ESC/Java2 Implementation Notes

David R. Cok
cok@frontiernet.net
Joseph R. Kiniry
joseph.kiniry@ucd.ie
Dermot Cochran
Dermot.Cochran@ucd.ie

October 2008

Abstract: ESC/Java2 is a tool for statically checking program specifications. It expands significantly upon ESC/Java, on which it is built. It is consistent with the definition of JML and of Java 1.4. It adds additional static checking to that in ESC/Java; most significantly, it adds support for checking frame conditions and annotations containing method calls. This document describes the status of the final release of ESC/Java2, along with some notes regarding the details of that implementation.

Keywords: Behavioral interface specification, Java, JML, Escjava, ESC/Java, ESC/Java2, model-based specification, assertion, precondition, postcondition, frame.

2003 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, tools, theory, JML, ESC/Java, ESC/Java2; D.2.4 [*Software Engineering*] Software/Program Verification — assertion checkers, class invariants, formal methods, programming by contract; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, invariants, logics of programs, pre- and postconditions, specification techniques.

Copyright © 2003-2004 David R. Cok, 2005-2008 Joseph R. Kiniry, 2008 Dermot Cochran

Table of Contents

1 Introduction **1**
1.1 Motivation and Background 1
1.2 Acknowledgements 1
1.3 Dependencies and license restrictions 2
1.4 Contacts and information 2

2 Running ESC/Java2 . . . . . . **4**

3 Status of JML features **5**
3.1 File finding and refinement sequences 5
3.2 Format of annotations 8
3.3 Compilation unit annotations 10
3.3.1 refine statements 10
3.3.2 model import statements 12
3.3.3 automatic imports 13
3.4 Access (privacy) modifiers 13
3.5 Type modifiers 13
3.5.1 pure (JML) 14
3.5.2 model (JML) 14
3.5.3 weakly (JML) 14
3.5.4 non_null_by_default (JML) 14
3.5.5 nullable_by_default (JML) 15
3.5.6 final (Java) 15
3.5.7 abstract (Java) 16
3.5.8 strictfp (Java) 16
3.5.9 static (Java) 16
3.6 Annotations pertinent to a class or interface 16
3.6.1 Ghost fields 16
3.6.2 Model fields 17
3.6.3 Model methods 17
3.6.4 Model constructors 17
3.6.5 Model class and model interface declarations 18
3.6.6 Java initializer blocks 18
3.6.7 initializer 18
3.6.8 static_initializer 19
3.6.9 Java method, constructor and field declarations 19
3.6.10 Nested Java type declarations 19
3.7 Annotation clauses for a class or interface 19
3.7.1 invariant, invariant_redundantly 19
3.7.2 constraint, constraint_redundantly 19
3.7.3 represents, represents_redundantly 20
3.7.4 axiom 20

3.7.5 initially, initially_redundantly . . . 20
3.7.6 readable and writable . . . 20
3.7.7 monitors_for . . . 21
3.8 Annotations for a method or constructor . . . 21
3.8.1 Lightweight and heavyweight specifications . . . 21
3.8.2 also . . . 24
3.8.3 model_program . . . 24
3.8.4 code_contract . . . 24
3.8.5 implies_that . . . 24
3.8.6 for_example specification . . . 25
3.9 Annotation clauses for a method or constructor . . . 25
3.9.1 forall . . . 25
3.9.2 old . . . 25
3.9.3 requires, requires_redundantly, pre, pre_redundantly . . . 26
3.9.4 ensures, ensures_redundantly, post, post_redundantly . . . 26
3.9.5 signals, signals_redundantly, exsures, exsures_redundantly . . . 26
3.9.6 modifies, modifiable, assignable, modifies_redundantly, modifiable_redundantly, assignable_redundantly . . . 26
3.9.7 diverges, diverges_redundantly . . . 28
3.9.8 when . . . 28
3.9.9 duration . . . 28
3.9.10 working_space . . . 29
3.9.11 accessible . . . 29
3.9.12 callable . . . 29
3.9.13 measured_by . . . 29
3.9.14 Redundancy . . . 29
3.10 Annotation modifiers for a method or constructor . . . 29
3.10.1 pure (JML) . . . 30
3.10.2 non_null (JML - methods only) . . . 30
3.10.3 nullable (JML - methods only) . . . 30
3.10.4 helper (JML) . . . 31
3.10.5 final (Java - methods only) . . . 31
3.10.6 static (Java - methods only) . . . 31
3.10.7 synchronized (Java - methods only) . . . 32
3.10.8 native (Java - methods only) . . . 32
3.10.9 strictfp (Java - methods only) . . . 32
3.11 Annotation assertions for a field declaration . . . 32
3.11.1 in (JML) . . . 32
3.11.2 maps, \into (JML) . . . 32
3.12 Annotation modifiers for a field declaration . . . 33
3.12.1 non_null (JML) . . . 33
3.12.2 nullable (JML) . . . 33
3.12.3 monitored (JML) . . . 33
3.12.4 final (Java) . . . 34
3.12.5 volatile (Java) . . . 34
3.12.6 transient (Java) . . . 34
3.12.7 static (Java) . . . 34

3.12.8 instance (JML) 34
3.13 Annotation modifiers for formal parameters 35
3.13.1 non_null (JML) 35
3.13.2 nullable (JML) 35
3.13.3 final (Java) 36
3.14 Annotation statements within the body of a method or constructor 36
3.14.1 assume, assume_redundantly 36
3.14.2 assert, assert_redundantly 36
3.14.3 set 36
3.14.4 unreachable 36
3.14.5 hence_by, hence_by_redundantly 37
3.14.6 loop_invariant, loop_invariant_redundantly, maintaining, maintaining_redundantly 37
3.14.7 decreases, decreasing, decreases_redundantly, decreasing_redundantly 37
3.14.8 ghost declarations 37
3.15 Modifiers that may be applied to local declarations 37
3.15.1 non_null (JML) 38
3.15.2 nullable (JML) 38
3.15.3 uninitialized (JML) 38
3.15.4 final (Java) 38
3.16 JML functions (extensions to expressions) 38
3.16.1 New operators in JML 38
3.16.2 New JML expressions (functions and values) 40
3.16.3 New JML types 42
3.16.4 quantified expressions - \forall, \exists, \num_of, \max, \min, \sum, \product 43
3.16.5 Set comprehension 43
3.16.6 \not_specified 44
3.16.7 \private_data 44
3.16.8 \other 44
3.16.9 Other Java operators and expression syntax 44
3.17 store-ref expressions 44
3.17.1 [*ident* | `super` | `this`] . [*ident* | `this`] 45
3.17.2 [*ident* | `super` | `this`] `.*` 45
3.17.3 [*ident* | `super` | `this`] `[`*expr*`]` 45
3.17.4 [*ident* | `super` | `this`] `[`*expr*`..`*expr2*`]` 45
3.17.5 [*ident* | `super` | `this`] `[*]` 45
3.17.6 *classname.** 45
3.17.7 \nothing 45
3.17.8 \everything 45
3.18 Statements within model programs 46
3.19 Other issues 46
3.19.1 universe type system 46
3.19.2 infinite-precision arithmetic 46
3.19.3 nowarn annotations 46
3.19.4 Java and JML assert statements 46

3.19.5 Methods and constructors without bodies in Java files ... 48
3.19.6 Methods and constructors in annotation expressions 48
3.19.7 Original ESC/Java also_ specifications ... 48
3.19.8 anonymous classes ... 49
3.19.9 block-level class declarations ... 49
3.19.10 `field`, `method` and `constructor` keywords ... 49
3.19.11 Equivalence of `\TYPE` and `java.lang.Class` ... 50
3.19.12 exceptions in annotation expressions ... 51
3.19.13 Specifications and inheritance ... 51
3.19.13.1 Desugaring in the presence of inheritance ... 51
3.19.13.2 Defaults and inheritance ... 53
3.19.13.3 Inheritance and non_null ... 54

4 ESC/Java2 features ... 59

4.1 Error and warning messages ... 59
4.2 Nowarn annotations and warnings ... 59
4.2.1 nowarn annotations ... 59
4.2.2 nowarn warning types ... 60
4.3 Command-line options ... 61
4.4 Environment variables affecting ESC/Java2 ... 63

5 Changes to static checking in ESC/Java2 ... 65

5.1 Handling of specification inheritance ... 65
5.2 non_null ... 65
5.3 Translation of the Java 1.4 assert statement ... 65
5.4 Semantics of String ... 65
5.4.1 Concatentation operators ... 65
5.4.2 Explicit String literals ... 66
5.4.3 The intern method ... 66
5.5 The `\TYPE` and `java.lang.Class` types ... 66
5.6 The initially clause ... 66
5.7 The constraint clause ... 66
5.8 Use of modifies clauses in checking routine bodies ... 67
5.9 Defaults for modifies clauses ... 67
5.10 modifies \everything ... 67
5.11 Checking of modifies ... 67
5.12 \typeof ... 67
5.13 Use of pure routines in annotations ... 68
5.14 Checking of model fields ... 68
5.15 \not_modified ... 68

6 Incompatibilities . . . 69
6.1 Major features of Java not implemented in ESC/Java2 . . . 69
6.1.1 Java 1.5 . . . 69
6.1.2 anonymous and block-level classes . . . 69
6.1.3 serialization . . . 69
6.1.4 most multi-threading considerations . . . 69
6.1.5 Java generics . . . 69
6.2 Major features of JML not implemented in ESC/Java2 . . . 69
6.2.1 code_contract clauses . . . 69
6.2.2 some aspects of store-ref expressions . . . 69
6.2.3 implies_that and for_example behavior . . . 69
6.2.4 splitting of annotations across comments . . . 69
6.3 Limitations of static checking . . . 70
6.4 Incompatibilities with ESC/Java . . . 70
6.4.1 Error messages and warnings . . . 70
6.4.2 also . . . 70
6.4.3 inheritance of specifications . . . 70
6.4.4 non_null on formal parameters and results of routines . . . 70
6.4.5 monitored_by . . . 70
6.4.6 readable_if . . . 70
6.4.7 initially (old style) . . . 70
6.4.8 semicolon termination . . . 70
6.4.9 Routine bodies in spec files . . . 71
6.5 Non-JML features in ESC/Java2 . . . 71
6.5.1 annotation comments beginning with `//-@` or `/*-@` . . . 71
6.5.2 order of clauses . . . 71
6.5.3 splitting of annotations . . . 71
6.5.4 helper annotations . . . 71
6.5.5 \typeof applied to primitive types . . . 71
6.5.6 unreachable . . . 71
6.5.7 \not_modified . . . 71
6.5.8 specifications of default constructor . . . 71
6.5.9 loop_predicate . . . 71
6.5.10 skolem_constant . . . 71
6.5.11 still_deferred . . . 71
6.5.12 writable_deferred . . . 71
6.5.13 writable_if . . . 71
6.5.14 readable_if . . . 71
6.5.15 monitored_by . . . 71
6.5.16 dttfsa . . . 72
6.5.17 uninitialized . . . 72
6.5.18 placement of annotations . . . 72
6.5.19 semicolon termination . . . 72
6.5.20 need for the field, method, constructor keywords . . . 72
6.5.21 omission of method bodies . . . 72
6.5.22 Errors and cautions . . . 72
6.5.23 membership in lockset . . . 73
6.6 JML features needing clarification . . . 73

6.6.1 model programs 73
6.6.2 callable, accessible 73
6.6.3 when, measured_by 73
6.6.4 initializer, static_initializer 73
6.6.5 desugaring of forall 73
6.6.6 weakly 73
6.6.7 hence_by 73
6.6.8 use of \result in duration and working_space clauses 73
6.6.9 instance fields and this in constructor preconditions 73
6.6.10 rules about splitting annotations across comments 73
6.6.11 \typeof applied to primitive types 73
6.7 Desired extensions 73

Appendix A Modifier Summary 74

Bibliography 77

Concept Index 78

1 Introduction

1.1 Motivation and Background

ESC/Java2 extends the pioneering work on ESC/Java by a group [Flanagan-etal02] at the Systems Research Center at DEC, later Compaq, now HP [http://www.hpl.hp.com/research/]. ESC/Java parses JML-like annotations in a Java program and warns, in a modular way, about annotations that may not be justified by the Java source of the given classes and the specifications of other classes. The program works accurately enough and fast enough that it has been found to be a useful tool. Its usefulness is diminished by limitations in the kind of annotations that it can parse and check and also in that its annotation language is similar to but is neither a subset nor a superset of JML.

The goal of the ESC/Java2 work is to extend the use of ESC/Java by

a. updating the parser of ESC/Java so that it is consistent with the current definition of JML and Java,
b. packaging the updated tool so that it is more easily available to a larger set of users, consistent with the source code license provisions of the ESC/Java source code,
c. and extending the range of JML annotations that can be checked by the tool, where possible and where consistent with the engineering goals of ESC/Java.

This document records the status of this implementation. It is not intended to be a tutorial or a reference guide for either JML or ESC/Java or ESC/Java2. Rather it records the status of the features of JML: the status of their implementation in ESC/Java2, the degree to which the annotation is logically checked, and any differences between ESC/Java2 and JML.

- More detailed information on JML is available at the web site http://www.jmlspecs.org/; the details of the JML definition are published in "Preliminary Design of JML" [LeavensBakerRuby02] and in "The JML Reference Manual" [Leavens-etal03] both available from the JML website.
- Information on the original ESC/Java tool, most of which still applies, is provided in "ESC/Java User's Manual", SRC Technical note 2000-002 (Leino, Nelson, Saxe), available at

  http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/SRC-2000-002.html

1.2 Acknowledgements

To date, the work on ESC/Java2 has been carried out primarily by David Cok, Joe Kiniry, Patrice Chalin, Radu Grigore, Mikolas Janota, Michal Moskal, George Karabotsos, Perry James, Julien Charles and Dermot Cochran. Gary Leavens has provided guidance on the semantics and the current and future state of JML. K. Rustan M. Leino has provided advice with respect to the original ESC/Java.

The website for the project through version 2.0a8 was hosted by the Security of Systems group at the Radboud University Nijmegen (what used to be known as

Katholieke Universiteit Nijmegen or the University of Nijmegen) in Nijmegen, Netherlands. (http://www.niii.kun.nl/sos/research/escjava/). The project is now hosted by the Systems Research Group in the School of Computer Science and Informatics at University College Dublin from version 2.0a8 onward. (http://kind.ucd.ie/)

The work of producing ESC/Java2 stands on the very much more considerable effort of the ESC/Java team in conceiving of and producing ESC/Java, JavaFE, Simplify and related tools in the first place.

It also is built upon the work in designing JML and providing tools for JML led by Gary Leavens at Iowa State University, with contributions from several other individuals and groups, as described on the JML web page.

1.3 Dependencies and license restrictions

The ESC/Java2 tool relies on the following software packages that are separately available and may have their own license restrictions.

- The original source for ESC/Java, JavaFE, Simplify and related tools, available at

  http://www.hpl.hp.com/downloads/crl/jtk/

- Part of the Mocha tool from UCBerkeley (optional), available at

  http://www-cad.eecs.berkeley.edu/~mocha/download/j-mocha/

- The CM3 compiler for Modula-3, which is needed to build the Simplify tool, available from http://www.elegosoft.com/ or fink.sourceforge.net.
- The testing framework JUnit version 3.8.1, available at http://www.junit.org/.
- Specifications of the Java system classes. The most useful set of these specifications are those available with the releases of JML.
- ESC/Java2 does not depend on the JML tool set, but it is useful to use the two in combination. ESC/Java2 is obviously dependent on the grammar and semantics of JML. JML is available at http://www.jmlspecs.org/.
- The Java 1.4 front-end for ESC/Java2 http://kind.ucd.ie/products/opensource/JavaFE/.

1.4 Contacts and information

Further information about JML and ESC/Java2 can be obtained from these sources.

- The KUN ESC/Java2 website: http://www.niii.kun.nl/sos/research/escjava/
- The JML web site: http://www.jmlspecs.org/
- The JML project on sourceforge: http://sourceforge.net/projects/jmlspecs/
- The JML interest mailing list on sourceforge:

  jmlspecs-interest@lists.sourceforge.net

- The JML development mailing list on sourceforge:

`jmlspecs-developers@lists.sourceforge.net`

- The ESC/Java2 mailing list on sourceforge:

`jmlspecs-escjava@lists.sourceforge.net`

- The Kind Software website: `http://kind.ucd.ie/products/opensource/ESCJava2/`

JML utilized (the Java subset of) the multijava compiler. Information about multijava can be obtained from these sources.

- The multijava website: `http://www.multijava.org/`
- The multijava project on sourceforge: `http://sourceforge.net/projects/multijava/`

Future versions of JML will not be based on multijava but instead will use, for example, the Eclipse JDT or OpenJDK See the JML mailing list for more details.

2 Running ESC/Java2

There are three essential items that you need in order to run ESC/Java2: a build of ESC/Java2 itself, an executable for Simplify for your platform, and a version of the JML specifications for the Java system classes. These are available together as a single release from the Kind Software ESC/Java2 website (`http://kind.ucd.ie/products/opensource/ESCJava2/`). However, you may want to substitute an alternate version of the JML specifications that you generate yourself or you obtain from the JML website. The specifications included with ESC/Java2 are a snapshot of the JML specifications at the time of release.

