The Hacker's Guide to Javac

David Erni and Adrian Kuhn

University of Bern, March 2008

Abstract

This document is an introduction to hacking the Java compiler. An introduction to the Java compiler is given. Two examples are exercised, a simple hello world and a AST rewriting plugin.

1 Introduction

With the release of Java 6, the Java compiler is available as open source. The compiler is part of the OpenJDK project, and can be downloaded from the Java compiler group website at http://www.openjdk.org/groups/compiler/. However, for the purpose of this tutorial, any Java 6 installation will work fine. The examples do not recompile the compiler, they just extend it.

This document introduces the internals of the Java compiler. First we give a broad overview of its internal parts and phases, then we exercise two examples. Both examples use the compiler's novel plugin mechanism, which has been specified as JSR 269. However, the second example goes way beyond JSR 269. Casting to the JSR objects to their compiler internal interfaces, we implement AST rewriting. Our example replaces assertions statements with explicit if-throw expressions.

2 Java Compiler in a Nutshell

This section summarizes the OpenJDKJava compiler passes and their documentation respectively. This section contains a short introduction to the main compiler passes.

The compilation process is managed by the Java Compiler class that is defined in com.sun.tools.javac.main. When the Java compiler is invoked with default compile policy it performs the following passes (see also Figure 1):

1. **parse:** Reads a set of *.java source files and maps the resulting token sequence into AST-Nodes.
2. **enter:** Enters symbols for the definitions into the symbol table.
3. **process annotations:** If Requested, processes annotations found in the specifed compilation units.
4. **attribute:** Attributes the Syntax trees. This step includes name resolution, type checking and constant folding.
5. **flow:** Performs dataflow analysis on the trees from the previous step. This includes checks for assignments and reachability.
6. **desugar:** Rewrites the AST and translates away some syntactic sugar.
7. **generate:** Generates Source Files or Class Files.

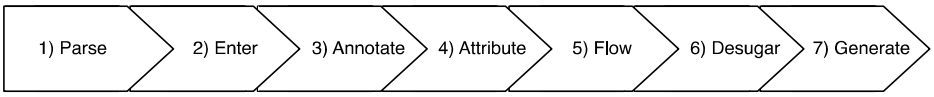

Figure 1: When the Java compiler is invoked with default compile policy it performs these passes.

2.1 Parse

To Parse the files, the compiler relies on the classes available under com.sun.tools.javac.parser.*. As a first step, the lexical analyzer maps an input stream consisting of characters into a token sequence. Then the parser maps the generated token sequence into an abstract syntax tree.

2.2 Enter

During this step the compiler registers symbols for all the definitions that were found into their enclosing scope. Enter consists of the following two phases:

During the first phase, all class symbols are registered with their corresponding scope. This is implemented by using a visitor that descends down the hierarchy, visiting all classes, including inner classes. The compiler attaches a MemberEnter object to each class symbol that will be used for completion in phase 2.

In the second phase these classes are completed using the MemberEnter object mentioned above. First step during completion is the determination of a class's parameters, supertype and interfaces. In a second step those symbols are entered into the class's scope (ignoring the class symbols that have been entered during the first step). Unlike the first phase, the second one is lazily executed. The members of a class are entered as soon as the contents of a class are first accessed. This is achieved by installing a completer object in the class symbols. Those objects can invoke the member-enter phase on-demand.

Finally, enter adds all top-level classes to a todo-queue that will be used during the Attribution pass.

2.3 Process Annotations

If an annotation processor is present and annotation processing is requested this pass processes any annotations found in the specified compilation units. JSR 269 defined an interface for writing such plugins [2]. However, this interface is very limited and does, in particular, not allow to extend the language with Collective Behavior. The main limitation being that JSR 269 does not provide sub-method reflection.

2.4 Attribute

Attributes all parse trees that are found in the todo-queue generated by the Enter pass. Note that attributing classes may cause additional files to be parsed and entered via the SourceCompleter.

Most of the context-dependent analysis happens during this pass. This includes name resolution, type checking and constant folding as subtasks. Some subtasks involve but are not limited to the following auxiliary classes.

- **Check:** This is the class for type checking. It will report errors such as completion errors or type errors.
- **Resolve:** Class for name resolution. It will report errors if the resolution fails.
- **ConstFold:** This is the class for constant folding. Constant folding is the process of simplifying constant expressions at compile time.
- **Infer:** Class for type parameter inference.

2.5 Flow

This pass performs data flow checks on the previously attributed parse trees. Liveness analysis checks that every statement is reachable. Exception analysis ensures that every checked exception that is thrown is declared or caught. Definite assignment analysis ensures that each variable is assigned when used. Definite unassignment analysis ensures that no final variable is assigned more than once.

