Lombok

Version: db71f39c271f1f8124fac96daa68d8b012fbf390

Parents:

0de56b76e6e9ba738232a3eb6c66c16df7346c82

17972d59fa7e2eec6b73ba5da8234f5fa7ac2536

Merge base:

6c1993659cd53f601520209771d116cb94e9b825

lombok/src/core/lombok/eclipse/HandlerLibrary.java

Chunk 1: (concatenarion/import declaration)

```
import lombok.core.AnnotationValues.AnnotationValueDecodeFail;
<<<<<<< HEAD
import lombok.core.configuration.ConfigurationKeysLoader;
=======
import lombok.core.BooleanFieldAugment;
>>>>>>> 17972d59fa7e2eec6b73ba5da8234f5fa7ac2536
import lombok.core.HandlerPriority;
```

```
import lombok.core.AnnotationValues.AnnotationValueDecodeFail;
import lombok.core.configuration.ConfigurationKeysLoader;
import lombok.core.BooleanFieldAugment;
import lombok.core.HandlerPriority;
```

lombok/src/core/lombok/eclipse/handlers/EclipseHandlerUtil.java

Chunk 2: (combination/import declaration)

```
import lombok.core.AnnotationValues.AnnotationValue;
<<<<<<< HEAD
=======
import lombok.core.BooleanFieldAugment;
import lombok.core.ReferenceFieldAugment;
import lombok.core.TransformationsUtil;
>>>>>>> 17972d59fa7e2eec6b73ba5da8234f5fa7ac2536
import lombok.core.TypeResolver;
```

```
import lombok.core.AnnotationValues.AnnotationValue;
import lombok.core.BooleanFieldAugment;
import lombok.core.ReferenceFieldAugment;
import lombok.core.TypeResolver;
```

lombok/src/core/lombok/javac/HandlerLibrary.java

Chunk 3: (combination/import declaration)

```
import lombok.core.TypeResolver;
<<<<<<< HEAD
import lombok.core.AnnotationValues.AnnotationValueDecodeFail;
import lombok.core.configuration.ConfigurationKeysLoader;
=======
>>>>>>> 17972d59fa7e2eec6b73ba5da8234f5fa7ac2536
import lombok.javac.handlers.JavacHandlerUtil;
```

```
import lombok.core.TypeResolver;
import lombok.core.configuration.ConfigurationKeysLoader;
import lombok.javac.handlers.JavacHandlerUtil;
```

lombok/src/core/lombok/javac/handlers/JavacHandlerUtil.java

Chunk42: (combination/import declaration)

```
import lombok.core.AnnotationValues.AnnotationValue;
<<<<<<< HEAD
import lombok.core.handlers.HandlerUtil;
=======
import lombok.core.ReferenceFieldAugment;
import lombok.core.TransformationsUtil;
>>>>>>> 17972d59fa7e2eec6b73ba5da8234f5fa7ac2536
import lombok.core.TypeResolver;
```

```
import lombok.core.AnnotationValues.AnnotationValue;
import lombok.core.ReferenceFieldAugment;
import lombok.core.TypeResolver;
```

Version: f956ba1e337699206052a016da65f4f02ac6825b

Parents:

e5574133363c8b718329e07a73bf161416485da5

fbab1ca77cb8306843e26c5bad91186b34563282

Merge base:

7d51842ca381c491d5dfb44bc76b0cea345e7170

Chunk 5: (new code/method invocation, variable)

```
                        ClassLiteralAccess loggingType = selfType(owner, source);

<<<<<<< HEAD
                        FieldDeclaration fieldDeclaration = createField(framework, source,
loggingType, logFieldName, useStatic);
=======
                        FieldDeclaration fieldDeclaration = createField(framework, source,
loggingType, loggerCategory);
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
                        fieldDeclaration.traverse(new SetGeneratedByVisitor(source),
typeDecl.staticInitializerScope);
```

```
                        ClassLiteralAccess loggingType = selfType(owner, source);

                        FieldDeclaration fieldDeclaration = createField(framework, source,
loggingType, logFieldName, useStatic, loggerTopic);
                        fieldDeclaration.traverse(new SetGeneratedByVisitor(source),
typeDecl.staticInitializerScope);
```

Chunk 6: (new code/method signature)

```
        }

<<<<<<< HEAD
        private static FieldDeclaration createField(LoggingFramework framework, Annotation
source, ClassLiteralAccess loggingType, String logFieldName, boolean useStatic) {
=======
        public static FieldDeclaration createField(LoggingFramework framework, Annotation
source, ClassLiteralAccess loggingType, String loggerCategory) {
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
                int pS = source.sourceStart, pE = source.sourceEnd;
```

```
                        ClassLiteralAccess loggingType = selfType(owner, source);

                        FieldDeclaration fieldDeclaration = createField(framework, source,
loggingType, logFieldName, useStatic, loggerTopic);
                        fieldDeclaration.traverse(new SetGeneratedByVisitor(source),
typeDecl.staticInitializerScope);
```

Chunk 7: (combination/method invocation)

```
                @Override public void
handle(AnnotationValues<lombok.extern.apachecommons.CommonsLog> annotation, Annotation
source, EclipseNode annotationNode) {
<<<<<<< HEAD
```

```
                        handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_COMMONS_FLAG_USAGE,                          "@apachecommons.CommonsLog",
ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any @Log");

                        processAnnotation(LoggingFramework.COMMONS,     annotation,     source,
annotationNode);
=======
                        processAnnotation(LoggingFramework.COMMONS,     annotation,     source,
annotationNode, annotation.getInstance().topic());
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
                }
```

```
                @Override                                   public                                  void
handle(AnnotationValues<lombok.extern.apachecommons.CommonsLog>     annotation,     Annotation
source, EclipseNode annotationNode) {
                        handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_COMMONS_FLAG_USAGE,                          "@apachecommons.CommonsLog",
ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any @Log");
                        processAnnotation(LoggingFramework.COMMONS,     annotation,     source,
annotationNode, annotation.getInstance().topic());
                }
```

chunk 8:(combination/method invocation)

```
                @Override    public    void    handle(AnnotationValues<lombok.extern.java.Log>
annotation, Annotation source, EclipseNode annotationNode) {
<<<<<<< HEAD
                        handleFlagUsage(annotationNode,  ConfigurationKeys.LOG_JUL_FLAG_USAGE,
"@java.Log", ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any @Log");

                        processAnnotation(LoggingFramework.JUL,      annotation,      source,
annotationNode);
=======
                        processAnnotation(LoggingFramework.JUL,      annotation,      source,
annotationNode, annotation.getInstance().topic());
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
                }
```

```
                @Override    public    void    handle(AnnotationValues<lombok.extern.java.Log>
annotation, Annotation source, EclipseNode annotationNode) {
                        handleFlagUsage(annotationNode,  ConfigurationKeys.LOG_JUL_FLAG_USAGE,
"@java.Log", ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any @Log");
                        processAnnotation(LoggingFramework.JUL,      annotation,      source,
annotationNode, annotation.getInstance().topic());
                }
```

src/core/lombok/extern/apachecommons/CommonsLog.java

Chunk 9: (version 2/ annotation element, commentary)

```
public @interface CommonsLog {
<<<<<<< HEAD
}
=======
        /**
         * Sets the category of the constructed Logger. By default, it will use the type
where the annotation is placed.
         */
        String topic() default "";
}
```

```
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
```

```
public @interface CommonsLog {
	/**
	 * Sets the category of the constructed Logger. By default, it will use the type
where the annotation is placed.
	 */
	String topic() default "";
}
```

src/core/lombok/extern/java/Log.java

Chunk 10: (version 2/ annotation element, commentary)

```
 public @interface Log {
<<<<<<< HEAD
}
=======
	/**
	 * Sets the category of the constructed Logger. By default, it will use the type
where the annotation is placed.
	 */
	String topic() default "";
}
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
```

```
public @interface Log {
	/**
	 * Sets the category of the constructed Logger. By default, it will use the type
where the annotation is placed.
	 */
	String topic() default "";
}
```

src/core/lombok/extern/log4j/Log4j.java

Chunk 11: (version 2/ annotation element, commentary)

```
public @interface Log4j {
<<<<<<< HEAD
}
=======
	/**
	 * Sets the category of the constructed Logger. By default, it will use the type
where the annotation is placed.
	 */
	String topic() default "";
}
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
```

```
public @interface Log4j {
	/**
	 * Sets the category of the constructed Logger. By default, it will use the type
where the annotation is placed.
	 */
	String topic() default "";
}
```

lombok/src/core/lombok/extern/log4j/Log4j2.java

Chunk 12: (version 2/ annotation element, commentary)

```
public @interface Log4j2 {
<<<<<<< HEAD
}
=======
	/**
	 * Sets the category of the constructed Logger. By default, it will use the type
where the annotation is placed.
	 */
	String topic() default "";
}
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
```

```
public @interface Log4j2 {
	/**
	 * Sets the category of the constructed Logger. By default, it will use the type
where the annotation is placed.
	 */
	String topic() default "";
}
```

lombok/src/core/lombok/javac/handlers/HandleLog.java

chunk 13: (version 1/ if statement)

```
			}
<<<<<<< HEAD

			if		(fieldExists(logFieldName,		typeNode)!=
MemberExistsResult.NOT_EXISTS) {
				annotationNode.addWarning("Field	'"	+	logFieldName	+	"'
already exists.");
=======

			if (fieldExists("log", typeNode)!= MemberExistsResult.NOT_EXISTS) {
				annotationNode.addWarning("Field 'log' already exists.");
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
				return;
```

```
			}
			if		(fieldExists(logFieldName,		typeNode)		!=
MemberExistsResult.NOT_EXISTS) {
				annotationNode.addWarning("Field	'"	+	logFieldName	+	"'
already exists.");
				return;
```

Chunk 14: (new code/method incocation)

```
			JCFieldAccess loggingType = selfType(typeNode);
<<<<<<< HEAD
			createField(framework,	typeNode,	loggingType,	annotationNode.get(),
logFieldName, useStatic);
=======
			createField(framework,	typeNode,	loggingType,	annotationNode.get(),
loggerCategory);
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
			break;
```

```
                        JCFieldAccess loggingType = selfType(typeNode);
                        createField(framework, typeNode, loggingType, annotationNode.get(),
logFieldName, useStatic, loggerTopic);
                        break;
```

Chunk 15: (new code/method incocation)

```
}

<<<<<<< HEAD
        private static boolean createField(LoggingFramework framework, JavacNode typeNode,
JCFieldAccess loggingType, JCTree source, String logFieldName, boolean useStatic) {
=======
        public static boolean createField(LoggingFramework framework, JavacNode typeNode,
JCFieldAccess loggingType, JCTree source, String loggerCategory) {
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
                JavacTreeMaker maker = typeNode.getTreeMaker();
```

```
        }

        private static boolean createField(LoggingFramework framework, JavacNode typeNode,
JCFieldAccess loggingType, JCTree source, String logFieldName, boolean useStatic, String
loggerTopic) {
                JavacTreeMaker maker = typeNode.getTreeMaker();
```

Chunk 16: (combination/method incocation)

```
                @Override                                public                                void
handle(AnnotationValues<lombok.extern.apachecommons.CommonsLog>   annotation,   JCAnnotation
ast, JavacNode annotationNode) {
<<<<<<< HEAD
                        handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_COMMONS_FLAG_USAGE,                        "@apachecommons.CommonsLog",
ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any @Log");

                        processAnnotation(LoggingFramework.COMMONS,               annotation,
annotationNode);
=======
                        processAnnotation(LoggingFramework.COMMONS,               annotation,
annotationNode, annotation.getInstance().topic());
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
                }
```

```
                @Override                                public                                void
handle(AnnotationValues<lombok.extern.apachecommons.CommonsLog>   annotation,   JCAnnotation
ast, JavacNode annotationNode) {
                        handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_COMMONS_FLAG_USAGE,                        "@apachecommons.CommonsLog",
ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any @Log");
                        processAnnotation(LoggingFramework.COMMONS,               annotation,
annotationNode, annotation.getInstance().topic());
                }
```

Chunk 17: (combination/method incocation)

```
                @Override    public    void    handle(AnnotationValues<lombok.extern.java.Log>
annotation, JCAnnotation ast, JavacNode annotationNode) {
<<<<<<< HEAD
```

```
                        handleFlagUsage(annotationNode, ConfigurationKeys.LOG_JUL_FLAG_USAGE,
"@java.Log", ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any @Log");

                        processAnnotation(LoggingFramework.JUL, annotation, annotationNode);
=======
                        processAnnotation(LoggingFramework.JUL, annotation, annotationNode,
annotation.getInstance().topic());
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
                }
```

```
                @Override public void handle(AnnotationValues<lombok.extern.java.Log>
annotation, JCAnnotation ast, JavacNode annotationNode) {
                        handleFlagUsage(annotationNode, ConfigurationKeys.LOG_JUL_FLAG_USAGE,
"@java.Log", ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any @Log");
                        processAnnotation(LoggingFramework.JUL, annotation, annotationNode,
annotation.getInstance().topic());
                }
```

