

Does your software do what it should?  
Specification and verification with the Java Modeling  
Language and OpenJML.  
The OpenJML User Guide  
  
DRAFT IN PROGRESS

David R. Cok  
GrammaTech, Inc.

December 30, 2016

The most recent version of this document is available at  
<http://jmlspecs.sourceforge.net/OpenJMLUserGuide.pdf>.

Copyright (c) 2010-2015 by David R. Cok. Permission is granted to make and distribute copies of this document for educational or research purposes, provided that the copyright notice and permission notice are preserved and acknowledgment is given in publications. Modified versions of the document may not be made. Please forward corrections to the author. Incorporating this document within a larger collection, or distributing it for commercial purposes, or including it as part or all of a product for sale is allowed only by separate written permission from the author.



# Contents

<b>1</b>	<b>Why specify? Why check?</b>	<b>5</b>
<b>2</b>	<b>Background of verification, JML, and OpenJML</b>	<b>6</b>
<b>3</b>	<b>Organization of this document</b>	<b>7</b>
<b>4</b>	<b>Some details</b>	<b>8</b>
4.1	Disambiguating ‘annotation’ . . . . .	8
4.2	Syntactic conflicts with @ . . . . .	8
4.3	.jml files and .java files . . . . .	9
<b>5</b>	<b>Quick start to OpenJML</b>	<b>10</b>
5.1	Installing OpenJML . . . . .	10
<b>6</b>	<b>Other resources</b>	<b>11</b>
<b>I</b>	<b>Tutorial introduction to specifying and checking Java programs</b>	<b>12</b>
<b>7</b>	<b>Tutorial</b>	<b>13</b>
<b>II</b>	<b>The Java Modeling Language (JML)</b>	<b>14</b>
<b>8</b>	<b>JML concepts</b>	<b>15</b>
8.1	JML modifiers and Java annotations . . . . .	15
8.2	Model and Ghost . . . . .	17
8.3	Visibility . . . . .	17
8.4	JML types . . . . .	17
8.5	Evaluation and well-formedness of JML expressions . . . . .	17
8.6	Null and non-null references . . . . .	17
8.7	Static and Instance . . . . .	17
8.8	Location sets . . . . .	17
8.9	Arithmetic modes . . . . .	17
8.10	org.jmlspecs.lang package . . . . .	17
<b>9</b>	<b>Summary of JML Features</b>	<b>18</b>
9.1	JML Syntax . . . . .	18
9.1.1	Syntax of JML specifications . . . . .	18
9.1.2	Conditional JML specifications . . . . .	18
9.1.3	Finding specification files and the refine statement . . . . .	19

9.1.4	Model import statements . . . . .	20
9.1.5	Modifiers . . . . .	20
9.1.6	Method specification clauses . . . . .	22
9.1.7	Class specification clauses . . . . .	22
9.1.8	Visibility of specifications . . . . .	22
9.1.9	Statement specifications . . . . .	22
9.1.10	JML expressions . . . . .	23
9.1.11	JML types . . . . .	26
9.1.12	Non-Null and Nullable . . . . .	26
9.1.13	Observable purity: \query and \secret . . . . .	26
9.1.14	Race condition detection . . . . .	26
9.1.15	Arithmetic modes . . . . .	26
9.1.16	Universe types . . . . .	26
9.1.17	Dynamic frames . . . . .	26
9.1.18	Code contracts . . . . .	27
9.1.19	redundantly suffixes . . . . .	27
9.1.20	nowarn lexical construct . . . . .	27
9.2	Interaction with Java features . . . . .	27
9.3	Other issues . . . . .	27
9.3.1	Interaction with JSR-308 . . . . .	27
9.3.2	Interaction with FindBugs . . . . .	27

### **III The OpenJML tool 28**

#### **10 Introduction 29**

10.1	OpenJML . . . . .	29
10.1.1	Command-line tool . . . . .	29
10.1.2	Eclipse plug-in . . . . .	29
10.1.3	Development of OpenJML . . . . .	30
10.2	JML . . . . .	30
10.3	OpenJDK . . . . .	30
10.4	License . . . . .	30

#### **11 The command-line tool 31**

11.1	Installation and System Requirements . . . . .	31
11.2	Running OpenJML . . . . .	31
11.2.1	Exit values . . . . .	32
11.2.2	Files . . . . .	32
11.2.3	Specification files . . . . .	32
11.2.4	Annotations and the runtime library . . . . .	34
11.2.5	Java properties and the openjml.properties file . . . . .	34
11.2.6	OpenJML options . . . . .	35
11.2.7	Options: Runtime Assertion Checking . . . . .	38
11.2.8	Options related to Static Checking . . . . .	40
11.2.9	Options related to parsing and typechecking . . . . .	41
11.2.10	Java options related to annotation processing . . . . .	41
11.2.11	Other JML Options . . . . .	41

#### **12 The Eclipse Plug-in 42**

12.1	Installation and System Requirements . . . . .	42
12.2	GUI Features . . . . .	43

12.2.1	Commands . . . . .	43
12.2.2	Menubar additions . . . . .	44
12.2.3	Toolbar additions . . . . .	44
12.2.4	OpenJML Problems, Markers and highlights . . . . .	44
12.2.5	OpenJML console and error log . . . . .	44
12.2.6	Preferences . . . . .	45
12.2.7	Editor embellishments . . . . .	45
12.2.8	OpenJML Views . . . . .	45
12.2.9	Help . . . . .	45
12.2.10	Other GUI elements . . . . .	45
<b>13</b>	<b>OpenJML tools</b>	<b>47</b>
13.1	Parsing and Type-checking . . . . .	47
13.1.1	Classpaths, sourcepaths, and specification paths in OpenJML . . . . .	47
13.1.2	Command-line options for type-checking . . . . .	48
13.2	Static Checking and Verification . . . . .	49
13.2.1	Options specific to static checking . . . . .	49
13.3	Runtime Assertion Checking . . . . .	49
13.3.1	Options specific to runtime checking . . . . .	49
13.4	Generating Documentation . . . . .	49
13.5	Generating Specification File Skeletons . . . . .	49
13.6	Generating Test Cases . . . . .	50
13.7	Limitations of OpenJML's implementation of JML . . . . .	50
13.7.1	model import statement . . . . .	50
13.7.2	purity checks and system library annotations . . . . .	50
13.7.3	TBD - other unimplemented features . . . . .	51
<b>14</b>	<b>Using OpenJML and OpenJDK within user programs</b>	<b>52</b>
14.1	Concepts . . . . .	53
14.1.1	Compilation Contexts . . . . .	53
14.1.2	JavaFileObjects . . . . .	54
14.1.3	Interfaces and concrete classes . . . . .	54
14.1.4	Object Factories . . . . .	55
14.1.5	Abstract Syntax Trees . . . . .	55
14.1.6	Compilation Phases and The tool registry . . . . .	55
14.2	OpenJML operations . . . . .	57
14.2.1	Methods equivalent to command-line operations . . . . .	57
14.2.2	Parsing . . . . .	59
14.2.3	Type-checking . . . . .	61
14.2.4	Static checking . . . . .	61
14.2.5	Compiling run-time checks . . . . .	61
14.2.6	Creating JML-enhanced documentation . . . . .	61
14.3	Working with ASTs . . . . .	61
14.3.1	Printing parse trees . . . . .	61
14.3.2	Source location information . . . . .	61
14.3.3	Exploring parse trees with Visitors . . . . .	61
14.3.4	Creating parse trees . . . . .	67
14.4	Working with JML specifications . . . . .	67
14.5	Utilities . . . . .	67
14.6	Extending or modifying JML . . . . .	67
14.6.1	Adding new command-line options . . . . .	67
14.6.2	Altering IAPI . . . . .	67

14.6.3	Changing the Scanner . . . . .	67
14.6.4	Enhancing the parser . . . . .	67
14.6.5	Adding new modifiers and annotations . . . . .	67
14.6.6	Adding new AST nodes . . . . .	67
14.6.7	Modifying a compiler phase . . . . .	67
<b>IV</b>	<b>Contributing to OpenJML</b>	<b>68</b>
<b>V</b>	<b>Semantics and translation of Java and JML in OpenJML</b>	<b>69</b>
<b>15</b>	<b>Introduction</b>	<b>70</b>
<b>16</b>	<b>Statement translations</b>	<b>71</b>
16.1	While loop . . . . .	71
<b>17</b>	<b>Java expression translations</b>	<b>72</b>
17.1	Implicit or explicit arithmetic conversions . . . . .	72
17.2	Arithmetic expressions . . . . .	72
17.3	Bit-shift expressions . . . . .	73
17.4	Relational expressions . . . . .	73
17.5	Logical expressions . . . . .	73

## **Chapter 1**

### **Why specify? Why check?**

## **Chapter 2**

# **Background of verification, JML, and OpenJML**



## Chapter 3

# Organization of this document

This document addresses three related topics: how to read, write, and use specifications; the Java Modeling Language (JML) in which specifications are written; and the OpenJML tool that provides editing and checking support for Java programs using JML.

These three topics are best learned in an interleaved fashion. The tutorial section (Part I) does just this. It introduces the simpler topics of specification, using Java programs with JML as the specification language, and using OpenJML as the tool to aid editing and checking, with motivating examples. A reader new to JML or to using specifications will find this tutorial to be the easiest introduction to the topics of this book.

However, it is also useful to have a compact description of each of the JML language and the OpenJML tool. These descriptions are found in Parts II and III respectively. After some introduction, a reader may well want to take a break from the tutorial to read through and experiment with the details of JML and OpenJML. Once a reader has graduated from the tutorial and is specifying and verifying new examples, the description of JML serves as a summary of the JML language and the description of OpenJML is the user guide and reference manual for the tool. These two parts stand on their own.

Part IV contains information for those interested in contributing to the document. Contributions in the form of bug reports and experience reports with substantial use cases or experience in teaching are always welcome; this information can be shared directly with the developers or through the `jmlspecs` mailing list. Part IV, however, contains information primarily of interest to those developing and extending the OpenJML code itself.

The final part of the document, Part V, describes details of how OpenJML translates the combination of Java and JML. This is not meant to be read through and is only intended for the reader interested in the detailed semantics of JML and the implementation of OpenJML.

FIXME - what about a mailing list

# Chapter 4

## Some details

### 4.1 Disambiguating ‘annotation’

. Formal specifications for code are often called annotations; in this document we often use the term ‘JML annotations’ to refer to specifications written in JML. There is also a specific syntactic construct in Java called ‘annotations’: the interfaces labeled with ‘@’ symbols that can modify various syntactic elements of Java. Thus the simple term ‘annotation’ can be ambiguous. The ambiguity is heightened by the fact that JML annotations, such as `/*@ pure */`, can be expressed as Java annotations, `@Pure`.

In this document, we will generally disambiguate the term ‘annotation’ as either ‘JML annotation’ or ‘Java annotation’; if used alone, ‘annotation’ will generally mean a JML annotation.

### 4.2 Syntactic conflicts with @

For historical reasons, specifications are often written as structured programming language comments, with the `@` symbol denoting a comment containing specifications. Java comments begin with either `//` or `/*`; those comments that contain JML specifications begin with `//@` or `/*@`. Similarly, `//` or `/*` are also used for comments in C and C++; the ANSI-C Specification Language also uses `//@` or `/*@` to indicate comments containing specifications.

Unfortunately, since the `@` symbol is also used for Java annotations, the following problem can arise. Some Java code is written something like (the particular Java annotation and its content are irrelevant)

```
@SuppressWarnings("...")
class X
```

and then the user comments out the Java annotation without any whitespace:

```
//@SuppressWarnings("...")
class X
```

Now JML tools will interpret the `//@` as the beginning of a JML annotation that will generally have parsing errors.

If the user includes whitespace, as in

```
// @SuppressWarnings(...) class X
```

there is then no problem. The workaround for this conflict is to edit the original Java source to include the whitespace. In some situations, placing all JML annotations in a .jml file may solve the problem; however, some tools, including OpenJML, may still parse the .java file, including the erroneous apparently-JML annotations, even though those annotations are ignored when a .jml file is present.

### 4.3 .jml files and .java files

-TBD - .jml files hide annotations in .java files, except those in body of methods

## Chapter 5

# Quick start to OpenJML

The details of installing and running OpenJML are presented in Part III. However, an installation of the tool is needed to work through the tutorial. Some impatient readers may also wish to have a quick installation of the tool prior to diving into the full description. This section provides initial installation and use instructions.