There are a number of ways to run ESC/Java2. The various command-line options are described in Section 4.3 [Command-line options], page 61.

1. Double-click the esctools2.jar file that comes with the release. This launches a GUI tool that runs ESC/Java2. The jar file has the JML specs built-in and uses them by default. You will have to tell the GUI tool the location of the appropriate Simplify executable for your platform, what value of `CLASSPATH` to use, and what the input files and other options should be. Versions of Simplify for several platforms are supplied with the release.

2. Execute the esctools2.jar file using the command `java -jar esctools2.jar` . This allows you to launch the GUI tool from the command-line and to add other command-line options as initial settings of the GUI tool. In particular you can specify the location of the Simplify executable with the option `-simplify` *PathToSimplify*, in which you supply an absolute path to the correct Simplify executable. A directory path for alternate Java specifications can be supplied as the argument to the `-specs` option. The classpath is specified with the `-classpath` option.

3. Execute a non-GUI version of ESC/Java2 using the script supplied with the release: `./escj` . The script can guess the location of the release and of the Simplify executable. You can help it by defining the environment variable `ESCTOOLS_ROOT` as the absolute path to the location of the directory containing the release. and by defining the variable `SIMPLIFY` as the name of (not the path to) the Simplify executable. If your working directory is something other than the directory containing the release, you will need to provide a suitable path to the `escj` script when you invoke it.

4. Execute a non-GUI version of ESC/Java2 using the command `java -cp esctools2.jar escjava.Main` . In this case you need to specify the location (path and name) of the Simplify executable using the `-simplify` option and the location of the reference specifications using the `-specs` option. You also need to specify the classpath using `-classpath` and any other input files and options.

5. Using the Eclipse plugin for ESC/Java2, which is a component of the MOBIUS Program Verification Environment (PVE) for Eclipse: `http://kind.ucd.ie/products/opensource/Mobius`

3 Status of JML features

ESC/Java2 parses correctly formatted JML files, with the exceptions described in this document. JML files must be correct Java source with correctly formatted annotations, which appear to Java as comments. Although ESC/Java2 does some error reporting during parsing, it does not report all parsing or type errors in either Java or JML, nor does it necessarily terminate normally if the input is not legal Java/JML. There are a number of tools supporting JML that can be used to check the well-formedness of the JML annotations in a file; a Java compiler can be used to check the format of the Java source code.

The authors encourage any report of a legal Java/JML file that ESC/Java2 will not parse. Furthermore, despite the caveat above, the authors do want ESC/Java2 to be a useful tool; hence they are interested in examples of legal or illegal Java/JML source code that cause abnormal termination and in examples in which the absence of error messages or the occurrence of an inappropriate error message is misleading to the user. Examples that generate unsound or incomplete behavior beyond that already documented are also of interest.

The organization of ESC/Java2's error and warning messages is described in Section 4.1 [Error and warning messages], page 59.

3.1 File finding and refinement sequences

JML follows Java conventions in file naming. ESC/Java2 recognizes Java source and class files organized into directory hierarchies matching the package definitions, including source and class files packaged in jar files. The file names themselves typically consist of a type name as a prefix and either `.java` or `.class` as a suffix. Specification files typically have the type name as a prefix and one of the specification suffixes (`.refines-java`, `.refines-spec`, `.refines-jml`, `.java`, `.spec`, `.jml`, `.java-refined`, `.spec-refined`, `.jml-refined`) as a suffix, though arbitrary suffixes are permitted.

The ESC/Java2 program utilizes a classpath and a sourcepath, which are standard sequences of directories or jar files separated by a platform-dependent path separator character (a colon on Linux and MacOSX, a semicolon on Windows). The classpath is specified by the -classpath command-line option, or by the CLASSPATH environment variable if no command-line option is given, and is just the current working directory if neither is specified. The sourcepath is specified by the -sourcepath command-line option; if that is not provided, the sourcepath is identical to the classpath.

The tool also needs a set of specifications of Java system classes (and of other library classes you may be using). The directory path to these specifications can be included in the sourcepath, but it is convenient to specify them to the tool using the `-specs` option, which takes a standard directory path as its single argument.

The command-line arguments consist of options and their arguments (described in Section 4.3 [Command-line options], page 61) and input entries. Input entries may be files, directories, package names, or class names. These input entries designate the classes on which ESC/Java2 operates.

Files and directories specified on the command-line are found with respect to the current working directory (if the paths to the files are relative paths). Specifying a directory is shorthand for listing all of the files in the directory with suffixes known to JML, namely .refines-java, .refines-spec, .refines-jml, .java, .spec, and .jml.

Input entries may also be fully-qualified package or class names. In this case, the package or class is found by searching the directories of the sourcepath.

Each file on the command-line (or file implied by a directory, package, or class) is parsed to determine the package to which it belongs and the name of the type that it declares. ESC/Java2 and JML tools then use the following procedure to find the refinement sequence for the given fully qualified type. When the specifications of types referenced in files being processed by ESC/Java2 are needed, they are found using the same procedure.

- Search each directory of the sourcepath in turn, looking for the first sourcepath directory containing a directory hierarchy for the given package containing a file whose name has the type name as its prefix and one of the following suffixes: `.refines-java`, `.refines-spec`, `.refines-jml`, `.java`, `.spec`, `.jml`. If the directory contains more than one such file, the one with a suffix closest to the beginning of the list of suffixes is used. This file is called the *Most-refined compilation unit* (MRCU).
- If the MRCU contains a `refine` statement, then the file named in it is sought in the sourcepath in the same package (but not necessarily the same directory) as the MRCU. It is an error if a file named in a `refine` statement cannot be found in the sourcepath. Files found in this way are parsed in turn and the files named in each `refine` statement are sought. This procedure is repeated recursively until a file is found that has no `refine` statement. The sequence of files so found is called the *refinement sequence*. The refinement sequence may contain the .java or .class file for the type. ESC/Java2 will parse source code up to Java version 1.4.2 and bytecode up to Java version 1.5.
- The rules above do not restrict the filenames of the files of the refinement sequence, other than that they must be in the same package (but not necessarily in the same directory). In particular, aside from the suffix for the MRCU (and java and class files), there is no restriction on the suffixes that the files may have, nor on the order of suffixes in the refinement sequence. There is also no restriction on the prefixes of the file names, other than that the MRCU, the .java file and the .class file must have the typename as the prefix. However, it is good style if all of the files in the refinement sequence have the same prefix. If a file has a filename prefix that does not match the type declared within it, it is in danger of being misinterpreted as belonging to a different type. Consequently a caution is issued to the user if this situation is discovered.
- Once an MRCU is found, the remainder of the refinement sequence is determined by the `refine` statements. But which file is found as the MRCU may depend on the contents *and order* of the directories in the sourcepath. This order dependence is by design as it is thought that the user may use this feature to choose different starting points along the refinement sequence for processing. It may also lead to inadvertent errors.
- It is an error if a sequence of refine statements defines a circular sequence of refinement files.
- The .java source file and the .class file for the given fully-qualified type are found as defined by Java, independently of determining the refinement sequence, using the sourcepath and the classpath, respectively. Note that if the .java file declares more

than one type, then there may be more than one .class file relevant to the refinement sequence.

- Note that if the .java file declares more than one type, then the files of the corresponding refinement sequence must contain the specifications for all of the declared types.
- The specifications for the classes declared in the files of a refinement sequence are the combination of the specifications in all the files of the refinement sequence. The Java signature of the classes is obtained from the relevant .java or .class files. The source code implementation of the classes is determined from the .java file. If no .java file exists (or it does not contain an implementation of a method), then the check of that method will be a trivial pass; checks of the usage of the method within other routine bodies will still be performed.
- It may be that a refinement sequence exists and does not contain the file specified on the command-line. ESC/Java2 issues a caution to the user in this case. However, if no refinement sequence is found, no caution is issued even if the command-line file is not on the sourcepath; it is simply used as the specification of the declared classes.
- It may be that the refinement sequence exists, and a corresponding .java file exists in the sourcepath, but the refinement sequence does not contain the .java file. In this case the .java file is used as the source code of the implementation and to define the signature of the class, but no specifications are obtained from it. A caution is issued to the user in this case.
- It is not required that a .java or a .class file exist, since it is desired to be able to write specifications in advance of an implementation. However, if either one does exist then the following rules are used:
   a. if the implementation is needed (because this is a type whose implementation is being checked by ESC/Java2) as well as the signature, then the .java file is used for both the implementation and the signature regardless of time stamp, if it exists.
   b. if only a signature is needed, then which file is used is determined by a command-line option (e.g. `-preferSource` as described in Section 4.3 [Command-line options], page 61). The default is to use the most recently modified of the two to define the Java signature of the class (whether or not it is in the refinement sequence). Specifications are not permitted to add new (non-model, non-ghost) declarations of fields, routines, or enclosed classes or interfaces to those defined in the Java implementation.
- Currently, .class files do not contain specifications. However, we would like to leave open the possibility that in the future a binary version of parsed and checked specifications could be created that would improve processing time.

Status: The above rules are implemented in ESC/Java2 with the following exceptions.

- ESC/Java2 does not yet use the search order for the MRCU as described. Rather it finds the file with the most active suffix anywhere in the sourcepath, regardless of its position in the sourcepath.
- When finding a package named as an input entry, ESC/Java2 combines all of the packages by that name in any directory of the sourcepath, rather than just using the first one.

3.2 Format of annotations

- **Comment format:** JML annotations are included in a Java program as specially formatted comments. In particular, JML annotations recognized by ESC/Java2 are either
  - single-line comments beginning with `//@`, or
  - multi-line comments enclosed between `/*@` and either `*/` or `@*/` , or
  - annotations embedded in a javadoc comment between any of the four pairs of markers `<esc>` and `</esc>`, `<ESC>` and `</ESC>`, `<jml>` and `</jml>`, or `<JML>` and `</JML>`. The original ESC/Java only recognized the first pair. These annotation pairs may not be nested, but there may be multiple annotations in sequence. ESC/Java2 and JML do not restrict where in the javadoc comment an annotation may occur. However, javadoc requires the annotation to be a part of the textual description and to precede any tag descriptions that are part of the comment. The jmldoc tool allows multiple annotations to be intermixed with the tag descriptions. Neither ESC/Java2 nor the JML tools require the annotation to be enclosed between `<pre>` and `</pre>` tags; however, if you expect reasonable formatting in a javadoc-produced HTML page, you will likely wish to do so. The jmldoc tool does not require `<pre>` and `</pre>` tags to produce good formatting.

  Both the JML tools and ESC/Java2 allow multiple `@` symbols in the opening and closing comment markers (e.g. `//@@@@` is equivalent to `//@`).

  ESC/Java2 recognizes these additional comment forms:
  - single-line comments beginning with `//-@` ;
  - multi-line comments enclosed between `/*-@` and either `*/` or `@*/` ;

  These are used for (primarily experimental) constructs that are known to ESC/Java2 but are not part of JML.

  Note that JML recognizes additional annotations in these forms:
  - single-line comments beginning with `//+@` ;
  - multi-line comments enclosed between `/*+@` and either `*/` or `@*/` or `@+*/` ;

  These latter forms are part of JML but not ESC/Java2 to allow for syntax defined by JML but ignored by ESC/Java2. It is hoped that the result of the current work on ESC/Java2 will diminish the need for the JML-only comments. They may remain useful as a way to retain JML annotations that are not processed (though they could be) by ESC/Java2.
  Status: All of these annotation markers are implemented.
  Differences: None.

  There is also an interaction between javadoc comments and embedded annotations of which the annotation writer should be aware. Consider the text

```
/**  Javadoc material.
<esc>
... annotations ...
</esc>
     More javadoc material.
*/
public void m();
```

It is somewhat ambiguous as to whether (a) to associate 'Javadoc material' with the embedded annotations and 'More javadoc material' with the method declaration or (b) to associate all of the javadoc material with the method declaration. The javadoc tool will do the latter, but the writer, and the flow of the text, may well have meant the former. It is better to avoid embedded annotations if this confusion may arise.

- **-parsePlus option:** The `-parsePlus` command-line option instructs ESC/Java2 to parse all annotations recognized by JML (particularly including the `//+@` and `/*+@` annotation markers). This is used mainly in testing to find and attempt to process the JML-only annotations, but may be useful in other circumstances. See [-parsePlus], page 62.
- **Initial '`@`' symbols in annotations:** Within a multi-line annotation, a sequence of '`@`' symbols that follow whitespace at the beginning of a line are treated as white space. Within an annotation embedded in a Javadoc comment, a sequence of '*' symbols (but not '`@`' symbols) that follow whitespace at the beginning of a line are treated as white space.
- **Splitting annotations across comments:** JML tools will parse and process annotations that are split across multiple comments (e.g. a multi-line annotation in which each line begins with `//@`). ESC/Java2 expects an annotation to be entirely contained within one single- or multi-line comment. The latter behavior is 'correct' JML; however, the JML tools will correctly process and not warn about annotations split across multiple comments. To be specific:
  - ESC/Java2 requires that any clause beginning with a keyword (e.g. invariant, requires) and ending with a semicolon must be contained within one annotation comment. For example, write

    ```
    //@ requires i != 0 && j != 0;
    ```

    or

    ```
    /*@ requires i != 0 &&
      @          j != 0;
      @*/
    ```

    not

    ```
    //@ requires i != 0 &&
    //@          j != 0;
    ```

  - ESC/Java2 requires that model methods, model constructors and model programs be defined within one annotation comment. For example, write

    ```
     /*@ public model int m(int i, int j) {
                 return i+j;
             }
       @*/
    ```

    not

    ```
    //@ public
    //@ model int m(int i, int j) {
    //@          return i+j;
    //@ }
    ```

   - The tool also requires that a Java modifier (e.g. `public`) be in the same comment as a JML annotation (e.g. `behavior` or model method) that it modifies. For example, write

     ```
     //@ public behavior
     ```

     not

     ```
     //@ public
     //@ behavior
     ```

   - Finally, any `in` or `maps` clauses following a ghost or model field declaration must be within the same annotation comment as the declaration. Thus, write

     ```
     //@ model T t; in a;
     ```

     not

     ```
     //@ model T t;
     //@ in a;
     ```

  Thus, `requires` and `ensures` clauses must each be wholly within a single annotation comment; individual keywords such as `pure`, `normal_behavior`, `also`, `{|` or `implies_that` may be in annotation comments by themselves (with any relevant access modifiers) if permitted JML.

- **Multiple annotations per comment:** It is legal JML to include multiple annotations per comment; in fact it is common practice and good style to include many related annotations within one multi-line comment. ESC/Java2 supports this practice (though ESC/Java had some difficulties).
- **Terminating semicolons:** JML requires annotations to be terminated by semicolons. The original ESC/Java did not. The absence of semicolons is illegal JML, but is sometimes tolerated by ESC/Java2. ESC/Java2 will warn if a semicolon is missing. Such warnings can be suppressed with the `-noSemicolonWarnings` command-line option.

3.3 Compilation unit annotations

Compilation unit annotations are placed prior to the declaration of any type within the compilation unit.

3.3.1 refine statements

- **Description:** A JML refine statement indicates that the containing compilation unit adds additional specifications to those contained in the referenced file. If present, it must be located after any Java package statement and before any Java or model import statements. There may be only one refine statement in a compilation unit. It has the form

  `//@ refine "`*filename*`";`

  The refine statements define a *refinement sequence* as described in Section 3.1 [File finding], page 5. Here we focus on the combining of the compilation units in a refinement sequence to produce a single set of specifications for a type. Each compilation unit has its own set of declarations and specifications, all of which must be consistent. They are subject to the following rules, violations of which provoke error messages.

- All files of the refinement sequence must belong to the same package (though not necessarily the same directory); the type names of the declared types must be identical (including case).
- If a .java or a .class file exists for a type, the specifications may not add any Java (that is, non-model, non-ghost) declarations to the signature. They may only repeat declarations. The specification files may declare specifications for a method that is not implemented in the Java implementation if the declaration overrides a method in a superclass or superinterface or, for interfaces, a method in `java.lang.Object`. This enables the specification writer to write specifications for a routine in a class or interface that must be obeyed by subtypes, even if the class or interface itself does not provide a new implementation.
- If a field is redeclared, it must be redeclared with the same type and the same Java modifiers. An initializer of a java field may be present only in the .java file. An initializer of a ghost field may be declared in only one file of a refinement sequence.
- These JML modifiers must be consistent across all redeclarations of a field: `model`, `ghost`, `instance`. The modifiers `spec_public`, `spec_protected`, `non_null`, `nullable` and `monitored` may be added by a refinement file, but may not be removed.
- If a method or constructor is redeclared, it must be redeclared with the same return type, the same Java modifiers, and the same names for its formal parameters. An implementation may be present only in the .java file. (The restriction on the formal parameter names is to simplify reading and to avoid having the implementation have to rename variables in specifications.)
- These JML modifiers must be consistent across all method and constructor redeclarations: `model`. These JML modifiers may be added by a refinement but may not be removed: `spec_public`, `spec_protected`, `helper`, `non_null`, `pure`.
- The Java modifier `final`, as applied to a formal parameter, must be consistent across all redeclarations of a method or constructor. The JML modifier `non_null` may be added, but not removed.
- If a refinement file redeclares a method or constructor from a previous refinement, or if the method is overriding a method in a superclass or interface (including the case where a type redeclares a method with specifications even though there is no Java declaration), the specification for that redeclared or overriding method must begin with '`also`' (and must not begin with '`also`' when those conditions are not satisfied).
- A type redeclaration must have the same set of Java modifiers. In addition the JML modifier `model` must be consistent; the JML modifiers `pure`, `spec_public`, and `spec_protected` may be added by a refinement but not removed.