Chunk 18: (combination/method incocation)

```
                @Override public void handle(AnnotationValues<lombok.extern.log4j.Log4j>
annotation, JCAnnotation ast, JavacNode annotationNode) {
<<<<<<< HEAD
                        handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_LOG4J_FLAG_USAGE, "@Log4j", ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any
@Log");

                        processAnnotation(LoggingFramework.LOG4J, annotation,
annotationNode);
=======
                        processAnnotation(LoggingFramework.LOG4J, annotation, annotationNode,
annotation.getInstance().topic());
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
                }
```

```
                @Override public void handle(AnnotationValues<lombok.extern.log4j.Log4j>
annotation, JCAnnotation ast, JavacNode annotationNode) {
                        handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_LOG4J_FLAG_USAGE, "@Log4j", ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any
@Log");
                        processAnnotation(LoggingFramework.LOG4J, annotation, annotationNode,
annotation.getInstance().topic());
                }
```

Chunk 19: (combination/method incocation)

```
                @Override public void handle(AnnotationValues<lombok.extern.log4j.Log4j2>
annotation, JCAnnotation ast, JavacNode annotationNode) {
<<<<<<< HEAD
                        handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_LOG4J2_FLAG_USAGE, "@Log4j2", ConfigurationKeys.LOG_ANY_FLAG_USAGE,
"any @Log");

                        processAnnotation(LoggingFramework.LOG4J2, annotation,
annotationNode);
=======
                        processAnnotation(LoggingFramework.LOG4J2, annotation,
annotationNode, annotation.getInstance().topic());
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
                }
```

```
		@Override public void handle(AnnotationValues<lombok.extern.log4j.Log4j2>
annotation, JCAnnotation ast, JavacNode annotationNode) {
			handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_LOG4J2_FLAG_USAGE, "@Log4j2", ConfigurationKeys.LOG_ANY_FLAG_USAGE,
"any @Log");
			processAnnotation(LoggingFramework.LOG4J2, annotation,
annotationNode, annotation.getInstance().topic());
		}
```

Chunk 20: (combination/method incocation)

```
		@Override public void handle(AnnotationValues<lombok.extern.slf4j.Slf4j>
annotation, JCAnnotation ast, JavacNode annotationNode) {
<<<<<<< HEAD
			handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_SLF4J_FLAG_USAGE, "@Slf4j", ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any
@Log");

			processAnnotation(LoggingFramework.SLF4J, annotation,
annotationNode);
=======
			processAnnotation(LoggingFramework.SLF4J, annotation, annotationNode,
annotation.getInstance().topic());
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
		}
```

```
		@Override public void handle(AnnotationValues<lombok.extern.slf4j.Slf4j>
annotation, JCAnnotation ast, JavacNode annotationNode) {
			handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_SLF4J_FLAG_USAGE, "@Slf4j", ConfigurationKeys.LOG_ANY_FLAG_USAGE, "any
@Log");
			processAnnotation(LoggingFramework.SLF4J, annotation, annotationNode,
annotation.getInstance().topic());
		}
```

Chunk 21: (combination/method incocation)

```
		@Override public void handle(AnnotationValues<lombok.extern.slf4j.XSlf4j>
annotation, JCAnnotation ast, JavacNode annotationNode) {
<<<<<<< HEAD
			handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_XSLF4J_FLAG_USAGE, "@XSlf4j", ConfigurationKeys.LOG_ANY_FLAG_USAGE,
"any @Log");

			processAnnotation(LoggingFramework.XSLF4J, annotation,
annotationNode);
=======
			processAnnotation(LoggingFramework.XSLF4J, annotation,
annotationNode, annotation.getInstance().topic());
>>>>>>> fbab1ca77cb8306843e26c5bad91186b34563282
		}
```

```
		@Override public void handle(AnnotationValues<lombok.extern.slf4j.XSlf4j>
annotation, JCAnnotation ast, JavacNode annotationNode) {
			handleFlagUsage(annotationNode,
ConfigurationKeys.LOG_XSLF4J_FLAG_USAGE, "@XSlf4j", ConfigurationKeys.LOG_ANY_FLAG_USAGE,
"any @Log");
			processAnnotation(LoggingFramework.XSLF4J, annotation,
annotationNode, annotation.getInstance().topic());
```

```
}
```

Version: 78b2d6919e35887940f9f11b6ae1731245739b83

Parents:

5deb185591904d275cb06eea85c0d739587fc738

83b7e77b0cce6cd5993b17f36164271accdd281c

Merge base:

deed98be16e5099af52d951fc611f86a82a42858

lombok/src/delombok/lombok/delombok/DelombokApp.java

Chunk 22: (combination/annotation, commentary, method invocation, variable)

```
                }

<<<<<<< HEAD
                @SuppressWarnings("resource")
                // The jar file is used for the lifetime of the classLoader, therefore the
lifetime of delombok.
                // Since we only read from it, not closing it should not be a problem.
                final JarFile toolsJarFile = new JarFile(toolsJar);
=======
                @SuppressWarnings({"resource", "all"}) final JarFile toolsJarFile = new
JarFile(toolsJar);
>>>>>>> 83b7e77b0cce6cd5993b17f36164271accdd281c

                ClassLoader loader = new ClassLoader() {
```

```
                }

                // The jar file is used for the lifetime of the classLoader, therefore the
lifetime of delombok.
                // Since we only read from it, not closing it should not be a problem.
                @SuppressWarnings({"resource", "all"}) final JarFile toolsJarFile = new
JarFile(toolsJar);

                ClassLoader loader = new ClassLoader() {
```

Version: 86a635876dd75c4f3a61593491fa2ce53f8444b8

Parents:

7ee868659f4ff3cb286b676d649e8c57e9248d87

72b55dccb18f38b8aefd0ac8e7c2e8bd2dd5c057

Merge base:

deed98be16e5099af52d951fc611f86a82a42858

lombok/src/core/lombok/core/Version.java

Chunk 23: (version 2/commentary, variable)

```
        private static final String VERSION = "0.12.1";
<<<<<<< HEAD
        private static final String RELEASE_NAME = "Edgy Guinea Pig";
//      private static final String RELEASE_NAME = "Angry Butterfly";
=======
        private static final String RELEASE_NAME = "Angry Butterfly";
>>>>>>> 72b55dccb18f38b8aefd0ac8e7c2e8bd2dd5c057

        private Version() {
```

```
        private static final String VERSION = "0.12.1";
        private static final String RELEASE_NAME = "Edgy Guinea Pig";

        private Version() {
```

lombok/src/core/lombok/javac/handlers/HandleConstructor.java

Chunk 24: (Version 1/method invocation, variable)

```
                        List<JCAnnotation>      nullables      =      findAnnotations(fieldNode,
TransformationsUtil.NULLABLE_PATTERN);
<<<<<<< HEAD
                        JCVariableDecl   param   =   maker.VarDef(maker.Modifiers(Flags.FINAL   |
Flags.PARAMETER, nonNulls.appendList(nullables)), field.name, field.vartype, null);
=======
                        JCVariableDecl   param   =   maker.VarDef(maker.Modifiers(Flags.FINAL,
nonNulls.appendList(nullables)), fieldName, field.vartype, null);
>>>>>>> 72b55dccb18f38b8aefd0ac8e7c2e8bd2dd5c057
                        params.append(param);
```

```
                        List<JCAnnotation>      nullables      =      findAnnotations(fieldNode,
TransformationsUtil.NULLABLE_PATTERN);
                        JCVariableDecl   param   =   maker.VarDef(maker.Modifiers(Flags.FINAL   |
Flags.PARAMETER, nonNulls.appendList(nullables)), fieldName, field.vartype, null);
                        params.append(param);
```

Chunk 25: (Version 1/method invocation, variable)

```
                        List<JCAnnotation>      nullables      =      findAnnotations(fieldNode,
TransformationsUtil.NULLABLE_PATTERN);
<<<<<<< HEAD
                        JCVariableDecl   param   =   maker.VarDef(maker.Modifiers(Flags.FINAL   |
Flags.PARAMETER, nonNulls.appendList(nullables)), field.name, pType, null);
=======
```

```
                        JCVariableDecl param = maker.VarDef(maker.Modifiers(Flags.FINAL,
nonNulls.appendList(nullables)), fieldName, pType, null);
>>>>>>> 72b55dccb18f38b8aefd0ac8e7c2e8bd2dd5c057
                        params.append(param);
```

```
                        List<JCAnnotation> nullables = findAnnotations(fieldNode,
TransformationsUtil.NULLABLE_PATTERN);
                        JCVariableDecl param = maker.VarDef(maker.Modifiers(Flags.FINAL |
Flags.PARAMETER, nonNulls.appendList(nullables)), fieldName, pType, null);
                        params.append(param);
```

Version: 45697b50816df79475a8bb69dc89ff68747fbfe6

Parents:

4c03e3d2209004310858978 78d4888bf530b31ec

deed98be16e5099af52d951fc611f86a82a42858

Merge base:

620616bf8a73ea78863a5507aff631799b3a7a2e

lombok/src/core/lombok/javac/handlers/JavacHandlerUtil.java

Chunk 26: (new code/ method invocation, return statement, variable)

```
                JCExpression npe = chainDots(variable, "java", "lang",
"NullPointerException");
<<<<<<< HEAD
                JCTree exception = maker.NewClass(null, List.<JCExpression>nil(), npe,
List.<JCExpression>of(maker.Literal(fieldName.toString())), null);
                JCStatement throwStatement = maker.Throw(exception);
                return maker.If(Javac.makeBinary(maker, CTC_EQUAL, maker.Ident(fieldName),
Javac.makeLiteral(maker, CTC_BOT, null)), throwStatement, null);
=======
                JCTree exception = treeMaker.NewClass(null, List.<JCExpression>nil(), npe,
List.<JCExpression>of(treeMaker.Literal(fieldName.toString())), null);
                JCStatement throwStatement = treeMaker.Throw(exception);
                JCBlock throwBlock = treeMaker.Block(0, List.of(throwStatement));
                return treeMaker.If(treeMaker.Binary(CTC_EQUAL, treeMaker.Ident(fieldName),
treeMaker.Literal(CTC_BOT, null)), throwBlock, null);
>>>>>>> deed98be16e5099af52d951fc611f86a82a42858
        }
```

```
                JCExpression npe = chainDots(variable, "java", "lang",
"NullPointerException");
                JCTree exception = maker.NewClass(null, List.<JCExpression>nil(), npe,
List.<JCExpression>of(maker.Literal(fieldName.toString())), null);
                JCStatement throwStatement = maker.Throw(exception);
                JCBlock throwBlock = maker.Block(0, List.of(throwStatement));
                return maker.If(Javac.makeBinary(maker, CTC_EQUAL, maker.Ident(fieldName),
Javac.makeLiteral(maker, CTC_BOT, null)), throwBlock, null);
        }
```

lombok/src/utils/lombok/javac/CommentCatcher.java

Chunk 27: (new code/if statement, method invocation)

```
                            Class<?> parserFactory =
Class.forName("lombok.javac.java6.CommentCollectingParserFactory");
<<<<<<< HEAD
                            parserFactory.getMethod("setInCompiler",JavaCompiler.class,
Context.class, Map.class).invoke(null, compiler, context, commentsMap);
                    } else if (JavaCompiler.version().startsWith("1.7") ||
JavaCompiler.version().startsWith("1.8")) {
                            Class<?> parserFactory =
Class.forName("lombok.javac.java7.CommentCollectingParserFactory");
                            parserFactory.getMethod("setInCompiler",JavaCompiler.class,
Context.class, Map.class).invoke(null, compiler, context, commentsMap);
                    } else {
                            throw new IllegalStateException("No comments parser for
compiler version " + JavaCompiler.version());
```

```
=======
                            parserFactory.getMethod("setInCompiler",   JavaCompiler.class,
Context.class, Map.class).invoke(null, compiler, context, commentsMap);
                    } else {
                            Class<?>                    parserFactory                    =
Class.forName("lombok.javac.java7.CommentCollectingParserFactory");
                            parserFactory.getMethod("setInCompiler",   JavaCompiler.class,
Context.class, Map.class).invoke(null, compiler, context, commentsMap);
>>>>>>> deed98be16e5099af52d951fc611f86a82a42858
                    }
```

```
                            parserFactory                                              =
Class.forName("lombok.javac.java6.CommentCollectingParserFactory");
                    } else {
                            parserFactory                                              =
Class.forName("lombok.javac.java7.CommentCollectingParserFactory");
                    }
                    parserFactory.getMethod("setInCompiler",          JavaCompiler.class,
Context.class, Map.class).invoke(null, compiler, context, commentsMap);
            }
```

lombok/src/utils/lombok/javac/Javac.java

Chunk 28: (new code/ import declaration)

```
package lombok.javac;

<<<<<<< HEAD
import java.lang.reflect.Method;
import java.util.Objects;
import java.util.regex.Pattern;

import com.sun.tools.javac.main.JavaCompiler;
import com.sun.tools.javac.tree.JCTree.JCBinary;
=======
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import com.sun.tools.javac.code.TypeTags;
import com.sun.tools.javac.main.JavaCompiler;
import com.sun.tools.javac.tree.JCTree;
import com.sun.tools.javac.tree.JCTree.JCClassDecl;
>>>>>>> deed98be16e5099af52d951fc611f86a82a42858
import com.sun.tools.javac.tree.JCTree.JCExpression;
```

```
package lombok.javac;

import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
import javax.lang.model.type.NoType;
import javax.lang.model.type.TypeKind;
import javax.lang.model.type.TypeVisitor;

import lombok.Lombok;

import com.sun.tools.javac.code.Type;
import com.sun.tools.javac.main.JavaCompiler;
import com.sun.tools.javac.tree.JCTree;
import com.sun.tools.javac.tree.JCTree.JCBinary;
import com.sun.tools.javac.tree.JCTree.JCClassDecl;
import com.sun.tools.javac.tree.JCTree.JCExpression;
```

