUTFIELD:
      return getStackValue(f, 1);
    case INVOKEVIRTUAL:
    case INVOKESPECIAL:
```

```
    case INVOKEINTERFACE:
      String desc = ((MethodInsnNode) insn).desc;
      return getStackValue(f, Type.getArgumentTypes(desc).length);
    }
    return null;
  }
  private static Value getStackValue(Frame f, int index) {
    int top = f.getStackSize() - 1;
    return index <= top ? f.getStack(top - index) : null;
  }
}
```

The `findNullDereferences` method analyzes the given method node with an `IsNullInterpreter`. It then tests, for each instruction, if the set of possible values of its reference operand (if any) is the `NULL` or the `NONNULL` set. If it is the case this instruction may lead to a null pointer exception, so it is added to the list of such instructions that is returned by this method.

The `getTarget` method returns the `Value` corresponding to the object operand of `insn`, in the frame `f`, or `null` if `insn` does not have an object operand. Its main role is to compute the offset of this value from the top of the operand stack, which depends on the type of instruction.

7.2.5 Control flow analysis

Control flow analyses can have many applications. A simple example is to compute the *cyclomatic complexity* of methods. This metric is defined as the number of edges in the control flow graph, minus the number of nodes, plus two. For instance the cyclomatic complexity of the `checkAndSetF` method, whose control flow graph is shown in section 7.1.2, is $11 - 12 + 2 = 1$. This metric gives a good indication of the “complexity” of a method (there is a correlation between this number and the average number of bugs in a method). It also gives the recommended number of test cases that are necessary to “correctly” test a method.

The algorithm to compute this metric can be implemented with the ASM analysis framework (there are more efficient ways, based on the core API alone, but they require to write more code). The first step consits in constructing the control flow graph. As we said at the begining of this chapter, this can be done by overriding the `newControlFlowEdge` method of the `Analyzer` class. This class represents nodes as `Frame` objects. If we want to store the graph in these objects, we need to extend the `Frame` class:

```
class Node extends Frame {
  Set<Node> successors = new HashSet<Node>();
  public Node(int nLocals, int nStack) {
    super(nLocals, nStack);
  }
  public Node(Frame src) {
    super(src);
  }
}
```

Then we can provide an **Analyzer** subclass that constructs our control flow graph, and use its result to compute the number of edges, the number of nodes, and finally the cyclomatic complexity:

```
public class CyclomaticComplexity {
  public int getCyclomaticComplexity(String owner, MethodNode mn)
      throws AnalyzerException {
    Analyzer a = new Analyzer(new BasicInterpreter()) {
      protected Frame newFrame(int nLocals, int nStack) {
        return new Node(nLocals, nStack);
      }
      protected Frame newFrame(Frame src) {
        return new Node(src);
      }
      protected void newControlFlowEdge(Frame src, Frame dst) {
        Node s = (Node) getFrames()[src];
        s.successors.add((Node) getFrames()[dst]);
      }
    };
    a.analyze(owner, mn);
    Frame[] frames = a.getFrames();
    int edges = 0;
    int nodes = 0;
    for (int i = 0; i < frames.length; ++i) {
      if (frames[i] != null) {
        edges += ((Node) frames[i]).successors.size();
        nodes += 1;
      }
    }
    return edges - nodes + 2;
  }
}
```

8 Metadata

This chapter presents the tree API for compiled Java classes metadata, such as annotations. It is very short because these metadata have already been presented in chapter 4, and because the tree API is simple once the corresponding core API is known.

8.1 Generics

The tree API does not provide any support for generic types! Indeed it represents generic types with signatures, as in the core API, but does not provide a `SignatureNode` class corresponding to `SignatureVisitor`, although this would be possible (in fact it would be convenient to use several `Node` classes to distinguish between type, method and class signatures, at least).

8.2 Annotations

The tree API for annotations is based on the `AnnotationNode` class, whose public API is the following:

```
public class AnnotationNode implements AnnotationVisitor {
  String desc;
  List<Object> values;
  AnnotationNode(String desc);
  ... // methods of the AnnotationVisitor interface
  void accept(AnnotationVisitor av);
}
```

The `desc` field contains the annotation type, while the `values` field contains the name value pairs, where each name is followed by its associated value (the representation of values is described in the Javadoc).

As you can see the `AnnotationNode` class implements the `AnnotationVisitor` interface, and also provides an `accept` method that takes as parameter an object implementing this interface, like the ClassNode and MethodNode classes with the class and method visitor interfaces. The patterns that we have seen for classes and methods can therefore also be used for composing the core and tree API components for annotations. For example the "anonymous inner class" variant of the pattern based on inheritance (see section 6.2.2), adapted to annotations, gives:

```
public AnnotationVisitor visitAnnotation(String desc, boolean visible) {
  return new AnnotationNode(desc) {
    @Override public void visitEnd() {
      // put your annotation transformation code here
      accept(cv.visitAnnotation(desc, visible));
    }
  };
}
```

8.3 Debug

The source file from which a class was compiled is stored in the sourceFile field in ClassNode. The information about source line numbers is stored in LineNumberNode objects, whose class inherits from AbstractInsnNode. Similarly to the core API, where information about line numbers is visited at the same time as instructions, LineNumberNode objects are part of the instruction list. Finally the name and type of source local variables is stored in the MethodNode's localVariables field, which is a list of LocalVariableNode objects.

A Appendix

A.1 Bytecode instructions

This section gives a short description of the bytecode instructions. For a complete description, see the Java Virtual Machine Specification.

Conventions: a and b represent `int`, `float`, `long` or `double` values (e.g. they mean `int` for `IADD` but `long` for `LADD`), o and p represent objet references, v represents any value (or, for stack instructions, a value of size 1), w represents a `long` or `double`, and i, j and n represent `int` values.

Local variables

Instruction	Stack before	Stack after
`ILOAD`, `LLOAD`, `FLOAD`, `DLOAD` *var*	...	... , a
`ALOAD` *var*	...	... , o
`ISTORE`, `LSTORE`, `FSTORE`, `DSTORE` *var*	... , a	...
`ASTORE` *var*	... , o	...
`IINC` *var incr*	...	...

Stack

`POP`	... , v	...
`POP2`	... , v_1 , v_2	...
	... , w	...
`DUP`	... , v	... , v , **v**
`DUP2`	... , v_1 , v_2	... , v_1 , v_2 , $\mathbf{v_1}$, $\mathbf{v_2}$
	... , w	... , w, **w**
`SWAP`	... , v_1 , v_2	... , $v_{\mathbf{2}}$, $v_{\mathbf{1}}$
`DUP_X1`	... , v_1 , v_2	... , $\mathbf{v_2}$, v_1 , v_2

DUP_X2	… , v_1 , v_2 , v_3	… , $\mathbf{v_3}$, v_1 , v_2 , v_3
	… , w , v	… , **v** , w , v
DUP2_X1	… , v_1 , v_2 , v_3	… , $\mathbf{v_2}$, $\mathbf{v_3}$, v_1 , v_2 , v_3
	… , v , w	… , **w** , v , w
DUP2_X2	… , v_1 , v_2 , v_3 , v_4	… , $\mathbf{v_3}$, $\mathbf{v_4}$, v_1 , v_2 , v_3 , v_4
	… , w , v_1 , v_2	… , $\mathbf{v_1}$, $\mathbf{v_2}$, w , v_1 , v_2
	…. , v_1 , v_2 , w	… , **w** , v_1 , v_2 , w
	… , w_1 , w_2	… , $\mathbf{w_2}$, w_1 , w_2

Constants

ICONST_***n*** ($-1 \le n \le 5$)	…	… , n
LCONST_***n*** ($0 \le n \le 1$)	…	… , nL
FCONST_***n*** ($0 \le n \le 2$)	…	… , nF
DCONST_***n*** ($0 \le n \le 1$)	…	… , nD
BIPUSH b, $-128 \le b < 127$	…	… , b
SIPUSH s, $-32768 \le s < 32767$	…	… , s
LDC *cst* (int, float, long, double, String or Type)	…	… , *cst*
ACONST_NULL	…	… , null

Arithmetic and logic

IADD, LADD, FADD, DADD	… , a , b	… , a + b
ISUB, LSUB, FSUB, DSUB	… , a , b	… , a - b
IMUL, LMUL, FMUL, DMUL	… , a , b	… , a * b
IDIV, LDIV, FDIV, DDIV	… , a , b	… , a / b
IREM, LREM, FREM, DREM	… , a , b	… , a % b
INEG, LNEG, FNEG, DNEG	… , a	… , -a
ISHL, LSHL	… , a , n	… , a << n
ISHR, LSHR	… , a , n	… , a >> n
IUSHR, LUSHR	… , a , n	… , a >>> n
IAND, LAND	… , a , b	… , a & b
IOR, LOR	… , a , b	… , a | b
IXOR, LXOR	… , a , b	… , a ^ b
LCMP	… , a , b	… , a == b ? 0 : (a < b ? -1 : 1)

FCMPL, FCMPG	… , a , b	… , a == b ? 0 : (a < b ? -1 : 1)
DCMPL, DCMPG	… , a , b	… , a == b ? 0 : (a < b ? -1 : 1)

Casts

I2B	… , i	… , (byte) i