### 5.1 Installing OpenJML

OpenJML is available as a command-line tool and as an Eclipse plug-in. Complete the following steps to install the command-line tool:

- Create or identify a directory (folder) in which to place the installation. Let `$OPENJML` represent the path to this installation directory.
- Download into `$OPENJML` either the zip file at <http://jmlspecs.sourceforge.net/openjml.zip> or a gzipped tar file from <http://jmlspecs.sourceforge.net/openjml.tar.gz>
- `cd` into the directory and either `unzip openjml.zip` or `untar tar xvzf openjml.tar.gz` the downloaded file.
-

## Chapter 6

# Other resources

There are several other useful resources related to JML and OpenJML:

- <http://www.jmlspecs.org> is a web site describing current on JML, including references to many publications, other tools, and links to various groups using JML.
- <http://www.jmlspecs.org/OldReleases/jmlrefman.pdf> is the official reference manual for JML, though it sometimes lags behind agreed-upon changes that are implemented in tools. (FIXME - make a better link)
- <http://www.openjml.org> contains a set of on-line resources for OpenJML
- <http://jmlspecs.sourceforge.net/OpenJMLUserGuide.pdf> is the most current version of this document
- <http://jmlspecs.sourceforge.net/OpenJMLUserGuide.html> is an HTML version, with frames, of this document; <http://jmlspecs.sourceforge.net/OpenJMLUserGuide-onepage.html> is the same material in one large HTML page.
- The source code for OpenJML, the original JML tools, and some other JML projects is contained in the jmlspecs sourceforge project at <http://sourceforge.net/projects/jmlspecs>.

There are also other tools that make use of JML. An incomplete list follows:

- Key - FIXME - need url
- The previous generation of JML tools prior to OpenJML is available at <http://www.jmlspecs.org/download.shtml>.
- FIXME - need others

## **Part I**

# **Tutorial introduction to specifying and checking Java programs**

## **Chapter 7**

## **Tutorial**

## **Part II**

# **The Java Modeling Language (JML)**



## Chapter 8

# JML concepts

### 8.1 JML modifiers and Java annotations

The Java Modeling Language was defined prior to the introduction of annotations in Java. Some, but not all, of the features of JML can now be textually represented as Java annotations. Currently JML supports both the old and new syntactic forms.

#### Modifiers

Modifiers are JML keywords that specify JML characteristics of methods, classes, fields, or variables. Examples are `pure`, `model`, and `ghost`. They are syntactically placed just like Java modifiers, such as `public`.

Each such modifier has an equivalent Java annotation. For example

```
/*@ pure */ public int m(int i) ...
```

can be written equivalently as

```
org.jmlspecs.annotation.Pure public int m(int i) ...
```

The `org.jmlspecs.annotation` prefix can be made implicit in the usual way by including the import statement

```
import org.jmlspecs.annotation.Pure;
```

Note that in the second form, the `pure` designation is now part of the *Java* program and so the import of the `org.jmlspecs.annotation` package must also be in the Java program, and the package must be available to the Java compiler.

All of the modifiers, their corresponding Java annotations, and the locations in which they may be used are described in §9.1.5.

#### Type modifiers

Some modifiers are actually type modifiers. In particular `non_null` and `nullable` are in this category. Thus the description of the previous subsection (§8.1) apply to these as well.

However, Java 1.8 allows Java annotations to be applied to types wherever type names may appear. For example

`(@NonNull String)toUpper(s)`

is allowed in Java 1.8 but is forbidden in Java 1.7.

*Need additional discussion of the change in JML for Java 1.8, especially for arrays.*

### **Method specification clauses**

*This section will be added later.*

### **Class specification clauses**

*This section will be added later.*

### **Statement specifications**

JML specifications that are statements within the body of a method have no equivalent as Java annotations. These include loop specifications, assert and assume statements, ghost declarations, set and debug statements, and specifications on individual statements.

## 8.2 Model and Ghost

## 8.3 Visibility

## 8.4 JML types

## 8.5 Evaluation and well-formedness of JML expressions

## 8.6 Null and non-null references

## 8.7 Static and Instance

## 8.8 Location sets

## 8.9 Arithmetic modes

## 8.10 org.jmlspecs.lang package

Some JML features are defined in the `org.jmlspecs.lang` package. The `org.jmlspecs.lang` package is included as a model import by default, just as the `java.lang` package is included by default in a Java file. `org.jmlspecs.lang.*` contains these elements:

- `JML.informal(<string>)` : This method is a replacement for (and is equivalent to) the informal expression syntax (§??) `(* ... *)`. Both expressions return a boolean value, which is always `true`.
- TBD

## Chapter 9

# Summary of JML Features

The definition of the Java Modeling Language is contained in the JML reference manual.[?] This document does not repeat that definition in detail. However, the following sections summarize the features of JML, indicate what is and is not implemented in OpenJML, describes any extensions to JML contained in OpenJML, and includes comments about relevant implementation aspects of OpenJML.

### 9.1 JML Syntax

#### 9.1.1 Syntax of JML specifications

JML specifications may be written as Java annotations. Currently these are only implemented for modifiers (cf. section TBD). In Java 8, the use of Java annotations for JML features will be expanded.

JML specifications may also be written in specially formatted Java comments: a JML specification includes everything between either (a) an opening `/*@` and closing `*/` or (b) an opening `//@` and the next line ending character (`\n` or `\r`) that is not within a string or character literal.

Such comments that occur within the body of a class or interface definition are considered to be a specification of the class, a field, or a method, depending on the kind of specification clause it is. JML specifications may also occur in the body of a method.

**Obsolete syntax.** In previous versions of JML, JML specifications could be placed within javadoc comments. Such specifications are no longer standard JML and are not supported by OpenJML.

#### 9.1.2 Conditional JML specifications

JML has a mechanism for conditional specifications, based on a system of keys. A key is an identifier (consisting of ASCII alphanumeric and underscore characters, and beginning with a non-digit). A conditional JML comment is guarded by one or more positive or negative keys (or both). The keys are placed just before the `@` character that is part of the opening sequence of the JML comment (the `//@` or the `/*@`). Each key is preceded by a `'+'` or a `'-'` sign, to indicate whether it is a positive or negative key, respectively. *No white-space is allowed.* If there is white-space anywhere between the initial `//` or `/*` and the first `@` character, the comment will appear to be a normal Java comment and will be silently ignored.

The keys are interpreted as follows. Each tool that processes the Java+JML input will have a means (e.g. by command-line options) to specify the set of keys that are enabled.

- If the JML annotation has no keys, the annotation is always processed.
- If there are only positive keys, the annotation is processed only if at least one of the keys is enabled.
- If there are only negative keys, the annotation is processed unless one of the keys is enabled.
- If there are both positive and negative keys, the annotation is processed only if (a) at least one of the positive keys is enabled AND (b) none of the negative keys are enabled.

JML previously defined one conditional annotation: those that began with `/*+@` or `//+@`. ESC/Java2 also defined `/*-@` and `//-@`. Both of these are now deprecated. OpenJML does have an option to enable the `+-style` comments.

The particular keys do not have any defined meaning in the JML reference manual. OpenJML implicitly enables the following keys:

- **ESC** : the ESC key is enabled when OpenJML is performing ESC static checking;
- **RAC** : the RAC key is enabled when OpenJML is performing Runtime-Assertion-Checking.
- **DEBUG** : The DEBUG key is not implicitly enabled. However it is defined as the key that enables the **debug** JML statement. That is the **debug** statement is ignored by default and is used by OpenJML if the user enables the DEBUG key.
- **OPENJML** : The OPENJML key is enabled whenever OpenJML is processing annotations (and presumably is not enabled by other tools).
- **KEY**: The KEY key is reserved for annotations recognized by the KeY tool [?]. It is ignored by OpenJML.

Thus, for example, one can turn off a non-executable assert statement for RAC-processing but retain it for ESC and for type-checking by writing `//-RAC@ assert ...`

### 9.1.3 Finding specification files and the refine statement

JML allows specifications to be placed directly in the `.java` files that contain the implementation of methods and classes. Indeed, specifications such as assert statements or loop invariants are necessarily placed directly in a method body. Other specifications, such as class invariants and method pre- and post-conditions, may be placed in auxiliary files. For classes which are only present as `.class` files and not as `.java` files, the auxiliary file is a necessity.

Current JML allows one such auxiliary file per `.java` file or corresponding `.class` file. It is similar to the corresponding `.java` file except that

- it has a `.jml` suffix
- it contains no method bodies (method declarations are terminated with semi-colons, as if they were abstract)
- TBD - field initializations?

The `.jml` file must be in the same package as the corresponding `.java` file and has the same name, except for the suffix. It need not be in the same folder, though the tail of the path to the folder containing the `.jml` file must still correspond to the package containing the `.java` and `.jml` files. If there is no source file, then there is a `.jml` file for each compilation unit that has a specification. All the nested, inner, or top-level classes that are defined in one Java compilation unit will have their specifications in one corresponding `.jml` file.

The search for specification files is analogous to the way in which `.class` files are found on the *classpath*, except that the *specspath* is used instead. To find the specifications for a public top-level class *T*:

- look in each element of the *specspath* (cf. section TBD), in order, for a fully-qualified file whose name is *T.jml*. If found, the contents of that file are used as the specifications of *T*.
- if no such `.jml` file is found, look in each element of the *specspath*, in order, for a fully-qualified file whose name is *T.java*.

There are two (silent) consequences of this search algorithm that can be confusing:

- If both a `.jml` and a `.java` file exist on the *specspath* and both contain JML specification text, the specifications in the `.java` file will be (silently) ignored.
- If a `.java` file is listed on the command-line it will be compiled (for its Java content), but if it is not a member of an element of the *specspath*, it will (silently) not be used as the source of specifications for itself.

**Obsolete syntax.** The `refine` and `refines` statements are no longer recognized. The previous (complicated) method of finding specification files and merging the specifications from multiple files is also no longer implemented. The only specification file suffix allowed is `.jml`; the others — `.spec`, `.refines-java`, `.refines-spec`, `.refines-jml` — are no longer implemented.

In addition, the `.jml` file is now sought before seeking the `.java` file; if a `.jml` file is found anywhere in the *specs path*, then any specifications in the `.java` file are ignored. This is a different search algorithm than was previously used.

### 9.1.4 Model import statements

*This section will be added later.* Java `import` statements introduce class names into the namespace of a `.java` file. JML has a `model import` statement:

```
//@ model import ...
```

The effect of a JML `model import` statement is the same as a Java `import` statement, except that the names imported by the JML statement are only visible within JML annotations. If the `model import` statement is within a `.jml` file, the imported names are visible only within annotations in the `.jml` file, and not outside JML annotations and not in the `.java` file.

*Note:* Most tools only approximately implement this feature. For example, see FIXME for a discussion of this feature in OpenJML.

### 9.1.5 Modifiers

Modifiers are JML keywords that specify JML characteristics of methods, classes, fields, or variables. They are syntactically placed just like Java modifiers, such as `public`.

*C*heck the table; *a*dd section references; *a*dd where allowed; *i*ndicate which are type modifiers; *t*urn headings 90 degrees.

**Obsolete syntax.** JML no longer defines the modifier `weakly`.

JML Keyword	Java annotation	class	interface	method	field declaration	variable declaration
code	Code			X		
code_bigint_math	CodeBigintMath					
code_java_math	CodeJavaMath					
code_safe_math	CodeSafeMath					
extract	Extract					
ghost	Ghost				X	X
helper	Helper			X		
instance	Instance					
model	Model					
monitored	Monitored					
non_null	NonNull			X	X	X
non_null_by_default	NonNullByDefault	X	X	X		
nullable	Nullable			X	X	X
nullable_by_default	NullableByDefault	X	X	X		
peer	Peer					
pure	Pure	X	X	X		
query	Query					
readonly	Readonly					
rep	Rep					
secret	Secret					
spec_bigint_math	SpecBigintMath					
spec_java_math	SpecJavaMath					
spec_protected	SpecProtected					
spec_public	SpecPublic					
spec_safe_math	SpecSafeMath					
static	Static					
uninitialized	Uninitialized					

Table 9.1: Summary of JML modifiers. All Java annotations are in the `org.jmlspecs.annotation` package.

### **9.1.6 Method specification clauses**

*This section will be added later.*

### **Behaviors**

### **9.1.7 Class specification clauses**

*This section will be added later.*

### **9.1.8 Visibility of specifications**

*This section will be added later.*

### **9.1.9 Statement specifications**

*This section will be added later.*

JML defines some statements that are used in the body of a method's implementation. These are not method specifications per se; rather, they are assertions or assumptions that are used to aid the proof of the specifications themselves, in the way that lemmas are aids to proving a resulting theorem. They can also be used to state predicates that the user believes to be true, and wants checked, or assumptions that are true but are too difficult for the prover to prove itself.