- **Status:** The refine statement is partly implemented in ESC/Java2; not all the rules above regarding consistency of modifiers are enforced.
- **Comment on combining refinements:** There are (at least) 3 ways to carry out the combining of refinements:

a. by syntactically combining the relevant text;
b. by typechecking each compilation unit independently and then combining the signatures;
c. by typechecking each compilation unit in turn, in the context of the compilation units it is refining.

ESC/Java2 uses (a).

3.3.2 model import statements

- **Description:** A model import statement has the form

  `//@ model` *java-import-statement*`;`

  Note that simply writing

  `//@` *java-import-statement*`;`

  is not legal JML. A model import statement may occur wherever a Java import statement may be placed. A model import statement introduces types that are used only by annotations. Annotations may also use types introduced by Java import statements.
- **Status:** Model import statements are fully implemented.
- **Differences from JML or Java:** This feature is implemented in ESC/Java2 as it is in JML. However, both have the following problem. The model import statements are parsed by JML tools and by ESC/Java2 as if they were Java import statements. Thus they may introduce or resolve an ambiguity in class name resolution of names used in the Java source code in a compilation unit, or cause misinterpretation of a type name. For example, in

  ```
  import java.io.*;
  //@ model import myclasses.File;
  public class C extends File {}
  ```

  the use of `File` as the superclass is interpreted as `java.io.File` by the Java compiler but as `myclasses.File` by JML tools and ESC/Java2. Similarly, in

  ```
  import java.io.*;
  //@ model import myclasses.*; // class myclasses.File exists
  public class C extends File {}
  ```

  the use of `File` is interpreted as `java.io.File` by a Java compiler but will be deemed ambiguous between `java.io.File` and `myclasses.File` by the JML and ESC/Java2 tools. These are as yet unresolved bugs.
- **Comment:** This form is also illegal:

  `/*@ model @*/ import` *typename*`;`

  Either use a Java import statement (without a `model` keyword) or enclose the entire model import statement in an annotation comment.

3.3.3 automatic imports

- **Description:** In Java programs, the package `java.lang.*` is automatically imported into each compilation unit. Similarly, in JML, the package `org.jmlspecs.lang.*` is automatically imported, as a model import, into each compilation unit.
- **Status:** Fully implemented in ESC/Java2.
- **Differences:** None.

3.4 Access (privacy) modifiers

Java allows the programmer to modify fields, methods, constructors, class and interface declarations with one of the privacy or access modifiers `public`, `protected`, `private` or to omit these implying default (or package) access. These modifiers affect the visibility of the associated declaration in other classes. ESC/Java2 issues compile-time errors for (some) misuses of access, but the access of any given syntactic entity does not affect the static checking that is performed.

JML also imposes some rules about access modifiers. Some JML constructs are allowed to be modified by an access modifier: the class-level clauses described in Section 3.6 [Type Annotations], page 16, such as `invariant`, and the behavior and example keywords (`behavior`, `normal_behavior`, `exceptional_behavior`, `example`, `normal_example`, and `exceptional_example`). In addition the method-level clauses (e.g. `requires`, see Section 3.9 [Routine Annotation clauses], page 25) are assigned the privacy level of the behavior case of which they are a part (if in a heavyweight specification case) or the privacy level of the method they modify (if in a lightweight specification case). A specification clause may not use program entities with tighter access restrictions than it itself has. For example, a requires clause in a protected `normal_behavior` specification case may not use `private` fields.

Java program constructs that may be modified with an access modifier may also be modified with one or the other of `spec_public` and `spec_protected`. A program construct modified with `spec_public` is considered to have public access for any specification and may be used in any specification clause; a program construct modified with `spec_protected` may be used in any specification clause in a derived type. JML constructs may not be modified with `spec_public` or `spec_protected`.

Note that `spec_public`, but not `spec_protected`, was present in ESC/Java

Status: Parsing of access modifiers is fully implemented. The access modifiers do not affect static checking. Checking that access is used consistently is not implemented.

3.5 Type modifiers

A class may be modified with the Java modifiers `public`, `final`, `abstract` and `strictfp` and the JML modifiers `pure`, `model`, `spec_public`, and `spec_protected`. An interface may be modified with the Java modifiers `public`, `strictfp` and the JML modifiers `pure`, `model`, `spec_public`, and `spec_protected`. Nested classes and interfaces may have the additional modifiers `static`, `protected`, and `private`. The access modifiers are described in Section 3.4 [Access modifiers], page 13. In addition the superclass and superinterfaces may be modified with the keyword `weakly`.

3.5.1 pure (JML)

- **Description:** The `pure` modifier, when applied to a class or interface, indicates that every method and constructor of the class or interface is `pure`. Thus, no method may assign to variables other than those declared within the body of the routine. Constructors may only assign to the instance fields of the object being constructed (and its superclasses).
- **Status:** Parsed and fully implemented.
- **Differences from JML or Java:** None.
- **Comment:** A method inherits purity from the methods it overrides; that is, if an overridden method is pure, the overriding method will be pure whether or not it is declared pure. This is not the case for classes or for interfaces. A subclass may add non-pure methods, even if it has a pure superclass. Declaring a class pure is precisely equivalent to declaring all of its methods and constructors pure.

3.5.2 model (JML)

- **Description:** The `model` modifier indicates that the class or interface is only to be used in annotations. It is not part of the Java program.
- **Status:** Parsed and fully implemented.
- **Differences from JML or Java:** JML does not yet properly handle model classes, especially those at the top level. Both JML and ESC/Java2 parse model types as if they were Java types and so will not detected erroneous uses of model types in Java code; both tools may also have some related name lookup bugs.

3.5.3 weakly (JML)

- **Description:** This annotation is used to modify superclasses and superinterfaces in a class or interface declaration. An example of its syntax is this:

  ```
  public class A extends B /*@ weakly */
     implements C /*@ weakly */, D /*@ weakly */ { ... }
  ```

  The semantics are not described here.
- **Status:** Parsed and ignored by ESC/Java2.
- **Differences from JML or Java:** Parsed but ignored by ESC/Java2. This feature was not present in ESC/Java.

3.5.4 non_null_by_default (JML)

- **Description:** This annotation is used to denote a `non_null` default semantics for reference types in a given class. An example of its syntax is this:

  ```
  public /*@ non_null_by_default @*/ class A { ... }
  ```

  If a class is labeled with the annotation `non_null_by_default` then every field, formal parameter, and method has a default specification of `non_null`. Local variables do not have any default specification.

  If a class is annotated `non_null_by_default` and a formal parameter, method, or field is annotated `non_null` then the latter spec is redundant and the user is notified of such.

If a class is annotated `non_null_by_default` and a formal parameter, method, or field is annotated with `nullable` then this specified annotation overrides the class annotation and the reference may be null. The scope of this annotation to the class includes nested (possibly anonymous) classes, but is not inherited by subclasses.

If an interface may be annotated with `non_null_by_default`, then all constants, formal parameters, and methods in the interface have a default specification of `non_null`. All concrete implementations of this interface must be consistent with these default specifications.

- **Status:** The modifier is parsed. Implemented for method return types and parameters, but class level modifiers are ignored.

  Left to do: Fields and bound variables.
- **Differences from JML or Java:** In the current JML2 implementation the `non_null_by_default` annotation is file-scoped rather than class-scoped.

3.5.5 nullable_by_default (JML)

- **Description:** This annotation is used to denote a `nullable` default semantics for reference types in a given class. An example of its syntax is this:

  ```
  public /*@ nullable_by_default @*/ class A { ... }
  ```

  If a class is labeled with the annotation `nullable_by_default` then every field, formal parameter, and method has a default specification of `nullable`. Local variables do not have any default specification.

  If a class is annotated `nullable_by_default` and a formal parameter, method, or field is annotated `nullable` then the latter spec is redundant and the user is notified of such.

  If a class is annotated `nullable_by_default` and a formal parameter, method, or field is annotated with `non_null` then this specified annotation overrides the class annotation and the reference may be null.

  [[The scope of this annotation to the class to which it is applied as well as nested (possibly anonymous) classes. The class-scoped annotation is not inherited.]]

  [[An interface may be annotated with `nullable_by_default`. All constants, formal parameters, and methods in the interface have a default specification of `nullable`. All concrete implementations of this interface must be consistent with these default specifications.]]
- **Status:** The modifier is parsed. Typechecking implementation is underway. This modifier is not yet used by the static checker.
- **Differences from JML or Java:** In the current JML2 implementation the `nullable_by_default` annotation is file-scoped rather than class-scoped.

3.5.6 final (Java)

- **Description:** A final class may not have subclasses.
- **Status:** Parsed, typechecked, and used by the static checker.
- **Differences from JML or Java:** None.

3.5.7 abstract (Java)

- **Description:** A class must be declared abstract if it has abstract methods. An abstract class may not be instantiated; only non-abstract subclasses of an abstract class may be instantiated. All interfaces are by definition abstract; using the `abstract` modifier on an interface has been deprecated.
- **Status:** This modifier is parsed and checked. It does not need any static checking.
- **Differences from JML or Java:** None.

3.5.8 strictfp (Java)

- **Description:** The `strictfp` modifier determines the semantics of floating point operations within the class so modified.
- **Status:** This modifier is parsed and typechecked. The static checker does not make use of this information.
- **Differences from JML or Java:** None.

3.5.9 static (Java)

- **Description:** `static` is a Java modifier that may be applied to classes and interfaces that are members of enclosing classes.
- **Status:** Implemented.
- **Differences from JML or Java:** None.

3.6 Annotations pertinent to a class or interface

These annotations may appear anywhere a declaration within a class or interface may appear. They define specification-only ghost or model fields of the type and state specifications that apply to the whole object (not just to individual methods).

3.6.1 Ghost fields

- **Description:** A ghost field is a field of the object that can hold a primitive value or a reference to an object, but is used only in specifications. Its value is changed using the `set` annotation within the body of a method or constructor (see Section 3.14.3 [set], page 36). A ghost field may have an initializer, just as a Java program field may, but the ghost field may be initialized in only one compilation unit of a refinement sequence. A ghost field may have modifiers that a Java field declaration would have (namely, access modifiers (see Section 3.4 [Access modifiers], page 13), `static`, `final`, but not `volatile`, `transient`) as well as the JML modifiers `non_null`, `nullable`, `monitored` and `instance` (see Section 3.12 [Field Annotation modifiers], page 33).

  An interface may also declare ghost fields; these fields may be referenced by annotations in the interface or its subtypes. Such ghost fields are by default static, but may be modified by the JML modifier `instance`, in which case they are a field of every object that implements the interface.
- **Status:** Ghost fields are completely supported.
- **Differences from JML or Java:** None.

3.6.2 Model fields

- **Description:** Model fields are declarations within an annotation prefixed by the modifier `model`. They do not represent actual specification fields as do `ghost` values. Rather, their values are implied by the concrete representation of the class, either by an explicit expression in a `represents` clause or implicitly by a boolean condition in a `\such_that` form of the `represents` clause. They are used to supply values that *model* the behavior of the class.

  A model field may have these modifiers: access modifiers (see Section 3.4 [Access modifiers], page 13), `static`, and the JML modifiers `non_null`, `nullable`, and `instance` (see Section 3.12 [Field Annotation modifiers], page 33). Model fields may not have initializers.
- **Status:** Model fields are parsed and used in typechecking, but ESC/Java2 does not find inconsistencies between multiple represents clauses.
- **Differences from JML or Java:** None.

3.6.3 Model methods

- **Description:** Model methods are method declarations within an annotation and modified with the modifier `model`. They declare methods that may be used in model programs and (if pure) in specifications. Model methods may have these Java modifiers: `public`, `protected`, `private`, `static`, `abstract`, `final`, `synchronized`, `strictfp`; they may have these JML modifiers: `pure`, `non_null`, `nullable`, and `helper`.

  Model methods may be declared in multiple specification files, but may have an implementation in at most one.
- **Status:** Model methods are parsed and converted to regular Java methods within ESC/Java2.
- **Differences from JML or Java:** Model methods are parsed and converted to regular Java methods within ESC/Java2. Consequently, ESC/Java2 will not detect their (illegal) use within the implementation or inheritance of a Java routine.

3.6.4 Model constructors

- **Description:** Model constructors are constructor declarations within an annotation and modified with the modifier `model`. They declare constructors that may be used in model programs and (if pure) in specifications. A model constructor may have these Java modifiers: `public`, `protected`, `private`; it may have these JML modifiers: `pure`, `helper`. Constructors may be `strictfp` only by virtue of the entire class being declared `strictfp`.

  Model constructors may be declared in multiple specification files, but may have an implementation in at most one.
- **Status:** Model constructors are parsed and converted to regular Java constructors within ESC/Java2.
- **Differences from JML or Java:** Model constructors are parsed and converted to regular Java constructors within ESC/Java2. Consequently, ESC/Java2 will not detect their (illegal) use within the implementation of a Java routine.

3.6.5 Model class and model interface declarations

- **Description:** A model type (class or interface) declaration is a conventional type declaration modified by the JML keyword `model` (and in an annotation comment). The entire declaration must be within one annotation comment. The model type may be used within annotation expressions and statements. Model types may have the same modifiers as top-level and nested java type declarations, as appropriate (see Appendix A).
- **Status:** Implemented.
- **Differences from JML or Java:** Model types are in the same name space as conventional Java types. Thus in some cases the resolution of a type name in Java code could resolve to a model type name rather than to the correct Java type. This is a bug in the name scoping of both JML and ESC/Java2; the workaround is to rename the model type so that it does not hide a Java type name.

3.6.6 Java initializer blocks

- **Description:** Java permits blocks of code within braces in the body of a class (but not interface) declaration. In the process of loading a class, each initializer of a static field and each initializer code block with a static modifier is executed in textual sequential order. Similarly, when an instance of a class is created, each initializer of a non-static field and each non-static initializer block is executed in textual sequential order. Each initializer block may be preceded by a specification, just like a method specification. The preconditions and postconditions specified must hold just before and just after the execution of the initializer block. Note that no invariants or other class-level specifications are required to hold until all static initialization and class loading is complete (for static invariants) or until a constructor has completed execution (for instance invariants).
- **Status:** Most specifications are parsed but not all and no reasoning is implemented.
- **Differences from JML or Java:** None.

3.6.7 initializer

- **Description:** The JML `initializer` and `static_initializer` keywords are used in specification files as stand-ins for all of the instance and class initialization that is performed as part of object creation or class loading. Within a class declaration in a compilation unit there may be just one each of the `initializer` and `static_initializer` keywords, each preceded by specifications (like those preceding a routine declaration or a Java initialization block). If more than one compilation unit of a refinement sequence has these keywords, then the associated specifications are combined just like routine specifications are combined. The composite specifications associated with an `initializer` keyword give preconditions that must hold before any instance initialization and postconditions that must hold after any instance initialization (but before constructors are executed). Similarly, the specifications of a `static_initializer` keyword hold before and after the static initialization of the class. Note that these are different than the specifications for a Java initializer block, which apply only to that block.
- **Status:** Not yet parsed or implemented in static checking.

- **Differences from JML or Java:** None.

3.6.8 static_initializer

- **Description:** See the description above.
- **Status:** Not yet parsed or implemented in the static checker.
- **Differences from JML or Java:** None.

3.6.9 Java method, constructor and field declarations

- **Description:** These declarations are identical to those defined by Java. An implementation or initialization for such a declaration may appear only in the .java file, not in any repeated declaration in a specification file. The modifiers allowed are listed in Appendix A.
- **Status:** Parsed and typechecked fully. Java fields may be used in annotations. JML and ESC/Java2 also allow pure methods and pure constructors to be used in annotations.
- **Differences from JML or Java:** None

3.6.10 Nested Java type declarations

- **Description:** Java allows declarations of classes and interfaces within a class or interface. These are called nested classes or interfaces. Inner classes or interfaces are nested classes or interfaces that are not `static`. The modifiers allowed are listed in Appendix A.
- **Status:** Implemented.
- **Differences from JML or Java:** None

3.7 Annotation clauses for a class or interface

These clauses provide a specification of the behavior of the class and of objects of the class. They may be specified in any order, within annotation comments, anywhere an element of a type declaration may appear. They may individually have access modifiers (see Section 3.4 [Access modifiers], page 13) `public`, `protected`, or `private`.

3.7.1 invariant, invariant_redundantly

- **Description:** An `invariant` clause specifies a boolean condition that must hold before and after any call of a (non-helper) method of the containing type. Invariants must hold after any (non-helper) constructor call of the containing type. In checking the implementation of a method, invariants are assumed as part of the preconditions and must be established as part of the postconditions.

  An `invariant` clause may be declared `static`, in which case it may only reference static fields and routines.
- **Status:** Fully implemented, except that ESC/Java2 does not check the restriction on static invariants.
- **Differences from JML or Java:** None

3.7.2 constraint, constraint_redundantly

- **Description:** A `constraint` clause specifies a relation that must hold between the pre- and post-conditions of any (non-helper) method of the containing type. If the clause is declared `static`, then all field and routine references within the constraint predicate must be static.