2.6 Desugar

Removes syntactic sugar, such as inner classes, class literals, assertions, and foreach loops.

2.7 Generate

This (final) pass generates the source or class file for a list of classes. It decides to generate a source file or a class file depending on the compiler's options.

3 What is JSR 269?

Annotations have been introduced in Java 5 as a way to attach meta-information in source code. With JSR 269, Java 6 takes annotations to the next level. JSR 269, the Pluggable Annotation Processing API, extends the Java compiler with a plugin mechanism. With this JSR, it is now possible to write custom annotation processors that can be plugged-in to the compilation process.

JSR 269 has two basic pieces, an API that models the Java programming language and an API for writing annotation processors. Those pieces are in the new packages `javax.lang.model.*` and `javax.annotation.processing`, respectively. The JSR 269 functionality is exposed by new options of the javac command.

```
-proc:{none,only}      Control whether annotation processing
                       and/or compilation is done.
-processor <classes>   Names of the annotation processors to
                       run; bypasses default discovery process
-processorpath <path>  Specify where to find annotation
```

`processors`

Annotation processing is enabled by default in javac. To just run annotation processing without compiling source code to class files, use the `-proc:only` option.

4 How to print "Hello World!" with Javac

In this first example, we write a simple annotation processor that prints "hello worlds!" when compiling Java classes. We use the internal messaging of the compiler to print the hello world as a compiler note.

First, we define a `HelloWorld` annotation as follows

```
1 public @interface HelloWorld {
2
3 }
```

And a class `Dummy` that uses above annotation

```
1 @HelloWorld
2 public class Dummy {
3
4 }
```

Annotation processing happens in a sequence of rounds[1]. On each round, a processor may be asked to process a subset of the annotations found on the source and class files produced by a prior round. The inputs to the first round of processing are the initial inputs to a run of the tool; these initial inputs can be regarded as the output of a virtual zeroth round of processing. If a processor was asked to process on a given round, it will be asked to process on subsequent rounds, including the last round, even if there are no annotations for it to process. The tool infrastructure may also ask a processor to process files generated implicitly by the tool's operation.

The latter method processes a set of annotation types on type elements originating from the prior round and returns whether or not these annotations are claimed by this processor. If true is returned, the annotations are claimed and subsequent processors will not be asked to process them; if false is returned, the annotations are unclaimed and subsequent processors may be asked to process them. A processor may always return the same boolean

[1] http://java.sun.com/javase/6/docs/api/javax/annotation/processing/AbstractProcessor.html

value or may vary the result based on chosen criteria. To write an annotation processor, we subclass from `AbstractProcessor` and annotate our subclass with `SupportedAnnotationTypes` and `SupportedSourceVersion`. The subclass must override two methods:

- `public synchronized void init(ProcessingEnvironment processingEnv)`
- `public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)`

Both methods are called by the Java compiler during the annotation pass. The first method is called once to initialize the plugin, the latter method is called for each annotation round, plus once after all rounds are done.

Our simple `HelloWorldProcessor` is implemented as follows:

```
1  import java.util.Set;
2
3  import javax.annotation.processing.*;
4  import javax.lang.model.SourceVersion;
5  import javax.lang.model.element.TypeElement;
6  import javax.tools.Diagnostic;
7
8  @SupportedAnnotationTypes("HelloWorld")
9  @SupportedSourceVersion(SourceVersion.RELEASE_6)
10 public class HelloWorldProcessor extends AbstractProcessor {
11
12     @Override
13     public synchronized void init(ProcessingEnvironment
           processingEnv) {
14         super.init(processingEnv);
15     }
16
17     @Override
18     public boolean process(Set<? extends TypeElement> annotations,
           RoundEnvironment roundEnv) {
19         if (!roundEnv.processingOver()) {
20             processingEnv.getMessager().printMessage(
21                 Diagnostic.Kind.NOTE, "Hello Worlds!");
22         }
23         return true;
24     }
25
26 }
```

Line 8 registers the processor for the HelloWorld annotation. That is, we get called with a set of all program element where that annotation is present. Line 9 sets the supported source version.

Lines 12–15 override the initialization method, for now we simply delegate to the superclass.

Lines 17–24 override the processing method. This method gets called with a set of annotated program elements. The method is called once for each round, plus once in the end with an empty set of elements. Thus, we use a simple `if` to do nothing in that last round. For other rounds, we print a hello world message. We do not `System.out.print`, but rather use the message framework of the compiler to print a note. Other possible options would be to print a warning or an error.

The method returns `true` in order to claim the passed elements as processed.