#### **JML assert statement**

*This section will be added later.*

#### **Java assert statement**

*This section will be added later.*

#### **JML assume statement**

*This section will be added later.*

#### **Ghost declaration**

*This section will be added later.*

#### **set statement**

*This section will be added later.*



**debug statement**

*This section will be added later.*

**loop invariants**

*This section will be added later.*

**loop variants**

*This section will be added later.*

**loop assignable declarations**

*This section will be added later.*

**Refining statement specifications**

*This section will be added later.*

**9.1.10 JML expressions**

Expressions in JML annotations are Java expressions with three adjustments:

- Expressions with side-effects are not allowed. Specifically, JML excludes
  - the ++ and - pre- and post- increment and decrement operations
  - the assignment operator
  - assignment operators that combine an operation with assignment (e.g., +=)
  - method invocations that are not explicitly declared pure (cf. §TBD)
- JML adds additional operators to the Java set of operators, discussed in subsection §9.1.10 below.
- JML adds specific keywords that are used as constants or function-like expressions within JML expressions, discussed in subsection §9.1.10 below

**JML operators**

Table ?? lists all of the Java and JML operators, in order of precedence. The JML operators have identified by a comment and a reference to a subsection describing them. All of the JML keyword expressions in §9.1.10 are primitives with precedence higher than any operator.

Most operators are left-associative, if associativity is applicable. The exceptions are TBD...

new () [] . and method calls		
unary + unary - ! (typecast)	-	
* / %	L	
+ (binary) - (binary)	L	
« » »>	L	
< <= > >= <: instanceof <# <#=	-	<: is the JML subtype operation (§9.1.10); <# and <#= are lock ordering operators (§9.1.10)
== !=	L	
&	L	
^	L	
	L	
&&	L	
	L	
==> <==	?	JML implies and reverse implies (§9.1.10)
<==> <!=>	?	JML equivalence and inequivalence (§9.1.10)
?:	-	
= *= /= %= += -= <= >= >>= &= &=  =	L	Java only

Table 9.2: Java and JML operators, in order of precedence, from highest (most tightly binding) to lowest precedence. Operators on the same line have the same precedence. The associativity is given in the central column.

**JML implies (==>) and reverse implies (<==)** The ==> and <== are the JML implication and reverse implication operators. They are short circuit operations taking boolean operands with these equivalences:

- $p ==> q$  is equivalent to  $!p \ || \ q$
- $p <== q$  is equivalent to  $p \ || \ !q$

Note that because of the short-circuit characteristic,  $p <== q$  is not quite equivalent to  $q ==> p$

**JML equivalence (<==>) and inequivalence (<!=>)** The <==> and <!=> are the JML equivalence and inequivalence operators. They are equivalent to == and != except that they take only boolean operands and have much lower precedence.

**JML subtype operator (<:)** In JML expressions, <: denotes the subtype operation among classes and interfaces; the operands both have type \TYPE (cf. §??). Note that the subtype operation (despite not including the = character) includes both type equality and proper subtypes. Note also that in JML, types can express the full parameterized type, not just the erased type in runtime Java. More discussion of JML types is found in §??.

**JML lock ordering operators (<#) and <#=** ) The lock ordering operators are used to determine ordering among objects used for locking in a multi-threaded application; the operands are any Java objects. The only predefined property of these operators is that for any two object references o and oo,  $o <# oo$  is equivalent to  $o == oo \ || \ o <# oo$ ; that is <# is like less than and <#= is like less-than-or-equals. There is no predefined ordering among objects. The user must define an intended ordering with some axioms or invariants. An example of using the lock ordering operators for specification and reasoning about concurrency is found in §??.

TBD - add ++ – into the table as Java only; check precedence

## JML expression keywords

*This section will be added later.*

**Referring to the result of a method:** `\result`

**informal specification** . In some situations a specification needs to be expressed informally, in natural language, perhaps in anticipation of a formal expression or because a formal expression is too complex. The original JML informal expression is

`(* ... *)` where the text between the delimiters is any natural language text not including the characters `*`. An alternate expression (cf. §8.10) is the form `JML.informal(...)`, where the argument is a `"`-delimited String, as permitted by Java.

In both cases the expression always has the value `true`.

**Advanced quantifiers:** `\max` `\min` `\num_of` `\product` `\sum`

**Evaluating expressions in previous program states:** `\old`, `\pre`, `\past`

**Newly allocated objects:** `\fresh`

**Type expressions:** `\type`, `\typeof`, `\elemtype`

**Loop expressions:** `\index and` `\values`

`\not_modified`

`\nonnulllements`

`\not_assigned`, `\only_accessed`, `\only_assigned`, `\only_called`, `\only_captured` TBD -  
reach, duration, space, working\_space

TBD - lockset, max

TBD - is\_initialized, invariant\_for

TBD - lblneg, lblpos, lbl (JML extension)

TBD - reach (object sets?)

### 9.1.11 JML types

Specifications are sometimes best written using infinite-precision mathematical types, rather than the fixed bit-width types of Java. JML's arithmetic modes (§??) allow choosing among various numerical precisions. In this section we simply note the type names that JML defines.

All of the Java type names are legal and useful in JML: `int` `short` `long` `byte` `char` `boolean` `double` `real` and class and interface types. In addition, JML defines the following:

- `\bigint` - the type of infinite-precision integers, represented as `java.lang.BigInteger` during run-time checking
- `\real` - the type of mathematical real numbers, represented as `TBD` during runtime-checking
- `\TYPE` - the type of JML type objects

The familiar operators are defined on values of the `\bigint` and `\real` types: unary and binary `+` and `-`, `*`, `/`, `%`. Also, these types can be used in quantified expressions and variables of these types can be declared as ghost or model variables.

The set of `\TYPE` values includes non-generic types such as `\type(org.lang.Object)`, fully parameterized generic types, such as `\type(org.utils.List<Integer>)`, and primitive types, such as `\type(int)`. The subtype operator (`<:`) is defined on values of type `\TYPE`.

TBD - what about other constructors or accessors of `\TYPE` values

### 9.1.12 Non-Null and Nullable

*This section will be added later.*

### 9.1.13 Observable purity: `\query` and `\secret`

*This section will be added later.*

### 9.1.14 Race condition detection

*This section will be added later.*

### 9.1.15 Arithmetic modes

*This section will be added later.*

### 9.1.16 Universe types

*This section will be added later.*

### 9.1.17 Dynamic frames

*This section will be added later.*

### **9.1.18 Code contracts**

*This section will be added later.*

### **9.1.19 redundantly suffixes**

*This section will be added later.*

### **9.1.20 nowarn lexical construct**

*This section will be added later.*

## **9.2 Interaction with Java features**

*This section will be added later.*

## **9.3 Other issues**

### **9.3.1 Interaction with JSR-308**

*This section will be added later.*

### **9.3.2 Interaction with FindBugs**

*This section will be added later.*

## **Part III**

# **The OpenJML tool**

# Chapter 10

## Introduction

### 10.1 OpenJML

OpenJML is a tool for processing Java Modeling Language (JML) specifications of Java programs. The tool parses and type-checks the specifications and performs static or run-time checking of the validity of the specifications. Other tools are anticipated, such as test case generation and an enhanced version of javadoc that includes the specifications in the javadoc documentation.

The functionality is available

- as a command-line tool to do type-checking, static checking or runtime checking,
- as an Eclipse plug-in to perform those tasks, and
- programmatically from a user's Java program

OpenJML uses program specifications written in JML, the Java Modeling Language, and it is constructed by extending OpenJDK, the open source Java compiler. OpenJML currently requires Java 1.7 to run. Releases of Java can be obtained from the Oracle release site (<http://www.oracle.com/technetwork/java/javase/downloads>).

The source code for OpenJML is kept in SourceForge as a module of the JML Project (<https://sourceforge.net/p/jmlspecs/code/HEAD/tree/>). The JMLAnnotations and Specs projects (also modules of JML) are used by OpenJML. Information about creating a developer's environment for the OpenJML source can be found below (section 10.1.3).

#### 10.1.1 Command-line tool

The OpenJML command line tool can be downloaded from <http://jmlspecs.sourceforge.net/openjml.tar.gz>.

The command line tool is described in chapter 11.

#### 10.1.2 Eclipse plug-in

The Update site for the Eclipse plug-in that encapsulates the OpenJML tool is <http://jmlspecs.sourceforge.net/openjml-updatesite>.

The plug-in is described in section 12 and in the online documentation available in Eclipse Help.

### 10.1.3 Development of OpenJML

Developers wishing to contribute to OpenJML can retrieve a project-set file to download source code from SVN and create the corresponding projects within Eclipse from <http://jmlspecs.sourceforge.net/OpenJML-projectSet.psf>. <p>Alternately, the set of SVN commands needed to checkout all the pieces of the OpenJML source code into the directory structure expected by Eclipse is found at this link: [http://jmlspecs.sourceforge.net/svn\\_commands](http://jmlspecs.sourceforge.net/svn_commands).

The general instructions for setting up a development environment are found at the JML wiki: <https://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJmlSetup>.

## 10.2 JML

The Java Modeling Language (JML) is a language that enables logical assertions to be made about Java programs. The assertions are expressed as structured Java comments or Java annotations. Various tools can then read the JML information and do static checking, runtime checking, display for documentation, or other useful tasks.

More information about JML can be found on the JML web site: <http://www.jmlspecs.org>. The information includes publications, a list of groups using or contributing to JML, mailing lists, etc. There is also a SourceForge project for JML : <https://sourceforge.net/projects/jmlspecs/>.

## 10.3 OpenJDK

OpenJDK (<http://openjdk.net>) is the project that produces the Java JDK and JRE releases. OpenJML extends OpenJDK to produce the OpenJML tools. OpenJML is a fully encapsulated, stand-alone tool, so the OpenJDK foundation is only of interest to OpenJML developers. Users, however, can be assured that OpenJML is built on 'official' Java tooling and can readily stay up to date with changes in the Java language.

## 10.4 License

The OpenJML command-line tool is built from OpenJDK, which is licensed under GPLv2 (<http://openjdk.java.net/legal/>). Hence OpenJML is correspondingly licensed.

The OpenJML plug-in is a pure Eclipse plug-in, and therefore is not required to be licensed under the EPL.

The source code for both tools is available as a SourceForge project at <https://sourceforge.net/p/jmlspecs/code/HEAD/tree/OpenJML/trunk>.



# Chapter 11

## The command-line tool

### 11.1 Installation and System Requirements

The command-line tool is supplied as a .zip or a .tar.gz file, downloadable from <http://jmlspecs.sourceforge.net/>. Download the file to a directory of your choice and unzip or untar it in place. It contains the following files:

- openjml.jar - the main jar file for the application
- jmlruntime.jar - a library needed on the classpath when running OpenJML's runtime-assertion-checking
- jmlspecs.jar - a library containing specification files
- openjml-template.properties - a sample file, which should be copied and renamed openjml.properties, containing definitions of properties whose values depend on your local system
- LICENSE.rtf - a copy of the modified GPL license that applies to OpenJDK and OpenJML
- OpenJMLUserGuide.pdf - this document

You must run OpenJML in a Java 1.7 JRE; it is not compatible with Java 1.8 or 1.9.

You should ensure that the jmlruntime.jar and jmlspecs.jar files remain in the same folder as the openjml.jar file.

In addition to OpenJML itself, you will also need a SMT solver if you intend to use the static checking capability of OpenJML.

TBD - more on installing SMT solvers

### 11.2 Running OpenJML

To run OpenJML, be sure that the java command uses a 1.7 JVM and use the following command line. Here *\$OPENJML* designates the folder in which the openjml.jar file resides.

```
java -jar $OPENJML/openjml.jar <options> <files>
```

Here *<files>* and *<options>* stand for text described below.

The following command is currently a viable alternative as well.

```
java -cp $OPENJML/openjml.jar org.jmlspecs.openjml.Main <options> <files>
```

The valid options are listed in Table 11.1 and are described in subsections below. Options and files can appear in any order.

### 11.2.1 Exit values

When OpenJML runs as a command-line tool, it emits one of several values on exit:

- 0 (EXIT\_OK) : successful operation, no errors, there may be warnings (including static checking warnings)
- 1 (EXIT\_ERROR) : normal operation, but with errors (parsing or type-checking)
- 2 (EXIT\_CMDERR) : an error in the formulation of the command-line, such as invalid options
- 3 (EXIT\_SYSERR) : a system error, such as out of memory
- 4 (EXIT\_ABNORMAL) : a fatal error, such as a program crash or internal inconsistency, caused by an internal bug

In the itemized list above, the symbolic names are defined in `org.jmlspecs.openjml.Main`. For example, when executing OpenJML programmatically (cf. section TBD), the user's Java program should use the symbol `org.jmlspecs.openjml.Main.EXIT_ERROR`, rather than a specific integer.

Compiler warnings and static checking warnings will be reported as errors if the `-Werror` option is used. This may change an EXIT\_OK result to an EXIT\_ERROR result.

### 11.2.2 Files

In the command templates above, `<files>` refers to a list of `.java` or `.jml` files. Each file must be specified with an absolute file system path or with a path relative to the current working directory (in particular, not with respect to the classpath or the sourcepath).