- **Status:** Parsed and typechecked. The static checker issues a warning if the constraint is not true as part of the postconditions of any method (but does not check constraints in association with constructors).
- **Differences from JML or Java:** ESC/Java2 does not check the restriction on static constraints.

3.7.3 represents, represents_redundantly

- **Description:** A `represents` clause designates how a model field is related to the concrete fields or other model fields of the implementation. The represents clause must be declared static when and only when the model field for which it is providing a representation is declared static; if the clause is static then all field and routine references must be static.
- **Status:** Parsed, typechecked, and used by the static checker.
- **Differences from JML or Java:** None. ESC/Java2 does not check the restrictions on static.

3.7.4 axiom

- **Description:** An `axiom` is used to specify a mathematical property, independent of the implementation of classes or objects. Axioms are always considered to be static.
- **Status:** Implemented.
- **Differences from JML or Java:** None.

3.7.5 initially, initially_redundantly

- **Description:** This clause specifies a condition that must hold in the post-state of any (non-helper) constructor (including the default constructor). Within the body of a constructor, any superclass `initially` clauses are assumed to hold after the execution of a (non-helper) `super(...)` call, including a possible implied call of the default superclass constructor. A class does not inherit any superclass `initially` clauses; `initially` clauses are not permitted in interfaces. If the clause is declared static, it may only reference static fields and routines.
- **Status:** Implemented. `Initially` clauses are typechecked as additional postconditions on every constructor of a class. Failures provoke an 'Initially' warning.
- **Differences from JML or Java:** None.

3.7.6 readable and writable

- **Description:** JML allows class-level clauses of the form

  `readable` *field* `if` *predicate* ;

  and

  `writable` *field* `if` *predicate* ;

These specify a predicate that must be true in the state in which a read or write access of a class field is attempted. These are useful to specify the access protocol for a variable shared across threads, but can also be used simply to indicate under what circumstances a field's value is defined.

- **Status:** Implemented. However, the semantics is not well-defined for the situation in which the field referred to in the clause is declared in a superclass of the class declaration containing the clause.
- **Differences from JML or Java:** ESC/Java also allows a `readable_if` and `writable_if` modifier for field declarations (each taking simply a predicate and are positioned just prior to the field declaration, as, for example, a `non_null` modifier would be). `readable_if` is deprecated in JML and `writable_if` is not defined at all. Hence these forms are discouraged in ESC/Java2 as well.

3.7.7 monitors_for

- **Description:** This clause associates a list of expressions with a given field name. The field identified must be a field of the class containing the declaration. All of the expressions must evaluate to objects (not to primitive values). If the field is static, all of the objects must be static. The effect is to associate the expression values as monitors for the given object.
- **Status:** Implemented.
- **Differences from JML or Java:** None.

3.8 Annotations for a method or constructor

Specifications of the behavior of an individual method or constructor typically appear within an annotation comment, just prior to the declaration of the method or constructor. The specifications consist of zero or more lightweight or heavyweight behavior sections, an optional `code_contract` section, an optional `implies_that` section, and an optional `for_example` section. Model methods and constructors may also be annotated with these specifications.

3.8.1 Lightweight and heavyweight specifications

- **Description:** Lightweight specification cases are simply a series of specification clauses and correspond to the specification style of ESC/Java. Heavyweight specification cases are introduced with a `behavior`, `normal_behavior`, or `exceptional_behavior` keyword. Heavyweight specifications may have optional privacy modifiers (see Section 3.4 [Access modifiers], page 13) and have different defaults than do lightweight specifications.

  Within a heavyweight specification, if a particular clause type is omitted, the default for that clause is as follows:

  ```
  ensures true;
  signals (java.lang.Exception) true;
  diverges false;
  assignable \everything;
  accessible \everything;
  callable \everything;
  ```

```
when true;
duration \not_specified;
working_space \not_specified;
```

The defaults defined by JML for lightweight specifications are `\not_specified` in each case. This is interpreted within ESC/Java2 as follows.

```
ensures true;
signals (java.lang.Exception) true;
diverges true;
assignable \everything;
accessible \everything;
callable \everything;
when true;
duration \not_specified;
working_space \not_specified;
```

The default for the requires clause is determined as follows:[1]

- If there are some other clauses explicitly given, but no requires clause, the default is `requires true;`
- If there is no specification at all and the routine is a constructor or is a method that does not override any superclass or superinterface methods, the default is `requires true;`
- If there is no specification at all (including no `non_null` modifier) and the method does override some superclass or superinterface method, the default is `requires false;`
- For the default constructor, if no specifications have been given, the default requires specification is the same as the requires specification of the corresponding superclass constructor.

The reasons for these defaults are explained in Section 3.19.13 [Specifications and inheritance], page 51.

The modifies clause has this exception to the above rule: For the default constructor, if no specifications have been given, the default modifies specification is the same as the modifies specification of the corresponding superclass constructor. (This is not a textual replication; rather the subclass constructor may modify the same set of locations, under the same conditions, as the superclass constructor.) Note that the defaults for the `diverges` clause are different between the lightweight and heavyweight forms. The lightweight default, `diverges true`, puts the least restrictions on the implementation; it states that the implementation is allowed to be non-terminating (but not required to be). The heavyweight default, `diverges false`, requires the implementation to terminate with either an exception or a normal return, but this is not checked by ESC/Java2.

- **Status:** The lightweight and heavyweight specification forms, including nesting using `{|` and `|}` and combination with `also`, are fully parsed and implemented (though not all clause types, as described below, are fully implemented or handled by the static checker).

[1] Michael Möller contributed to this formulation of the defaults for requires.

- **Differences from JML or Java:** None, except that ESC/Java2 does not constrain the order of clauses as rigidly as do JML tools. JML requires forall, old and requires clauses to precede any other clauses; with a warning level of -w2 (not the default), JML will also warn about deviations from a recommended order of the other clause types. ESC/Java2 will accept clauses in any order (but note that the scope of `forall` and `old` does not include clauses that precede them). ESC/Java2 does not check the access modifiers on behavior and example keywords.
- **Comment:** The ESC/Java2 (and ESC/Java) translator, which produces the verification conditions to be checked by the static checker, accepts a set of specification clauses in lightweight form. In order to handle the nested and heavyweight forms and the combination of specifications using `also`, either within one source file or across a refinement sequence, ESC/Java2 *desugars* the more complicated syntax into a simpler form. For this purpose ESC/Java2 does not need to desugar all clauses of a given type down to one instance of that clause type. The equivalent but slightly simpler version used in ESC/Java2 is outlined here. Note that the desugaring process must take care not to lose the location information that is helpful to the user when warning messages are issued.

  The desugaring process first eliminates nesting by replicating and distributing the `forall`, `old`, and `requires` clauses across the nested groups of clauses. Any `old` clauses are desugared by replacing any references to them by the expression with which they are initialized (evaluated in the pre-state); any `forall` clause is desugared by wrapping any clause within its scope in a `\forall` quantified expression. Also, each lightweight specification case, `normal_behavior` and `exceptional_behavior` keyword is desugared into a `behavior` specification. Specifications from corresponding methods in the refinement sequence are combined, connected by `also`. That produces a series of specification cases, connected by `also`, each consisting of one group of clauses (that is, one specification case). ESC/Java2's static checker will accept a single specification case. For each specification case, a composite precondition predicate is formed by taking the conjunction of the predicates in each of the `requires` clauses in that specification case, and making that conjunction the argument of an `\old` function:

  pre-predicate = `\old(`*p1* `&&` *p2* ... `)`

  Each clause within the specification case is altered by constraining its action using that conjunction. Calling that combined predicate *pre-predicate*, we transform each clause as follows:

  - `ensures` *pred*`;`
    becomes
    `ensures` *pre-predicate* `==>` *pred*`;`
  - `diverges` *pred*`;`
    becomes
    `diverges` *pre-predicate* `==>` *pred*`;`
  - `modifies` clauses: Multiple modifies clauses within one specification case are combined into one clause, along with the composite precondition for that specification case; modifies clauses from different specification cases are not combined.

  - `signals` (*type id*) *pred*;
    becomes
    `signals` (*type id*) *pre-predicate* `==>` *pred*;

  The requires clauses are desugared by replacing all of the requires clauses in all of the specification cases with a single requires clause whose predicate is the disjunction of the conjuctions formed for each specification case, without the enclosing `\old`. When the static checker creates a VC to be used as a precondition for calling a method, it forms a disjunction of the requires clauses for the method and all the declarations that it overrides.

3.8.2 also

- **Description:** JML allows multiple specifications for a single method declaration; these are separated and connected by the `also` keyword. Furthermore, if the method has additional declarations (with or without specifications) in an earlier source file in the refinement sequence or an overridden method in a superclass or interface, then (and only then) the specification must begin with `also` to indicate that there are some previous declarations, with possible specifications, of which the reader should be aware.
- **Status:** The use of `also` is completely implemented, with desugaring occurring as described above.
- **Differences from JML or Java:** None.
- **Comment:** This syntax for combining specifications is different than and not backwards compatible with the syntax used in ESC/Java. That tool did not allow combining multiple specifications using `also` and did not support refinement sequences. Where there was inheritance of specification clauses from a superclass or interface, the keywords `also_requires`, `also_ensures`, `also_modifies`, and `also_exsures` were required. These keywords are not supported in ESC/Java2 and such specifications will need to be rewritten using the new `also` syntax.

3.8.3 model_program

- **Description:** Model programs are an alternate way to provide specifications for a method. Rather than stating logical conditions that the pre- and post-states must satisfy, the behavior is specified by a *model program*, which specifies the behavior using typical imperative programming constructs. However, a model program allows some non-deterministic (and non-executable) constructs as well. Note that there are a number of JML constructs which are only used within model programs. These are described in Section 3.18 [Statements within model programs], page 46.
- **Status:** Model programs are parsed and ignored. They are permitted as a specification case, as defined by JML. Any constructs unique to model programs are simply skipped over by the parser.
- **Differences from JML or Java:** None.

3.8.4 code_contract

3.8.5 implies_that

- **Description:** The `implies_that` keyword introduces specification cases that are logical consequences of the usual behavior and lightweight specifications. They could be used as specifications to be checked in the same way that the other specifications are. Alternatively, it could be verified that they are logical consequences of the other specifications; the results could then be used as additional useful statements of behavior; these in turn could help with proofs involving use of the method or constructor with which the `implies_that` specification is associated.

- **Status:** The specifications in an `implies_that` section are parsed, but not used within any static checking.
- **Differences from JML or Java:** None.

3.8.6 for_example specification

- **Description:** The `for_example` keyword introduces specification cases that are useful and instructional examples for the reader of the specifications. Hence they must be logical consequences of the other specifications. Each case may be lightweight or be introduced by one of the keywords `example`, `normal_example`, and `exceptional_example`; these keywords may have associated access modifiers (see Section 3.4 [Access modifiers], page 13).
- **Status:** The specifications in a `for_example` section are parsed, including the `example`, `normal_example`, and `exceptional_example` keywords. However, they are not used within any static checking, nor is it verified that they follow from the other specifications.
- **Differences from JML or Java:** None.

3.9 Annotation clauses for a method or constructor

In this section we describe the clause types that may be part of specifications, including `implies_that` and `for_example` sections. Note that some clauses have alternate keywords reflecting different personal preferences or different usages among ESC/Java2 and other JML tools. These alternates are complete synonyms.

3.9.1 forall

- **Description:** The `forall` declaration declares a universally quantified variable; the scope of the declaration is all subsequent clauses for the same routine up to the `also` or `|}` marking the end of the specification case containing the `forall` declaration, or until end of the behavior, implies_that or for_example section. No initializer is allowed. The clause is desugared by wrapping each desugared clause that is in scope in an appropriate `\forall` expression.
- **Status:** Fully implemented. However, the semantics need clarifying and the static checker objects to quantified expressions.
- **Differences from JML or Java:** None.

3.9.2 old

- **Description:** The `old` declaration is used within a routine specification to define a value that may be used in subsequent clauses of the specification. The variable declared must be initialized. The initialization expression is always evaluated in the pre-state, regardless of how the variable is used in subsequent clauses. The scope of the variable extends from its declaration (including the initializer), to the 'also' or '`|}`' that marks the end of the specification case sequence containing the `old` declaration, or until the end of the behavior, implies_that or for_example section. The uses of `old` variables are desugared by substituting the initialization expression, wrapped in an appropriate `\old` expression, at the point of use.

- **Status:** Implemented, but array initializers are not supported.
- **Differences from JML or Java:** None.

3.9.3 requires, requires_redundantly, pre, pre_redundantly

- **Description:** A requires clause specifies a condition that must hold in the pre-state of the method, in the context where it is called in the program. The remaining clauses of the specification case must hold whenever the requires clause (or the conjunction of multiple requires clauses) holds. The expression in the clause may also be `\not_specified`, which is equivalent to omitting the clause. The expression must have boolean type and is evaluated in the pre-state.
- **Status:** The requires clause is implemented and is utilized by ESC/Java2 in generating verification conditions.
- **Differences from JML or Java:** None.

3.9.4 ensures, ensures_redundantly, post, post_redundantly

- **Description:** An ensures clause states a condition that must hold in the post-state of a method or constructor whenever the associated preconditions hold in the pre-state and the method or constructor exits normally. The expression in the clause may also be `\not_specified`, which is equivalent to omitting the clause. The expression must have boolean type and is evaluated in the post-state.
- **Status:** Implemented and used by the static checker.
- **Differences from JML or Java:** None.

3.9.5 signals, signals_redundantly, exsures, exsures_redundantly

- **Description:** A signals clause states a condition that must hold in the post-state of a method or constructor whenever the associated preconditions hold in the pre-state and the method or constructor exits with an exception of (or a subclass of) the designated type. The expression in the clause may also be `\not_specified`, which is equivalent to omitting the clause. The expression must have boolean type and is evaluated in the post-state (though the keyword `\result` is not valid in a `signals` clause).
- **Status:** Implemented and used by the static checker.
- **Differences from JML or Java:** None.

3.9.6 modifies, modifiable, assignable, modifies_redundantly, modifiable_redundantly, assignable_redundantly

- **Description:** The modifies clause indicates which memory locations may be assigned within the associated routine when the routine is called in a pre-state that satisfies the associated preconditions. The list of locations in the modifies clause may also contain the special keywords `\nothing`, `\everything`, `\not_specified`, and the wild-card forms *expr*`.*` , *typename*`.*` , *array*`[*]`, and *array*`[`*expr* `..` *expr*`]`. Any expressions in the clause are evaluated in the pre-state.

Each location listed in the clause implicitly includes all the elements of its datagroup, if a datagroup is associated with the location. Other memory locations are added to a location's datagroup using the `in` and `maps` clauses (see Section 3.11 [Field Annotations], page 32).

- **Status:** Implemented and used by the static checker. ESC/Java2 checks that assignments within a routine are consistent with the routine's modifies clauses and checks that the modifies clauses of called routines are consistent with those of the caller.

  However, ESC/Java2 is not yet able to properly handle the forms `\everything`, `array[*]` and `array[e1..e2]` when in the modifies clause of a routine called within the body of a calling routine (and remember that `modifies \everything` is the default). For example, given

  ```
  public int i;
  public int[] a;

  //@ modifies \everything;
  void m() { ... }

  //@ modifies a[*];
  void n() { ... }

  //@ modifies a[2..10];
  void nn() { ... }

  //@ modifies \nothing;
  void p() {
          i = 0;
          m();
          //@ assert i == 0;
          a[3] = 0;
          n();
          //@ assert a[3] == 0;
          a[3] = 0;
          nn();
          //@ assert a[3] == 0;
  }
  ```

  ESC/Java2 should complain that the first assert statement in routine `p` is not established, because `m`, which claims to possibly modify everything, might have modified field `i`. It will complain that the modifies clauses of the two routines are in conflict. It will also complain about the assert statement if `m` had a modifies clause of `modifies i;`. Similarly the forms of array range designators in the modifies clauses of `n` and `nn` are not fully handled, so the second and third assert statements do not provoke complaints. Store-ref expressions that are specific array elements or the forms `expr.*` and `typename.*` are handled properly.

  A second aspect of modifies clauses not being fully handled is the following. Within a class that has a field named `i`

  ```
   //@ modifies this.*;
  ```

```
public void m() {
        n();
}

//@ modifies i;
public void n();
```

provokes no complaints, since `i` is recognized as part of `this.*`. However, in

```
//@ modifies i;
public void m() {
        n();
}

//@ modifies this.*;
public void n();
```

ESC/Java2 will issue a warning, since it does not check that all possible fields of the class (any of which might be modified by `n`) are listed in the modifies clause of `m`. The similar situation holds for static members.

3.9.7 diverges, diverges_redundantly

- **Description:** This clause states a predicate that must hold (in the pre-state) if the method never terminates (given that the associated precondition holds in the pre-state). The expression in the clause may also be `\not_specified`, which is equivalent to omitting the clause. The expression must have boolean type and is evaluated in the pre-state.
- **Status:** Parsed and typechecked, but not used in any static checking.
- **Differences from JML or Java:** None.

3.9.8 when

- **Description:** The expression in the clause may also be `\not_specified`, which is equivalent to omitting the clause. The expression must have boolean type and is evaluated in the pre-state.
- **Status:** Parsed and typechecked, but not used in any static checking.
- **Differences from JML or Java:** None.