Chunk 29: (combination/ import declaration)

```
import com.sun.tools.javac.tree.JCTree.JCLiteral;
<<<<<<< HEAD
import com.sun.tools.javac.tree.JCTree.JCPrimitiveTypeTree;
import com.sun.tools.javac.tree.JCTree.JCUnary;
import com.sun.tools.javac.tree.TreeMaker;
=======
import com.sun.tools.javac.tree.JCTree.JCModifiers;
import com.sun.tools.javac.tree.JCTree.JCTypeParameter;
import com.sun.tools.javac.tree.TreeMaker;
import com.sun.tools.javac.util.List;
import com.sun.tools.javac.util.Name;
>>>>>>> deed98be16e5099af52d951fc611f86a82a42858

/**
```

```
import com.sun.tools.javac.tree.JCTree.JCLiteral;
import com.sun.tools.javac.tree.JCTree.JCModifiers;
import com.sun.tools.javac.tree.JCTree.JCPrimitiveTypeTree;
import com.sun.tools.javac.tree.JCTree.JCTypeParameter;
import com.sun.tools.javac.tree.JCTree.JCUnary;
import com.sun.tools.javac.tree.TreeMaker;
import com.sun.tools.javac.util.List;
import com.sun.tools.javac.util.Name;

/**
```

Chunk 30: (new code/method declaration, static block, variable)

```
	}

<<<<<<< HEAD
	public static JCLiteral makeLiteral(TreeMaker maker, Object ctc, Object argument) {
		try {
			Method createLiteral;
			if (JavaCompiler.version().startsWith("1.8")) {
				createLiteral	=	TreeMaker.class.getMethod("Literal",
Class.forName("com.sun.tools.javac.code.TypeTag"), Object.class);
			} else {
				createLiteral	=	TreeMaker.class.getMethod("Literal",
Integer.TYPE, Object.class);
			}
			return (JCLiteral) createLiteral.invoke(maker, ctc, argument);
		} catch (NoSuchMethodException e) {
			throw new RuntimeException(e);
		} catch (IllegalAccessException e) {
			throw new RuntimeException(e);
```

```
                } catch (Exception e) {
                        if (e instanceof RuntimeException) throw (RuntimeException) e;
                        throw new RuntimeException(e);
                }
        }

        public static JCUnary makeUnary(TreeMaker maker, Object ctc, JCExpression argument)
{
                try {
                        Method createUnary;
                        if (JavaCompiler.version().startsWith("1.8")) {
                                createUnary        =        TreeMaker.class.getMethod("Unary",
Class.forName("com.sun.tools.javac.code.TypeTag"), JCExpression.class);
                        } else {
                                createUnary = TreeMaker.class.getMethod("Unary", Integer.TYPE,
JCExpression.class);
                        }
                        return (JCUnary) createUnary.invoke(maker, ctc, argument);
                } catch (NoSuchMethodException e) {
                        throw new RuntimeException(e);
                } catch (IllegalAccessException e) {
                        throw new RuntimeException(e);
                } catch (Exception e) {
                        if (e instanceof RuntimeException) throw (RuntimeException) e;
                        throw new RuntimeException(e);
                }
        }

        public  static  JCBinary  makeBinary(TreeMaker  maker,  Object  ctc,  JCExpression
rhsArgument, JCExpression lhsArgument) {
                try {
                        Method createUnary;
                        if (JavaCompiler.version().startsWith("1.8")) {
                                createUnary        =        TreeMaker.class.getMethod("Binary",
Class.forName("com.sun.tools.javac.code.TypeTag"), JCExpression.class, JCExpression.class);
                        } else {
                                createUnary        =        TreeMaker.class.getMethod("Binary",
Integer.TYPE, JCExpression.class, JCExpression.class);
                        }
                        return    (JCBinary)    createUnary.invoke(maker,    ctc,    rhsArgument,
lhsArgument);
                } catch (NoSuchMethodException e) {
                        throw new RuntimeException(e);
                } catch (IllegalAccessException e) {
                        throw new RuntimeException(e);
                } catch (Exception e) {
                        if (e instanceof RuntimeException) throw (RuntimeException) e;
                        throw new RuntimeException(e);
                }
        }


=======
        private static final Field JCTREE_TAG;
        private static final Method JCTREE_GETTAG;
        static {
                Field f = null;
                try {
                        f = JCTree.class.getDeclaredField("tag");
                } catch (NoSuchFieldException e) {}
                JCTREE_TAG = f;
```

```
                Method m = null;
                try {
                        m = JCTree.class.getDeclaredMethod("getTag");
                } catch (NoSuchMethodException e) {}
                JCTREE_GETTAG = m;
        }

        public static int getTag(JCTree node) {
                if (JCTREE_GETTAG != null) {
                        try {
                                return (Integer) JCTREE_GETTAG.invoke(node);
                        } catch (Exception e) {}
                }
                try {
                        return (Integer) JCTREE_TAG.get(node);
                } catch (Exception e) {
                        throw new IllegalStateException("Can't get node tag");
                }
        }

        private static Method method;

        public static JCClassDecl ClassDef(TreeMaker maker, JCModifiers mods, Name name,
List<JCTypeParameter> typarams, JCExpression extending, List<JCExpression> implementing,
List<JCTree> defs) {
                if (method == null) try {
                        method        =        TreeMaker.class.getDeclaredMethod("ClassDef",
JCModifiers.class, Name.class, List.class, JCExpression.class, List.class, List.class);
                } catch (NoSuchMethodException ignore) {}
                if (method == null) try {
                        method        =        TreeMaker.class.getDeclaredMethod("ClassDef",
JCModifiers.class, Name.class, List.class, JCTree.class, List.class, List.class);
                } catch (NoSuchMethodException ignore) {}

                if (method == null) throw new IllegalStateException("Lombok bug #20130617-
1310: ClassDef doesn't look like anything we thought it would look like.");
                if (!Modifier.isPublic(method.getModifiers()) && !method.isAccessible()) {
                        method.setAccessible(true);
                }

                try {
                        return  (JCClassDecl)  method.invoke(maker,  mods,  name,  typarams,
extending, implementing, defs);
                } catch (InvocationTargetException e) {
                        throw sneakyThrow(e.getCause());
                } catch (IllegalAccessException e) {
                        throw sneakyThrow(e.getCause());
                }
        }

        private static RuntimeException sneakyThrow(Throwable t) {
                if (t == null) throw new NullPointerException("t");
                Javac.<RuntimeException>sneakyThrow0(t);
                return null;
        }

        @SuppressWarnings("unchecked")
        private static <T extends Throwable> void sneakyThrow0(Throwable t) throws T {
                throw (T)t;
        }
>>>>>>> deed98be16e5099af52d951fc611f86a82a42858
}
```

```
	}

	public static JCLiteral makeLiteral(TreeMaker maker, Object ctc, Object argument) {
		try {
			return (JCLiteral) createLiteral.invoke(maker, ctc, argument);
		} catch (IllegalAccessException e) {
			throw Lombok.sneakyThrow(e);
		} catch (InvocationTargetException e) {
			throw Lombok.sneakyThrow(e.getCause());
		}
	}

	public static JCUnary makeUnary(TreeMaker maker, Object ctc, JCExpression argument)
{
		try {
			return (JCUnary) createUnary.invoke(maker, ctc, argument);
		} catch (IllegalAccessException e) {
			throw Lombok.sneakyThrow(e);
		} catch (InvocationTargetException e) {
			throw Lombok.sneakyThrow(e.getCause());
		}
	}

	public static JCBinary makeBinary(TreeMaker maker, Object ctc, JCExpression
lhsArgument, JCExpression rhsArgument) {
		try {
			return (JCBinary) createBinary.invoke(maker, ctc, lhsArgument,
rhsArgument);
		} catch (IllegalAccessException e) {
			throw Lombok.sneakyThrow(e);
		} catch (InvocationTargetException e) {
			throw Lombok.sneakyThrow(e.getCause());
		}
	}

	private static final Class<?> JC_VOID_TYPE, JC_NO_TYPE;

	static {
		Class<?> c = null;
		try {
			c = Class.forName("com.sun.tools.javac.code.Type$JCVoidType");
		} catch (Exception ignore) {}
		JC_VOID_TYPE = c;
		c = null;
		try {
			c = Class.forName("com.sun.tools.javac.code.Type$JCNoType");
		} catch (Exception ignore) {}
		JC_NO_TYPE = c;
	}

	public static Type createVoidType(TreeMaker maker, Object tag) {
		if (Javac.getJavaCompilerVersion() < 8) {
			return new JCNoType(((Integer) tag).intValue());
		} else {
			try {
				if (compareCTC(tag, CTC_VOID)) {
					return (Type) JC_VOID_TYPE.newInstance();
				} else {
					return (Type) JC_NO_TYPE.newInstance();
				}
```

```
                        } catch (IllegalAccessException e) {
                                throw Lombok.sneakyThrow(e);
                        } catch (InstantiationException e) {
                                throw Lombok.sneakyThrow(e);
                        }
                }
        }

        private static class JCNoType extends Type implements NoType {
                public JCNoType(int tag) {
                        super(tag, null);
                }

                @Override
                public TypeKind getKind() {
                        if (Javac.compareCTC(tag, CTC_VOID)) return TypeKind.VOID;
                        if (Javac.compareCTC(tag, CTC_NONE)) return TypeKind.NONE;
                        throw new AssertionError("Unexpected tag: " + tag);
                }

                @Override
                public <R, P> R accept(TypeVisitor<R, P> v, P p) {
                        return v.visitNoType(this, p);
                }
        }

        private      static      final      Field      JCTREE_TAG,      JCLITERAL_TYPETAG,
JCPRIMITIVETYPETREE_TYPETAG;
        private static final Method JCTREE_GETTAG;
        static {
                Field f = null;
                try {
                        f = JCTree.class.getDeclaredField("tag");
                } catch (NoSuchFieldException e) {}
                JCTREE_TAG = f;

                f = null;
                try {
                        f = JCLiteral.class.getDeclaredField("typetag");
                } catch (NoSuchFieldException e) {}
                JCLITERAL_TYPETAG = f;

                f = null;
                try {
                        f = JCPrimitiveTypeTree.class.getDeclaredField("typetag");
                } catch (NoSuchFieldException e) {}
                JCPRIMITIVETYPETREE_TYPETAG = f;

                Method m = null;
                try {
                        m = JCTree.class.getDeclaredMethod("getTag");
                } catch (NoSuchMethodException e) {}
                JCTREE_GETTAG = m;
        }

        public static Object getTag(JCTree node) {
                if (JCTREE_GETTAG != null) {
                        try {
                                return JCTREE_GETTAG.invoke(node);
                        } catch (Exception e) {}
                }
                try {
```

```
                        return JCTREE_TAG.get(node);
                } catch (Exception e) {
                        throw new IllegalStateException("Can't get node tag");
                }
        }

        public static Object getTypeTag(JCLiteral node) {
                try {
                        return JCLITERAL_TYPETAG.get(node);
                } catch (Exception e) {
                        throw new IllegalStateException("Can't get JCLiteral typetag");
                }
        }

        public static Object getTypeTag(JCPrimitiveTypeTree node) {
                try {
                        return JCPRIMITIVETYPETREE_TYPETAG.get(node);
                } catch (Exception e) {
                        throw   new   IllegalStateException("Can't   get   JCPrimitiveTypeTree
typetag");
                }
        }

        private static Method classDef;

        public static JCClassDecl ClassDef(TreeMaker maker, JCModifiers mods, Name name,
List<JCTypeParameter> typarams, JCExpression extending, List<JCExpression> implementing,
List<JCTree> defs) {
                if (classDef == null) try {
                        classDef        =        TreeMaker.class.getDeclaredMethod("ClassDef",
JCModifiers.class, Name.class, List.class, JCExpression.class, List.class, List.class);
                } catch (NoSuchMethodException ignore) {}
                if (classDef == null) try {
                        classDef        =        TreeMaker.class.getDeclaredMethod("ClassDef",
JCModifiers.class, Name.class, List.class, JCTree.class, List.class, List.class);
                } catch (NoSuchMethodException ignore) {}

                if (classDef == null) throw new IllegalStateException("Lombok bug #20130617-
1310: ClassDef doesn't look like anything we thought it would look like.");
                if (!Modifier.isPublic(classDef.getModifiers()) && !classDef.isAccessible())
{
                        classDef.setAccessible(true);
                }

                try {
                        return (JCClassDecl) classDef.invoke(maker, mods, name, typarams,
extending, implementing, defs);
                } catch (InvocationTargetException e) {
                        throw sneakyThrow(e.getCause());
                } catch (IllegalAccessException e) {
                        throw sneakyThrow(e.getCause());
                }
        }

        private static RuntimeException sneakyThrow(Throwable t) {
                if (t == null) throw new NullPointerException("t");
                Javac.<RuntimeException>sneakyThrow0(t);
                return null;
        }

        @SuppressWarnings("unchecked")
        private static <T extends Throwable> void sneakyThrow0(Throwable t) throws T {
```

```
                throw (T)t;
        }
}
```

Version: 87f763a94c87b03da269d110c44e7e750ddf5211

Parents:

eb4cbcd8bbd7bf7784aa229e9b6c5fe0670fa7a5

34055fcdff786c9b809ce1a08c1c9218968ebc7d

Merge base:

1865bd7309b9d1dc743f83ccdbd7204fb0939ecd

lombok/src/core/lombok/javac/handlers/JavacHandlerUtil.java

Chunk 31: (concatenation/import declaration)

```
import com.sun.tools.javac.tree.JCTree.JCStatement;
<<<<<<< HEAD
import com.sun.tools.javac.tree.JCTree.JCTypeParameter;
=======
import com.sun.tools.javac.tree.JCTree.JCTypeApply;
>>>>>>> 34055fcdff786c9b809ce1a08c1c9218968ebc7d
import com.sun.tools.javac.tree.JCTree.JCVariableDecl;
```

```
import com.sun.tools.javac.tree.JCTree.JCStatement;
import com.sun.tools.javac.tree.JCTree.JCTypeParameter;
import com.sun.tools.javac.tree.JCTree.JCTypeApply;
import com.sun.tools.javac.tree.JCTree.JCVariableDecl;
```

Version: 19466a5413d0c451b89d0d70a8ba8f5fe0fc98aa

Parents:

e98d226cfb9a4b76b12e38e8ac590fb6c6ebbacc

a264677ffcbd929acef5f6fde4915f4c3117b052

Merge base:

4689d2a9bf79f592690a71e7ad7d25cb38b2344b

lombok/src/core/lombok/javac/handlers/JavacHandlerUtil.java

Chunk 32: (version 2/import declaration)

```
import lombok.Getter;
<<<<<<< HEAD
import lombok.core.AnnotationValues;
import lombok.core.TransformationsUtil;
import lombok.core.TypeResolver;
=======
>>>>>>> a264677ffcbd929acef5f6fde4915f4c3117b052
import lombok.core.AST.Kind;
```

```
import lombok.Getter;
import lombok.core.AST.Kind;
```

Chunk 33: (new code/import declaration)

```
import lombok.core.AnnotationValues.AnnotationValue;
<<<<<<< HEAD
import lombok.experimental.Accessors;
=======
import lombok.core.TypeResolver;
>>>>>>> a264677ffcbd929acef5f6fde4915f4c3117b052
import lombok.javac.Javac;
```

```
import lombok.core.AnnotationValues.AnnotationValue;
import lombok.core.TransformationsUtil;
import lombok.core.TypeResolver;
import lombok.experimental.Accessors;
import lombok.javac.Javac;
```

Version: a514af4dcdd87cdae64e87b9d8a8d1a489a8e474

Parents:

aa5d3b8bb2cb2bf068f4b4728a9e765968c673d4

0c927175af39f2b8d66d25b735ee0e5249107286

Merge base:

6ca2a91d6bb7054328a845771af0a4e618002f14

lombok/src/core/lombok/eclipse/handlers/HandleGetter.java

Chunk 34: (combination/for statement, switch statement)

```
		int modifier = toEclipseModifier(level) | (field.modifiers &
ClassFileConstants.AccStatic);

<<<<<<< HEAD
		for (String altName : toAllGetterNames(fieldNode, isBoolean)) {
			switch (methodExists(altName, fieldNode, false)) {
=======
		for (String altName : TransformationsUtil.toAllGetterNames(fieldName,
isBoolean)) {
			switch (methodExists(altName, fieldNode, false, 0)) {
>>>>>>> 0c927175af39f2b8d66d25b735ee0e5249107286
			case EXISTS_BY_LOMBOK:
```

```
		int modifier = toEclipseModifier(level) | (field.modifiers &
ClassFileConstants.AccStatic);

		for (String altName : toAllGetterNames(fieldNode, isBoolean)) {
			switch (methodExists(altName, fieldNode, false, 0)) {
			case EXISTS_BY_LOMBOK:
```

lombok/src/core/lombok/eclipse/handlers/HandleSetter.java

Chunk 35: (combination/for statement, switch statement)

```
		int modifier = toEclipseModifier(level) | (field.modifiers &
ClassFileConstants.AccStatic);

<<<<<<< HEAD
		for (String altName : toAllSetterNames(fieldNode, isBoolean)) {
			switch (methodExists(altName, fieldNode, false)) {
=======
		for (String altName : TransformationsUtil.toAllSetterNames(new
String(field.name), isBoolean)) {
			switch (methodExists(altName, fieldNode, false, 1)) {
>>>>>>> 0c927175af39f2b8d66d25b735ee0e5249107286
			case EXISTS_BY_LOMBOK:
```

```
		int modifier = toEclipseModifier(level) | (field.modifiers &
ClassFileConstants.AccStatic);

		for (String altName : toAllSetterNames(fieldNode, isBoolean)) {
			switch (methodExists(altName, fieldNode, false, 1)) {
			case EXISTS_BY_LOMBOK:
```

lombok/src/core/lombok/javac/handlers/HandleGetter.java

Chunk 36: (combination/for statement, if statement, switch statement)

```
                String methodName = toGetterName(fieldNode);

<<<<<<< HEAD
                if (methodName == null) {
                        source.addWarning("Not generating getter for this field: It does not
fit your @Accessors prefix list.");
                        return;
                }

                for (String altName : toAllGetterNames(fieldNode)) {
                        switch (methodExists(altName, fieldNode, false)) {
=======
                for (String altName : toAllGetterNames(fieldDecl)) {
                        switch (methodExists(altName, fieldNode, false, 0)) {
>>>>>>> 0c927175af39f2b8d66d25b735ee0e5249107286
                        case EXISTS_BY_LOMBOK:
```

```
                String methodName = toGetterName(fieldNode);

                if (methodName == null) {
                        source.addWarning("Not generating getter for this field: It does not
fit your @Accessors prefix list.");
                        return;
                }

                for (String altName : toAllGetterNames(fieldNode)) {
                        switch (methodExists(altName, fieldNode, false, 0)) {
                        case EXISTS_BY_LOMBOK:
```

lombok/src/core/lombok/javac/handlers/HandleSetter.java

Chunk 37: (combination/for statement, if statement, switch statement)

```
                String methodName = toSetterName(fieldNode);

<<<<<<< HEAD
                if (methodName == null) {
                        source.addWarning("Not generating setter for this field: It does not
fit your @Accessors prefix list.");
                        return;
                }

                for (String altName : toAllSetterNames(fieldNode)) {
                        switch (methodExists(altName, fieldNode, false)) {
=======
                for (String altName : toAllSetterNames(fieldDecl)) {
                        switch (methodExists(altName, fieldNode, false, 1)) {
>>>>>>> 0c927175af39f2b8d66d25b735ee0e5249107286
                        case EXISTS_BY_LOMBOK:
```

```
                String methodName = toSetterName(fieldNode);

                if (methodName == null) {
                        source.addWarning("Not generating setter for this field: It does not
fit your @Accessors prefix list.");
                        return;
```

```
		}

		for (String altName : toAllSetterNames(fieldNode)) {
			switch (methodExists(altName, fieldNode, false, 1)) {
			case EXISTS_BY_LOMBOK:
```

Version: 3796efe82e73fe60a15c0fd1a827dd417dfbcb57

Parents:

302761816eb1e58c77cedb73040ed2967208d1fa

bf354e3b5ced16913726afc8247b1dd0321c9d62

Merge base:

6c9b3d54de988665b64a0114cac5c20059e4af2a

lombok/src/eclipseAgent/lombok/eclipse/agent/EclipsePatcher.java

Chunk 38: (version 2/ commentary, method invocation)

```
	private static void patchExtractInterface(ScriptManager sm) {
<<<<<<< HEAD
=======
		/* Fix sourceEnding for generated nodes to avoid null pointer */
		sm.addScript(ScriptBuilder.wrapMethodCall()
				.target(new
MethodTarget("org.eclipse.jdt.internal.compiler.SourceElementNotifier",
"notifySourceElementRequestor",                                                    "void",
"org.eclipse.jdt.internal.compiler.ast.AbstractMethodDeclaration",
"org.eclipse.jdt.internal.compiler.ast.TypeDeclaration",
"org.eclipse.jdt.internal.compiler.ast.ImportReference"))
				.methodToWrap(new
Hook("org.eclipse.jdt.internal.compiler.util.HashtableOfObjectToInt",     "get",     "int",
"java.lang.Object"))
				.wrapMethod(new        Hook("lombok.eclipse.agent.PatchFixes",
"getSourceEndFixed", "int", "int", "org.eclipse.jdt.internal.compiler.ast.ASTNode"))
				.requestExtra(StackRequest.PARAM1)
				.transplant().build());

		/* Make sure the generated source element is found instead of the annotation
*/
		sm.addScript(ScriptBuilder.wrapMethodCall()
		.target(new
MethodTarget("org.eclipse.jdt.internal.corext.refactoring.structure.ExtractInterfaceProcesso
r", "createMethodDeclaration", "void",

	"org.eclipse.jdt.internal.corext.refactoring.structure.CompilationUnitRewrite",
				"org.eclipse.jdt.core.dom.rewrite.ASTRewrite",
				"org.eclipse.jdt.core.dom.AbstractTypeDeclaration",
				"org.eclipse.jdt.core.dom.MethodDeclaration"
			))
			.methodToWrap(new   Hook("org.eclipse.jface.text.IDocument",   "get",
"java.lang.String", "int", "int"))
			.wrapMethod(new               Hook("lombok.eclipse.agent.PatchFixes",
"getRealMethodDeclarationSource",          "java.lang.String",          "java.lang.String",
"org.eclipse.jdt.core.dom.MethodDeclaration"))
			.requestExtra(StackRequest.PARAM4)
			.build());

>>>>>>> bf354e3b5ced16913726afc8247b1dd0321c9d62
		/* get real generated node in stead of a random one generated by the
annotation */
```

```
	private static void patchExtractInterface(ScriptManager sm) {
		/* Fix sourceEnding for generated nodes to avoid null pointer */
		sm.addScript(ScriptBuilder.wrapMethodCall()
```

```
                                .target(new
MethodTarget("org.eclipse.jdt.internal.compiler.SourceElementNotifier",
"notifySourceElementRequestor",                                                    "void",
"org.eclipse.jdt.internal.compiler.ast.AbstractMethodDeclaration",
"org.eclipse.jdt.internal.compiler.ast.TypeDeclaration",
"org.eclipse.jdt.internal.compiler.ast.ImportReference"))
                                .methodToWrap(new
Hook("org.eclipse.jdt.internal.compiler.util.HashtableOfObjectToInt",      "get",      "int",
"java.lang.Object"))
                                .wrapMethod(new        Hook("lombok.eclipse.agent.PatchFixes",
"getSourceEndFixed", "int", "int", "org.eclipse.jdt.internal.compiler.ast.ASTNode"))
                                .requestExtra(StackRequest.PARAM1)
                                .transplant().build());

                /* Make sure the generated source element is found instead of the annotation
*/
                sm.addScript(ScriptBuilder.wrapMethodCall()
                .target(new
MethodTarget("org.eclipse.jdt.internal.corext.refactoring.structure.ExtractInterfaceProcesso
r", "createMethodDeclaration", "void",

        "org.eclipse.jdt.internal.corext.refactoring.structure.CompilationUnitRewrite",
                                "org.eclipse.jdt.core.dom.rewrite.ASTRewrite",
                                "org.eclipse.jdt.core.dom.AbstractTypeDeclaration",
                                "org.eclipse.jdt.core.dom.MethodDeclaration"
                        ))
                        .methodToWrap(new    Hook("org.eclipse.jface.text.IDocument",    "get",
"java.lang.String", "int", "int"))
                        .wrapMethod(new                Hook("lombok.eclipse.agent.PatchFixes",
"getRealMethodDeclarationSource",           "java.lang.String",           "java.lang.String",
"org.eclipse.jdt.core.dom.MethodDeclaration"))
                        .requestExtra(StackRequest.PARAM4)
                        .build());

                /*  get  real  generated  node  in  stead  of  a  random  one  generated  by  the
annotation */
```

Chunk 39: (version 2/commentary)

```
                                .build());

<<<<<<< HEAD
                /* Do not add  @Override's for generated methods */
=======
                /* Do not add @Override's for generated methods */
>>>>>>> bf354e3b5ced16913726afc8247b1dd0321c9d62
                sm.addScript(ScriptBuilder.exitEarly()
```

```
                                .build());

                /* Do not add @Override's for generated methods */
                sm.addScript(ScriptBuilder.exitEarly()
```

Chunk 40: (version 2/ commentary, method invocation)

```
                                .build());

<<<<<<< HEAD
=======
                /* Do not add comments for generated methods */
                sm.addScript(ScriptBuilder.exitEarly()
```

```
                                .target(new
MethodTarget("org.eclipse.jdt.internal.corext.refactoring.structure.ExtractInterfaceProcesso
r", "createMethodComment"))
                                .decisionMethod(new    Hook("lombok.eclipse.agent.PatchFixes",
"isGenerated", "boolean", "org.eclipse.jdt.core.dom.ASTNode"))
                                .request(StackRequest.PARAM2)
                                .build());

>>>>>>> bf354e3b5ced16913726afc8247b1dd0321c9d62
        }
```

```
                                .build());

                /* Do not add comments for generated methods */
                sm.addScript(ScriptBuilder.exitEarly()
                                .target(new
MethodTarget("org.eclipse.jdt.internal.corext.refactoring.structure.ExtractInterfaceProcesso
r", "createMethodComment"))
                                .decisionMethod(new    Hook("lombok.eclipse.agent.PatchFixes",
"isGenerated", "boolean", "org.eclipse.jdt.core.dom.ASTNode"))
                                .request(StackRequest.PARAM2)
                                .build());
```

lombok/src/eclipseAgent/lombok/eclipse/agent/PatchFixes.java

chunk 41: (version 2/import declaration)

```
import org.eclipse.jdt.core.IMethod;
<<<<<<< HEAD
import org.eclipse.jdt.core.JavaModelException;
import org.eclipse.jdt.core.dom.AbstractTypeDeclaration;
=======
import org.eclipse.jdt.core.IType;
import org.eclipse.jdt.core.JavaModelException;
>>>>>>> bf354e3b5ced16913726afc8247b1dd0321c9d62
import org.eclipse.jdt.core.dom.MethodDeclaration;
```

```
import org.eclipse.jdt.core.IMethod;
import org.eclipse.jdt.core.IType;
import org.eclipse.jdt.core.JavaModelException;
import org.eclipse.jdt.core.dom.MethodDeclaration;
```