You can also specify directories on the command line using the `-dir` and `-dirs` options. The `-dir <directory>` option indicates that the `<directory>` value (an absolute or relative path to a folder) should be understood as a folder; all `.java` or specification files recursively (TBD - check recursively) within the folder are included as if they were individually listed on the command-line. The `-dirs` option indicates that each one of the remaining command-line arguments is interpreted as either a source file (if it is a file with a `.java` or `.jml` suffix) or as a folder (if it is a folder) whose contents are processed as if listed on the command-line. Note that the `-dirs` option must be the last option.

As described later in section 11.2.3, JML specifications for Java programs can be placed either in the `.java` files themselves or in auxiliary `.jml` files. The format of `.jml` files is defined by JML. OpenJML can type-check `.jml` files as well as `.java` files if they are placed on the command-line. Doing so can be useful to check the syntax in a specific `.jml` file, but is usually not necessary: when a `.java` file is processed by OpenJML, the corresponding `.jml` file is automatically found (cf. ??).

### 11.2.3 Specification files

JML specifications for Java classes (either source or binary) are written in files with a `.jml` suffix or are written directly in the source `.java` file. When OpenJML needs specifications for a given class, it looks for a `.jml` file on the `specspath`. If one is not found, OpenJML then looks for a `.java` file on the

Options specific to JML	
-	no more options
-check	[11.2.6] typecheck only (-command check)
-checkSpecsPath	[11.2.6] warn about non-existent specs path entries
-command <action>	[11.2.6] which action to do: check esc rac compile
-compile	[11.2.6] TBD
-counterexample	[11.2.6] show a counterexample for failed static checks
-dir <dir>	[11.2.6] argument is a folder or file
-dirs	[11.2.6] remaining arguments are folders or files
-esc	[11.2.6] do static checking (-command esc)
-internalRuntime	[11.2.6] add internal runtime library to classpath
-internalSpecs	[11.2.6] add internal specs library to specspath
-java	[11.2.6] use the native OpenJDK tool
-jml	[11.2.6] process JML constructs
-jmldebug	[11.2.7] very verbose output (includes -progress)
-jmlverbose	[11.2.7] JML-specific verbose output
-keys	[11.2.6] define keys for optional annotations
-method	
-nonnullByDefault	[11.2.6] values are not null by default
-normal	[11.2.7]
-nullableByDefault	[11.2.6] values may be null by default
-progress	[11.2.7]
-purityCheck	[11.2.6] check for purity
-quiet	[11.2.7] no informational output
-rac	[11.2.6] compile runtime assertion checks (-command rac)
-racCheckAssumptions	[11.2.7] enables (default on) checking assume statements as if they were asserts
-racCompileToJavaAssert	[11.2.7] compile RAC checks using Java asserts
-racJavaChecks	[11.2.7] enables (default on) performing JML checking of violated Java features

JML options, continued	
-racShowSource	[11.2.7] includes source location in RAC warning messages
-showNotImplemented	warn if feature not implemented
-specspath	[11.2.6] location of specs files
-stopIfParseErrors	stop if there are any parse errors
-subexpressions	[11.2.6] show subexpression detail for failed static checks
-trace	[11.2.6] show a trace for failed static checks

Options inherited from Java	
-Akey	
-bootclasspath <path>	See Java documentation.
-classpath <path>	location of class files
-cp <path>	location of class files
-d <directory>	location of output class files
-encoding <encoding>	
-endorsedirs <dirs>	
-extdirs <dirs>	
-deprecation	
-g	
-help	output help information
-implicit	
-J<flag>	
-nowarn	show only errors, no warnings
-proc	
-processor <classes>	
-processorpath <path>	where to find annotation processors
-s <directory>	location of output source files
-source <release>	the Java version of source files
-sourcepath <path>	location of source files
-target <release>	the Java version of the output class files
-X	Java non-standard extensions
-verbose	verbose output
-version	output (OpenJML) version
-Werror	treat warnings as errors

Table 11.1: OpenJML options. See the text for more detail on each option.

specspath. Note that this rule requires that source files (that have specifications you want to use) must be listed on the specspath. Note also that there need not be a source file; a `.jml` file can be (and often is) used to provide specifications for class files.

Previous versions of JML had a more complicated scheme for constructing specifications for a class involving refinements, multiple specification files, and various prefixes. This complicated process is now deprecated and no longer supported.

[ TBD: some systems might find the first `.java` or `.jml` file on the specspath and use it, even if there were a `.jml` file later.]

## 11.2.4 Annotations and the runtime library

JML uses Java annotations as introduced in Java 1.6. Those annotation classes are in the package `org.jmlspecs.annotation`. In order for files using these annotations to be processed by Java, the annotation classes must be on the classpath. They may also be required when a compiled Java program that uses such annotations is executed. In addition, running a program that has JML runtime assertion checks compiled in will require the presence of runtime classes that define utility functions used by the assertion checking code.

Both the annotation classes and the runtime checking classes are provided in a library named `jmlruntime.jar`. The distribution of OpenJML contains this library, as well as containing a version of the library within `openjml.jar`. When OpenJML is applied to a set of classes, by default it finds a version of the runtime classes and appends the location of the runtime classes to the classpath.

You can prevent OpenJML from automatically adding `jmlruntime.jar` to the classpath with the option `-noInternalRuntime`. If you use this option, then you will have to supply your own annotation classes and (if using Runtime Assertion Checking) runtime utility classes on the classpath. You may wish to do this, for example, if you have newer versions of the annotation classes that you are experimenting with. You could simply put them on the classpath, since they would be in front of the automatically added classes and used in favor of default versions; however, if you want to be sure that the default versions are not present, use the `-noInternalRuntime` option.

The symptom that no runtime classes are being found at all is error messages that complain that the `org.jmlspecs.annotation` package is not found.

## 11.2.5 Java properties and the `openjml.properties` file

OpenJML uses a number of properties that may be defined in the environment; these properties are typically characteristics of the local environment that vary among different users or different installations. They can also be to set default values of options, so they do not need to be set on the command-line. An example is the file system location of a particular solver.

The tool looks for a file named `openjml.properties` in several locations. It loads the properties it finds in each of these, in order, so later definitions will supplant earlier ones.

- System properties, including those defined with `-D` options on the command-line
- On the system classpath
- In the users home directory (the value of the Java property `user.home`)
- In the current working directory (the value of the Java property `user.dir`)

[TBD: Check the above]

The properties that are currently recognized are these:

- `openjml.defaultProver` - the value is the name of the prover to use by default
- `openjml.prover.<name>`, where `<name>` is the name of a prover, and the value is the file system path to the executable to be invoked for that prover

[TBD: Check the above]

The distribution includes a file named `openjml-template.properties` that contains stubs for all the recognized options. You should copy that file, rename it as `openjml.properties`, and edit it to reflect your system configuration. (If you are an OpenJML developer, take care not to commit your system's `openjml.properties` file into the OpenJML shared SVN repository.)

## 11.2.6 OpenJML options

There are many options that control or modify the behavior of OpenJML. Some of these are inherited from the Java compiler on which OpenJML is based. Options for the command-line tool are expressed as standard command-line options. In the Eclipse GUI, the values of options are set on a typical Eclipse preference or properties page for OpenJML.

The command-line options follow the style of the OpenJDK compiler — they begin with a single hyphen and there are no two-hyphen versions. OpenJML (but not OpenJDK) options that require a parameter may either use an `=` followed directly by the argument with no whitespace or may provide the argument as the subsequent entry of the argument list. For example, either `-racbin=output` or `-racbin output` is permitted. If the argument is optional but present, the `=` form must be used. Values of options that contain whitespace must be quoted as appropriate for the operating system being used.

Options that are boolean in nature can be enabled and disabled by either

- adding a prefix `-no`, as in `-showRacSource` and `-no-showRacSource`
- or using the `=` form, as in `-showRacSource=true` and `-showRacSource=false`

### Options: Operational modes

These operational modes are mutually exclusive.

- **-jml** (default) : use the OpenJML implementation to process the listed files, including embedded JML comments and any `.jml` files
- **-no-jml**: uses the OpenJML implementation to type-check and possibly compile the listed files, but ignores all JML annotations in those files
- **-java**: processes the command-line options and files using only OpenJDK functionality. No OpenJML functionality is invoked. Must be the first option and overrides the others.

### Options: JML tools

The following mutually exclusive options determine which OpenJML tool is applied to the input files. They presume that the `-jml` mode is in effect.

- **-command <tool>** : initiates the given function; the value of `<tool>` may be one of `check`, `esc`, `rac`, `TBD`. The default is to use the OpenJML tool to do only typechecking of Java and JML in the source files.
- **-check** : causes OpenJML to do only type-checking of the Java and JML in the input files (alias for `-command=check`)
- **-compile** : TBD

- **-esc** : causes OpenJML to do (type-checking and) static checking of the JML specifications against the implementations in the input files (alias for `-command=esc`)
- **-rac** : compiles the given Java files as OpenJDK would do, but with JML checks included for checking at runtime (alias for `-command=rac`)
- **-doc** : executes javadoc but adds JML specifications into the javadoc output files. (alias for `-command=doc`) *Not yet implemented.*

### Options: Finding files and classes: class, source, and specs paths

A common source of confusion is the various different paths used to find files, specifications and classes in OpenJML. OpenJML is a Java application and thus a *classpath* is used to find the classes that constitute the OpenJML application; but OpenJML is also a tool that processes Java files, so it uses a (different) classpath to find the files that it is processing. As is the case for other Java applications, a *<path>* contains a sequence of individual paths to folders or jar files, separated by the path separator character (a semicolon on Windows systems and a colon on Unix and MacOSX systems). You should distinguish the following:

- the classpath used to run the application: specified by one of
  - the CLASSPATH environment variable
  - the .jar file given with the `java -jar` form of the command is used
  - the value for the `-classpath` (equivalently, `-cp`) option when OpenJML is run with the `java -cp openjml.jar org.jmlspecs.openjml.Main` command

This classpath is not of much concern to OpenJML, but is the classpath that Java users will be familiar with. The value is implicitly given in the `-jar` form of the command. The application classpath is explicitly given in the alternate form of the command, and it may be omitted; if it is omitted, the value of the system property CLASSPATH is used and it must contain the `openjml.jar` library.

- the classpath used by OpenJML. This classpath determines where OpenJML will find .class files for classes referenced by the .java files it is processing. The classpath is specified by

`-classpath <path>`

or

`-cp <path>`

after the executable is named on the commandline. That is,

```
java -jar openmjml.jar -cp <openjml-classpath> ...
```

or

```
java -cp openjml.jar org.jmlspecs.openjml.Main -cp <openjml-classpath> ...
```

If the OpenJML classpath is not specified, its value is the same as the application classpath.

- the OpenJML sourcepath - The sourcepath is used by OpenJML as the list of locations in which to find .java files that are referenced by the files being processed. For example, if a file on the command-line, say `T.java`, refers to another class, say `class U`, that is not listed on the command-line, then `U` must be found. OpenJML (just as is done by the Java compiler) will look for a source file for `U` in the sourcepath and a class file for `U` in the classpath. If both are found then TBD.

The OpenJML sourcepath is specified by the `-sourcepath <path>` option. If it is not specified, the value for the sourcepath is taken to be the same as the OpenJML classpath.

In fact, the sourcepath is rarely used. Users often will specify a classpath containing both .class and .java files; by not specifying a sourcepath, the same path is used for both .java and .class files. This is simpler to write, but does mean that the application must search through all source and binary directories for any particular source or binary file.

- the OpenJML specspath - The specspath tells OpenJML where to look for specification (.jml) files. It is specified with the `-specspath <path>` option. If it is not specified, the value for the specspath is the same as the value for the sourcepath. In addition, by default, the specspath has added to it an internal library of specifications. These are the existing (and incomplete) specifications of the Java standard library classes.

The addition of the Java specifications to the specspath can be disabled by using the `-noInternalSpecs` option. For example, if you have your own set of specification files that you want to use instead of the internal library, then you should use the `-noInternalSpecs` option and a `-specspath` option with a path that includes your own specification library.

Note also that often source (.java) files contain specifications as well. Thus, if you are specifying a specspath yourself, you should be sure to include directories containing source files in the specspath; this rule also includes the .java files that appear on the command-line: they also should appear on the specspath.

TBD - describe what happens if the above guidelines are not followed. (Can we make this more user friendly).

**The `-noInternalSpecs` option.** As described above, this option turns off the automatic adding of the internal specifications library to the specspath. If you use this option, it is your responsibility to provide an alternate specifications library for the standard Java class library. If you do not you will likely see a large number of static checking warnings when you use Extended Static Checking to check the implementation code against the specifications.

The internal specifications are written for programs that conform to Java 1.7. [ TBD - change this to adhere to the `-source` option?] [TBD - what about the specs in jmlspecs for different source levels.]