3.9.9 duration

- **Description:** This specification asserts that the execution of the routine (with the given preconditions) will not exceed the stated number of virtual machine cycles. The expression in the clause may also be `\not_specified`, which is equivalent to omitting the clause. The expression must have `long` type and is evaluated in the post-state.
- **Status:** Parsed and typechecked but not used in any static checking.
- **Differences from JML or Java:** None.

3.9.10 working_space

- **Description:** This specification asserts that the execution of the routine will not utilize more than the stated number of bytes of heap space. The expression in the clause may also be `\not_specified`, which is equivalent to omitting the clause. The expression must have `long` type and is evaluated in the post-state.
- **Status:** Parsed and typechecked, but not used in any static checking.
- **Differences from JML or Java:** None.

3.9.11 accessible

- **Description:** The list of locations in the clause may also contain the special keywords `\nothing`, `\everything`, Any expressions (e.g. array indices) are evaluated in the pre-state.
- **Status:** Parsed and ignored.

3.9.12 callable

- **Description:** The list of locations in the clause may also contain the special keywords `\nothing`, `\everything`, Any expressions (e.g. array indices) are evaluated in the pre-state.
- **Status:** Parsed and ignored.

3.9.13 measured_by

- **Description:** The expression in the clause may also be `\not_specified`, which is equivalent to omitting the clause.
- **Status:** Parsed and ignored.

3.9.14 Redundancy

- **Description:** Many clauses have a redundant form, indicated by using a keyword with a **_redundantly** suffix. The intention of these clauses is to indicate specifications that are implied by other, nonredundant, specifications. The writer may choose to include the redundant specifications in order to point out some non-obvious implications of other specifications, either to facilitate understanding by the reader or to assist the prover in verifying conclusions.

  Note that the `implies_that` and `for_example` specifications are additional forms of redundancy.
- **Status:** Currently in ESC/Java2 a command-line option selects between using redundant specifications (those with keywords ending in `_redundantly`) in the same way as nonredundant specifications (the default) or ignoring them (when the option `-noredundancy` is chosen).

3.10 Annotation modifiers for a method or constructor

Annotation modifiers can appear between the last specification clause or javadoc comment and the type designator or class name that is part of the method or constructor. JML modifiers and Java modifiers may appear in any order. Though less common and not the usual style (and discouraged), ESC/Java2 (following ESC/Java) allows JML modifiers to appear after the method declaration and before the opening left brace of the body or the terminating semicolon if there is no body.

Besides the modifiers listed here, methods and constructors may also have the access modifiers described in Section 3.4 [Access modifiers], page 13.

3.10.1 pure (JML)

- **Description:** The `pure` modifier applied to a method indicates that the method does not assign to any non-local memory location during its execution; it may not even modify and then restore the original value. It is equivalent to having `modifies \nothing;` in the specification. In the case of a constructor, the only fields that may be modified are, at most, the fields of the object itself, which are initialized as a result of the action of the constructor. In this case the `pure` modifier is equivalent to specifications of the form

  ```
  modifies this.*;
  ```

  However, a constructor may have modifies clauses that are more restrictive than `modifies this.*;`. Note that if a method is declared pure, then all overriding methods are also pure, whether or not they have an explicit declaration to that effect. Note that a pure constructor may not modify static fields.

  It is also worth mentioning that it is possible to call non-pure methods from within the pure ones provided that the changed state is locally allocated in the pure method.
- **Status:** Fully supported.
- **Differences from JML or Java:** None.

3.10.2 non_null (JML - methods only)

- **Description:** Modifying a routine with the `non_null` modifier is valid only for methods that return objects as return values (and not for constructors). The modifier specifies that the return value is never null. It is equivalent to a specification of

  ```
  ensures \result != null;
  ```

  added to each specification case of the method's specification in all of the files of the refinement sequence (but not of a superclass's specification of that method).

  Note that superclass and subclass declarations of a method each independently may have or not have `non_null` declarations. A method's implementation must satisfy the superclass specification and independently satisfy the subclass specification. There is a more thorough discussion in Section 3.19.13.3 [Inheritance and non_null], page 54.
- **Status:** Parsed, typechecked and supported by the static checker.
- **Differences from JML or Java:** None.

3.10.3 nullable (JML - methods only)

- **Description:** Modifying a routine with the `nullable` modifier is valid only for methods that return objects as return values (and not for constructors). The modifier specifies that the return value may be null.

Note that this modifier is *not* the opposite of `non_null` since the negation of the equivalent specification would be

```
ensures !(\result != null);
```

which is equivalent to

```
ensures \result == null;
```

which is not the intent of `nullable`.

Note that superclass and subclass declarations of a method each independently may have or not have `nullable` (and `non_null`) declarations. A method's implementation must satisfy the superclass specification and independently satisfy the subclass specification. There is a more thorough discussion in ⟨undefined⟩ [Inheritance and non-null], page ⟨undefined⟩.

In short, a `nullable` method may be overridden by a `non_null` method but must then be `non_null` thereafter.

- **Status:** The modifier is parsed. Typechecking implementation is underway. This modifier is not yet used by the static checker.
- **Differences from JML or Java:** None.

3.10.4 helper (JML)

- **Description:** This modifier indicates that the method or constructor in question is used as an internal helper routine and that the method or constructor is therefore not expected to satisfy any of the class-level invariants or constraints, in either its pre-state or its post-state. The method or constructor is still expected to satisfy any specifications (e.g. ensures clauses) that are explicitly associated with this method or constructor.
- **Status:** Parsed, typechecked and supported by the static checker.
- **Differences from JML or Java:** JML only allows private methods and constructors to be helpers. ESC/Java2 also allows any constructors, final methods or methods of final classes to be helpers. (ESC/Java2's rule is that only routines that cannot be overridden may be helpers.)

3.10.5 final (Java - methods only)

- **Description:** This Java modifier indicates that a method may not be overridden. It may also be applied to model methods.
- **Status:** Fully implemented. This feature does not affect the static checking; it simply produces a typechecking error if a final method is overridden.
- **Differences from JML or Java:** None.

3.10.6 static (Java - methods only)

- **Description:** `static` is a Java modifier (which may be used on JML annotations as well) that indicates that the declaration in question is a member only of the class and not of each instance of the class.

- **Status:** Fully implemented.
- **Differences from JML or Java:** None.

3.10.7 synchronized (Java - methods only)

- **Description:** This modifier indicates that processing must wait until a monitor lock is available and must obtain that lock before the execution of the method is begun; the lock is released when the method execution is completed. Each object has, implicitly, a monitor lock associated with the object. Before executing an instance method, it is the lock associated with receiver object that is obtained; before executing a static method, it is the lock associated with the class object (T.class for type T) that is obtained.
- **Status:** Implemented.
- **Differences from JML or Java:** None.

3.10.8 native (Java - methods only)

- **Description:** A native method is one whose implementation is provided outside of the Java language. Consequently such a method will not have an implementation.
- **Status:** Fully implemented. The static checker will indicate that the method passes its checks since there is no body to check.
- **Differences from JML or Java:** None.

3.10.9 strictfp (Java - methods only)

- **Description:** This modifier indicates that all of the floating point operations within the method must hold to strict semantics.
- **Status:** Parsed and ignored by ESC/Java2. No static checking of floating point semantics is performed. Note that the Java Language Specification stipulates that constructors may be `strictfp` only by virtue of the entire class being declared `strictfp`.
- **Differences from JML or Java:** None.

3.11 Annotation assertions for a field declaration

A field declaration (including ghost and model field declarations) may be **followed** by field assertions, which are introduced by the `in` and `maps` keywords. These declarations associate a field or its sub-fields with specific datagroups, which can then be used as items in a modifies clause.

3.11.1 in (JML)

- **Description:** This assertion follows a field declaration and lists the datagroups of which the field is a part. A field is automatically a part of the datagroup with the same name as itself.
- **Status:** Implemented.
- **Differences from JML or Java:** None.

3.11.2 maps, \into (JML)

- **Description:**
- **Status:** Parsed, typechecked and static checked using a limited unrolling of any recursive definitions.

3.12 Annotation modifiers for a field declaration

A field declaration (including ghost and model field declarations) may have both Java and JML modifiers. In addition to the access modifiers (see Section 3.4 [Access modifiers], page 13), the following modifiers are relevant to field declarations.

3.12.1 non_null (JML)

- **Description:** This modifier on a field declaration indicates that the field in question never has a null value, after the object is constructed. It is equivalent to a class invariant of the form

  ```
  invariant field != null;
  ```

  with the same access modifiers as the field itself.

  Note that the field is not null after *construction*. The field may indeed be null after class initialization and during the execution of constructors or helper methods called by constructors.
- **Status:** Fully implemented and supported by the static checker.
- **Differences from JML or Java:** None.

3.12.2 nullable (JML)

- **Description:** This modifier on a field declaration indicates that the field in question may have a null value. It is equivalent to a class invariant of the

  ```
  invariant field != null || field == null;
  ```

  with the same access modifiers as the field itself which, since this specification simplifies to `true` because all references in Java either have `null` value or they do not, is not added at all.
- **Status:** The modifier is parsed. Typechecking implementation is underway. This modifier is not yet used by the static checker.
- **Differences from JML or Java:** None.

3.12.3 monitored (JML)

- **Description:** This modifier indicates that the field modified is guarded by the monitor associated with `this` if the field is not static and by the monitor associated with the class object if the field is static. On any write to the field, all monitors guarding the field must be held by the thread executing the write; on any read of the field, at least one of the monitors guarding the field must be held by the thread executing the read.

  Modifying a field `x` in class `C` with `monitored` is equivalent to the declaration

  ```
  monitored_by x = C.class;
  ```

  if `x` is static, or to

  ```
  monitored_by x = this;
  ```

if `x` is not static.

- **Status:** Implemented. The implementation improves on that in ESC/Java by implementing monitors for static fields and by making the error messages more informative and accurate.
- **Differences from JML or Java:** None.

3.12.4 final (Java)

- **Description:** This Java modifier indicates that the field it modifies may not be assigned to, after it has been initialized (either by an initializer or in a constructor). Java has some complex rules about definite-assignment that are relevant but not discussed here.
- **Status:** It is supported by the ESC/Java2 parser and type checker. It does not have any effect on the static checking.
- **Differences from JML or Java:** None.

3.12.5 volatile (Java)

- **Description:** This Java modifier affects the optimizations that might be applied and the semantics of the order of writing the values of object fields to memory. It affects multi-threaded programs only. It is not applicable to a ghost or model declaration.
- **Status:** Parsed but ignored by the static checker.
- **Differences from JML or Java:** None.

3.12.6 transient (Java)

- **Description:** Fields designated `transient` are not saved as part of an object's persistent state. This is not applicable to a ghost or model declaration.
- **Status:** Parsed but ignored by the static checker. Any implications of this modifier would be part of the specifications of serialization for the object.
- **Differences from JML or Java:** None.

3.12.7 static (Java)

- **Description:** `static` is a Java modifier (which may be used on JML ghost and model field declarations as well) that indicates that the declaration in question is a member of the class and not of each instance of the class.
- **Status:** Fully implemented.
- **Differences from JML or Java:** None.

3.12.8 instance (JML)

- **Description:** `instance` is a JML modifier that indicates the opposite of `static`, that is, that the field in question is a member of each instance of the object, not just of the class. Within a class declaration, field declarations are non-static by default (and `instance` is not needed). However, within an interface, Java field declarations are `static` and `final` by default. Ghost and model field declarations are `static` by default (but not `final`). The `instance` modifier may be applied to ghost or model field declarations in an interface, making those fields non-static. (JML allows the modifier `instance` only on ghost and model field declarations, in either classes or interfaces.) As in Java, ghost and model declarations in a public interface are implicitly public.

- **Status:** Implemented.
- **Differences from JML or Java:**

3.13 Annotation modifiers for formal parameters

These modifiers may precede the type name of a formal parameter within the declaration of the method or constructor.

3.13.1 non_null (JML)

- **Description:** Modifying a formal parameter with a `non_null` modifier is equivalent to adding a precondition stating that the parameter is not equal to `null` and requiring that any assignment to that variable assign a non-null value. It is only legal for reference types, not for primitive types. The `non_null` condition is in addition to the composite specification formed from the complete refinement sequence. Hence it is equivalent to adding a precondition requiring the parameter be `non_null` to the desugared specification of the routine (as well as the conditions on subsequent assignments within the body of the routine).

  Note that parameters of superclass and subclass declarations of a method each independently may have or not have `non_null` declarations. A method's implementation must satisfy the superclass specification and independently satisfy the subclass specification. A more thorough discussion is given in Section 3.19.13.3 [Inheritance and non_null], page 54.
- **Status:** The modifier is parsed, checked and supported by the static checker.
- **Differences from JML or Java:** None known.
- **Comment:** The original ESC/Java did not permit a subclass to modify a formal parameter as `non_null` in an overriding declaration. The declarations of the top-most declaration were used for all overriding declarations.

3.13.2 nullable (JML)

- **Description:** Modifying a formal parameter with a `nullable` modifier does not modify a method specification in any way. It is only legal for reference types, not for primitive types.

  Note that parameters of superclass and subclass declarations of a method each independently may have or not have `nullable` declarations. A method's implementation must satisfy the superclass specification and independently satisfy the subclass specification. A more thorough discussion is given in Section 3.19.13.3 [Inheritance and non_null], page 54.
- **Status:** The modifier is parsed. Typechecking implementation is underway. This modifier is not yet used by the static checker.
- **Differences from JML or Java:** None known.
- **Comment:** The original ESC/Java did not support this annotation at all, as it did not exist in JML until late 2005.

3.13.3 final (Java)

- **Description:** This Java modifier indicates that a formal parameter may not be the target of an assignment in the body of the routine.
- **Status:** Fully implemented. This feature does not affect the static checking; it simply produces a typechecking error if a final parameter is the target of an assignment.
- **Differences from JML or Java:** None.

3.14 Annotation statements within the body of a method or constructor

These annotation statements may be intermixed with the statements within the body of a method or constructor. They may also be used within model programs.

3.14.1 assume, assume_redundantly

- **Description:** This annotation statement (inherited from the original ESC/Java) states a predicate which is then assumed by the static checker. It is typically used to state a predicate at some point in the body of a routine that the static checker is unable to establish. Checking of the remainder of the body can then proceed with this assumption.
- **Status:** Implemented.
- **Differences from JML or Java:** None.

3.14.2 assert, assert_redundantly

- **Description:** This statement will generate a static checker warning if a program execution can be found in which the associated predicate is false. Note that this is different than the Java assert statement, both in syntax and in semantics. The command-line options `-jmlAssertions` will cause Java assertions to behave like JML assertions (see Section 3.19.4 [Java and JML assert statements], page 46).
- **Status:** Fully implemented.
- **Differences from JML or Java:** None.

3.14.3 set

- **Description:** The JML set statement is used as a statement within the body of a routine to assign a value to a ghost field or variable. ESC/Java allowed only fields to be set. ESC/Java2 allows in addition the declaration of local ghost variables and the assignment of values to them using set statements. Only assignments (with =) are allowed and the right-hand-sides of the assignment statements must be pure expressions.
- **Status:** Implemented.
- **Differences from JML or Java:** None.

3.14.4 unreachable

- **Description:** This is a statement that asserts that the command-flow of the program will never reach this point. If the static checker suspects that there is a program execution that can do so, it will issue a warning. It is equivalent to an annotation stating `assert` `false`. There is no conditional unreachable statement but the equivalent can be created using an assert statement.

- **Status:** Implemented.
- **Differences from JML or Java:** There is no unreachable annotation in JML.

3.14.5 hence_by, hence_by_redundantly

- **Description:** This keyword is followed by a predicate.
- **Status:** Parsed and type-checked. Ignored by the static checker.
- **Differences from JML or Java:** None.

3.14.6 loop_invariant, loop_invariant_redundantly, maintaining, maintaining_redundantly

- **Description:** A `loop_invariant` pragma is followed by a predicate. The pragma must appear immediately prior to a loop statement (for, while, do, or Java labeled statement). The predicate must hold at the start of each loop iteration. Details are given in the ESC/Java User's Manual. Note that loops are checked only by unrolling them a few iterations. The number of iterations can be set by the `-loop` command-line option.
- **Status:** Implemented.
- **Differences from JML or Java:** None.

3.14.7 decreases, decreasing, decreases_redundantly, decreasing_redundantly

- **Description:** This pragma specifies a quantity (type `int`) which must decrease but must always be non-negative at the start of any loop iteration. Note that loops are checked only by unrolling them a few iterations. The number of iterations can be set by the `-loop` command-line option.
- **Status:** Implemented.
- **Differences from JML or Java:** None.

3.14.8 ghost declarations

- **Description:** JML and ESC/Java2 (but not ESC/Java) allow the declaration of local ghost variables within the body of a routine, just as Java allows local declarations. These can then be used in subsequent annotation statements within the body, such as assert, assume, or set statements. Such declarations may also be declared `final`, `non_null`, or `nullable` and may have initializers.
- **Status:** Implemented.
- **Differences from JML or Java:** None.

3.15 Modifiers that may be applied to local declarations

Declarations within the body of a method, constructor or initialization code introduce local variables used only during the execution of that body of code. Privacy modifiers, `static`, and `instance` are not applicable to these declarations. JML defines local ghost declarations as well (but not local model declarations). The following modifiers are allowed.