Chunk 42: (version 2/commentary, for statement, if statement, method signature, variable, while statement)

```
        }


<<<<<<< HEAD
//
        lombok.eclipse.agent.PatchFixes.getRealMethodDeclarationNode(Lorg/eclipse/jdt/core/I
Method;Lorg/eclipse/jdt/core/dom/CompilationUnit;)Lorg/eclipse/jdt/core/dom/MethodDeclaratio
n;
        public                static                org.eclipse.jdt.core.dom.MethodDeclaration
getRealMethodDeclarationNode(org.eclipse.jdt.core.IMethod                        sourceMethod,
org.eclipse.jdt.core.dom.CompilationUnit cuUnit) throws JavaModelException {
                MethodDeclaration                    methodDeclarationNode                    =
ASTNodeSearchUtil.getMethodDeclarationNode(sourceMethod, cuUnit);
                if (isGenerated(methodDeclarationNode)) {
                        String typeName = sourceMethod.getTypeRoot().getElementName();
```

```
                        String methodName = sourceMethod.getElementName();
                        for (Object type : cuUnit.types()) {
                                org.eclipse.jdt.core.dom.AbstractTypeDeclaration
typeDeclaration = (AbstractTypeDeclaration)type;
                                if ((typeDeclaration.getName()+".java").equals(typeName)) {
                                        for           (Object           declaration           :
typeDeclaration.bodyDeclarations()) {
                                                if            (declaration            instanceof
org.eclipse.jdt.core.dom.MethodDeclaration) {

        org.eclipse.jdt.core.dom.MethodDeclaration             methodDeclaration             =
(org.eclipse.jdt.core.dom.MethodDeclaration) declaration;
                                                        if
(methodDeclaration.getName().toString().equals(methodName)) {
                                                                return methodDeclaration;
                                                        }
=======
        public                static                org.eclipse.jdt.core.dom.MethodDeclaration
getRealMethodDeclarationNode(org.eclipse.jdt.core.IMethod                        sourceMethod,
org.eclipse.jdt.core.dom.CompilationUnit cuUnit) throws JavaModelException {
                MethodDeclaration                    methodDeclarationNode                    =
ASTNodeSearchUtil.getMethodDeclarationNode(sourceMethod, cuUnit);
                if (isGenerated(methodDeclarationNode)) {
                        IType declaringType = sourceMethod.getDeclaringType();
                        Stack<IType> typeStack = new Stack<IType>();
                        while (declaringType != null) {
                                typeStack.push(declaringType);
                                declaringType = declaringType.getDeclaringType();
                        }

                        IType rootType = typeStack.pop();
                        org.eclipse.jdt.core.dom.AbstractTypeDeclaration   typeDeclaration   =
findTypeDeclaration(rootType, cuUnit.types());
                        while (!typeStack.isEmpty() && typeDeclaration != null) {
                                typeDeclaration     =     findTypeDeclaration(typeStack.pop(),
typeDeclaration.bodyDeclarations());
                        }

                        if (typeStack.isEmpty() && typeDeclaration != null) {
                                String methodName = sourceMethod.getElementName();
                                for (Object declaration : typeDeclaration.bodyDeclarations())
{
                                        if                (declaration                instanceof
org.eclipse.jdt.core.dom.MethodDeclaration) {
                                                org.eclipse.jdt.core.dom.MethodDeclaration
methodDeclaration = (org.eclipse.jdt.core.dom.MethodDeclaration) declaration;
                                                if
(methodDeclaration.getName().toString().equals(methodName)) {
                                                        return methodDeclaration;
>>>>>>> bf354e3b5ced16913726afc8247b1dd0321c9d62
                                                }
```

```
        }


        public                static                org.eclipse.jdt.core.dom.MethodDeclaration
getRealMethodDeclarationNode(org.eclipse.jdt.core.IMethod                        sourceMethod,
org.eclipse.jdt.core.dom.CompilationUnit cuUnit) throws JavaModelException {
                MethodDeclaration                    methodDeclarationNode                    =
ASTNodeSearchUtil.getMethodDeclarationNode(sourceMethod, cuUnit);
                if (isGenerated(methodDeclarationNode)) {
```

```
                        IType declaringType = sourceMethod.getDeclaringType();
                        Stack<IType> typeStack = new Stack<IType>();
                        while (declaringType != null) {
                                typeStack.push(declaringType);
                                declaringType = declaringType.getDeclaringType();
                        }

                        IType rootType = typeStack.pop();
                        org.eclipse.jdt.core.dom.AbstractTypeDeclaration   typeDeclaration   =
findTypeDeclaration(rootType, cuUnit.types());
                        while (!typeStack.isEmpty() && typeDeclaration != null) {
                                typeDeclaration     =     findTypeDeclaration(typeStack.pop(),
typeDeclaration.bodyDeclarations());
                        }

                        if (typeStack.isEmpty() && typeDeclaration != null) {
                                String methodName = sourceMethod.getElementName();
                                for (Object declaration : typeDeclaration.bodyDeclarations())
{
                                        if               (declaration               instanceof
org.eclipse.jdt.core.dom.MethodDeclaration) {
                                                org.eclipse.jdt.core.dom.MethodDeclaration
methodDeclaration = (org.eclipse.jdt.core.dom.MethodDeclaration) declaration;
                                                if
(methodDeclaration.getName().toString().equals(methodName)) {
                                                        return methodDeclaration;
                                                }
```

Chunk 43: (version 2/method declaration)

```
        }

<<<<<<< HEAD
=======
        private            static            org.eclipse.jdt.core.dom.AbstractTypeDeclaration
findTypeDeclaration(IType searchType, List<?> nodes) {
                for (Object object : nodes) {
                        if                            (object                            instanceof
org.eclipse.jdt.core.dom.AbstractTypeDeclaration) {
                                org.eclipse.jdt.core.dom.AbstractTypeDeclaration
typeDeclaration = (org.eclipse.jdt.core.dom.AbstractTypeDeclaration) object;
                                if
(typeDeclaration.getName().toString().equals(searchType.getElementName()))
                                        return typeDeclaration;
                        }
                }
                return null;
        }

>>>>>>> bf354e3b5ced16913726afc8247b1dd0321c9d62
        public          static          int          getSourceEndFixed(int          sourceEnd,
org.eclipse.jdt.internal.compiler.ast.ASTNode node) throws Exception {
```

```
        }

        private            static            org.eclipse.jdt.core.dom.AbstractTypeDeclaration
findTypeDeclaration(IType searchType, List<?> nodes) {
                for (Object object : nodes) {
                        if                            (object                            instanceof
org.eclipse.jdt.core.dom.AbstractTypeDeclaration) {
```

```
                                org.eclipse.jdt.core.dom.AbstractTypeDeclaration
typeDeclaration = (org.eclipse.jdt.core.dom.AbstractTypeDeclaration) object;
                                if
(typeDeclaration.getName().toString().equals(searchType.getElementName()))
                                        return typeDeclaration;
                        }
                }
                return null;
        }

        public          static          int          getSourceEndFixed(int          sourceEnd,
org.eclipse.jdt.internal.compiler.ast.ASTNode node) throws Exception {
```

Version: a5c7d134c168f6f9e9ab6203cb54b1030057c790

Parents:

aaf3101393d4f87ea8e256ba35a5b5374e6a0161

4e831b05ec08399795d27c343b6324b5b6de3443

Merge base:

ef820d8d5ab76c6db8335201da3c7ab9de7cb56a

lombok/src/eclipseAgent/lombok/eclipse/agent/PatchFixes.java

Chunk 44: (version 1/commentary, method declaration)

```
        }

<<<<<<< HEAD
        /* Very practical implementation, but works for getter and setter even with type
parameters */
        public   static   java.lang.String   getRealMethodDeclarationSource(java.lang.String
original, org.eclipse.jdt.core.dom.MethodDeclaration declaration) {
                if(isGenerated(declaration)) {
                        String returnType = declaration.getReturnType2().toString();
                        String params = "";
                        for (Object object : declaration.parameters()) {
                                org.eclipse.jdt.core.dom.ASTNode          parameter          =
((org.eclipse.jdt.core.dom.ASTNode)object);
                                params += ","+parameter.toString();
                        }
                        return                 returnType                 +                 "
"+declaration.getName().getFullyQualifiedName()+"("+(params.isEmpty()       ?       ""       :
params.substring(1))+");";
                }
                return original;
        }

        public          static          int          getSourceEndFixed(int          sourceEnd,
org.eclipse.jdt.internal.compiler.ast.ASTNode node) throws Exception {
                if (sourceEnd == -1) {
                        org.eclipse.jdt.internal.compiler.ast.ASTNode         object         =
(org.eclipse.jdt.internal.compiler.ast.ASTNode)node.getClass().getField("$generatedBy").get(
node);
                        if (object != null) {
                                return object.sourceEnd;
                        }
                }
                return sourceEnd;
        }

=======
>>>>>>> 4e831b05ec08399795d27c343b6324b5b6de3443
        public static int fixRetrieveStartingCatchPosition(int original, int start) {
```

```
        }

        /* Very practical implementation, but works for getter and setter even with type
parameters */
        public   static   java.lang.String   getRealMethodDeclarationSource(java.lang.String
original, org.eclipse.jdt.core.dom.MethodDeclaration declaration) {
                if(isGenerated(declaration)) {
```

```
                        String returnType = declaration.getReturnType2().toString();
                        String params = "";
                        for (Object object : declaration.parameters()) {
                                org.eclipse.jdt.core.dom.ASTNode          parameter          =
((org.eclipse.jdt.core.dom.ASTNode)object);
                                params += ","+parameter.toString();
                        }
                        return                  returnType                  +                  "
"+declaration.getName().getFullyQualifiedName()+"("+(params.isEmpty()       ?       ""       :
params.substring(1))+");";
                }
                return original;
        }

        public          static          int          getSourceEndFixed(int          sourceEnd,
org.eclipse.jdt.internal.compiler.ast.ASTNode node) throws Exception {
                if (sourceEnd == -1) {
                        org.eclipse.jdt.internal.compiler.ast.ASTNode         object         =
(org.eclipse.jdt.internal.compiler.ast.ASTNode)node.getClass().getField("$generatedBy").get(
node);
                        if (object != null) {
                                return object.sourceEnd;
                        }
                }
                return sourceEnd;
        }

        public static int fixRetrieveStartingCatchPosition(int original, int start) {
```

Version: aaf3101393d4f87ea8e256ba35a5b5374e6a0161

Parents:

6d6a191c67827b626c67ddfbce071c17be58723b

53ce4f61788ab62263d9e267b947303973d11a7f

Merge base:

ef820d8d5ab76c6db8335201da3c7ab9de7cb56a

lombok/src/eclipseAgent/lombok/eclipse/agent/PatchFixes.java

Chunk 45: (version 2/ commentary, method declaration)

```
	}

<<<<<<< HEAD
=======
	/* Very practical implementation, but works for getter and setter even with type parameters */
	public static java.lang.String getRealMethodDeclarationSource(java.lang.String original, org.eclipse.jdt.core.dom.MethodDeclaration declaration) {
		if(isGenerated(declaration)) {
			String returnType = declaration.getReturnType2().toString();
			String params = "";
			for (Object object : declaration.parameters()) {
				org.eclipse.jdt.core.dom.ASTNode parameter = ((org.eclipse.jdt.core.dom.ASTNode)object);
				params += ","+parameter.toString();
			}
			return returnType + " "+declaration.getName().getFullyQualifiedName()+"("+(params.isEmpty() ? "" : params.substring(1))+");";
		}
		return original;
	}

	public static int getSourceEndFixed(int sourceEnd, org.eclipse.jdt.internal.compiler.ast.ASTNode node) throws Exception {
		if (sourceEnd == -1) {
			org.eclipse.jdt.internal.compiler.ast.ASTNode object = (org.eclipse.jdt.internal.compiler.ast.ASTNode)node.getClass().getField("$generatedBy").get(node);
			if (object != null) {
				return object.sourceEnd;
			}
		}
		return sourceEnd;
	}

>>>>>>> 53ce4f61788ab62263d9e267b947303973d11a7f
	public static int fixRetrieveStartingCatchPosition(int original, int start) {
```

```
	}

	/* Very practical implementation, but works for getter and setter even with type parameters */
	public static java.lang.String getRealMethodDeclarationSource(java.lang.String original, org.eclipse.jdt.core.dom.MethodDeclaration declaration) {
		if(isGenerated(declaration)) {
```

```
                        String returnType = declaration.getReturnType2().toString();
                        String params = "";
                        for (Object object : declaration.parameters()) {
                                org.eclipse.jdt.core.dom.ASTNode          parameter          =
((org.eclipse.jdt.core.dom.ASTNode)object);
                                params += ","+parameter.toString();
                        }
                        return                  returnType                  +                 "
"+declaration.getName().getFullyQualifiedName()+"("+(params.isEmpty()       ?       ""       :
params.substring(1))+");";
                }
                return original;
        }

        public          static          int          getSourceEndFixed(int          sourceEnd,
org.eclipse.jdt.internal.compiler.ast.ASTNode node) throws Exception {
                if (sourceEnd == -1) {
                        org.eclipse.jdt.internal.compiler.ast.ASTNode         object         =
(org.eclipse.jdt.internal.compiler.ast.ASTNode)node.getClass().getField("$generatedBy").get(
node);
                        if (object != null) {
                                return object.sourceEnd;
                        }
                }
                return sourceEnd;
        }

        public static int fixRetrieveStartingCatchPosition(int original, int start) {
```