### Options: OpenJML options applicable to all OpenJML tools

- **`-dir <folder>`** : abbreviation for listing on the command-line all of the .java files in the given folder, and its subfolders; if the argument is a file, use it as is
- **`-dirs`** : treat all subsequent command-line arguments as if each were the argument to `-dir`
- **`-specspath <path>`** : defines the specifications path, cf. section TBD
- **`-keys <keys>`** : the argument is a comma-separated list of options JML keys (cf. section TBD)
- **`-strictJML`** : warns about an OpenJML extensions to standard JML
- **`-nullableByDefault`** : sets the global default to be that all declarations are implicitly `@Nullable`
- **`-nonnullByDefault`** : sets the global default to be that all declarations are implicitly `@NonNull` (the default)
- **`-purityCheck`** : turns on (default is on) purity checking (recommended since the Java library specifications are not complete for `@Pure` declarations)

## Options: Extended Static Checking

These options apply only when performing ESC:

- **-prover** *<prover>* : the name of the prover to use: one of `z3_4_3`, `cvc4`, `yices2`
- **-exec** *<file>* : the path to the executable corresponding to the given prover
- **-boogie** : enables using boogie (-prover option ignored; -exec must specify the Z3 executable for Boogie to use)
- **-method** *<methodlist>* : a comma-separated list of method names to check (default is all methods in all listed classes)
- **-exclude** *<methodlist>* : a comma-separated list of method names to exclude from checking
- **-checkFeasibility** *<where>* : checks feasibility of the program at various points — a comma-separated list of one of `none`, `all`, `exit` [TBD, finish list, give default]
- **-escMaxWarnings** *<int>* : the maximum number of assertion violations to look for; the argument is either a positive integer or `All`; the default is `All`
- **-counterexample** : prints out a counterexample for failed proofs
- **-trace** : prints out a counterexample trace for each failed assert (includes -counterexample)
- **-subexpressions** : prints out a counterexample trace with model values for each subexpression (includes -trace)

## 11.2.7 Options: Runtime Assertion Checking

These options apply only when doing RAC:

- **-showNotExecutable** : warns about the use of features that are not executable (and thus ignored); turn off with `-no-showNotExecutable`
- **-showRacSource** : enables including source code information in RAC error messages (default is enabled; disable with `-no-showRacSource`)
- **-racCheckAssumptions** : enables checking assume statements as if they were asserts (default is enabled; disable with `-no-racCheckAssumptions`)
- **-racJavaChecks** : enables performing JML checking of violated Java features (which will just proceed to throw an exception anyway) (default is enabled; disable with `-no-racJavaChecks`)
- **-racCompileToJavaAssert** : compile RAC checks using Java asserts (which must then be enabled using `-ea`) (default is disabled; disable with `-no-racCompileToJavaAssert`)

## Options: Version of Java language or class files

- **-source** *<level>* : this option specifies the Java version of the source files, with values of `1.4`, `1.5`, `1.6`, `1.7...` or `4`, `5`, `6`, `7`, ... . This controls whether some syntax features (e.g. annotations, extended for-loops, autoboxing, enums) are permitted. The default is the most recent version of Java, in this case `1.7`. Note that the classpath should include the Java library classes that correspond to the source version being used.
- **-target** *<level>* : this option specifies the Java version of the output class files



## Options: Other Java compiler options applicable to OpenJML

All the OpenJDK compiler options apply to OpenJML as well. The most commonly used or important OpenJDK options are listed here.

These options control where output is written:

- **-d <dir>** : specifies the directory in which output class files are placed; the directory must already exist
- **-s <dir>** : specifies the directory in which output source files are placed; such as those produced by annotation processors; the directory must already exist

These are Java options relevant to OpenJML whose meaning is unchanged in OpenJML.

- **-cp** or **-classpath**: the parameter gives the classpath to use to find unnamed but referenced class files (cf. section TBD)
- **-sourcepath**: the parameter gives the sequence of directories in which to find source files for unnamed but referenced classes (cf. section TBD)
- **-deprecation**: enables warnings about the use of deprecated features (applies to deprecated JML features as well)
- **-nowarn**: shuts off all compiler warnings, *including the static check warnings produced by ESC*
- **-Werror**: turns all warnings into errors, including JML (and static check) warnings
- **@filename**: the given *filename* contains a list of arguments
- **-source**: specifies the Java version to use (default 1.7)
- **-verbose**: turn on Java verbose output
- **-Xprefer:source** or **-Xprefer:newer**: when both a .java and a .class file are present, whether to choose the .java (source) file or the file that has the more recent modification time [ TBD - check that this works ]
- **-stopIfParseErrors**: if enabled (disabled by default), processing stops after parsing if there are any parsing errors (TBD - check this, describe the default)

Other Java options, whose meaning and use is unchanged from javac:

- **@<filename>** : reads the contents of <filename> as a sequence of command-line arguments (options, arguments and files)
- **-Akey**
- **-bootclasspath**
- **-encoding**
- **-endorsedirs**
- **-extdirs**
- **-g**
- **-implicit**
- **-J**
- **-nowarn** : only print errors, not warnings, *including not printing static check warnings*
- **-Werror** : turns all warnings into errors
- **-X...** : Java's extended options

These Java options are discussed elsewhere in this document:

- **-cp** *<path>* or **-classpath** *<path>* : section 11.2.6
- **-sourcepath** *<path>* : section 11.2.6
- **-verbose** : section 11.2.7
- **-source** :
- **-target** :

### Options: Information and debugging

These options print summary information and immediately exit (despite the presence of other command-line arguments):

- **-help** : prints out help information about the command-line options
- **-version** : prints out the version of the OpenJML tool software

The following options provide different levels of verbosity. If more than one is specified, the last one present overrides earlier ones.

- **-quiet** : no informational output, only errors and warnings
- **-normal** : (default) some informational output, in addition to errors and warnings
- **-progress** : prints out summary information as individual files are processed (includes -normal)
- **-verbose** : prints out verbose information about the Java processing in OpenJDK (does not include other OpenJML information)
- **-jmlverbose** : prints out verbose information about the JML processing (includes -verbose and -progress)
- **-jmldebug** : prints out (voluminous) debugging information (includes -jmlverbose)
- **-verbosity** *<int>* : sets the verbosity level to a value from 0 - 4, corresponding to -quiet, -normal, -progress, -jmlverbose, -jmldebug

Other debugging options:

- **-show** : prints out rewritten versions of the Java program files for informational and debugging purposes
- **-showNotImplemented** : prints warnings about JML features that are ignored because they are not implemented; disable with -no-showNotImplemented; the default is disabled.

An option used primarily for testing:

- **-jmltesting**: adjusts the output so that test output is more stable

### 11.2.8 Options related to Static Checking

- -counterexample
- -trace
- -subexpressions
- -method

### 11.2.9 Options related to parsing and typechecking

- -Werror
- -nowarn
- -stopIfParseError
- -checkSpecsPath
- -purityCheck
- -nonnullbydefault
- -nullablebydefault
- -keys

### 11.2.10 Java options related to annotation processing

- -proc
- -processor
- -processorpath

### 11.2.11 Other JML Options

- -roots

#### General options

- **-dir**: Indicates that its argument is a directory. All the .java and .jml files in the directory and its subdirectories are processed. (TBD - is this necessary?)
- **-dirs**: Indicates that all subsequent command-line arguments are directories, to be processed as for -dir, until an argument is reached that begins with a hyphen.
- **-specspath**: the parameter gives the sequence of directories in which to find .jml specification files for unnamed but referenced classes (cf. section TBD)
- **-checkSpecsPath**: if enabled (the default), warns about specspath elements that do not exist
- **-keys**: comma-separated list of the optional JML comment keys to enable (empty by default)
- **-strictJML**: (disabled by default) warns about the use of any OpenJML extensions to standard JML; disable with -no-strictJML
- **-showNotImplemented**: (disabled by default) warns about the use of features that are not implemented; disable with -no-showNotImplemented

*This section will be completed later.*

## Chapter 12

# The Eclipse Plug-in

Since OpenJML operates on Java files, it is natural that it be integrated into the Eclipse IDE for Java. OpenJML provides a conventional Eclipse plug-in that encapsulates the OpenJML command-line tool and integrates it with the Eclipse Java development environment. The plug-in also provides GUI functionality for working with JML specifications.

### 12.1 Installation and System Requirements

Your system must have the following:

- A Java 1.7 JRE as described in section 11.2. This must be the JRE in use in the environment in which Eclipse is invoked. If you start Eclipse by a command in a shell, it is straightforward to make sure that the correct Java JRE is defined in that shell. However, if you start Eclipse by, for example, double-clicking a desktop icon, then you must ensure that the Java 1.7 JRE is set by the system at startup.
- Eclipse 4.2 or later. The OpenJML plug-in is currently tested for the Juno, Kepler, and Luna releases of Eclipse.
- One or more SMT solvers, if the static checking functionality will be used.

Installation of the plug-in follows the conventional Eclipse procedure.

- Invoke the "Install New Software" dialog under the Eclipse "Help" menubar item.
- "Add" a new location, giving the URL `http://jmlspecs.sourceforge.net/openjml-updatesite` and some name of your choice (e.g. OpenJML).
- Select the "OpenJML" category and push "Next"
- Proceed through the rest of the wizard dialogs to install OpenJML.
- Restart Eclipse when asked to obtain full functionality.

If the plug-in is successfully installed, a yellow coffee cup (the JML icon) will appear in the toolbar (along with other menubar/toolbar items). The installation will fail (without obvious error messages), if the underlying Java VM is not a suitable Java 1.7 VM.

## 12.2 GUI Features

Note that the JML logo is a JML-decorated yellow coffee cup; this logo is associated with various GUI elements.

### 12.2.1 Commands

The OpenJML plug-in adds a number of commands. These are visible in the Preferences»General»Keys dialog. All the OpenJML commands explicitly added by the OpenJML plug-in are in the 'JML' category. Some commands are automatically added by Eclipse and are in other categories; for example, Eclipse automatically adds commands to open each individual Preference Page and each kind of View. You can sort the table of commands by category and you can filter the table, in order to show just those commands related to JML. Also, this dialog allows binding a keyboard key-combination to a command, as is the case for all Eclipse commands.

The commands are listed in Table ??, with forward references to more detailed discussion.

- Add to JML specs path (§??). Allows editing the specs path
- Clear All Results (§??). Deletes all results of static checking operations
- Clear Selected Results (§??). Deletes some of the results of static checking
- Delete JML Markers (§??). Deletes all JML markers and highlighting on selected resources
- Disable JML on the project (§??).
- Edit JML Source/Specs Paths (§??).
- Enable JML on the project (§??).
- Generate JML doc (§??).
- insert \result (§??).
- insert .... (§??).
- Open a Specifications Editor (§??).
- Open the ESC Results View (§??).
- RAC - ... (§??).
- Remove from JML specs path (§??).
- Rerun Static Check (§??).
- Show Counterexample (§??).
- Show Counterexample Value (§??).
- Show Detailed Proof Attempt Information (§??).
- Show ESC Result Information (§??).
- Show JML paths (§??).
- Show JML Specifications (§??).
- Static Check (ESC) (§??).

- Typecheck JML (§??). Performs syntax, parsing and typechecking on selected projects, folders, and files.
- Show In ... (§??).
- Show View (OpenJML Static Checks) (§??).
- Show View (OpenJML Trace) (§??).
- Preferences (OpenJML > OpenJML Solvers) (§??). Opens the OpenJML Solvers subpage. Preferences (OpenJML) (§??). Opens the OpenJML Preferences page.

### 12.2.2 Menubar additions

The main Eclipse menubar contains an additional menu titled 'JML'. It contains the following submenu items; typically the action for a menu item is the similarly named command. Menu items are also added in the following Context menus (context menus are available by right-clicking):

- Context menu on elements of the Eclipse Package Explorer View, Project Explorer View, and Navigator View
- Context menu on elements of the Outline View
- Context menu within an editor
- ... Problem View...
- ... OpenJML Proof view ...

Individual menu items may be disabled when they are not applicable. For instance, some items are enabled only when something is selected, some only when exactly one appropriate item is selected, some only when a method is selected, etc.

### 12.2.3 Toolbar additions

The OpenJML plug-in adds three toolbar items; clicking the toolbar item executes a corresponding command.

- the JML coffee cup logo - executes the 'Typecheck JML' command
- ESC - executes the 'Static Check (ESC)' command
- RAC - executes the 'RAC - compile selected' command

### 12.2.4 OpenJML Problems, Markers and highlights

TBD

### 12.2.5 OpenJML console and error log

The plug-in adds an additional type of console. Eclipse has a Console View that manages the consoles for various plug-ins. The OpenJML Console is created automatically when output is generated. General informational output is sent to the console; errors and warnings are placed in the console and shown in pop-up dialog boxes. Egregious errors are also logged in the Eclipse Error Log.