3.15.1 non_null (JML)

- **Description:** A local declaration of a variable of reference type within the body of a method or constructor (including local ghost declarations) may be modified with the JML annotation **non_null**. This requires that the initial value and any subsequently assigned value for that variable must not be null.
- **Status:** Fully implemented (in ESC/Java and ESC/Java2) except that detection of failure to initialize or initialize with null value, or ghost declarations.
- **Differences from JML or Java:** None.

3.15.2 nullable (JML)

- **Description:** A local declaration of a variable of reference type within the body of a method or constructor (including local ghost declarations) may be modified with the JML annotation `nullable`. This places no requirements on the initial value and any subsequently assigned value for that variable.
- **Status:** The modifier is parsed. Typechecking implementation is underway. This modifier is not yet used by the static checker.
- **Differences from JML or Java:** None.

3.15.3 uninitialized (JML)

- **Description:** This JML modifier may be applied to a local variable declaration within the body of a block of code. It indicates that although the variable has been initialized with an initial value, it should be considered as uninitialized. That is, a warning will be issued if its value is used before having been assigned a new value.
- **Status:** Implemented (as in ESC/Java).
- **Differences from JML or Java:** This modifier is not part of JML.

3.15.4 final (Java)

- **Description:** This Java modifier has the usual meaning that the variable may not be altered (or even reassigned its same value) after it has been declared and initialized.
- **Status:** Implemented.
- **Differences from JML or Java:** ESC/Java2 does not detect invalid assignments to final variables that are not initialized in the declaration.

3.16 JML functions (extensions to expressions)

JML defines a number of new operators, functions, and other constructions for use within expressions that are part of annotations.

3.16.1 New operators in JML

JML adds to Java the operators described below. The `==>` and `<==` operators have the same precedence and they bind less tightly than the Java logical or expression. That is the expression `( p || q ==> r || s)` is equivalent to `( (p || q) ==> (r || s) )`. The `<==>` and `<=!=>` operators have the same precedence and bind less tightly than `==>` and `<==`. That is, the expression `( p ==> q <==> r <== s)` is equivalent to `((p==>q) <==> (r<==s))`.

The equivalence and inequivalence operators bind more tightly than the conditional (`?:`) operator.

The subtype operator (`<:`) has lower precedence than the shift operator `<<`, the same precedence as the comparison operators `<`, `>`, `<=`, and `>=`, and has higher precendence than the equality (`==`) and inequality (`!=`) operators.

- `<==>` : This operator (equivalence) takes two boolean arguments; it returns a boolean value of `true` if the two arguments are both `true` or both `false`, and `false` otherwise.

  Status: Implemented.
- `<=!=>` : This operator (inequivalence) takes two boolean arguments; it returns a boolean value of `false` if the two arguments are both `true` or both `false`, and `true` otherwise. `(A <==> B)` is equivalent to `!(A <=!=> B)`.

  Status: Implemented.
- `==>` : This operator (implies) takes two boolean arguments and returns a boolean value of `true` if the first is `false` or the second is `true`, and returns `false` if the first is `true` and the second is `false`.

  Status: Implemented.
- `<==` : This operator (reverse implication, or explies) takes two boolean arguments and returns a boolean value of `true` if the second is `false` or the first is `true`, and returns `false` if the second is `true` and the first is `false`. `(A ==> B)` is equivalent to `(B <== A)`.

  Status: Implemented.
- `<:` : This operator takes two arguments of type `\TYPE` (or, equivalently, of type `java.lang.Class`); it returns `true` if the left-hand argument is the same type as or a subtype of the right-hand argument.

  Status: Implemented.
- `<` , `<=` : In addition to their usual meaning in Java (and corresponding meanings in JML), the less-than and less-than-or-equal operators are used to compare locks. Any object that is listed as a monitor (in a `monitors_for` clause) or is identified implicitly or explicitly in a Java synchronization statement is a lock, in addition to its declared use as an object. In order to reason about deadlocks, the user must define a partial order on locks and have the code acquire the locks only in increasing order. The partial order may be defined using `axiom` clauses; the operators are used to compare locks by this partial order. They take two locks as arguments and return `true` if the left-hand object is less than (or less than or equal to, respectively) the right-hand object according to the partial order axioms, and returns `false` otherwise. The comparison is independent of whether the locks have actually been acquired; the `\lockset` expression supplies that information. The ESC/Java User's Manual provides additional information and examples about Deadlock and Race detection.

  Status: Implemented.

3.16.2 New JML expressions (functions and values)

- `\result` : This keyword denotes the result of a method that returns a value. It may be used in an `ensures` clause (but not a `signals` clause).

  Status: Implemented.
- `\old` : This pseudo-function takes one argument of any type and returns the value of its argument. It is used to indicate that the expression which is its argument must be evaluated in the pre-state. It is used in ensures and signals clauses to distinguish pre- and post-state values. It may also be used within annotations in the body of a routine (e.g. assert, assume, set statements).

  Status: Implemented. JML does not allow \old in set statements or local ghost variable initialization.
- `\not_modified` : This function of an arbitrary number of arguments returns a boolean value indicating whether all of the arguments are unchanged between pre-state and the current state. It may be used in an ensures or signals clause and in annotations in the body of a routine. It is equivalent to boolean AND of `x == \old(x)` for each argument.

  Status: Implemented (not part of the original ESC/Java).

  Differences: ESC/Java2 allows arbitrary (pure) expressions as the arguments of `\not_modified`; JML only allows store locations.
- `\fresh` : This function takes a single argument of any reference type and returns a `boolean`. The operator may be used in an expression that is evaluated in the post-state. The result is true if its argument is non-null and was not allocated in the pre-state.

  Status: Implemented.
- `\reach` : - deprecated
- `\duration` : This function returns a `long` value giving a maximum number of virtual machine cycles needed to execute the method or constructor call which is the argument. The argument is not executed (and so need not be a pure expression).

  Status: Parsed. No type checking is performed on the argument nor is any static checking performed.
- `\space` : This function returns a `long` value giving the number of bytes of heap space allocated to the object referred to by its argument. The argument must have reference type.

  Status: Parsed. No type checking is performed on the argument nor is any static checking performed.
- `\working_space` : This function returns a `long` value giving the number of bytes of heap space needed to execute the method or constructor call that is the argument. The argument is not executed (and so need not be a pure expression).

Status: Parsed. No type checking is performed on the argument nor is any static checking performed. [

- `\nonnullelements` : This function returns a boolean and takes an argument of a reference array type. The result is true if the argument is not null and no element of the array is null.

  Status: Implemented.
- `\typeof` : This function returns a value of type `\TYPE` and takes one argument of any type. The result is the dynamic type of the argument. The result of \typeof applied to a null expression is not equal to nor a subtype of the type of any reference or primitive type.

  Status: Implemented.

  Differences: JML allows arguments of primitive types; ESC/Java2 does also, but the original ESC/Java did not.
- `\elemtype` : This function takes an argument of type `\TYPE` and returns a value of type `\TYPE`. If the argument is an array type then the result is the (dynamic) most-specific type of the elements of the array; if the argument is not an array type, the result is unspecified.

  Status: Implemented.
- `\type` : This is a syntactic construct used to denote type literals. It denotes the type constant (of value `\TYPE`) corresponding to the type name given as the argument.

  Status: Implemented.

  Differences: None.
- `\is_initialized` : This construct takes a type name as its argument. It returns true just in the case that the type named has completed its static initialization.

  Status: Parsed and partly typechecked. No static checking is implemented.
- `\invariant_for` : This function takes one object of reference-type as its argument and returns a boolean. The result is true just when the object satisfies its class invariants.

  Status: Parsed and partly typechecked. No static checking is implemented.
- `\lblneg`, `\lblpos` : These syntactic constructs have the unusual form

  ```
  (\lblpos Identifier Expression)
  ```

  The enclosing parentheses are required; the middle item is an unquoted identifier that has no relationship to any other identifiers elsewhere in the program. The result of the expression is the value of the expression that is within the expression;

this expression must be a boolean expression. The effect of the operation is as follows. If the static checker would issue a warning that some condition does not hold, and the relevant expression contains a lblneg or a lblpos construct, and the expression within the construct evaluates to false (for lblneg) or true (for lblpos), then the identifier in the construct is listed as a label encountered in the counterexample context. This may be useful to the user in identifying which portion of an expression is causing some condition to hold or not to hold.

- `\lockset` : This value has type `\LockSet`. The value is the set of objects whose locks are held by the current thread.

  Status: This feature is fully implemented by ESC/Java2 (as inherited from ESC/Java).
- membership in a \lockset : A `\lockset` contains objects. Membership in a `\lockset` is tested using this syntax, for some object expression `o` : `\lockset[o]` . This expression returns a boolean, which is true if the object is in the lockset.

  Status: This feature is fully implemented by ESC/Java2 (as inherited from ESC/Java). It is not part of JML, but should be.
- `\max` : This function takes an argument of type `\LockSet`. It returns an object of type `java.lang.Object`. The result is one of the elements of the argument; the function satisfies the following for any `\LockSet s` and Object `o`: `s[o] ==> (o <= \max(s))`.

  Status: This feature is fully implemented by ESC/Java2 (as inherited from ESC/Java).
- Operators for overflow checking : These are under development in JML and are not yet implemented in ESC/Java2.
- *informal predicate* : An informal predicate consists of text enclosed within the delimiters `(*` and `*)`. Informal predicates are interpreted as boolean expressions that are always true in a positive context and always false in a negative context, and they may only be used in contexts where a boolean expression is allowed.

  Status: This feature is partly implemented in ESC/Java2 (as well as ESC/Java). Informal predicates are currently interpreted as always true, regardless of context, which may lead to unintended results.

3.16.3 New JML types

JML adds some new types that may be used as type names in declarations of variables within annotations.

- `\TYPE` : This is a JML type name used to denote the type of type designations. For example, `\typeof` and `\type` produce results of type `\TYPE`, and the `<:` operator takes arguments of type `\TYPE`. Values of type `\TYPE` can also be compared using the `==` or `!=` operators.

  Status: This is partly implemented. In JML \TYPE and java.lang.Class are now equivalent. This is not the case in ESC/Java2.

- `\bigint` : This is a new type name used in JML to denote an integral type equivalent to the mathematical integers. That is, it has infinite range and no underflow or overflow as a result of a fixed bit depth.

  Status: The type name is parsed but is equivalent to `long`.
- `\real` : This is a new type name used in JML to denote a type equivalent to the mathematical real numbers. That is, it has infinite range and precision and no underflow, overflow, or rounding error as do `float` and `double`.

  Status: The type name is parsed but is equivalent to `double`.
- `\LockSet` : This type may not be named (there is no token `\LockSet`). However the type is implicitly used as the type of the JML token `\lockset`, as the type of the argument of `\max`, and in the LockSet membership operation.

3.16.4 quantified expressions - \forall, \exists, \num_of, \max, \min, \sum, \product

As described in the JML documentation, quantified expressions take the form

(*quantifier-keyword type idlist* ; *range-expr* ; *expr*)

or

(*quantifier-keyword type idlist* ; ; *expr*)

or

(*quantifier-keyword type idlist* ; *expr*)

The *range-expr* is a boolean expression; its default value is `true`. The *idlist* is a comma-separated list of identifiers; these are the bound variables of the quantification. No side-effects are permitted in the predicate or expression.

Status:

- `\forall`, `\exists`: Fully implemented and used in static checking.
- `\num_of`, `\max`, `\min`, `\sum`, `\product`: Parsed and typechecked but not used in static checking.

Comment: The keyword `\max` is used both as a quantifier and as a function. The parser is able to distinguish the two usages.

3.16.5 Set comprehension

JML has a syntax for expressions whose value is a set. An example is

```
new JMLObjectSet { Integer i | o.has(i) && i.intValue() > 0 }
```

No side-effects are permitted in the predicate.

Status: ESC/Java2 parses and typechecks set comprehension expressions, with the following omissions:

- Set comprehension in class-level specifications is not handled.
- Expressions containing set comprehension are not used in static checking
- No restrictions on the type of the set are imposed
- The JML restrictions on the form of the predicate are not checked
- ESC/Java2 should check and forbid Java or JML modifiers in the bound variable declaration

3.16.6 \not_specified

This may be used, within the guidelines of the JML grammar, as the predicate or store-ref expression in an annotation clause. JML allows tools to provide their own behavior for `\not_specified`. Escjava2 treats such clauses as if they were not present, making them equivalent to the default for that clause type.

Status: Fully implemented.

3.16.7 \private_data

Status: Not implemented.

3.16.8 \other

Status: Not implemented.

3.16.9 Other Java operators and expression syntax

Java operators and expression syntax are all completely parsed and are typechecked with varying degrees of rigor. Most features are interpreted and can be reasoned about by the static checker. Some that are only partially handled by the static checker are these:

- new instance expressions - These are completely handled in program code. In specification expressions, these are interpreted according to their specifications by a procedure that inlines the specifications, but only to a limited depth of rewriting.
- new array expressions - These are handled adequately in both program code and in annotations, except for reasoning about the values of array elements when initializers are provided.
- anonymous classes

3.17 store-ref expressions

Some annotations require a list of *store-ref expressions*, which are expressions whose value is a set of references to fields of classes or objects; in particular, the value has type `org.jmlspecs.models.JMLObjectSet`. For example, the `modifies` clause designates a set (that is, a `JMLObjectSet` of *store-refs* that are allowed to be assigned to within a method. This section describes the syntactic features that designate such sets.

3.17.1 [*ident* | `super` | `this`] . [*ident* | `this`]

This designates a field of an object.

Status: Implemented within modifies clauses.

3.17.2 [*ident* | `super` | `this`] `.*`

This designates all static and instance fields, of any privacy level, including those inherited from superclasses and interfaces, of the given object.

Status: Not implemented.

3.17.3 [*ident* | `super` | `this`] `[`*expr*`]`

This designates an element of the given array object.

Status: Implemented within modifies clauses.

3.17.4 [*ident* | `super` | `this`] `[`*expr*`..`*expr2*`]`

This designates a range of elements of the given array object.

Status: Not implemented.

3.17.5 [*ident* | `super` | `this`] `[*]`

This designates all the elements of the given array object.

Status: Not implemented.

3.17.6 *classname.**

This designates all static fields, of any privacy level, including those in superclasses and interfaces, and including ghost and model fields, of the given class.

Status: Not implemented.

3.17.7 \nothing

This designates an empty set of store-refs. It may be used with the `modifies`, `accessible`, and `callable` clauses. If `\nothing` appears in a sequence of other store-refs, it is ignored, since a sequence of store-refs is essentially a union.

Status: Fully implemented.

3.17.8 \everything

This designates a universal set - the set of references to all object and class fields for every object and class allocated in the current state of the program. It may be used with the `modifies`, `accessible`, and `callable` clauses.

Status: Fully parsed - refer to the clause descriptions for specific behavior.

3.18 Statements within model programs

A model program consists of imperative programming statements and control structures, much like a typical Java program. However, a model program allows some non-deterministic (and non-Java) statements as well. These are described in this section.

3.19 Other issues

3.19.1 universe type system

JML has recently introduced a universe type system for alias control. This includes type modifiers \readonly, \rep, \peer. These have not yet been implemented in Esc/Java2, either in the parser or in the semantics.

3.19.2 infinite-precision arithmetic

JML defines primitive data types `\bigint` and `\real` which denote infinite precision integers and reals, for use in specifications. These type names are parsed, but the underlying semantics and the Simplify prover do not handle the finite-ness of numeric representations.

3.19.3 nowarn annotations

ESC/Java2 allows an annotation to suppress warnings otherwise produced by the static checker. These are discussed in Section 4.2 [Nowarn annotations and warnings], page 59. JML parses but ignores these annotations.

3.19.4 Java and JML assert statements

JML has an assert statement of the form

`//@ assert` *predicate*`;`

The static checker will evaluate the predicate in the appropriate context. If it cannot establish that the predicate is always true, an ESC/Java warning will be issued.

Java 1.4 also has an assert statement with the syntax

`assert` *predicate*`:` *value*`;`

If the predicate is not true, then a `java.lang.AssertionError` is created, with the given *value* as its argument, and the exception is thrown.

In versions of Java before 1.4, `assert` was a legitimate identifier name; in Java 1.4 it is a keyword and may not be used as an identifier. To achieve backwards compatibility, Java compilers have a `-source` command line option; ESC/Java2 behaves in a way similar to typical Java compilers:

- specifying `-source 1.4` causes ESC/Java2 to interpret `assert` in Java statements as a keyword;
- omitting the `-source` option or specifying an argument other than `1.4` causes ESC/Java2 to treat `assert` in Java statements as an identifier and to issue errors on encountering uses of the word as a keyword (i.e. to interpret assert in Java 1.3 mode).

There are three options for how ESC/Java2 should treat Java `assert` statements (when `-source 1.4` is specified). ESC/Java2 provides command-line options to achieve each of these behaviors.