To run the example, execute

```
javac HelloWorldProcessor.java
javac -processor HelloWorldProcessor *.java
```

This should output

```
Note: Hello Worlds!
```

5 How to abuse JSR269 for AST rewriting

In this example, we dive into the details of the compiler itself. Bending JSR269 beyond its limits, we implement an annotation processor that does AST rewriting. The processor will replace each assertion statements with an explicit throw expression. That is, each occurrence of

```
    assert cond : detail;
```

is replaced with

```
    if (!cond) throw new AssertionError(detail);
```

The latter does not generate an assert bytecode, but rather generates the usual series of bytecodes: an ordinary `if` statement with a throw clause. As a consequence, assertions are checked even when your VM runs without assertions enabled. This is most useful for libraries, where you have no control over your client's VM settings.

Again, we start by subclassing `AbstractProcessor`. However, this time we do not subscribe to a specific annotation, rather we pass the wildcard `"*"` to get invoked for the whole source code.

```
@SupportedAnnotationTypes("*")
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class ForceAssertions extends AbstractProcessor {

}
```

The initialize method is implement as follows:

```
30 private int tally;
31 private Trees trees;
32 private TreeMaker make;
33 private Name.Table names;
34
35 @Override
```

```
36 public synchronized void init(ProcessingEnvironment env) {
37     super.init(env);
38     trees = Trees.instance(env);
39     Context context = ((JavacProcessingEnvironment)
           env).getContext();
40     make = TreeMaker.instance(context);
41     names = Name.Table.instance(context);
42     tally = 0;
43 }
```

We use the processing environment to get a handle on necessary compiler components. Within the compiler, a single processing environment (or context, as it is called internally) is used for each invocation of the compiler. The context is then used to ensure a single copy of each compiler component exists per compiler invocation. In the compiler, we simply use Component.instance(context) to get the reference to the phase.

The components that we use are:

- Trees — a utility class of JSR269, that bridges between program elements and tree nodes. For example, given a method element, we can get its associated AST tree node.
- TreeMaker — an internal component of the compiler, that is a factory for creating tree nodes. The naming convention of this class is one of the many unorthodox idioms found in Javac's source code. Instead of common method names such as

  ```
  TreeMaker factory;
  factory.createNewClass( ... )
  ```

  their convention is to use

  ```
  TreeMaker make;
  make.NewClass( ... )
  ```

- Name.Table — another internal component of the compiler. Name is an abstraction for internal compiler strings. For efficiency reasons Javac uses hashed strings that are stored in a common large buffer.

Please note that, in line 39, we cast the JSR269 processing environment to its compiler internal type. This is our rabbit hole to wonderland.

Eventually, we initialize a counter variable to zero. The counter is used to report the number of applied replacements to the user.

The processing method is implemented as follows:

```
45 @Override
46 public boolean process(Set<? extends TypeElement> annotations,
       RoundEnvironment roundEnv) {
47     if (!roundEnv.processingOver()) {
48         Set<? extends Element> elements = roundEnv.getRootElements();
49         for (Element each : elements) {
50             if (each.getKind() == ElementKind.CLASS) {
51                 JCTree tree = (JCTree) trees.getTree(each);
52                 TreeTranslator visitor = new Inliner();
53                 tree.accept(visitor);
54             }
55         }
56     } else
57         processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE,
               tally + " assertions inlined.");
58     return false;
59 }
```

We iterate over all program elements and rewrite the AST of each class. In line 51, we go down the rabbit hole again, casting the JSR269 tree node to its compiler internal type. The difference between JSR269 tree nodes and internal tree node is, that JSR269 stops at method level, whereas internally all AST elements are accessible. We need full access in order to rewrite assertion statements.

Tree translation is done by subclassing `TreeTranslator` which is itself a subclass of `TreeVisitor`. Neither of these classes is part of JSR269, hence, from here on, all code that we present works on the internals of the Java compiler.

In line 57, the else part, we report the number of processed assertion statements. The else part is taken in the final processing round only.

The class `Inliner` implements the AST rewriting. It extends `TreeTranslator` and is an inner class of our processor. `TreeTranslator` implements the identity operation, that is, it does not transform any node.

```
private class Inliner extends TreeTranslator {

}
```

In order to transform assertion statements, we override the default behavior of `TreeTranslator.visitAssert(JCAssert)` as follows:

```
63 @Override
64 public void visitAssert(JCAssert tree) {
65     super.visitAssert(tree);
66     JCStatement newNode = makeIfThrowException(tree);
67     result = newNode;
68     tally++;
69 }
```

The node under transformation is passed in as method's argument. In line 67, the transformation result is returned by an assignment to the instance variable `TreeTranslator.result`.