*Note: Currently the OpenJML Console is not listed in the list of types of Consoles available in Console View menu.*

## 12.2.6 Preferences

The plug-in adds dialogs for setting OpenJML options. These are workspace preferences, affecting all projects in the workspace. There are two Preference pages:

- A top-level page named 'OpenJML' found in the top-level list of Eclipse preference pages. This page allows setting options that would otherwise be set on the command-line. There is also one UI option, named 'UI verbose', that enables verbose output to the OpenJML console about actions within the UI code.
- A sub-page named 'Solvers'. (Click the turnstile next to 'OpenJML' in the list of Preference pages to see the subpage). These preferences enable registering SMT solvers and setting the default solver to use.

There are currently no project-level preferences within the OpenJML plug-in.

## 12.2.7 Editor embellishments

TBD - fill out

- counterexample hovers
- quick fix proposals
- context-sensitive completions
- insertions

## 12.2.8 OpenJML Views

TBD

## 12.2.9 Help

There is a 'JML' entry in the table of contents under the Eclipse Help menu item. It provides an online user guide to JML and OpenJML.

*Note - the current information available under Help is outdated and will be replaced by this manual.*

## 12.2.10 Other GUI elements

- OpenJML decoration - A decoration is applied to names of projects in the Package Explorer View for which OpenJML has been enabled (cf. §??). The decoration is a miniature JML logo on the upper-right of the folder icon, covering the 'J' symbol that indicates an Eclipse Java project.
- The plug-in defines a JML Nature and a JML Builder. The Nature is associated with a project precisely when JML is enabled for the project. The Builder performs automatic type-checking. (cf. §??)
- .jml suffix. The plug-in adds a content type that associates the .jml filename suffix with the Java editor. This makes the Java editor the default editor for .jml files.

*Note - so we still get spurious errors on jml files?*

- Internationalization. TBD...
- classpath initializer. TBD ...
- definition as an Eclipse project. TBD ...
- Open JML Perspective. TBD ...



## Chapter 13

# OpenJML tools

### 13.1 Parsing and Type-checking

The foundational function of OpenJML is to parse and check the well-formedness of JML annotations in the context of the associated Java program. Such checking includes conventional type-checking and checking that names are used consistently with their visibility and purity status.

A set of Java files with JML annotations is type-checked with the command

```
java -jar $INSTALL/openjml.jar -check options files
```

or

```
java -jar $INSTALL/openjml.jar options files
```

since `-check` is the default action. The equivalent action in the Eclipse plug-in is the 'Typecheck JML' command, available through the toolbar or menu actions. Any `.jml` files are checked when the associated `.java` file is created. Only `.java` files either listed on the command-line or contained in folders listed on the command-line are certain to be checked. Some checking of other files may be performed where references are made to classes or methods in those non-listed files.

A key concept to understand is how class files, source files, and specification files are found and used by the OpenJML tool. This process is described in the following subsection. The command-line options relevant to parsing and type-checking are discussed in the subsequent subsection.

#### 13.1.1 Classpaths, sourcepaths, and specification paths in OpenJML

When a Java compiler compiles source files, it considers three types of files:

- Source files listed on the command-line
- Other source files referenced by those listed on the command-line, but not on the command-line themselves
- Already-compiled class files

The OpenJML tool considers the same files, but also needs

- Specification files associated with classes in the program

The OpenJML tool behaves in a way similar to a typical Java compiler, making use of three directory paths - the classpath, the sourcepath, and the specspath. These paths are standard lists of directories or jar files, separated either by colons (Unix) or semicolons (Windows). Java packages are subdirectories of these directories.

- **classpath:** The OpenJML classpath is set using one of these alternatives, in priority order:
  - As the argument to the OpenJML command-line option `-classpath`
  - As the value of the Java property `org.jmlspecs.openjml.classpath`
  - As the value of the system environment variable `CLASSPATH`
- **sourcepath:** The OpenJML sourcepath is set using one of these alternatives, in priority order:
  - As the argument of the OpenJML command-line option `-sourcepath`
  - As the value of the Java property `org.jmlspecs.openjml.sourcepath`
  - As the value of the OpenJML classpath (as determined above)
- **specspath:** The OpenJML specifications path is set using one of these alternatives, in priority order:
  - As the argument of the OpenJML command-line option `-specspath`
  - As the value of the Java property `org.jmlspecs.openjml.specspath`
  - As the value of the OpenJML sourcepath (as determined above)

Note that with no command-line options or Java properties set, the result is simply that the system `CLASSPATH` is used for all of these paths. A common practice is to simply use a single directory path, specified on the command-line using `-classpath`, for all three paths.

The paths are used as follows to find relevant files:

- Source files listed on the command-line are found directly in the file system. If the command-line element is an absolute path to a `.java` file, it is looked up in the file system as an absolute path; if the command-line element is a relative path, the file is found relative to the current working directory.
- Classes that are referenced by files on the command-line or transitively by other classes in the program, can be found in one of two ways:
  - The source file for the class is sought as a sub-file of an element of the `sourcepath`.
  - The class file for the class is sought as a sub-file of an element of the `classpath`.

If there is both a sourcefile and a classfile present, then

- if the option `-Xprefer:source` is present, the sourcefile is always recompiled
- if the option `-Xprefer:newer` is present, the sourcefile is recompiled only if its modification timestamp is newer than that of the class file.

The default is to use the newer of the source or class files.

### 13.1.2 Command-line options for type-checking

The following command line options are particularly relevant to type-checking.

- **-nullableByDefault:** sets the global default to be that all variable, field, method parameter, and method return type declarations are implicitly `@Nullable`
- **-nonnullByDefault:** sets the global default to be that all variable, field, method parameter, and method return typedeclarations are implicitly `@NonNull` (the default)
- **-purityCheck:** enables (default on) checking for purity; disable with `-no-purityCheck`
- **-internalSpecs:** enables (default on) using the built-in library specifications; disable with `-no-internalSpecs`
- **-internalRuntime:** enables (default on) using the built-in runtime library; disable with `-no-internalRuntime`

## 13.2 Static Checking and Verification

*This section will be added later.*

### 13.2.1 Options specific to static checking

- **-prover *prover***: the name of the prover to use: one of `z3_4_3`, `yices2` [TBD: expand list]
- **-exec *path***: the path to the executable corresponding to the given prover
- **-boogie**: enables using boogie (-prover option ignored; -exec must specify the Z3 executable)
- **-method *methodlist***: a comma-separated list of method names to check (default is all methods in all listed classes) [TBD - describe wildcards and fully]
- **-exclude *methodlist***: a comma-separated list of method names to exclude from checking
- **-checkFeasibility *where***: checks feasibility of the program at various points: one of `none`, `all`, `exit` [TBD, finish list, give default]
- **-escMaxWarnings *int***: the maximum number of assertion violations to look for; the argument is either a positive integer or `All` (or equivalently `all`, default is `All`)
- **-trace**: prints out a counterexample trace for each failed assert
- **-subexpressions**: prints out a counterexample trace with model values for each subexpression
- **-counterexample** or **-ce**: prints out counterexample information

## 13.3 Runtime Assertion Checking

*This section will be added later.*

### 13.3.1 Options specific to runtime checking

- **-showNotExecutable**: warns about the use of features that are not executable (and thus ignored)
- **-racShowSource**: includes source location in RAC warning messages [ TBD: default? ]
- **-racCheckAssumptions**: enables (default on [TBD - is this default correct?]) checking assume statements as if they were asserts
- **-racJavaChecks**: enables (default on) performing JML checking of violated Java features (which will just proceed to throw an exception anyway)
- **-racCompileToJavaAssert**: (default off) compile RAC checks using Java asserts (which must then be enabled using `-ea`), instead of using `org.jmlspecs.utils.JmlAssertionFailure`
- **-racPreconditionEntry**: (default off) enable distinguishing internal Precondition errors from entry Precondition errors, appropriate for automated testing; compiles code to generate `JmlAssertionError` exceptions (rather than RAC warning messages)[TBD - should this turn on `-racCheckAssumptions`?]

## 13.4 Generating Documentation

*This section will be added later.*

## 13.5 Generating Specification File Skeletons

*This section will be added later.*

## 13.6 Generating Test Cases

*This section will be added later.*

## 13.7 Limitations of OpenJML's implementation of JML

Currently OpenJML does not completely implement JML. The differences are explained in the following subsections.

### 13.7.1 model import statement

OpenJML translates a JML model import statement into a regular Java import statement [TBD - check this]. Consequently, names introduced in a model import statement are visible in both Java code and JML annotations. This has consequences in the situation in which a name is imported both through a Java import and a JML model import. Consider the following examples of involving packages `a` and `b`, each containing a class named `X`.

In these two examples,

```
import a.X; //@ model import b.X;
```

```
import a.*; //@ model import b.*;
```

the class named `X` is imported by both an import statement and a model import statement. In JML, the use of `X` in Java code unambiguously refers to `a.X`; the use of `X` in JML annotations is ambiguous. However, in OpenJML, the use of `X` in both contexts will be identified as ambiguous.

In

```
import a.*; //@ model import b.X;
```

a use of `X` in Java code refers to `a.X` and a use in JML annotations refers to `b.X`. However, in OpenJML, both uses will mean `b.X`.

However,

```
import a.X; //@ model import b.*;
```

is unproblematic. Both JML and OpenJML will interpret `X` as `a.X` in both Java code and JML annotations.

TBD - more to be said about `.jml` files

### 13.7.2 purity checks and system library annotations

JML requires that methods that are called within JML annotations must be pure methods (cf. section TBD). OpenJML does implement a check for this requirement. However, to be pure, a method must be annotated as such by either `/* pure */` or `@Pure`. A user should insert such annotations where appropriate in the user's own code. However, many system libraries still lack JML annotations, including indications of purity. Using an unannotated library call within JML annotation will provoke a warning from OpenJML. Until the system libraries are more thoroughly annotated, users may wish to use the `-no-purityCheck` option to turn off purity checking.

### **13.7.3 TBD - other unimplemented features**

## Chapter 14

# Using OpenJML and OpenJDK within user programs

The OpenJML software is available as a library so that Java and JML programs can be manipulated within a user's program. The developer needs only to include the `openjml.jar` library on the classpath when compiling a program and to call methods through the public API as described in this chapter. The public API is implemented in the interface `org.jmlspecs.openjml.IAPI`; it provides the ability to

- perform compilation actions as would be executed on the command-line
- parse files or Strings containing Java and JML source code, producing parse trees
- print parse trees
- walk over parse trees to perform user-defined actions
- type-check parse trees (both Java and JML checking)
- perform static checking
- compile modules with run-time checks
- emit javadoc documentation with JML annotations

The sections of this chapter describe these actions and various concepts needed to perform them correctly.

**CAUTION:** OpenJML relies on parts of the OpenJDK software that are labeled as internal, non-public and subject to change. Correspondingly, some of the OpenJML API may change in the future. The definition of the API class is intended to provide a buffer against such changes. However, the names and functionality of OpenJDK classes (e.g., the `Context` class in the next section) could change.

**List classes** **CAUTION #2:** The OpenJDK software uses its own implementation of Lists, namely `com.sun.tools.javac.util.List`. It is a different implementation than `java.util.List`, with a different interface. Since one or the other may be in the list of imports, the use of `List` in the code may not clearly indicate which type of `List` is being used. Error messages are not always helpful here. Users should keep these two types of `List` in mind to avoid confusion.

**Example source code** The subsections that follow contain many source code examples. Small source code snippets are shown in in-line boxes like this:

```
// A Java comment
```

Larger examples are shown as full programs. These are followed by a box of text with a gray background that contains the output expected if the program is run (if the program is error-free) or compiled (if there are compilation errors). Here is a “Hello, world” example program:

```
// DemoHelloWorld.java
public class DemoHelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
Hello, World!
```

All of these full-program example programs are working, tested examples. They are available in the demos directory of the OpenJML source code. The opening comment line (as well as the class name) of the example text gives the file name.

The full programs presume an appropriate environment. In particular, they expect the following

- the current working directory is the demos directory of the OpenJML source distribution
- the Java CLASSPATH contains the current directory and a release version of the OpenJML library (openjml.jar). For example, if the demos directory is the current working directory and a copy of openjml.jar is in the demos directory, then the CLASSPATH could be set as “. ; openjml.jar” (using the ; on Windows, a : on Mac and Linux)

Note that the examples often use other files that are in subdirectories of the demos directory.

```
// bash commands to compile and run the DemoHelloWorld example
cd OpenJML/demos                # Alter this to match your local installation
export CLASSPATH=". ; openjml.jar" # Use a : instead of ; on Unix or Mac
                                # Copy openjml.jar to the demo directory
javac DemoHelloWorld.java        # Be sure java tools from a 1.7 JDK
java DemoHelloWorld              # are on the PATH
```

## 14.1 Concepts

### 14.1.1 Compilation Contexts

All parsing and compilation activities within OpenJML are performed with respect to a *compilation context*, implemented in the code as a `com.sun.tools.javac.util.Context` object. There can be more than one Context at a given time, though this is rare. A context holds all of the symbol tables and cached values that represent the source code created in that context.