A. Parse but ignore them.
B. Treat them as Java does, namely, as equivalent to

    `if ( !` *predicate*`) throw new java.lang.AssertionError(`*value*`);`

C. Treat them as a JML assert statement, namely, as equivalent to

    `//@ assert` *predicate*`;`

The command line options for these three cases are

A. `-source 1.4`
B. `-javaAssertions` or `-eajava`
C. `-jmlAssertions` or `-eajml`

Note that there are, internally, three independent conditions:

- The version of Java being recognized. This is controlled by the `-source` option.
- Whether Java assert statements are enabled (given that Java 1.4 source is being processed). Java has them disabled by default. This is controlled by the options `-disableassertions`, `-enableassertions`, `-da`, `-ea`. The use, as in Java, of package names with these assertions or of enabling or disabling system assertions is not implemented in ESC/Java2.
- Whether to treat Java 1.4 assert statements in Java mode or JML mode (given that Java assertions are enabled). This is controlled by the `-javaAssertions` and `-jmlAssertions` options. `-javaAssertions` is the default; both options also effectively do `-source 1.4 -ea`.

Here are some examples of use:

-
  ```
  public void m() {
          assert false : "Message";
  }
  ```
  Java behavior: ESC/Java2 will issue an Unexpected Exception warning
  JML behavior: ESC/Java2 will issue an Assert warning
-
  ```
  public void m() {
          assert false : "Message"; //@ nowarn;
  }
  ```
  Java behavior: ESC/Java2 will issue an Unexpected Exception warning
  JML behavior: ESC/Java2 will not issue any warning (suppressed by the nowarn annotation)
-

```
public void m() {
        assert true : "Message";
}
```

Java behavior: No warnings - no exception ever thrown
JML behavior: No warnings - predicate is always true

-
  ```
  //@ exsures (java.lang.AssertionError e) true;
  public void m() {
          assert false : "Message";
  }
  ```

  Java behavior: ESC/Java2 will issue an Unexpected Exception warning
  JML behavior: ESC/Java2 will issue an Assert warning

-
  ```
  //@ exsures (java.lang.AssertionError e) true;
  public void m() throws AssertionError {
          assert false : "Message";
  }
  ```

  Java behavior: No warnings - the exsures clause allows the exception.
  JML behavior: Assert error.

-
  ```
  //@ exsures (java.lang.AssertionError e) false;
  public void m() throws AssertionError {
          assert false : "Message";
  }
  ```

  Java behavior: Postcondition warning - the exsures clause disallows the exception.
  JML behavior: Assert error.

3.19.5 Methods and constructors without bodies in Java files

Java requires non-abstract, non-native methods and constructors to have a defined body. Some tools, such as javadoc, allow the bodies to be stubbed with a semicolon in place of the usual block statement. ESC/Java2 also allows method and constructor bodies within .java files to be represented simply by a semicolon. This allows classes to be documented and specified before the implementation is completed. JML tools (except jmldoc) and ESC/Java require bodies to be present.

3.19.6 Methods and constructors in annotation expressions

JML allows the use of pure methods and constructors in annotation expressions; the original ESC/Java did not. ESC/Java2 follows JML.

3.19.7 Original ESC/Java also_ specifications

The original ESC/Java utilized the keywords `also_requires`, `also_ensures`, `also_exsures`, and `also_modifies` to add annotations to subclasses. These annotations were simply textually conjoined with the annotations from a superclass or interface. The syntax and semantics in JML are different and ESC/Java2 has changed to match JML. These keywords are now deprecated and are translated into the corresponding keyword without `also_`. However, the semantics of annotations in the presence of inheritance is now somewhat different and is described by the desugaring process (see [Desugaring], page 23). ESC/Java2 issues warnings if the original keywords are used and users are encouraged to change them.

3.19.8 anonymous classes

Java allows anonymous classes (named classes with some method overrides). An anonymous class is a subclass of the named class and is nested within the class in which it is defined (but is not a member of the enclosing class). One can add specifications to the anonymous class body and the members within it. However, since there are no quantities declared with the static type of the anonymous type, those specifications are not used in checking any uses of the routines of the anonymous class. ESC/Java2 does recognize that the anonymous class is a subclass and not the same class as its superclass.

Status: There is no static checking of the implementation of any of the methods of an anonymous class.

3.19.9 block-level class declarations

Java allows classes to be declared as local classes within a block statement. These constructs are parsed and type-checked. However,

- Although such a class (and its routines) may have annotations, the body of the implementation of these routines are not checked against the specifications.
- Any specifications are used when the methods of the class are called.

3.19.10 `field`, `method` and `constructor` keywords

JML is designed to work with any Java code. In particular that code may use JML keywords as variables and type names. This can create a problem in parsing annotations. For example, if `helper` has been declared a class, then an annotation that begins with

```
//@ helper model ...
```

could be interpreted as the beginning of a declaration with type `helper` and identifier `model` or as the beginning of a model method declaration with JML modifier helper. Though in some cases sufficient lookahead could distinguish these different uses, JML has defined the keywords `field`, `method`, and `constructor` to help disambiguate such situations. These are used in ghost field and model field, method, and constructor declarations. They have the effect that any identifiers appearing after these keywords in a JML annotation are not interpreted as JML keywords (except for `non_null` and `nullable` in formal parameters as described below). For example, in

```
//@ public model non_null helper method requires ensures() {};
```

`model`, `non_null`, `nullable`, `helper`, and `method` are JML keywords, but `requires` is expected to be a Java type name and `ensures` is interpreted as an identifier, the name of the method being declared.

The two exceptions to this rule are that `non_null` and `nullable` are needed as a JML modifier within the formal parameter declarations. A parser is capable of distinguishing these cases, however. Thus in

```
    //@ public model requires ensures(non_null Object o, non_null oo);
```

the first occurrence of `non_null` is a JML modifier, but the second is expected to be a Java type name. ESC/Java treats non_null specially in this case, but the JML tools do not. For JML you cannot use `non_null` here and need to write explicit pre/post-conditions instead.
`Status:` Fully implemented.

3.19.11 Equivalence of `\TYPE` and `java.lang.Class`

In Java, the types of classes can be treated as first class values of type `java.lang.Class`. In JML the type `\TYPE` has been defined as the type of type values. The following table shows the relationships between Java and JML syntax and quantities. In the following `T` is a type name, such as `int` or `java.lang.String`, `t` is an expression of `\TYPE` type, and `e` is an expression of any type.

	JML	Java
Type names denoting the type of type values	\TYPE	java.lang.Class
Expression that is a type literal corresponding to a given type name	`\type(T)`	`T.class`
Expression whose value is the type value corresponding to the name in a String		`Class.forName(s)`
Expression whose value is the type value corresponding to the dynamic type of an expression	`\typeof(e)`	`e.getClass()`
subtype relationship between two type values	`t1 <: t2`	`t2.isAssignableFrom(t1)`
Expression has type or is subtype of type name	`\typeof(t) <: \type(T)`	`t instanceof T`
A type value corresponding to the element type of an array type	`\elemtype(t)`	`t.getComponentType()`

In JML and ESC/Java2, `\TYPE` and `java.lang.Class` have been made equivalent; thus the corresponding quantities in each row may be used interchangeably (in annotations). The one exception in ESC/Java2 is that methods of `Class` (or `Object`) may not be invoked on a value of type `\TYPE`.

The location where the equivalence is particularly needed is in writing the specifications for a method such as `java.lang.Object.getClass()`, which is inherited by every reference type in Java. The natural specification (in java.lang.Object) is

```
    //@ ensures \result == \typeof(this);
    java.lang.Class getClass();
```

which can only be written if the value on the left-hand side of the ensures predicate, which has type `java.lang.Class`, can be compared to the value on the right-hand side, which has type `\TYPE`.

3.19.12 exceptions in annotation expressions

3.19.13 Specifications and inheritance

3.19.13.1 Desugaring in the presence of inheritance

ESC/Java has some limitations and some unsoundness in handling the inheritance of specifications. If a superclass stated a precondition (with a requires clause), then a subclass could not state an additional precondition. If a superinterface stated a precondition, an implementing class was permitted to state a precondition using the `also_requires` keyword, but the discussion in the ESC/Java User's Manual acknowledged this to be an unsound construct.

ESC/Java2 has corrected this problem by using the syntax and desugaring approach outlined by JML. JML allows subclasses to have additional `requires` clauses, and does not utilize the `also_requires` syntax. Consider the following example (the handling of superinterfaces is the same as the handling of super classes):

```
public interface Super {
        //@    requires P1;
        //@    ensures Q1;
        public void m();
}

public class D extends Super {
        //@ also
        //@    requires P3;
        //@    requires PP3;
        //@    ensures Q3;
        public void m();
}
```

ESC/Java would combine the interface and subclass specifications as follows:

```
        //@ requires P1;
        //@ requires P3;
        //@ requires PP3;
        //@ ensures Q1;
        //@ ensures Q3;
```

This does not satisfy behavior inheritance. The desugaring provided by ESC/Java2 is correct:

```
        //@ requires P1 || (P3 && PP3);
        //@ ensures P1 ==> Q1;
        //@ ensures (P3 && PP3) ==> Q3;
```

Here is a larger example:

```
public class Super {
        //@    requires P1;
        //@    ensures Q1;
        //@ also
        //@    requires P2;
```

```
                //@    ensures Q2;
                public void m();
        }

        public class D extends Super {
                //@ also
                //@    requires P3;
                //@    requires PP3;
                //@    ensures Q3;
                //@ also
                //@    requires P4;
                //@    ensures Q4;
                public void m();
        }
```

The specifications in `Super` desugar (in ESC/Java2) to

```
                //@ requires P1 || P2;
                //@ ensures P1 ==> Q1;
                //@ ensures P2 ==> Q2;
```

The specifications in `D` desugar to

```
                //@ requires (P3 && PP3) || P4;
                //@ ensures  (P3 && PP3) ==> Q3;
                //@ ensures  P4 ==> Q4;
```

but they are then combined with the superclass specifications to produce the composite specification:

```
                //@ requires P1 || P2 || (P3 && PP3) || P4;
                //@ ensures  P1 ==> Q1;
                //@ ensures  P2 ==> Q2;
                //@ ensures  (P3 && PP3) ==> Q3;
                //@ ensures  P4 ==> Q4;
```

With this approach the unsoundness of `requires` noted in section C.0.4 of the ESC/Java User's Manual is corrected. However, the behavior of some specifications will change, since specifications of an overriding method are no longer simply textually conjoined with the specifications of an overridden method. For example in ESC/Java, the specification of `m()` in this code

```
        public class Super {
                //@ requires i > 0;
                public int m(int i) {...}
        }
        public class D {
                //@ ensures \result > 0;
                public int m(int i) {...}
        }
```

was interpreted as

```
                //@ requires i > 0;
                //@ ensures \result > 0;
```

whereas in ESC/Java2 it is

```
            //@ requires i > 0 || true;
            //@ ensures i > 0 ==> true;
            //@ ensures true ==> \result > 0;
```

which is equivalent to

```
            //@ requires true;
            //@ ensures \result > 0;
```

3.19.13.2 Defaults and inheritance

The defaults for missing clauses were described in Section 3.8.1 [specifications], page 21. The effect of the default for `requires` is demonstrated in these examples.

Given

```
     class Super {
             //@ requires P;
             //@ ensures Q;
             public void m();
     }

     class Derived extends Super {
             // no spec given - default is 'requires false'
             public void m();
     }
```

the two classes behave as if they had these specifications

```
     class Super {
             //@ requires P;
             //@ ensures P ==> Q;
             public void m();
     }

     class Derived extends Super {
             //@ also
             //@ requires P;
             //@ ensures P ==> Q;
             public void m();
     }
```

Given

```
     class Super {
             // no spec given - default is 'requires true'
             public void m();
     }

     class Derived extends Super {
             //@ also
             //@ requires P;
             //@ ensures Q;
```

```
                public void m();
        }
```

the two classes behave as if they had these specifications

```
        class Super {
                //@ requires true;
                public void m();
        }

        class Derived extends Super {
                //@ also
                //@ requires true;
                //@ ensures P ==> Q;
                public void m();
        }
```

Finally, given

```
        class Super {
                // no spec given - default is 'requires true'
                public void m();
        }

        class Derived extends Super {
                // no spec given - default is 'requires false'
                public void m();
        }
```

the two classes behave as if they had these specifications

```
        class Super {
                //@ requires true;
                public void m();
        }

        class Derived extends Super {
                //@ also
                //@ requires true;
                public void m();
        }
```

3.19.13.3 Inheritance and non_null

The syntactic rules and semantic meaning of `non_null` on formal parameters in the presence of inheritance differ between ESC/Java and ESC/Java2.

The annotation `nullable` did not exist at all in JML until late 2005, thus it was not supported at all in ESC/Java.

A `non_null` annotation on a formal parameter is equivalent to (a) a requirement that the parameter not be assigned a null value in the body of the implementation of the routine, and (b) an additional precondition in each of the specification cases of the method's (or constructor's) specifications. Item (b) means that

```
     ... specifications ...
     public void m(/*@ non_null */ Object o, Object oo)
```

is equivalent to

```
     //@ requires o != null;
     //@ {|
          ... specifications ...
     //@ |}
```

Similarly a `non_null` annotation on the result is equivalent to an additional post-condition:

```
     ... specifications ...
     public /*@ non_null*/ Object m()
```

is equivalent to

```
     //@ ensures o != null;
     //@ {|
          ... specifications ...
     //@ |}
```

[This syntax is not legal JML since an ensures may not appear outside of the nested specifications, but the intent is evident: the ensures clause is distributed to each specification case.]

Confusing situations can arise if formal parameters in a overridden and overiding declaration differ in `non_null` annotations.

- Case 1: superclass has `non_null`, subclass has a specification, but no `non_null`

```
         class Super {
             public void nonnull(/*@ non_null */ Object o);

             public void m(/*@ non_null */ Object o) {
                 nonnull(o);           // Line A - OK
                 o = null;             // Line B - FAILS
             }
         }
         class Derived extends Super {
             //@ also
             //@ requires true;
             public void m(Object o) {
                 nonnull(o)            // Line C - FAILS
                 o = null;             // Line D - OK
             }

             public void mm() {
                 Object o;
                 m(o);                 // Line E - OK
                 (new S()).m(o);       // Line F - FAILS
             }
         }
```

  In this case, ESC/Java has `Derived.m` inherit the `non_null` specification from the overridden method. Hence ESC/Java reports errors for lines B, D, E and F, with A and C being OK. In ESC/Java2, however, `Derived.m` has its own specification and does not inherit the `non_null` specification. It must still satisfy the parent spec (which in this case simply has a trivial postcondition). So Lines B and F are errors and A is not. However, the combined spec for `Derived.m` is (ignoring other clause types)

```
requires o != null;
ensures true;
also
requires true;
ensures true;
```

which is equivalent to `requires true;`. Thus lines D and E are OK, and C will provoke an error.

If the precondition of `Derived.m` were `requires false;`, then the combined spec would be

```
requires o != null;
ensures true;
also
requires false;
```

which is equivalent to `requires o != null;`. Then E would fail, and C and D would be OK. In otherwords, it would be the same as Case 2 below.

- Case 2: superclass has `non_null`, subclass has no specification and no `non_null`

```
class Super {
    public void nonnull(/*@ non_null */ Object o);

    public void m(/*@ non_null */ Object o) {
        nonnull(o);           // Line A - OK
        o = null;             // Line B - FAILS
    }
}
class Derived extends Super {

    public void m(Object o) {
        nonnull(o); // Line C - OK - precondition limits values of o
        o = null;   // Line D - OK - no restriction on
                   //                assignments in this body
    }

    public void mm() {
        Object o;
        m(o);                 // Line E - FAILS
        (new S()).m(o);       // Line F - FAILS
    }
}
```

In this case Derived.m has no specification. Consequently it has a default specification of `requires false;` and effectively inherits the overridden method's specification, including the `non_null`. Lines B and F are still errors, and A is not, because of the specification of Super.m; line E is an error, and C and D are not.

- Case 3: superclass has no `non_null`, but subclass does

```
    class Super {
        public void nonnull(/*@ non_null */ Object o);

        public void m(Object o) {
            nonnull(o);           // Line A - FAILS
            o = null;             // Line B - OK
        }
    }
    class Derived extends Super {

        public void m(/*@ non_null */ Object o) {
            nonnull(o);      // Line C - FAILS - annotation ignored
            o = null;        // Line D - OK - annotation ignored
        }

        public void mm() {
            Object o;
            m(o);                 // Line E - OK
            (new S()).m(o);       // Line F - OK
        }
    }
```

In this case, it is the derived class that has the `non_null` specification, but not the superclass. This is a problematic case, since the body of the overriding method may be called through the overridden signature, in which case the actual argument may not be constrained to be non_null. Hence the non_null annotation on the overriding formal paramater can be misleading. Therefore in ESC/Java2, a `non_null` annotation is ignored if there is any overridden method that has that formal parameter declared without a `non_null` argument. A caution message is provided to warn the user of this behavior.

Thus, ESC/Java2 will allow lines B and F since the superclass does not limit the values of the parameter; similarly line A provokes an error. Although the parameter of `Derived.m` is declared `non_null`, the `non_null` annotation is ignored. Consequently, there is no limitation on the value of the formal parameter (since it may have been called through the superclass's signature), and hence line C fails; similarly, there is no limitation on what may be assigned to the formal parameter variable, so line D is ok. Finally, line E is ok since there is no additional precondition created by the ignored `non_null`.