I2C	… , i	… , (char) i
I2S	… , i	… , (short) i
L2I, F2I, D2I	… , a	… , (int) a
I2L, F2L, D2L	… , a	… , (long) a
I2F, L2F, D2F	… , a	… , (float) a
I2D, L2D, F2D	… , a	… , (double) a
CHECKCAST *class*	… , o	… , (*class*) o

Objects, fields and methods

NEW *class*	…	… , new *class*
GETFIELD *c f t*	… , o	… , o.*f*
PUTFIELD *c f t*	… , o , v	…
GETSTATIC *c f t*	…	… , *c*.*f*
PUTSTATIC *c f t*	… , v	…
INVOKEVIRTUAL *c m t*	… , o , v_1 , … , v_n	… , o.*m*(v_1, … v_n)
INVOKESPECIAL *c m t*	… , o , v_1 , … , v_n	… , o.*m*(v_1, … v_n)
INVOKESTATIC *c m t*	… , v_1 , … , v_n	… , *c*.*m*(v_1, … v_n)
INVOKEINTERFACE *c m t*	… , o , v_1 , … , v_n	… , o.*m*(v_1, … v_n)
INSTANCEOF *class*	… , o	… , o instanceof *class*
MONITORENTER	… , o	…
MONITOREXIT	… , o	…

Arrays

NEWARRAY *type* (for any primitive type)	… , n	… , new *type*[n]
ANEWARRAY *class*	… , n	… , new *class*[n]
MULTIANEWARRAY *[…[t n*	… , i_1 , … , i_n	… , new *t*[i_1]…[i_n]…
BALOAD, CALOAD, SALOAD	… , o , i	… , o[i]

IALOAD, LALOAD, FALOAD, DALOAD	... , o , i	... , o[i]
AALOAD	... , o , i	... , o[i]
BASTORE, CASTORE, SASTORE	... , o , i , j	...
IASTORE, LASTORE, FASTORE, DASTORE	... , o , i , a	...
AASTORE	... , o , i , p	...
ARRAYLENGTH	... , o	... , o.length

Jumps

IFEQ	... , i	...	jump if i == 0
IFNE	... , i	...	jump if i != 0
IFLT	... , i	...	jump if i < 0
IFGE	... , i	...	jump if i >= 0
IFGT	... , i	...	jump if i > 0
IFLE	... , i	...	jump if i <= 0
IF_ICMPEQ	... , i , j	...	jump if i == j
IF_ICMPNE	... , i , j	...	jump if i != j
IF_ICMPLT	... , i , j	...	jump if i < j
IF_ICMPGE	... , i , j	...	jump if i >= j
IF_ICMPGT	... , i , j	...	jump if i > j
IF_ICMPLE	... , i , j	...	jump if i <= j
IF_ACMPEQ	... , o , p	...	jump if o == p
IF_ACMPNE	... , o , p	...	jump if o != p
IFNULL	... , o	...	jump if o == null
IFNONNULL	... , o	...	jump if o != null
GOTO	...	...	jump always
TABLESWITCH	... , i	...	jump always
LOOKUPSWITCH	... , i	...	jump always

Return

IRETURN, LRETURN, FRETURN, DRETURN	... , a	
ARETURN	... , o	
RETURN	...	
ATHROW	... , o	

A.2 Subroutines

In addition two the bytecode instructions presented in the previous section, classes whose version is *lower* than or equal to `V1_5` can also contain the `JSR` and `RET` instructions, used for subroutines (`JSR` means Jump to SubRoutine, and `RET` means RETurn from subroutine). Classes whose version is higher than or equal to `V1_6` must *not* contain these instructions (they have been removed to simplify the new verifier architecture introduced in Java 6; this was possible because they are not strictly necessary).

The `JSR` instruction takes a label as argument, and jumps unconditionaly to this label. Before doing so however, it pushes on the operand stack a *return address*, which is the index of the instruction just after the `JSR`. This return address can be manipulated only by the stack instructions such as `POP`, `DUP` or `SWAP`, by the `ASTORE` instruction, and by the `RET` instruction.

The `RET` instruction takes a local variable index as argument. It loads the return address contained in this slot and jumps unconditionaly to the corresponding instruction. Since the return address can have several possible values, a `RET` instruction can return to several possible instructions.

Let's take an example to illustrate this. Consider the following code:

```
  JSR sub
  JSR sub
  RETURN
sub:
  ASTORE 1
  IINC 0 1
  RET 1
```

The first instruction pushes as return address the index of the second instruction, and jumps to the `ASTORE` instruction. This instruction stores the return address in local variable 1. Then local variable 0 is incremented by one. Finally the RET instruction loads the return address contained in local variable 1 and jumps to the corresponding instruction, i.e. the second instruction.

This second instruction is again a `JSR` instruction: it pushes as return address the index of the third instruction, and jumps to the `ASTORE` instruction. When the `RET` instruction is reached again the return address corresponds now to the `RETURN` instruction, and so execution jumps to this `RETURN` and stops.

The instructions after the ***sub*** label define what is called a subroutine. It is like a little "method", which can be "called" from different places, inside a normal method. Subroutines were used, prior to Java 6, to compile `finally` blocks in Java. But in fact subroutines are not strictly necessary: it is indeed possible to replace each `JSR` instruction with the body of the corresponding subroutine. This *inlining* produces duplicated code but removes the `JSR` and `RET` instructions. With the above example the result is very simple:

```
IINC 0 1
IINC 0 1
RETURN
```

ASM provides a `JSRInlinerAdapter` class which can perform this transformation automatically, in the `org.objectweb.asm.commons` package. You can use it to remove `JSR` and `RET` instructions in order to simplify code analysis, or to transform classes from version 1.5 or less to 1.6 or higher.

A.3 Attributes

As explained in section 2.1.1, it is possible to associate arbitrary *attributes* to classes, fields and methods. This extensibility mechanism is very useful to extend the class file format when new features are introduced. For example it has been used to extend this format in order to support annotations, generics, stack map frames, etc. This mechanism can also be used by users, as opposed to Sun, but since the introduction of annotations in Java 5, *it is much easier to use annotations than attributes*. That being said, if you *really* need to use your own attributes, or if you must manage non standard attributes defined by others, this can be done in ASM with the `Attribute` class.

By default, the `ClassReader` class creates an `Attribute` instance for each non standard attribute it finds, and calls the `visitAttribute` method (of the `ClassVisitor`, `FieldVisitor` or `MethodVisitor` interface, depending on the context) with this instance as argument. This instance contains the raw content of the attribute, in the form of a private byte array. The `ClassWriter` class, when visiting such unknow attributes, just copies this raw byte array in the class it constructs. *This default behavior is safe only if the optimization described in section* 2.2.4 *is used* (this gives another reason to use this optimization, besides the performance gain). Without this option the raw content may become inconsistent with the new constant pool created by the class writer, resulting in a corrupted class file.

By default non standard attributes are therefore copied as is in transformed classes, and their content is completely opaque to ASM and to users. If you need access to this content, you must first define an `Attribute` sub class that is able to decode the raw content, and to reencode it. You must also pass a prototype instance of this class in the `ClassReader.accept` method, so that this class can decode attributes of this type. Let's take an example to illustrate this. The following class can be used to support an imaginary "Comment" attribute, whose raw content is a `short` value that references an UTF8 string stored in the constant pool:

```
class CommentAttribute extends Attribute {
  private String comment;
  public CommentAttribute(final String comment) {
    super("Comment");
    this.comment = comment;
  }
  public String getComment() {
    return comment;
  }
  @Override
  public boolean isUnknown() {
    return false;
  }
  @Override
  protected Attribute read(ClassReader cr, int off, int len,
      char[] buf, int codeOff, Label[] labels) {
    return new CommentAttribute(cr.readUTF8(off, buf));
  }
  @Override
  protected ByteVector write(ClassWriter cw, byte[] code, int len,
      int maxStack, int maxLocals) {
    return new ByteVector().putShort(cw.newUTF8(comment));
  }
}
```

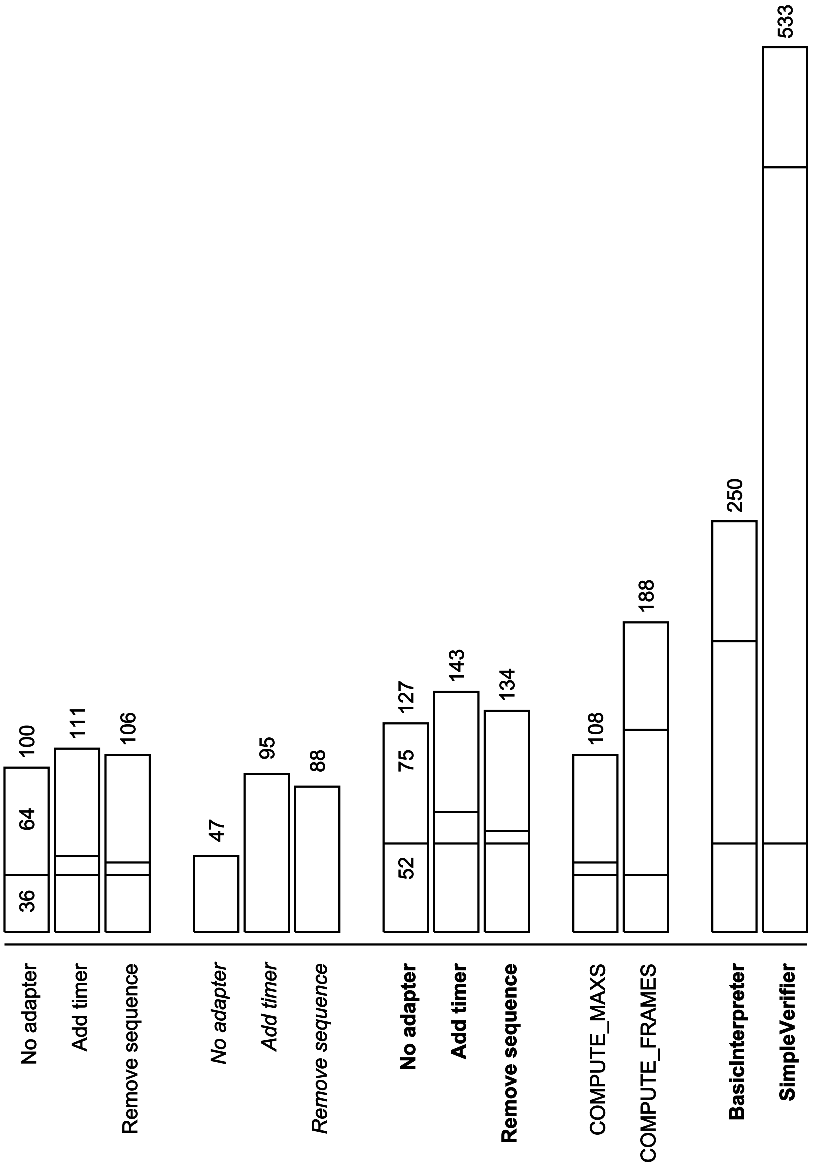
The most important methods are the `read` and `write` methods. The `read` method decodes the raw content of attributes of this type, and the `write` method performs the inverse operation. Note that the `read` method must return a *new* attribute instance. In order to decode attributes of this type when reading a class, you must use:

```
ClassReader cr = ...;
ClassVisitor cv = ...;
cr.accept(cv, new Attribute[] { new CommentAttribute("") }, 0);
```

The "Comment" attributes will then be recognized, and a `CommentAttribute` instance will be created for each of them (while unknown ones continue to be represented by `Attribute` instances).

A.4 Performances

The figure below gives the relative performances of the core and tree APIs, of the `ClassWriter` options, and of the analysis framework (shorter is faster):



The reference time 100 corresponds to a `ClassReader` chained directly to a `ClassWriter`. The “add timer” and “remove sequence” tests correspond to `AddTimerAdapter` and `RemoveGetFieldPutFieldAdapter` (*italic* means that the optimization described in section 2.2.4 is used, **bold** means that the tree API is used). The total transformation time is decomposed into three parts: class parsing (bottom), class transformation or analysis (middle) and class writing (top). For each test the measured value is the time needed to parse, transform and write a *byte array*, i.e. the time needed to load classes from disk and to load them inside the JVM is *not* taken into account. The results were obtained by running each test ten times, on the 12700+ classes of JDK 1.5 `rt.jar`, and by using the performance of the best run.

A quick analysis of these results shows that:

- 90% of the transformation time is due to class parsing and writing.
- The “copy constant pool” optimization gives a 15-20% speed up.
- Tree based transformations are about 30% slower than visitor based ones.
- The `COMPUTE_MAXS` option does not cost too much.
- The `COMPUTE_FRAMES` option costs a lot ⇒ do incremental frame updates.
- The cost of the analysis package is very high!

Index