There is little need for the user to create or manipulate Contexts. However it is essential that items created in one Context not be used in another context. There is no check for such misuse, but the subsequent

actions are likely to fail. For example, a Context contains interned versions of the names of source code identifiers (as Names). Consequently an identifier parsed in one Context will appear different than an identifier parsed in another Context, even if they have the same textual name. Do not try to reuse parse trees or other objects created in one Context in another Context.

Each instance of the IAPI interface creates its own Context object and most methods on that IAPI instance operate with respect to that Context. The `API.close` operation releases the Context object, allowing the garbage collector to reclaim space.<sup>1</sup>

### 14.1.2 JavaFileObjects

OpenJDK works with source files using `JavaFileObject` objects. This class abstracts the behavior of ordinary source files. Recall that the definition of the Java language allows source material to be held in containers other than ordinary files on disk; The `JavaFileObject` class accommodates such implementations.

OpenJML currently handles source material in ordinary files and source material expressed as `String` objects and contained in mock-file objects. Such mock objects make it easier to create source material programmatically, without having to create temporary files on disk.

Although the basic input unit to OpenJDK and OpenJML is a `JavaFileObject`, for convenience, methods that require source material as input have variations allowing the inputs to be expressed as names of files or `File` objects. If needed, the following methods create `JavaFileObjects`:

```
String filename = ...
File file = new java.io.File(filename);
IAPI m = Factory.makeAPI();
JavaFileObject jfo1 = m.makeJF0fromFilename(filename);
JavaFileObject jfo2 = m.makeJF0fromFile(file);
JavaFileObject jfo3 = m.makeJF0fromString(filename, contents);
```

The last of the methods above, `makeJF0fromString`, creates a mock-file object with the given contents (a `String`). The `contents` argument is a `String` holding the text that would be in a compilation unit. The mock-object must have a sensible filename as well. In particular, the given filename should match the package and class name as given in the `contents` argument. In addition to creating the `JavaFileObject` object, the mock-file is also added to an internal database of source mock-files; if a mock-file has a filename that would be on the source path (were it a concrete file), then the mock-file is used as if it were a real file in an OpenJML compilation. [TODO: Test this. Also, how to remove such files from the internal database. ]

### 14.1.3 Interfaces and concrete classes

A design meant to be extended should preferably be expressed as Java interfaces; if client code uses the interface and not the underlying concrete classes, then reimplementing functionality with new classes is straightforward. The OpenJDK architecture uses interfaces in some places, but often it is the concrete classes that must be extended.

Table 14.1 lists important interfaces, the corresponding OpenJDK concrete class, and the OpenJML replacement.

---

<sup>1</sup>The OpenJDK software was designed as a command-line tool, in which all memory is reclaimed when the process exits. Although in principle memory can be garbage collected when no more references to a Context or its constituent parts exist, the degree to which this is the case has not been tested.



Interface	OpenJDK class	OpenJML class
IAPI		API
	com.sun.tools.javac.main.Main	org.jmlspecs.openjml.Main
	Option	
IOption		JmlOption
IVisitor		
IJmlTree		
IJmlVisitor		
IProver		
IProverResult		ProverResult
IProverResult.ICounterexample		Counterexample
IProverResult.ICoreIds		
JCDiagnostic.DiagnosticPosition	SimpleDiagnosticPosition	DiagnosticPositionSE, DiagnosticPositionSES
Diagnostic<T>	JCDiagnostic	
	com.sun.tools.javac.main.JavaCompiler	JmlCompiler

Table 14.1: Interfaces and Classes

TODO: Add Parser, Scanner, other tools, JCTree nodes, JMLTree nodes, Option/JmlOption, Diagnostic-Position, Tool, OptionChecker

#### 14.1.4 Object Factories

#### 14.1.5 Abstract Syntax Trees

#### 14.1.6 Compilation Phases and The tool registry

Compilation in the OpenJDK compiler proceeds in a number of phases. Each phase is implemented by a specific tool. OpenJDK examples are the `DocCommentScanner`, `EndPosParser`, `Flow`, performing scanning, parsing and flow checks respectively; the OpenJML counterparts are `JmlScanner`, `JmlParser`, and `JmlFlow`.

In each compilation context there is one instance of each tool, registered with the context. The Context contains a map of keys to the singleton instance of the tool (or its factory) for that context. The scanner and parser are treated slightly differently: there is a singleton instance of a scanner factory and a parser factory, but a new instance of the scanner and the parser are created for each compilation unit compiled. Tables 14.2 and 14.3 list the tools most likely to be encountered when programming with OpenJML.

OpenJML implements alternate versions of many of the OpenJDK tools. The OpenJML versions are derived from the OpenJDK versions and are registered in the context in place of the OpenJDK versions. In that way, anywhere in the software that a tool is obtained (using the syntax `ZZZ.instance(context)` for a tool `ZZZ`), the appropriate version and instance of the tool is produced.

In some cases, a *tool factory* is registered instead of a tool instance. Then a tool instance is created on the first request for an instance of the tool. The reason for this is the following. Most tools use other tools and, for efficiency, request instances of those tools in their constructors. Circular dependencies can easily arise among these tool dependencies. Using factories helps mitigate this, though the problem still does easily arise.

Purpose	Java and JML tool	Notes
overall compiler	JavaCompiler, JmlCompiler	controls the flow of compilation phases
scanner factory	ScannerFactory, JmlScanner.Factory	
Token scanning	DocCommentScanner, JmlScanner	new instance created from the factory for each compilation unit
parser factory	ParserFactory, JmlFactory	
parser	EndPosParser, JmlParser	new instance created from the factory for each compilation unit
symbol table construction	Enter, JmlEnter	
annotation processing	Annotate	performed in JavaCompiler.processAnnotations
type determination and checking	Attr, JmlAttr	
flow-sensitive checks	Flow, JmlFlow	simple type-checking stops here
static checking	JmlEsc	invoked instead of desugaring if static checking is enabled (and processing ends here)
runtime assertion checking	JmlRac	invoked if RAC is enabled, and then proceeds with the remainder of compilation and code generation
desugaring generics		performed in the method JavaCompiler.desugar
code generation	Gen	not used for ESC

Table 14.2: Compilation phases and corresponding tools as implemented in JavaCompiler and JmlCompiler

Purpose	Java and JML tool	Notes
identifier table	Names	
symbol table	SymTab	
compiler and command-line options	Options, JmlOptions	
AST node factory	JCTree.Factory, JmlTree.Maker	
message reporting	Log	
printing ASTs	Pretty, JmlPretty	
name resolution	Resolve, JmlResolver	
AST utilities	TreeInfo, JmlTreeInfo	
type checks	Check, JmlCheck	
creating diagnostic message objects	JCDiagnostic.Factory	

Table 14.3: Some of the other registered tools

TBD: Others - MemberEnter, JmlMemberEnter, JmlRac, JmlCheck, Infer, Types, Options, Lint, Source, JavacMessages, DiagnosticListener, JavaFileManager/JavacFileManager, ClassReader/javadocClassReader, JavadocEnter, DocEnv/DocEnvJml, BasicBlocker, ProgressReporter?, ClassReader, ClassWriter, Todo, Annotate, Types, TaskListener, JavacTaskImpl, JavacTrees

TBD: Others - JmlSpecs, Utils, Nowarns, JmlTranslator, Dependencies

TBD: Is JmlTreeInfo still used

## 14.2 OpenJML operations

### 14.2.1 Methods equivalent to command-line operations

The `execute` methods of IAPI perform the same operation as a command on the command-line. These methods are different than others of IAPI in that they create and use their own `Context` object, ignoring that of the calling IAPI object.

The simple method is shown here:

```
import org.jmlspecs.openjml.IAPI;

IAPI m = new org.jmlspecs.openjml.API();
int returnCode = m.execute("--check", "--noPurityCheck", "src/demo/Err.java");
```

Each argument that would appear on the command-line is a separate argument to `execute`. All informational and diagnostic output is sent to `System.out`. The value returned by `execute` is the same as the exit code returned by the equivalent command-line operation. The String arguments are a varargs list, so

they can be provided to execute as a single array:

```
import org.jmlspecs.openjml.IAPI;
String[] args = new String[]{"-check","-noPurityCheck","src/demo/Err.java"};
IAPI m = new org.jmlspecs.openjml.API();
int returnCode = m.execute(args);
```

A full example of using execute on a file with a syntax error is shown below:

```
// DemoExecute.java
import org.jmlspecs.openjml.*;

public class DemoExecute {

    public static void main(String[] argv) {
        try {
            IAPI m = Factory.makeAPI();
            String[] args = new String[]{"-check","src/demo/Err.java"};
            int retcode = m.execute(null,args);
            System.out.println("Return Code: " + retcode);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

```
DemoExecute.java:9: error: ';' expected
        String[] args = new String[]{"-check","src/demo/Err.java"}
                                   ~
1 error
```

A longer form of execute takes two additional arguments: a Writer and a DiagnosticListener. The Writer receives all the informational output. The report method of the DiagnosticListener is

called for each warning or error diagnostic generated by OpenJML. Here is a full example of this method:

```
// DemoExecute2.java
import org.jmlspecs.openjml.*;
import javax.tools.*;

class MyDiagListener implements DiagnosticListener<JavaFileObject> {
    public int count = 0;
    public void report(Diagnostic<? extends JavaFileObject> diag) {
        System.out.println("Line: " + diag.getLineNumber());
        count++;
    }
}

public class DemoExecute2 {

    public static void main(String[] argv) {
        try {
            IAPI m = Factory.makeAPI();
            MyDiagListener listener = new MyDiagListener();
            int retcode = m.execute(new java.io.PrintWriter(System.out), listener, null,
                "-check", "-noPurityCheck", "src/demo/Err.java");
            System.out.println("Errors: " + listener.count);
            System.out.println("Return code: " + retcode);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

```
Line: 6
Line: 4
Errors: 2
Return code: 1
```

### 14.2.2 Parsing

There are two varieties of parsing. The first parses an individual Java or specification file, producing an AST that represents that source file. The second parses both a Java file and its specification file, if there is a separate one. The second form is generally more useful, since the specification file is found automatically. However, if the parse trees are being constructed programmatically, it may be useful to parse the files individually and then manually associate them.

Parsing constructs a parse tree. No symbols are created or entered into a symbol table. Nor is any type-checking performed. The only global effect is that identifiers are interned in the Names table, which is specific to the compilation context. Thus the only effect of discarding a parse tree is that there may be orphaned (no longer used) names in the Names table. The Names table cannot be cleared without the risk of dangling identifiers in parse trees.

Other than this consideration, parse trees can be created, manipulated, edited and discarded. Section TBD describes tools for manually creating parse trees and walking over them. Once a parse tree is type-checked, it should be considered immutable.

### Parsing individual files

There are two methods for parsing an individual file. The basic method takes a `JavaFileObject` as input and produces an AST. The convenience method takes a filename as input and produces an AST. The methods of section 14.1.1 enable you to produce `JavaFileObjects` from filenames, `File` objects, or `Strings` that hold the equivalent of the contents of a file (a compilation unit).

```
JmlCompilationUnit parseSingleFile(String filename);
JmlCompilationUnit parseSingleFile(JavaFileObject jfo);
```

The filename is relative to the current working directory.