Version: dc92425f85d2f2dd187b688ff6d218d3c8e657b6

Parents:

1cdd42ac10c128765d3ff642d808c00eab6a1782

5cc928f471f3875f141ab1ee737cfe2613e9cdd6

Merge base:

9433db4ecdf1a525541581a73161ababee0c352c

lombok/src/eclipseAgent/lombok/eclipse/agent/EclipsePatcher.java

Chunk 46: (version 1/method declaration)

```
        }
<<<<<<< HEAD

        private static void patchDisableLombokForCodeFormatterAndCleanup(ScriptManager sm) {
                sm.addScript(ScriptBuilder.setSymbolDuringMethodCall()
                                .target(new
MethodTarget("org.eclipse.jdt.internal.formatter.DefaultCodeFormatter",
"formatCompilationUnit"))
                                .callToWrap(new
Hook("org.eclipse.jdt.internal.core.util.CodeSnippetParsingUtil",    "parseCompilationUnit",
"org.eclipse.jdt.internal.compiler.ast.CompilationUnitDeclaration",                "char[]",
"java.util.Map", "boolean"))
                                .symbol("lombok.disable")
                                .build());

                sm.addScript(ScriptBuilder.exitEarly()
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.DoStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.EnhancedForStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.ForStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.IfStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.WhileStatement"))
                        .decisionMethod(new           Hook("lombok.eclipse.agent.PatchFixes",
"isGenerated", "boolean", "org.eclipse.jdt.core.dom.Statement"))
                        .request(StackRequest.PARAM1)
                        .valueMethod(new              Hook("lombok.eclipse.agent.PatchFixes",
"isGenerated", "boolean", "org.eclipse.jdt.core.dom.Statement"))
                        .build());
        }

        private static void patchListRewriteHandleGeneratedMethods(ScriptManager sm) {
                sm.addScript(ScriptBuilder.replaceMethodCall()
                                .target(new
MethodTarget("org.eclipse.jdt.internal.core.dom.rewrite.ASTRewriteAnalyzer$ListRewriter",
"rewriteList"))
                                .methodToReplace(new
Hook("org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent",                "getChildren",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent[]"))
```

```
                                .replacementMethod(new Hook("lombok.eclipse.agent.PatchFixes",
"listRewriteHandleGeneratedMethods",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent[]",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent"))
                                .build());
        }

=======

>>>>>>> 5cc928f471f3875f141ab1ee737cfe2613e9cdd6
        private static void patchDomAstReparseIssues(ScriptManager sm) {
```

```
        }

        private static void patchDisableLombokForCodeFormatterAndCleanup(ScriptManager sm) {
                sm.addScript(ScriptBuilder.setSymbolDuringMethodCall()
                                .target(new
MethodTarget("org.eclipse.jdt.internal.formatter.DefaultCodeFormatter",
"formatCompilationUnit"))
                                .callToWrap(new
Hook("org.eclipse.jdt.internal.core.util.CodeSnippetParsingUtil",    "parseCompilationUnit",
"org.eclipse.jdt.internal.compiler.ast.CompilationUnitDeclaration",                "char[]",
"java.util.Map", "boolean"))
                                .symbol("lombok.disable")
                                .build());

                sm.addScript(ScriptBuilder.exitEarly()
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.DoStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.EnhancedForStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.ForStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.IfStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.WhileStatement"))
                        .decisionMethod(new            Hook("lombok.eclipse.agent.PatchFixes",
"isGenerated", "boolean", "org.eclipse.jdt.core.dom.Statement"))
                        .request(StackRequest.PARAM1)
                        .valueMethod(new               Hook("lombok.eclipse.agent.PatchFixes",
"isGenerated", "boolean", "org.eclipse.jdt.core.dom.Statement"))
                        .build());
        }

        private static void patchListRewriteHandleGeneratedMethods(ScriptManager sm) {
                sm.addScript(ScriptBuilder.replaceMethodCall()
                                .target(new
MethodTarget("org.eclipse.jdt.internal.core.dom.rewrite.ASTRewriteAnalyzer$ListRewriter",
"rewriteList"))
                                .methodToReplace(new
Hook("org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent",                "getChildren",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent[]"))
                                .replacementMethod(new Hook("lombok.eclipse.agent.PatchFixes",
"listRewriteHandleGeneratedMethods",
```

```
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent[]",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent"))
                                .build());
        }

        private static void patchDomAstReparseIssues(ScriptManager sm) {
```

Version: 1cdd42ac10c128765d3ff642d808c00eab6a1782

Parents:

f3253a73c29c393bb572e05c992afa22b4de4748

b43cd3509311e25b64a559cd7dd02d11a45d9f0e

Merge base:

9433db4ecdf1a525541581a73161ababee0c352c

lombok/src/eclipseAgent/lombok/eclipse/agent/EclipsePatcher.java

Chunk 47: (concatenation/method invocation)

```
                        patchFixSourceTypeConverter(sm);
<<<<<<< HEAD
                        patchDisableLombokForCodeFormatterAndCleanup(sm);
=======
                        patchListRewriteHandleGeneratedMethods(sm);
>>>>>>> b43cd3509311e25b64a559cd7dd02d11a45d9f0e
                } else {
```

```
                        patchFixSourceTypeConverter(sm);
                        patchDisableLombokForCodeFormatterAndCleanup(sm);
                        patchListRewriteHandleGeneratedMethods(sm);
                } else {
```

Chunk 48: (concatenation/method declaration)

```
        }

<<<<<<< HEAD
        private static void patchDisableLombokForCodeFormatterAndCleanup(ScriptManager sm) {
                sm.addScript(ScriptBuilder.setSymbolDuringMethodCall()
                                .target(new
MethodTarget("org.eclipse.jdt.internal.formatter.DefaultCodeFormatter",
"formatCompilationUnit"))
                                .callToWrap(new
Hook("org.eclipse.jdt.internal.core.util.CodeSnippetParsingUtil",    "parseCompilationUnit",
"org.eclipse.jdt.internal.compiler.ast.CompilationUnitDeclaration",                "char[]",
"java.util.Map", "boolean"))
                                .symbol("lombok.disable")
                                .build());

                sm.addScript(ScriptBuilder.exitEarly()
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.DoStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.EnhancedForStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.ForStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.IfStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.WhileStatement"))
```

```
                        .decisionMethod(new            Hook("lombok.eclipse.agent.PatchFixes",
"isGenerated", "boolean", "org.eclipse.jdt.core.dom.Statement"))
                        .request(StackRequest.PARAM1)
                        .valueMethod(new               Hook("lombok.eclipse.agent.PatchFixes",
"isGenerated", "boolean", "org.eclipse.jdt.core.dom.Statement"))
                        .build());
        }

=======
        private static void patchListRewriteHandleGeneratedMethods(ScriptManager sm) {
                sm.addScript(ScriptBuilder.replaceMethodCall()
                                .target(new
MethodTarget("org.eclipse.jdt.internal.core.dom.rewrite.ASTRewriteAnalyzer$ListRewriter",
"rewriteList"))
                                .methodToReplace(new
Hook("org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent",                "getChildren",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent[]"))
                                .replacementMethod(new Hook("lombok.eclipse.agent.PatchFixes",
"listRewriteHandleGeneratedMethods",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent[]",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent"))
                                .build());
        }
>>>>>>> b43cd3509311e25b64a559cd7dd02d11a45d9f0e

        private static void patchDomAstReparseIssues(ScriptManager sm) {
```

```
        }

        private static void patchDisableLombokForCodeFormatterAndCleanup(ScriptManager sm) {
                sm.addScript(ScriptBuilder.setSymbolDuringMethodCall()
                                .target(new
MethodTarget("org.eclipse.jdt.internal.formatter.DefaultCodeFormatter",
"formatCompilationUnit"))
                                .callToWrap(new
Hook("org.eclipse.jdt.internal.core.util.CodeSnippetParsingUtil",    "parseCompilationUnit",
"org.eclipse.jdt.internal.compiler.ast.CompilationUnitDeclaration",                "char[]",
"java.util.Map", "boolean"))
                                .symbol("lombok.disable")
                                .build());

                sm.addScript(ScriptBuilder.exitEarly()
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.DoStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.EnhancedForStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.ForStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.IfStatement"))
                        .target(new
MethodTarget("org.eclipse.jdt.internal.corext.fix.ControlStatementsFix$ControlStatementFinde
r", "visit", "boolean", "org.eclipse.jdt.core.dom.WhileStatement"))
                        .decisionMethod(new            Hook("lombok.eclipse.agent.PatchFixes",
"isGenerated", "boolean", "org.eclipse.jdt.core.dom.Statement"))
                        .request(StackRequest.PARAM1)
```

```
                        .valueMethod(new               Hook("lombok.eclipse.agent.PatchFixes",
"isGenerated", "boolean", "org.eclipse.jdt.core.dom.Statement"))
                        .build());
        }

        private static void patchListRewriteHandleGeneratedMethods(ScriptManager sm) {
                sm.addScript(ScriptBuilder.replaceMethodCall()
                                .target(new
MethodTarget("org.eclipse.jdt.internal.core.dom.rewrite.ASTRewriteAnalyzer$ListRewriter",
"rewriteList"))
                                .methodToReplace(new
Hook("org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent",                "getChildren",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent[]"))
                                .replacementMethod(new Hook("lombok.eclipse.agent.PatchFixes",
"listRewriteHandleGeneratedMethods",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent[]",
"org.eclipse.jdt.internal.core.dom.rewrite.RewriteEvent"))
                                .build());
        }

        private static void patchDomAstReparseIssues(ScriptManager sm) {
```

Version: ddf54dcfaea71e50ae32b45785b8624b9137843b

Parents:

1c323332493148f0aaa936e668e1b0da5d09c8be

eae3e45ae7dd965cc642d7f03b833574e974fc1b

Merge base:

c8774389e7cb73e494267af3a87f70c7497b220a

lombok/src/core/lombok/javac/handlers/HandleGetter.java

Chunk 49: (concatenation/ commentary, method declaration, method invocation, static block, variable)

```
	}

<<<<<<< HEAD
	private List<JCStatement> createSimpleGetterBody(TreeMaker treeMaker, JavacNode
field) {
		return List.<JCStatement>of(treeMaker.Return(createFieldAccessor(treeMaker,
field, FieldAccess.ALWAYS_FIELD)));
	}

	private static final String AR = "java.util.concurrent.atomic.AtomicReference";
	private static final List<JCExpression> NIL_EXPRESSION = List.nil();

	private static final java.util.Map<Integer, String> TYPE_MAP;
	static {
		Map<Integer, String> m = new HashMap<Integer, String>();
		m.put(TypeTags.INT, "java.lang.Integer");
		m.put(TypeTags.DOUBLE, "java.lang.Double");
		m.put(TypeTags.FLOAT, "java.lang.Float");
		m.put(TypeTags.SHORT, "java.lang.Short");
		m.put(TypeTags.BYTE, "java.lang.Byte");
		m.put(TypeTags.LONG, "java.lang.Long");
		m.put(TypeTags.BOOLEAN, "java.lang.Boolean");
		m.put(TypeTags.CHAR, "java.lang.Character");
		TYPE_MAP = Collections.unmodifiableMap(m);
	}

	private List<JCStatement> createLazyGetterBody(TreeMaker maker, JavacNode fieldNode)
{
		/*
		java.util.concurrent.atomic.AtomicReference<ValueType> value =
this.fieldName.get();
		if (value == null) {
			synchronized (this.fieldName) {
				value = this.fieldName.get();
				if (value == null) {
					value = new
java.util.concurrent.atomic.AtomicReference<ValueType>(new ValueType());
					this.fieldName.set(value);
				}
			}
		}
		return value.get();
		*/

		List<JCStatement> statements = List.nil();
```

```
		JCVariableDecl field = (JCVariableDecl) fieldNode.get();
		field.type = null;
		if (field.vartype instanceof JCPrimitiveTypeTree) {
			String boxed =
TYPE_MAP.get(((JCPrimitiveTypeTree)field.vartype).typetag);
			if (boxed != null) {
				field.vartype = chainDotsString(maker, fieldNode, boxed);
			}
		}

		Name valueName = fieldNode.toName("value");

		/*	java.util.concurrent.atomic.AtomicReference<ValueType>	value	=
this.fieldName.get();*/ {
			JCTypeApply valueVarType = maker.TypeApply(chainDotsString(maker,
fieldNode, AR), List.of(copyType(maker, field)));
			statements	=	statements.append(maker.VarDef(maker.Modifiers(0),
valueName,	valueVarType,	callGet(fieldNode,	createFieldAccessor(maker,	fieldNode,
FieldAccess.ALWAYS_FIELD))));
		}

		/* if (value == null) { */ {
			JCSynchronized synchronizedStatement;
			/* synchronized (this.fieldName) { */ {
				List<JCStatement> synchronizedStatements = List.nil();
				/* value = this.fieldName.get(); */ {
					JCExpressionStatement newAssign =
maker.Exec(maker.Assign(maker.Ident(valueName), callGet(fieldNode,
createFieldAccessor(maker, fieldNode, FieldAccess.ALWAYS_FIELD))));
					synchronizedStatements =
synchronizedStatements.append(newAssign);
				}

				/* if (value == null) { */ {
					List<JCStatement> innerIfStatements = List.nil();
					/* value = new
java.util.concurrent.atomic.AtomicReference<ValueType>(new ValueType());*/ {
						JCTypeApply valueVarType =
maker.TypeApply(chainDotsString(maker, fieldNode, AR), List.of(copyType(maker, field)));

						JCNewClass newInstance = maker.NewClass(null,
NIL_EXPRESSION, valueVarType, List.<JCExpression>of(field.init), null);

						JCStatement statement =
maker.Exec(maker.Assign(maker.Ident(valueName), newInstance));
						innerIfStatements =
innerIfStatements.append(statement);
					}
					/* this.fieldName.set(value); */ {
						JCStatement statement = callSet(fieldNode,
createFieldAccessor(maker, fieldNode, FieldAccess.ALWAYS_FIELD), maker.Ident(valueName));
						innerIfStatements =
innerIfStatements.append(statement);
					}

					JCBinary isNull = maker.Binary(JCTree.EQ,
maker.Ident(valueName), maker.Literal(TypeTags.BOT, null));
					JCIf ifStatement = maker.If(isNull, maker.Block(0,
innerIfStatements), null);
					synchronizedStatements =
synchronizedStatements.append(ifStatement);
```

```
                            }

                            synchronizedStatement                                                =
maker.Synchronized(createFieldAccessor(maker,      fieldNode,      FieldAccess.ALWAYS_FIELD),
maker.Block(0, synchronizedStatements));
                        }

                        JCBinary  isNull  =  maker.Binary(JCTree.EQ,  maker.Ident(valueName),
maker.Literal(TypeTags.BOT, null));
                        JCIf      ifStatement      =      maker.If(isNull,      maker.Block(0,
List.<JCStatement>of(synchronizedStatement)), null);
                        statements = statements.append(ifStatement);
                }
                /* return value.get(); */
                statements        =        statements.append(maker.Return(callGet(fieldNode,
maker.Ident(valueName))));

                // update the field type and init last

                /*      private                                                                 final
java.util.concurrent.atomic.AtomicReference<java.util.concurrent.atomic.AtomicReference<Valu
eType>                          fieldName                          =                          new
java.util.concurrent.atomic.AtomicReference<java.util.concurrent.atomic.AtomicReference<Valu
eType>>(); */ {
                        field.vartype   =   maker.TypeApply(chainDotsString(maker,   fieldNode,
AR),     List.<JCExpression>of(maker.TypeApply(chainDotsString(maker,     fieldNode,     AR),
List.of(copyType(maker, field))));
                        field.init  =  maker.NewClass(null,  NIL_EXPRESSION,  copyType(maker,
field), NIL_EXPRESSION, null);
                }

                return statements;
        }

        private JCMethodInvocation callGet(JavacNode source, JCExpression receiver) {
                TreeMaker maker = source.getTreeMaker();
                return           maker.Apply(NIL_EXPRESSION,           maker.Select(receiver,
source.toName("get")), NIL_EXPRESSION);
        }

        private  JCStatement  callSet(JavacNode  source,  JCExpression  receiver,  JCExpression
value) {
                TreeMaker maker = source.getTreeMaker();
                return      maker.Exec(maker.Apply(NIL_EXPRESSION,      maker.Select(receiver,
source.toName("set")), List.<JCExpression>of(value)));
        }

        private JCExpression copyType(TreeMaker treeMaker, JCVariableDecl fieldNode) {
                return   fieldNode.type   !=   null   ?   treeMaker.Type(fieldNode.type)   :
fieldNode.vartype;
=======
        @Override public boolean isResolutionBased() {
                return false;
>>>>>>> eae3e45ae7dd965cc642d7f03b833574e974fc1b
        }
```

```
        }

        private   List<JCStatement>   createSimpleGetterBody(TreeMaker   treeMaker,   JavacNode
field) {
```

```
                return   List.<JCStatement>of(treeMaker.Return(createFieldAccessor(treeMaker,
field, FieldAccess.ALWAYS_FIELD)));
        }

        private static final String AR = "java.util.concurrent.atomic.AtomicReference";
        private static final List<JCExpression> NIL_EXPRESSION = List.nil();

        private static final java.util.Map<Integer, String> TYPE_MAP;
        static {
                Map<Integer, String> m = new HashMap<Integer, String>();
                m.put(TypeTags.INT, "java.lang.Integer");
                m.put(TypeTags.DOUBLE, "java.lang.Double");
                m.put(TypeTags.FLOAT, "java.lang.Float");
                m.put(TypeTags.SHORT, "java.lang.Short");
                m.put(TypeTags.BYTE, "java.lang.Byte");
                m.put(TypeTags.LONG, "java.lang.Long");
                m.put(TypeTags.BOOLEAN, "java.lang.Boolean");
                m.put(TypeTags.CHAR, "java.lang.Character");
                TYPE_MAP = Collections.unmodifiableMap(m);
        }

        private List<JCStatement> createLazyGetterBody(TreeMaker maker, JavacNode fieldNode)
{
                /*
                java.util.concurrent.atomic.AtomicReference<ValueType>        value        =
this.fieldName.get();
                if (value == null) {
                        synchronized (this.fieldName) {
                                value = this.fieldName.get();
                                if (value == null) {
                                        value                    =                    new
java.util.concurrent.atomic.AtomicReference<ValueType>(new ValueType());
                                        this.fieldName.set(value);
                                }
                        }
                }
                return value.get();
                */

                List<JCStatement> statements = List.nil();

                JCVariableDecl field = (JCVariableDecl) fieldNode.get();
                field.type = null;
                if (field.vartype instanceof JCPrimitiveTypeTree) {
                        String                              boxed                              =
TYPE_MAP.get(((JCPrimitiveTypeTree)field.vartype).typetag);
                        if (boxed != null) {
                                field.vartype = chainDotsString(maker, fieldNode, boxed);
                        }
                }

                Name valueName = fieldNode.toName("value");

                /*     java.util.concurrent.atomic.AtomicReference<ValueType>     value     =
this.fieldName.get();*/ {
                        JCTypeApply   valueVarType   =   maker.TypeApply(chainDotsString(maker,
fieldNode, AR), List.of(copyType(maker, field)));
                        statements    =    statements.append(maker.VarDef(maker.Modifiers(0),
valueName,    valueVarType,    callGet(fieldNode,    createFieldAccessor(maker,    fieldNode,
FieldAccess.ALWAYS_FIELD))));
                }
```

```
                /* if (value == null) { */ {
                        JCSynchronized synchronizedStatement;
                        /* synchronized (this.fieldName) { */ {
                                List<JCStatement> synchronizedStatements = List.nil();
                                /* value = this.fieldName.get(); */ {
                                        JCExpressionStatement            newAssign           =
maker.Exec(maker.Assign(maker.Ident(valueName),                              callGet(fieldNode,
createFieldAccessor(maker, fieldNode, FieldAccess.ALWAYS_FIELD))));
                                        synchronizedStatements                                =
synchronizedStatements.append(newAssign);
                                }

                                /* if (value == null) { */ {
                                        List<JCStatement> innerIfStatements = List.nil();
                                        /*              value              =              new
java.util.concurrent.atomic.AtomicReference<ValueType>(new ValueType());*/ {
                                                JCTypeApply           valueVarType           =
maker.TypeApply(chainDotsString(maker, fieldNode, AR), List.of(copyType(maker, field)));

                                                JCNewClass  newInstance  =  maker.NewClass(null,
NIL_EXPRESSION, valueVarType, List.<JCExpression>of(field.init), null);

                                                JCStatement             statement             =
maker.Exec(maker.Assign(maker.Ident(valueName), newInstance));
                                                innerIfStatements                             =
innerIfStatements.append(statement);
                                        }
                                        /* this.fieldName.set(value); */ {
                                                JCStatement   statement   =   callSet(fieldNode,
createFieldAccessor(maker, fieldNode, FieldAccess.ALWAYS_FIELD), maker.Ident(valueName));
                                                innerIfStatements                             =
innerIfStatements.append(statement);
                                        }

                                        JCBinary     isNull     =     maker.Binary(JCTree.EQ,
maker.Ident(valueName), maker.Literal(TypeTags.BOT, null));
                                        JCIf  ifStatement  =  maker.If(isNull,  maker.Block(0,
innerIfStatements), null);
                                        synchronizedStatements                                =
synchronizedStatements.append(ifStatement);
                                }

                                synchronizedStatement                                         =
maker.Synchronized(createFieldAccessor(maker,     fieldNode,     FieldAccess.ALWAYS_FIELD),
maker.Block(0, synchronizedStatements));
                        }

                        JCBinary  isNull  =  maker.Binary(JCTree.EQ,  maker.Ident(valueName),
maker.Literal(TypeTags.BOT, null));
                        JCIf      ifStatement      =      maker.If(isNull,      maker.Block(0,
List.<JCStatement>of(synchronizedStatement)), null);
                        statements = statements.append(ifStatement);
                }
                /* return value.get(); */
                statements        =        statements.append(maker.Return(callGet(fieldNode,
maker.Ident(valueName))));

                // update the field type and init last

                /*     private                                                          final
java.util.concurrent.atomic.AtomicReference<java.util.concurrent.atomic.AtomicReference<Valu
eType>                        fieldName                          =                          new
```

```
java.util.concurrent.atomic.AtomicReference<java.util.concurrent.atomic.AtomicReference<Valu
eType>>(); */ {
                        field.vartype  =  maker.TypeApply(chainDotsString(maker,  fieldNode,
AR),    List.<JCExpression>of(maker.TypeApply(chainDotsString(maker,    fieldNode,    AR),
List.of(copyType(maker, field)))));
                        field.init  =  maker.NewClass(null,  NIL_EXPRESSION,  copyType(maker,
field), NIL_EXPRESSION, null);
                }

                return statements;
        }

        private JCMethodInvocation callGet(JavacNode source, JCExpression receiver) {
                TreeMaker maker = source.getTreeMaker();
                return          maker.Apply(NIL_EXPRESSION,          maker.Select(receiver,
source.toName("get")), NIL_EXPRESSION);
        }

        private  JCStatement  callSet(JavacNode  source,  JCExpression  receiver,  JCExpression
value) {
                TreeMaker maker = source.getTreeMaker();
                return     maker.Exec(maker.Apply(NIL_EXPRESSION,     maker.Select(receiver,
source.toName("set")), List.<JCExpression>of(value)));
        }

        private JCExpression copyType(TreeMaker treeMaker, JCVariableDecl fieldNode) {
                return   fieldNode.type   !=   null   ?   treeMaker.Type(fieldNode.type)   :
fieldNode.vartype;
        }

        @Override public boolean isResolutionBased() {
                return false;
        }
```

Version: 3d4b27d6d288ecb418a2a1a09ed43cae90ec548e

Parents:

fe7f0db2fce7b4c80853b9aed100908ff1f55f40

57de0c3f6636181541a7712e8d506828420c13d1

Merge base:

98d8a9f63b3183005174abb7691a1692347b9a2e

lombok/eclipse/handlers/EclipseHandlerUtil.java

Chunk 50: (concatenation/ method declaration, variable)

```
	}

<<<<<<< HEAD
	private static final Annotation[] EMPTY_ANNOTATION_ARRAY = new Annotation[0];
	static Annotation[] getAndRemoveAnnotationParameter(Annotation annotation, String
annotationName) {

		List<Annotation> result = new ArrayList<Annotation>();
		if (annotation instanceof NormalAnnotation) {
			NormalAnnotation normalAnnotation = (NormalAnnotation)annotation;
			MemberValuePair[] memberValuePairs =
normalAnnotation.memberValuePairs;
			List<MemberValuePair> pairs = new ArrayList<MemberValuePair>();
			if (memberValuePairs != null) for (MemberValuePair memberValuePair :
memberValuePairs) {
				if (annotationName.equals(new String(memberValuePair.name))) {
					Expression value = memberValuePair.value;
					if (value instanceof ArrayInitializer) {
						ArrayInitializer array = (ArrayInitializer)
value;
						for(Expression expression : array.expressions)
{
							if (expression instanceof Annotation) {

	result.add((Annotation)expression);
							}
						}
					}
					else if (value instanceof Annotation) {
						result.add((Annotation)value);
					}
					continue;
				}
				pairs.add(memberValuePair);
			}

			if (!result.isEmpty()) {
				normalAnnotation.memberValuePairs = pairs.isEmpty() ? null :
pairs.toArray(new MemberValuePair[0]);
				return result.toArray(EMPTY_ANNOTATION_ARRAY);
			}
		}

		return EMPTY_ANNOTATION_ARRAY;
	}

=======
```

```
	static NameReference createNameReference(String name, Annotation source) {
		int pS = source.sourceStart, pE = source.sourceEnd;
		long p = (long)pS << 32 | pE;

		char[][] nameTokens = fromQualifiedName(name);
		long[] pos = new long[nameTokens.length];
		Arrays.fill(pos, p);

		QualifiedNameReference nameReference = new QualifiedNameReference(nameTokens,
pos, pS, pE);
		nameReference.statementEnd = pE;

		Eclipse.setGeneratedBy(nameReference, source);
		return nameReference;
	}
>>>>>>> 57de0c3f6636181541a7712e8d506828420c13d1
}
```

```
	}

	private static final Annotation[] EMPTY_ANNOTATION_ARRAY = new Annotation[0];
	static  Annotation[]  getAndRemoveAnnotationParameter(Annotation  annotation,  String
annotationName) {

		List<Annotation> result = new ArrayList<Annotation>();
		if (annotation instanceof NormalAnnotation) {
			NormalAnnotation normalAnnotation = (NormalAnnotation)annotation;
			MemberValuePair[]                 memberValuePairs                 =
normalAnnotation.memberValuePairs;
			List<MemberValuePair> pairs = new ArrayList<MemberValuePair>();
			if  (memberValuePairs  !=  null)  for  (MemberValuePair memberValuePair :
memberValuePairs) {
				if (annotationName.equals(new String(memberValuePair.name))) {
					Expression value = memberValuePair.value;
					if (value instanceof ArrayInitializer) {
						ArrayInitializer  array  =  (ArrayInitializer)
value;
						for(Expression  expression  :  array.expressions)
{
							if (expression instanceof Annotation) {

	result.add((Annotation)expression);
							}
						}
					}
					else if (value instanceof Annotation) {
						result.add((Annotation)value);
					}
					continue;
				}
				pairs.add(memberValuePair);
			}

			if (!result.isEmpty()) {
				normalAnnotation.memberValuePairs  =  pairs.isEmpty()  ?  null  :
pairs.toArray(new MemberValuePair[0]);
				return result.toArray(EMPTY_ANNOTATION_ARRAY);
			}
		}

		return EMPTY_ANNOTATION_ARRAY;
```

```
        }

        static NameReference createNameReference(String name, Annotation source) {
                int pS = source.sourceStart, pE = source.sourceEnd;
                long p = (long)pS << 32 | pE;

                char[][] nameTokens = fromQualifiedName(name);
                long[] pos = new long[nameTokens.length];
                Arrays.fill(pos, p);

                QualifiedNameReference nameReference = new QualifiedNameReference(nameTokens,
pos, pS, pE);
                nameReference.statementEnd = pE;

                Eclipse.setGeneratedBy(nameReference, source);
                return nameReference;
        }
}
```

Version: c88ae3af7432513987eafaf13c178baa77cb0667

Parents:

0951ea38fe11189cdc4c2778fdad9e9e3ad6a6ae

16f992c5adea8ed8ad183d27c247901d61b0635d

Merge base:

98d8a9f63b3183005174abb7691a1692347b9a2e

lombok/javac/handlers/HandleCleanup.java

Chunk 51: (version 2/commentary)

```
/*
<<<<<<< HEAD
 * Copyright Â© 2009-2010 Reinier Zwitserloot and Roel Spilker.
=======
 * Copyright Â© 2009-2010 Reinier Zwitserloot, Roel Spilker and Robbert Jan Grootjans.
>>>>>>> 16f992c5adea8ed8ad183d27c247901d61b0635d
 *
```

```
/*
 * Copyright Â© 2009-2010 Reinier Zwitserloot, Roel Spilker and Robbert Jan Grootjans.
 *
```

Version: 4e152f2f1485f904feb45ae614236d4aa4b6edc9

Parents:

0221e460b2e648b142284c6c462d5798f33a3ff7

fe0da3f53f1e88b704e21463cc5fea3d998e394a

Merge base:

2bdc1210d7a26df8b69563f0de22524398ba9bfd

lombok/src/lombok/eclipse/handlers/HandleData.java

Chunk 52: (combination/if statement, method invocation, variable)

```
                        long fieldPos = (((long)field.sourceStart) << 32) | field.sourceEnd;
<<<<<<< HEAD
                        args.add(new   Argument(field.name,   fieldPos,   copyType(field.type),
Modifier.FINAL));
=======
                        Argument    argument    =    new    Argument(field.name,    fieldPos,
copyType(field.type), 0);
                        Annotation[] nonNulls = findAnnotations(field, NON_NULL_PATTERN);
                        Annotation[] nullables = findAnnotations(field, NULLABLE_PATTERN);
                        if (nonNulls.length != 0) nullChecks.add(generateNullCheck(field));
                        Annotation[]     copiedAnnotations     =     copyAnnotations(nonNulls,
nullables);
                        if    (copiedAnnotations.length    !=    0)    argument.annotations    =
copiedAnnotations;
                        args.add(argument);
>>>>>>> fe0da3f53f1e88b704e21463cc5fea3d998e394a
                }
```

```
                        long fieldPos = (((long)field.sourceStart) << 32) | field.sourceEnd;
                        Argument    argument    =    new    Argument(field.name,    fieldPos,
copyType(field.type), Modifier.FINAL);
                        Annotation[] nonNulls = findAnnotations(field, NON_NULL_PATTERN);
                        Annotation[] nullables = findAnnotations(field, NULLABLE_PATTERN);
                        if (nonNulls.length != 0) nullChecks.add(generateNullCheck(field));
                        Annotation[]     copiedAnnotations     =     copyAnnotations(nonNulls,
nullables);
                        if    (copiedAnnotations.length    !=    0)    argument.annotations    =
copiedAnnotations;
                        args.add(argument);
                }
```

Chunk 53: (combination/if statement, method invocation, variable)

```
                        assigns.add(new SingleNameReference(field.name, fieldPos));
<<<<<<< HEAD
                        args.add(new   Argument(field.name,   fieldPos,   copyType(field.type),
Modifier.FINAL));
=======

                        Argument    argument    =    new    Argument(field.name,    fieldPos,
copyType(field.type), 0);
                        Annotation[] copiedAnnotations = copyAnnotations(
                                        findAnnotations(field,              NON_NULL_PATTERN),
findAnnotations(field, NULLABLE_PATTERN));
                        if    (copiedAnnotations.length    !=    0)    argument.annotations    =
copiedAnnotations;
```

```
                        args.add(new   Argument(field.name,   fieldPos,   copyType(field.type),
0));
>>>>>>> fe0da3f53f1e88b704e21463cc5fea3d998e394a
                }
```

```
                        assigns.add(new SingleNameReference(field.name, fieldPos));

                        Argument    argument    =    new    Argument(field.name,    fieldPos,
copyType(field.type), 0);
                        Annotation[] copiedAnnotations = copyAnnotations(
                                        findAnnotations(field,              NON_NULL_PATTERN),
findAnnotations(field, NULLABLE_PATTERN));
                        if    (copiedAnnotations.length    !=    0)    argument.annotations    =
copiedAnnotations;
                        args.add(new   Argument(field.name,   fieldPos,   copyType(field.type),
Modifier.FINAL));
                }
```

lombok/src/lombok/eclipse/handlers/HandleEqualsAndHashCode.java

Chunk 54: (version 1/if statement, method invocation)

```
                }

<<<<<<< HEAD
                if ( !isDirectDescendentOfObject && !callSuper && implicit ) {
                        errorNode.addWarning("Generating  equals/hashCode  implementation  but
without a call to superclass, even though this class does not extend java.lang.Object. If
this is intentional, add '@EqualsAndHashCode(callSuper=false)' to your type.");
=======
                if ( !isDirectDescendantOfObject && !callSuper ) {
                        errorNode.addWarning("Generating  equals/hashCode  implementation  but
without a call to superclass, even though this class does not extend java.lang.Object.");
>>>>>>> fe0da3f53f1e88b704e21463cc5fea3d998e394a
                }
```

```
                }

                if ( !isDirectDescendantOfObject && !callSuper && implicit ) {
                        errorNode.addWarning("Generating  equals/hashCode  implementation  but
without a call to superclass, even though this class does not extend java.lang.Object. If
this is intentional, add '@EqualsAndHashCode(callSuper=false)' to your type.");
                }
```

lombok/src/lombok/eclipse/handlers/HandleSetter.java

Chunk 55: (version 1/method invocation, variable)

```
                method.annotations = null;
<<<<<<< HEAD
                Argument param = new Argument(field.name, pos, Eclipse.copyType(field.type),
Modifier.FINAL);
=======
                Argument param = new Argument(field.name, pos, copyType(field.type), 0);
>>>>>>> fe0da3f53f1e88b704e21463cc5fea3d998e394a
                method.arguments = new Argument[] { param };
```

```
                method.annotations = null;
```

```
                Argument param = new Argument(field.name, pos, copyType(field.type),
Modifier.FINAL);
                method.arguments = new Argument[] { param };
```

lombok/src/lombok/javac/handlers/HandleData.java

Chunk 56: (combination/method invocation, variable)

```
                        JCVariableDecl field = (JCVariableDecl) fieldNode.get();
<<<<<<< HEAD
                        JCVariableDecl param = maker.VarDef(maker.Modifiers(Flags.FINAL),
field.name, field.vartype, null);
=======

                        List<JCAnnotation> nonNulls = findAnnotations(fieldNode,
NON_NULL_PATTERN);
                        List<JCAnnotation> nullables = findAnnotations(fieldNode,
NULLABLE_PATTERN);
                        JCVariableDecl param = maker.VarDef(maker.Modifiers(0,
nonNulls.appendList(nullables)), field.name, field.vartype, null);

>>>>>>> fe0da3f53f1e88b704e21463cc5fea3d998e394a
                        params = params.append(param);
```

```
                for ( Node fieldNode : fields ) {
                        JCVariableDecl field = (JCVariableDecl) fieldNode.get();
                        List<JCAnnotation> nonNulls = findAnnotations(fieldNode,
NON_NULL_PATTERN);
                        List<JCAnnotation> nullables = findAnnotations(fieldNode,
NULLABLE_PATTERN);
                        JCVariableDecl param = maker.VarDef(maker.Modifiers(Flags.FINAL,
nonNulls.appendList(nullables)), field.name, field.vartype, null);
                        params = params.append(param);
```

Chunk 57: (version 2/method invocation, variable)

```
                        } else pType = field.vartype;
<<<<<<< HEAD
                        JCVariableDecl param = maker.VarDef(maker.Modifiers(Flags.FINAL),
field.name, pType, null);
=======

                        List<JCAnnotation> nonNulls = findAnnotations(fieldNode,
NON_NULL_PATTERN);
                        List<JCAnnotation> nullables = findAnnotations(fieldNode,
NULLABLE_PATTERN);
                        JCVariableDecl param = maker.VarDef(maker.Modifiers(0,
nonNulls.appendList(nullables)), field.name, pType, null);
>>>>>>> fe0da3f53f1e88b704e21463cc5fea3d998e394a
                        params = params.append(param);
```

```
                        } else pType = field.vartype;
                        List<JCAnnotation> nonNulls = findAnnotations(fieldNode,
NON_NULL_PATTERN);
                        List<JCAnnotation> nullables = findAnnotations(fieldNode,
NULLABLE_PATTERN);
                        JCVariableDecl param = maker.VarDef(maker.Modifiers(Flags.FINAL,
nonNulls.appendList(nullables)), field.name, pType, null);
                        params = params.append(param);
```

lombok/src/lombok/javac/handlers/HandleEqualsAndHashCode.java

Chunk 58: (version 1/ if statement, method invocation)

```
                }

<<<<<<< HEAD
                if ( !isDirectDescendentOfObject && !callSuper && implicit ) {
                        errorNode.addWarning("Generating equals/hashCode implementation but
without a call to superclass, even though this class does not extend java.lang.Object. If
this is intentional, add '@EqualsAndHashCode(callSuper=false)' to your type.");
=======
                if ( !isDirectDescendantOfObject && !callSuper ) {
                        errorNode.addWarning("Generating equals/hashCode implementation but
without a call to superclass, even though this class does not extend java.lang.Object.");
>>>>>>> fe0da3f53f1e88b704e21463cc5fea3d998e394a
                }
```

```
                }

                if ( !isDirectDescendantOfObject && !callSuper && implicit ) {
                        errorNode.addWarning("Generating equals/hashCode implementation but
without a call to superclass, even though this class does not extend java.lang.Object. If
this is intentional, add '@EqualsAndHashCode(callSuper=false)' to your type.");
                }
```

lombok/src/lombok/javac/handlers/HandleSetter.java

Chunk 59: (new code/method invocation, variable)

```
                Name methodName = field.toName(toSetterName(fieldDecl));
<<<<<<< HEAD
                JCVariableDecl param = treeMaker.VarDef(treeMaker.Modifiers(Flags.FINAL),
fieldDecl.name, fieldDecl.vartype, null);
=======

                JCVariableDecl param = treeMaker.VarDef(treeMaker.Modifiers(0,
nonNulls.appendList(nullables)), fieldDecl.name, fieldDecl.vartype, null);
>>>>>>> fe0da3f53f1e88b704e21463cc5fea3d998e394a
                JCExpression methodType = treeMaker.Type(field.getSymbolTable().voidType);
```

```
                Name methodName = field.toName(toSetterName(fieldDecl));
                JCVariableDecl param = treeMaker.VarDef(treeMaker.Modifiers(Flags.FINAL,
nonNulls.appendList(nullables)), fieldDecl.name, fieldDecl.vartype, null);
                JCExpression methodType = treeMaker.Type(field.getSymbolTable().voidType);
```