By convention, a transformation method must be implemented as follows:

- Call super to ensure the transformation is applied to the node's children as well.
- Perform the actual transformation.
- Assign the transformed result to `TreeTranslator.result`. The type of the result does not have to be the same as passed in. Instead, we are free to return any type of tree node that is allowed by Java's grammar at that location. This constraint is not verified by the tree translator itself, but returning a wrong type leads to disaster later in the compilation process.

We delegate the transformation to a private method `makeIfThrowException` as follows:

```
71 private JCStatement makeIfThrowException(JCAssert node) {
72     // make: if (!(condition) throw new AssertionError(detail);
73     List<JCExpression> args = node.getDetail() == null
74             ? List.<JCExpression> nil()
75             : List.of(node.detail);
76     JCExpression expr = make.NewClass(
77             null,
78             null,
79             make.Ident(names.fromString("AssertionError")),
80             args,
81             null);
82     return make.If(
83             make.Unary(JCTree.NOT, node.cond),
84             make.Throw(expr),
85             null);
86 }
```

The method takes an assertion statement as argument and returns an `if` statement. This is a valid return value, since both tree nodes are statements and thus equivalent by Java's grammar. There is no production rule, that would prohibit the use of an `if` statement in place of an assertion statement.

`makeIfThrowException` is the workhorse of the AST rewriting. We use the `TreeMaker` to create new tree nodes. Given an expression of the pattern

```
    assert cond : detail;
```

we replace it by the following pattern

```
    if (!cond) throw new AssertionError(detail);
```

In lines 73–75 we cover the case of an assertion statement where detail has been omitted. In lines 76–81 we create a class creation AST node, that creates a new instance of `"AssertionError"`. In line 79, we use `Name.Table` to get an compiler internal string representation. In line 80, pass the arguments created above by lines 73–75. Lines 77, 78 and 81 pass a null value, as there is neither an outer instance, nor type paramaters, nor an anonymous class body.

In line 83, we invert the assert condition; in line 84, we create a throw expression, and eventually, in lines 82–85 we wrap the whole thing in an `if` statement.

NB: the `List` class is another remarkable implementation detail of the Java compiler. Rather than using the Java Collection Framework, the compiler uses its own data types. Both the `List` and the `Pair` datatype are implemented using cons, a Lisp idiom. Pairs are implemented as

```
public class Pair<A, B> {

    public final A fst;
    public final B snd;

    public Pair(A fst, B snd) {
        this.fst = fst;
        this.snd = snd;
    }

    ...

}
```

and lists are implemented as

```
public class List<A> extends AbstractCollection<A> implements
    java.util.List<A> {

    public A head;

    public List<A> tail;

    public List(A head, List<A> tail) {
        this.tail = tail;
        this.head = head;
    }

    ...

}
```

There are static convenience methods for the creation of lists, in particular

- `List.nil()`
- `List.of(A)`
- `List.of(A,A)`
- `List.of(A,A,A)`
- `List.of(A,A,A,A...)`

The same applies for pairs,

- `Pair.of(A, B)`

Again, the unorthodox naming convention leads to more beautiful code. Instead of Java's usual

```
List list = new List();
list.add(a);
list.add(b);
list.add(c);
```

you just write

```
List.of(a, b, c);
```

5.1 Running the AST Rewriter

In order to demo the AST rewriter, we implement

```
1 public class Example {
2
3     public static void main(String[] args) {
4         String str = null;
5         assert str != null : "Must not be null";
6     }
7
8 }
```

and execute

```
javac ForceAssertions.java
javac -processor ForceAssertions Example.java
```

This produces the output

```
Note: 1 assertions inlined.
```

Now, when executing `Example` with assertions disabled

```
java  -disableassertions Example
```

we nevertheless get

```
Exception in thread "main" java.lang.AssertionError: Must not be null
        at Example.main(Example.java:1)
```

Using the hidden compiler option `-printsource` we can even pretty print the rewritten AST as Java sources! Special care has to be taken to redirect the output, otherwise the original files may be overwritten.

Executing

```
javac -processor ForceAssertions -printsource -d gen Example.java
```

yields

```
1  public class Example {
2
3      public Example() {
4          super();
5      }
6
7      public static void main(String[] args) {
8          String str = null;
9          if (!(str != null)) throw new AssertionError("Must not be
               null");
10     }
11 }
```

As you can see, line 9 has been rewritten, and in lines 3–5 an empty default constructor has been inserted.