Here is a full example that shows both interfaces and shows how to attach a specification parse tree to its Java parse tree.

```
// DemoParseSingle.java
import javax.tools.JavaFileObject;

import org.jmlspecs.openjml.*;

public class DemoParseSingle {

    public static void main(String[] argv) {
        try {
            IAPI m = Factory.makeAPI();
            String f1 = "src/demo/A.java";
            JavaFileObject f2 = m.makeJFOfromFilename("specs/demo/A.jml");
            JmlTree.JmlCompilationUnit ast1 = m.parseSingleFile(f1);
            JmlTree.JmlCompilationUnit ast2 = m.parseSingleFile(f2);
            m.attachSpecs(ast1,ast2);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

### Parsing Java and JML files together

The more common action is to parse a Java file and its specification at the same time. The JML language defines how the specification file is found for a given source or binary class. In short, the specification file has syntax very similar to a Java file:

- it must be in the same package and have the same class name as the Java class
- if both are files, the filenames without suffix must be the same

- the specification file must be on the *specspath*
- if a .jml file meeting the above criteria is found anywhere on the specspath, it is used; otherwise a .java file on the specspath meeting the above criteria is used; otherwise only default specifications are used.<sup>2</sup>

Note that a Java file can be specified on the command-line that is not on the specspath. In that case (if there is no .jml file) no specification file will be found, although the user may expect that the Java file itself may serve as its own specifications. This is a confusing situation and should be avoided.

### 14.2.3 Type-checking

### 14.2.4 Static checking

### 14.2.5 Compiling run-time checks

### 14.2.6 Creating JML-enhanced documentation

## 14.3 Working with ASTs

### 14.3.1 Printing parse trees

TBD

### 14.3.2 Source location information

TBD

### 14.3.3 Exploring parse trees with Visitors

OpenJML defines some Visitor classes that can be extended to implement user-defined functionality while traversing a parse tree. The basic class is `JmlScanner`. An unmodified instance of `JmlScanner` will traverse a parse tree without performing any actions.

There are three modes of traversing an AST.

- `AST_JAVA_MODE` - traverses only the Java portion of an AST, ignoring any JML annotations
- `AST_JML_MODE` - traverses the Java and JML syntax that was part of the original source file
- `AST_SPEC_MODE` - traverses the Java syntax and its specifications (whether they came from the same source file or a different one). This mode is only available after the AST has been type-checked.

A derived class can affect the behavior of the visitor in two ways:

- By overriding the `scan` method, an action can be performed at every node of an AST

---

<sup>2</sup>In the past, JML allowed multiple specification files and defined an ordering and rules for combining the specifications contained in them. The JML has been simplified to allow just one specification file, just one suffix (.jml), and no combining of specifications from a .jml and a .java file if both exist.

- By overriding specific `visit...` methods, an action can be performed that is specific to the nodes of the corresponding type

In the example that follows, the `scan` method of the `Visitor` is modified to print the node type and count all nodes in the AST, the `visitBinary` method is modified to count Java binary operations, and the `visitJml-Binary` method is modified to count JML binary operations. The default constructor of the parent `Visitor`



class sets the traversal mode to AST\_JML\_MODE.

```
// DemoWalkTree1.java
import org.jmlspecs.openjml.*;

import com.sun.tools.javac.tree.JCTree;

public class DemoWalkTree1 {

    static class Walker extends JmlTreeScanner {

        int nodes = 0;
        int jmlopcount = 0;
        int allopcount = 0;

        @Override
        public void scan(JCTree node) {
            if (node != null) System.out.println("Node: " + node.getClass());
            if (node != null) nodes++;
            super.scan(node);
        }

        @Override
        public void visitJmlBinary(JmlTree.JmlBinary that) {
            jmlopcount++;
            allopcount++;
            super.visitJmlBinary(that);
        }

        @Override
        public void visitBinary(JCTree.JCBinary that) {
            allopcount++;
            super.visitBinary(that);
        }

    }

    public static void main(String[] argv) {
        try {
            IAPI m = Factory.makeAPI();
            Walker visitor = new Walker();
            JCTree.JCExpression expr = m.parseExpression("(a+b)*c", false);
            visitor.scan(expr);
            System.out.println("Counts: " + visitor.nodes + " " +
                               visitor.allopcount + " " + visitor.jmlopcount);
            expr = m.parseExpression("a <==> \\result", true);
            visitor.scan(expr);
            System.out.println("Counts: " + visitor.nodes + " " +
                               visitor.allopcount + " " + visitor.jmlopcount);

        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

```
Node: class com.sun.tools.javac.tree.JCTree$JCBinary
Node: class com.sun.tools.javac.tree.JCTree$JCParens
Node: class com.sun.tools.javac.tree.JCTree$JCBinary
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Counts: 6 2 0
Node: class org.jmlspecs.openjml.JmlTree$JmlBinary
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlSingleton
Counts: 9 3 1
```

The second example shows the differences among the three traversal modes. Note that the AST\_SPEC\_MODE

traversal fails when requested prior to type-checking the AST.

```
// DemoWalkTree2.java
import org.jmlspecs.openjml.*;

import com.sun.tools.javac.tree.JCTree;

public class DemoWalkTree2 {

    static class Walker extends JmlTreeScanner {

        public Walker(int mode) {
            super(mode);
        }

        int nodes = 0;
        int jmlopcount = 0;
        int allopcount = 0;

        @Override
        public void scan(JCTree node) {
            if (node != null) System.out.println("Node: " + node.getClass());
            if (node != null) nodes++;
            super.scan(node);
        }

        @Override
        public void visitJmlBinary(JmlTree.JmlBinary that) {
            jmlopcount++;
            allopcount++;
            super.visitJmlBinary(that);
        }

        @Override
        public void visitBinary(JCTree.JCBinary that) {
            allopcount++;
            super.visitBinary(that);
        }

    }

    public static void main(String[] argv) {
        try {
            java.io.File f = new java.io.File("src/demo/A.java");
            IAPI m = Factory.makeAPI("-specspath", "specs", "-sourcepath", "src", "-noPurityCheck");
            JmlTree.JmlCompilationUnit expr = m.parseFiles(f).get(0);
            Walker visitor = new Walker(Walker.AST_JAVA_MODE);
            visitor.scan(expr);
            System.out.println("Counts: " + visitor.nodes + " " +
                               visitor.allopcount + " " + visitor.jmlopcount);
            visitor = new Walker(Walker.AST_JML_MODE);
            visitor.scan(expr);
            System.out.println("Counts: " + visitor.nodes + " " +
                               visitor.allopcount + " " + visitor.jmlopcount);
            try {
                visitor = new Walker(Walker.AST_SPEC_MODE);
                visitor.scan(expr);
                System.out.println("Counts: " + visitor.nodes + " " +
                                   visitor.allopcount + " " + visitor.jmlopcount);
            } catch (Exception e) {

```

```
Node: class org.jmlspecs.openjml.JmlTree$JmlCompilationUnit
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlClassDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class com.sun.tools.javac.tree.JCTree$JCPrimitiveTypeTree
Node: class com.sun.tools.javac.tree.JCTree$JCBlock
```

```
Counts: 8 0 0
```

```
Node: class org.jmlspecs.openjml.JmlTree$JmlCompilationUnit
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlClassDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodDecl
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodSpecs
Node: class org.jmlspecs.openjml.JmlTree$JmlSpecificationCase
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodClauseExpr
Node: class com.sun.tools.javac.tree.JCTree$JCBinary
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class com.sun.tools.javac.tree.JCTree$JCLiteral
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class com.sun.tools.javac.tree.JCTree$JCPrimitiveTypeTree
Node: class com.sun.tools.javac.tree.JCTree$JCBlock
Node: class org.jmlspecs.openjml.JmlTree$JmlTypeClauseDecl
Node: class org.jmlspecs.openjml.JmlTree$JmlVariableDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class com.sun.tools.javac.tree.JCTree$JCAnnotation
Node: class com.sun.tools.javac.tree.JCTree$JCFieldAccess
Node: class com.sun.tools.javac.tree.JCTree$JCFieldAccess
Node: class com.sun.tools.javac.tree.JCTree$JCFieldAccess
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class com.sun.tools.javac.tree.JCTree$JCPrimitiveTypeTree
Node: class com.sun.tools.javac.tree.JCTree$JCLiteral
```

```
Counts: 25 1 0
```

```
Node: class org.jmlspecs.openjml.JmlTree$JmlCompilationUnit
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlClassDecl
```

```
EXCEPTION: java.lang.RuntimeException: AST_SPEC_MODE requires that the Class be type-checked; class
```

```
Node: class org.jmlspecs.openjml.JmlTree$JmlCompilationUnit
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlClassDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodSpecs
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class com.sun.tools.javac.tree.JCTree$JCBlock
Node: class com.sun.tools.javac.tree.JCTree$JCExpressionStatement
Node: class com.sun.tools.javac.tree.JCTree$JCMethodInvocation
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodSpecs
Node: class org.jmlspecs.openjml.JmlTree$JmlSpecificationCase
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodClauseExpr
Node: class com.sun.tools.javac.tree.JCTree$JCLiteral
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
```

There are two other points to make about these examples.

- Note that each derived method calls the superclass version of the method that it overrides. The superclass method implements the logic to traverse all the children of the AST node. If the super call is omitted, no traversal of the children is performed. If the derived class wishes to traverse only some of the children, a specialized implementation of the method will need to be created. It is easiest to create such an implementation by consulting the code in the super class.
- In the examples above, you can see that the `System.out.println` statement that prints the node's class occurs before the super call. The result is a pre-order traversal of the tree; if the print statement occurred after the super call, the output would show a post-order traversal.

#### **14.3.4 Creating parse trees**

### **14.4 Working with JML specifications**

### **14.5 Utilities**

– version – context – symbols

### **14.6 Extending or modifying JML**

JML is modified by providing new implementations of key classes, typically by derivation from those that are part of OpenJML. In fact, OpenJML extends many of the OpenJDK classes to incorporate JML functionality into the OpenJDK Java compiler.

#### **14.6.1 Adding new command-line options**

#### **14.6.2 Altering IAPI**

#### **14.6.3 Changing the Scanner**

#### **14.6.4 Enhancing the parser**

#### **14.6.5 Adding new modifiers and annotations**

#### **14.6.6 Adding new AST nodes**

#### **14.6.7 Modifying a compiler phase**

## **Part IV**

# **Contributing to OpenJML**

## **Part V**

# **Semantics and translation of Java and JML in OpenJML**

## Chapter 15

# Introduction

- JML section is a summary of material from the reference manual
- Sorted First-order-logic
- individual subexpressions; optional expression form; optimization; usefulness for tracing
- RAC vs. ESC
- nomenclature



## Chapter 16

# Statement translations

TODO: Need to insert both RAC and ESC in all of the following.

### 16.1 While loop

Java and JML statement:

```
//@ invariant invariant_condition ;
//@ decreases counter ;
while (condition) {
    body
}
```

Translation: *TODO: Needs variant condition, havoc information*

```
{
    //@ assert jmltranslate(invariant_condition) ;
    //@ assert jmltranslate(variant_condition) > 0 ;
    while (true) {
        stats(tmp,condition)
        if (!tmp) {
            //@ assume !tmp;
            break;
        }
        //@ assume tmp;
        stats(body)
    }
}
```

## Chapter 17

# Java expression translations

### 17.1 Implicit or explicit arithmetic conversions

*TODO*

### 17.2 Arithmetic expressions

*TODO: need arithmetic range assertions*

In these,  $T$  is the type of the result of the operation. The two operands in binary operations are already assumed to have been converted to a common type according to Java's rules.

```
stats(tmp, - a ) ==>
  stats(tmpa, a )
  T tmp = - tmpa ;
```

```
stats(tmp, a + b ) ==>
  stats(tmpa, a )
  stats(tmpb, b )
  T tmp = tmpa + tmpb ;
```

```
stats(tmp, a - b ) ==>
  stats(tmpa, a )
  stats(tmpb, b )
  T tmp = tmpa - tmpb ;
```

```
stats(tmp, a * b ) ==>
  stats(tmpa, a )
  stats(tmpb, b )
  T tmp = tmpa * tmpb ;
```

```
stats(tmp, a / b ) ==>
  stats(tmpa, a )
```

```

stats(tmpb, b )
/*@ assert tmpb != 0; //No division by zero
T tmp = tmpa / tmpb ;

```

```

stats(tmp, a % b ) ==>
  stats(tmpa, a )
  stats(tmpb, b )
  /*@ assert tmpb != 0; //No division by zero
  T tmp = tmpa % tmpb ;

```

## 17.3 Bit-shift expressions

*TODO*

## 17.4 Relational expressions

No assertions are generated for the relational operations `<` `>` `<=` `>=` `==` `!=`. The operands are presumed to have been converted to a common type according to Java's rules.

```

stats(tmp, a op b ) ==>
  stats(tmpa, a )
  stats(tmpb, b )
  T tmp = tmpa op tmpb ;

```

## 17.5 Logical expressions

```

stats(tmp, ! a ) ==>
  stats(tmpa, a )
  T tmp = ! tmpa ;

```

The `&&` and `||` operations are short-circuit operations in which the second operand is conditionally evaluated. Here `&` and `|` are the (FOL) boolean non-short-circuit conjunction and disjunction.

```

stats(tmp, a && b ) ==>
  boolean tmp ;
  stats(tmpa, a )
  if ( tmpa ) {
    /*@ assume tmpa ;
    stats(tmpb, b )
    tmp = tmpa & tmpb ;
  } else {
    /*@ assume ! tmpa ;
    tmp = tmpa ;

```

```

}

stats(tmp, a || b) ==>
  boolean tmp ;
  stats(tmpa, a)
  if ( ! tmpa ) {
    //@ assume ! tmpa ;
    stats(tmpb, b)
    tmp = tmpa | tmpb ;
  } else {
    //@ assume tmpa ;
    tmp = tmpa ;
  }

```

#### TODOs

- Fix the TITLE for the web pages
- on HTML pages boxed examples do not render correctly

*An index will be added later.*