5.2 How to register a plugin as service

Java offers a mechanism to register services. If an annotation processor is registered as service, the compiler automatically discovers the plugin. Registration works by placing a file name `javax.annotation.processing.Processor` in the folder `META-INF/services` somewhere in the classpath. The file format is straightforward, it must contain the fully qualified name of the to-be-registered annotation processors. Each name must be on a separate line.

5.3 Further Reading

Erni describes a more complex compiler modification in his bachelor's thesis. He does not rely on casting JSR269 but modifies the compiler directly by using several point cuts during the compilation process [1].

References

[1] David Erni. JAG - a Prototype for Collective Behavior in Java. Bachelors Thesis, University of Bern. March 2008.

[2] Joseph D. Darcy. JSR-000269 Pluggable Annotation Processing API, December 2006. http://jcp.org/en/jsr/detail?id=269.

A Installation of Java 6 under OSX

By default, current OSX installations are shipped with Java 5.0 only. In order to install Java 6, download the installation bundle from http://www.apple.com/support/downloads/javaformacosx105update1.html. The update requires Mac OS X 10.5.2 or later, and a 64-bit Intel-based Mac. The update does not replace the existing installation of J2SE 5.0 or change the default version of Java.

B Full Source code of ForceAssertions.java

```
1  import java.util.Set;
2
3  import javax.annotation.processing.AbstractProcessor;
4  import javax.annotation.processing.ProcessingEnvironment;
5  import javax.annotation.processing.RoundEnvironment;
6  import javax.annotation.processing.SupportedAnnotationTypes;
7  import javax.annotation.processing.SupportedSourceVersion;
8  import javax.lang.model.SourceVersion;
9  import javax.lang.model.element.Element;
10 import javax.lang.model.element.ElementKind;
11 import javax.lang.model.element.TypeElement;
12 import javax.tools.Diagnostic;
13
14 import com.sun.source.util.Trees;
15 import com.sun.tools.javac.processing.JavacProcessingEnvironment;
16 import com.sun.tools.javac.tree.JCTree;
17 import com.sun.tools.javac.tree.TreeMaker;
18 import com.sun.tools.javac.tree.TreeTranslator;
19 import com.sun.tools.javac.tree.JCTree.JCAssert;
20 import com.sun.tools.javac.tree.JCTree.JCExpression;
21 import com.sun.tools.javac.tree.JCTree.JCStatement;
22 import com.sun.tools.javac.util.Context;
23 import com.sun.tools.javac.util.List;
24 import com.sun.tools.javac.util.Name;
25
```

```
26 @SupportedAnnotationTypes("*")
27 @SupportedSourceVersion(SourceVersion.RELEASE_6)
28 public class ForceAssertions extends AbstractProcessor {
29
30     private int tally;
31     private Trees trees;
32     private TreeMaker make;
33     private Name.Table names;
34
35     @Override
36     public synchronized void init(ProcessingEnvironment env) {
37         super.init(env);
38         trees = Trees.instance(env);
39         Context context = ((JavacProcessingEnvironment)
               env).getContext();
40         make = TreeMaker.instance(context);
41         names = Name.Table.instance(context);
42         tally = 0;
43     }
44
45     @Override
46     public boolean process(Set<? extends TypeElement> annotations,
           RoundEnvironment roundEnv) {
47         if (!roundEnv.processingOver()) {
48             Set<? extends Element> elements =
                   roundEnv.getRootElements();
49             for (Element each : elements) {
50                 if (each.getKind() == ElementKind.CLASS) {
51                     JCTree tree = (JCTree) trees.getTree(each);
52                     TreeTranslator visitor = new Inliner();
53                     tree.accept(visitor);
54                 }
55             }
56         } else
57             processingEnv.getMessager().printMessage(
                   Diagnostic.Kind.NOTE, tally + " assertions
                   inlined.");
58         return false;
59     }
60
61     private class Inliner extends TreeTranslator {
62
63         @Override
64         public void visitAssert(JCAssert tree) {
65             super.visitAssert(tree);
66             JCStatement newNode = makeIfThrowException(tree);
67             result = newNode;
68             tally++;
69         }
```

```
70
71          private JCStatement makeIfThrowException(JCAssert node) {
72              // make: if (!(condition) throw new
                    AssertionError(detail);
73              List<JCExpression> args = node.getDetail() == null
74                      ? List.<JCExpression> nil()
75                      : List.of(node.detail);
76              JCExpression expr = make.NewClass(
77                      null,
78                      null,
79                      make.Ident(names.fromString("AssertionError")),
80                      args,
81                      null);
82              return make.If(
83                      make.Unary(JCTree.NOT, node.cond),
84                      make.Throw(expr),
85                      null);
86          }
87
88      }
89
90  }
91
```