Note that behavior equivalent to the `non_null` declaration on a formal parameter can be obtained by adding appropriate explicit preconditions requiring the formal parameter to be `non_null` in the appropriate circumstances. Furthermore, if it is actually desired to have the formal parameter be `non_null` within the body, code such as the following can be written. Instead of

```
        public void m(/*@ non_null */ Object o) {
                ...
        }
```

write

```
        public void m(Object oo) {
                /*@ non_null */ Object o = oo;
                ...
        }
```

Finally note that

- the original ESC/Java does not allow overriding methods to have `non_null` parameters, whether or not the overridden method does;
- there are no particular issues with respect to `non_null` on the result type; overriding and overridden methods may have any combination of `non_null` annotations or lack thereof.

4 ESC/Java2 features

This chapter describes some of the user-interface features and usage of ESC/Java2. A description of how to run ESC/Java2 is provided in Chapter 2 [Running ESCJava2], page 4.

4.1 Error and warning messages

ESC/Java2 continues ESC/Java's use of four levels of error messages:

- **fatal** errors are problems (usually invalid syntax) that prevent ESC/Java from proceeding further in parsing and checking files;
- **errors** indicate illegally formed input files, though processing may continue to find other errors or even to attempt static checking of the files (errors or checks subsequent to the first problem may be erroneous as a result of earlier problems);
- **cautions** indicate situations that are not illegal, but may be misleading to the user - a common example is features that are parsed but not checked (also some illegal JML constructs are reported using cautions if ESC/Java2 can unambiguously correct them);
- **warnings** indicate situations in which the static checking phase could not determine that annotation specifications were satisfied, such as an inability to determine that an object reference is non-Null when it is dereferenced.

The reporting of cautions and warnings can be controlled by command-line options (`-noCautions`, `-nowarn`, `-warn`, `-nocheck`).

4.2 Nowarn annotations and warnings

4.2.1 nowarn annotations

- **Description:** A nowarn annotation has the form

  `//@ nowarn` *`comma-separated-list-of-warning-types`*`;`

  or simply

  `//@ nowarn ;`

  The annotation is associated with the line in the source file on which it is located, rather than with a grammatical construct. It is used to suppress warnings from the static checker associated with a construct on that line; the annotation will suppress the warning either if it is on the line where the offending action took place or if it is on the line of the associated declaration. If there is no list of warning types, then all warnings associated with this line are suppressed. The warnings of a given type can also be suppressed using command-line options. The nowarn annotations have no effect on errors or cautions, since these are produced by parsing or semantic errors in the source code itself.
- **Status:** Implemented.
- **Differences from JML or Java:** JML supports the parsing of these annotations, but since JML does no static checking, it ignores them. ESC/Java did not require a ter-

minating semicolon, but JML does. ESC/Java2 accepts nowarn annotations with or without a terminating semicolon; it issues a caution if the semicolon is missing.

4.2.2 nowarn warning types

The static checker used in ESC/Java2 (and in ESC/Java) produces warnings (as opposed to errors or cautions) when it detects source code that might violate the specifications. These warnings are categorized into types and can be suppressed or enabled by warning type name, using either the `nowarn` annotation or the command-line options `-nowarn`, which suppresses individual warning types, or `-warn`, which enables individual warning types, or `-nocheck`, which turns off all static checking. The following list of the warning types is excerpted (except where additions are explicitly noted) and quoted from the "ESC/Java User's Manual".

The additional warning pseudo-type name "All" may be used with the `-nowarn` command-line option to turn off all warnings; `-warn` may then be used to selectively turn on individual warning types. The "All" type may not be used with `-warn`.

- **ArrayStore** warns that the control may reach an assignment `A[I] = E` when the value of `E` is not assignment compatible with the actual element type of `A`.
- **Assert** warns that control may reach a pragma `assert E` when the value of `E` is false.
- **Cast** warns that control may reach a cast `(T)E` when the value of `E` cannot be cast to the type `E`.
- **Constraint** warns that a method does not establish the post-condition stated in a `constraint` clause [added in ESC/Java2].
- **Deadlock** warns that control may reach a `synchronized` statement that would acquire a lock in violation of the locking order, or that a `synchronized` method may start by acquiring a lock in violation of the locking order.
- **Exception** warns that a routine may terminate abruptly by throwing an exception that is not an instance of any type listed explicitly in the routine's throws clause.
- **IndexNegative** warns that control may reach an array access `A[I]` when the value of the index `I` is negative.
- **IndexTooBig** warns that control may reach an array access `A[I]` when `A.length <= I`.
- **Initially** warns that a constructor does not establish the post-condition stated in an `initially` clause [added in ESC/Java2].
- **Invariant** warns that some object invariant may not hold when control reaches a routine call, or that some object invariant may not hold on exit from the current body.
- **LoopInv** warns that some loop invariant may not hold when it is supposed to.
- **Modifies** warns that an assignment or method call violates the assignable (modifies) clauses of a routine [added in ESC/Java2].
- **OwnerNull** warns that a constructor may violate the implicit postcondition `this.owner != null`.
- **NegSize** warns of a possible attempt to allocate an array of negative length.
- **NonNull** warns of a possible attempt to assign the value `null` to a variable whose declaration is modified by a `non_null` pragma, or to call a routine with an actual parameter value of `null` when the declaration of the corresponding formal parameter is modified by a `non_null` pragma.

- **NonNullInit** warns that a constructor may fail to establish a non-null value for an instance field of the constructed object when the declaration of that instance field is modified by a `non_null` pragma.
- **Null** warns of a possible attempt to dereference null, for example, by field access `O.f`, an array access `O[i]`, a method call `O.m(...)`, a synchronized statement `synchronized (O) ...`, or a throw statement `throw O`, where `O` evaluates to `null`.
- **Post** warns that a routine body may fail to establish some normal postcondition (on terminating normally) or some exceptional postcondition (when terminating by throwing an exception of a relevant type).
- **Pre** warns that control may reach a routine call when some precondition of the routine does not hold.
- **Race** warns of a possible attempt to access a monitored field while not holding the requisite lock(s).
- **RaceAllNull** warns of a possible attempt to access a monitored field at a time when all of the objects designated as monitors for the field are null.
- **Reachable** warns that control may reach an `unreachable` pragma.
- **Uninit** warns that control may reach a read access to a local variable before execution of any assignment to the variable other than an initializer in a declaration modified by an `uninitialized` pragma.
- **Unreadable** warns that control may reach a read access of a field or variable `x` when the expression in a `readable_if` pragma modifying `x`'s declaration is false.
- **ZeroDiv** warns of a possible attempt to apply the integer division (`/`) or remainder (`%`) operator with zero as the divisor.

4.3 Command-line options

ESC/Java2 has implemented a number of command-line options that are not present in ESC/Java, as well as documenting some of those in ESC/Java. ESC/Java2 also kept many experimental (and undocumented) options from ESC/Java. The following is a partial list of command-line options available in ESC/Java2; some of these have been added to those available in ESC/Java or had their behavior altered.

Note that in ESC/Java all command-line options and their arguments must precede all file names. ESC/Java2 allows file names and other input entries to be intermixed with options.

- `-bootclasspath` *directory-path* : specifies the location of system binary files; this directory path is appended to any specification for the classpath in finding class files; the default is the platform-dependent classpath specified by Java preferences (using the -v option will show the full classpath being used)
- `-class` *fully-qualified-class-name* : the named class will be one of the input entries upon which the program acts
- `-classpath` *directory-path* : specifies the directories in which binary (.class) files for types are sought; the default is to use the value of the CLASSPATH environment variable, or to use just the current directory if neither `-classpath` nor CLASSPATH is specified

- `-da` : synonym for `-disableassertions`
- `-dir` *directory* : providing a directory as an input entry is equivalent to (but much shorted and more robust than) listing all JML-relevant files within that directory
- `-disableassertions` : as in Java, causes parsing to read Java 1.4 source but ignores any Java assert statements
- `-ea` : synonym for `-enableassertions`
- `-eajava` : synonym for `-javaAssertions`
- `-eajml` : synonym for `-jmlAssertions`
- `-enableassertions` : as in Java, causes parsing to read Java 1.4 source and to use Java assert statements
- `-era` : enable reachability analysis
- `-erst`: enable reachability analysis with specification testing
- `-f` *filename* : the text within the named file is inserted at that location in the command-line; this option is typically used to insert standard sets of options or input elements into the command-line.
- `-file` *filename* : the file given is one of the input elements upon which the program will act
- `-gui` : this must be the first command-line argument; it causes the GUI version of ESC/Java2 to start
- `-help` : causes the program to print information about command-line options and then to terminate
- `-javaAssertions` : causes parsing to read Java 1.4 source and to treat Java assert statements as Java does, as a throw of a `java.lang.AssertionError` exception
- `-jmlAssertions` : causes parsing to read Java 1.4 source and to treat Java assert statements like JML assert statements
- `-list` *filename* : the given file must contain a sequence of input entries, one per line; the option can be used to include a standard set of files, directories, packages, or classes (see also `-f`)
- `-loop` *n*[| .0 | .5] : ESC/Java analyzes loops by unrolling them. The argument of the option specifies the number of times the loop is unrolled. The default is 1.5. The precise meaning of the argument (including the inclusion of the .5) is described in the ESC/Java User's Manual.
- `-neverBinary` : do not use any class files, even if there is no java file for a type
- `-neverSource` : do not use any java files, even if there is no class file for a type
- `-noCautions` : suppresses the reporting of any cautions
- `-nocheck` : will not execute the static checking phase, though does all the parsing, typechecking and verification condition generation
- `-nowarn` *warning-type* : turns off the reporting of warnings of the given type; use a warning-type of 'All' to turn off all warnings
- `-package` *package-name* : includes all the source files in the given package in the list of source files being processed

- `-parsePlus` : causes ESC/Java2 to process annotations following the markers `//+@` and `/*+@` (see Section 3.2 [Format of annotations], page 8).
- `-preferBinary` : if both java and class files exist on the source and classpath for a type, then always use the class file
- `-preferRecent` : if both java and class files exist on the source and classpath for a type, then always use the one with the most recent modification time
- `-preferSource` : if both java and class files exist on the source and classpath for a type, then always use the java file
- `-quiet` : turns off any informational messages, leaving only reports of errors, warnings and cautions.
- `-simplify` *filename* : used to specify the Simplify executable that should be used by ESC/Java2. The executable chosen must be the appropriate one for the platform on which you are running.
- `-source` *version* : interprets the source code according to the definition of Java for the given version. The only implemented effect of this option is the interpretation of `assert` as a keyword if the version is `1.4` and as a normal identifier if the version is not `1.4`. The default is version `1.3`. In the absence of other options, enabling Java 1.4 will cause Java assertions to be treated as exceptions (the `-javaAssertions` mode). See Section 3.19.4 [Java and JML assert statements], page 46.
- `-sourcepath` *directory-path* : specifies the directories in which source and specification files for types are sought; the default is to use the classpath
- `-specs` *directory-path* : specifies a directory path of specification files. ESC/Java2 will not work well without a set of specifications. The jar file constituting a release contains a current version of the JML specifications; if ESC/Java2 is run from that jar file, those specification will be the default value of this option.
- `-v` : verbose output describing the steps of processing (ESC/Java2 has added more output to that produced by ESC/Java)
- `-warn` *warning-type* : turns on the reporting of warnings of the given type. Types of wanings include: UnsoundIncomplete,

4.4 Environment variables affecting ESC/Java2

There are several environment variables which alter the behavior of ESC/Java2 or Simplify, which is called by ESC/Java2. These are described in this section.

- `ESCTOOLS_RELEASE`
- `ESCTOOLS_ROOT`
- `SIMPLIFY` - the name of the Simplify executable appropriate to the platform on which ESC/Java2 is running. Current options are Simplify-1.5.4.linux, Simplify-1.5.5.macosx, Simplify-1.5.4.solaris, and Simplify-1.5.4.exe (for Windows). Only Mac OS X, Linux and Windows have been tested
- `CLASSPATH`
- `BOOTCLASSPATH`
- `ESC_SPECS` - sets the default value for `-specs`

- `ESCJ_VERBOSE` - normally not set. When set to some value, then more verbose output is obtained. This is simply output describing the environment variables used in invoking the tool.
- `ESCJ_STDARGS` - the command-line options to be used (in addition to any additional ones on the command-line for escjava2). (Default: -nowarn Deadlock)
- `ESC_REMOTE_DEBUG` - normally not set. When set to some value, then the value of `JAVA_VM_DEBUG_FLAGS` is included in the options supplied to the virtual machine, intended to be used to invoke the virtual machine in a debugging mode.
- `JAVA` - the name of the executable of the Java virtual machine (default: java)
- `JAVA_VM_FLAGS` - the set of options to be supplied to the Java virtual machine (the default is an empty string)
- `JAVA_VM_DEBUG_FLAGS` - a set of flags to be used when ESC_REMOTE_DEBUG is set, intended to be used to invoke the virtual machine in a debugging mode (default is a set of options appropriate for debugging on my machine!)

The following variables affect the running of the Simplify prover.

- `ESCJ_SIMPLIFY_ARGS` - the command-line arguments to use in running simplify (default: -noprune -noplunge)
- `PROVER_KILL_TIME` - the maximum number of seconds to spend on any one proof attempt (default: 300)
- `PROVER_CC_LIMIT` - the maximum number of static checker warnings to report for any given routine (default: 10)
- ... and many others, undocumented, but all beginning with PROVER_ ...

The following variables are set automatically to match a standard release. For normal operation they should not be set. They would be set to another value only in debugging or other unusual circumstances. However, if they are set in the environment to an arbitrary value, they would cause a malfunction of the tool.

- `ESC_CLASSPATH` - the classpath needed to run the ESC/Java2 tool (i.e. to find its class files). This is different than the classpath needed to find the source and class files being checked; those classpaths are set with the `-classpath` and `-sourcepath` command-line options.
- `ESCJ_SIMPLIFY_DIR` - the directory in which to find the Simplify executable

5 Changes to static checking in ESC/Java2

Many of the changes from ESC/Java to ESC/Java2 are changes to the parser so that ESC/Java2 will parse and typecheck all of current JML, as well as being upgraded to handle Java 1.3 and 1.4. However, there are several ways in which the static checking has been modified. They are described in this chapter.

5.1 Handling of specification inheritance

The semantics of the inheritance of specifications has changed from ESC/Java to ESC/Java2; ESC/Java2 now matches the semantics of JML. The implementation required changes in the way that translation of specifications into guarded commands is performed. The change in semantics is described with examples in Section 3.19.13 [Specifications and inheritance], page 51.

5.2 non_null

The semantics of `non_null` applied to formal parameters in the presence of inheritance has changed from ESC/Java to ESC/Java2. If an overriding declaration declares a formal parameter `non_null` then all overridden declarations must have that formal parameter declared `non_null`. This is described in more detail in Section 3.19.13.3 [Inheritance and non_null], page 54.

5.3 Translation of the Java 1.4 assert statement

The handling of the Java 1.4 assert statement is discussed in Section 3.19.4 [Java and JML assert statements], page 46. The assert statement is translated into equivalent Java or JML constructs and the static checker is used without modification.

5.4 Semantics of String

Although java.lang.String is a reference type in Java, it has a number of built-in properties that were not modeled in Esc/Java, but have been added in Esc/Java2. These properties are the following:

- The built-in + and += String concatentation operators
- Explicit String literals are interned
- The intern method of the String class

5.4.1 Concatentation operators

Note that the += operator is equivalent to an expression using the + operator, namely x += y is equivalent to x = x + y.

The key semantic property of the String concatenation property is that it produces a fresh (newly allocated) object. This is important because within a pure method a fresh object may be modified, just as a newly constructed object may be modified.

5.4.2 Explicit String literals

The important semantic property of String literals in Java is that two syntactically identical String literals also compare equal as objects. Also, the concatenation of two String literals produces a String literal (at compile time). Thus,

```
String a,b,c;
a = "sample";
b = "sample";
c = "sam" + "ple";
boolean t = (a==b);        // This is true
t = (a==c);                // This is also true
b = new String("sample");
t = (a==b);                // This is NOT true
```

The Simplify prover does not have any built-in knowledge of Strings. Consequently, Esc/Java2 makes a note of all of the String literals in a module, replacing each one by a unique symbol, but using the same symbol for syntactically identical literals. The concatenation of String literals is performed by the parser.

5.4.3 The intern method

The intern operation maps Strings that compare equal using the equals method to the same Object. String literals are automatically interned. Thus

```
boolean t;
String a,b,c;
a = "sample";
b = new String("sample");
c = b.intern();
t = (a==b);                // This is NOT true
t = (a==c);                // This IS true
```

Interning is not yet implemented in ESC/Java2.

5.5 The `\TYPE` and `java.lang.Class` types

In JML and ESC/Java2, uses of `\TYPE` values and `java.lang.Class` values are interchangeable, with the exception, in ESC/Java2, that methods of class `java.lang.Class` cannot be applied to a `\TYPE` value.

5.6 The initially clause

JML defines an initially annotation that states a predicate which must be true of an object after any constructor call. This is implemented in ESC/Java2 by adding these assertions as additional postconditions on each non-helper constructor,

5.7 The constraint clause

JML defines a constraint annotation that states a predicate which must be true of an object after any method call. This is implemented in ESC/Java2 by adding constraints as additional postconditions on each non-helper method,

5.8 Use of modifies clauses in checking routine bodies

ESC/Java2 mitigates problems caused by errors in modifies clauses by maintaining pre-state values for any variable mentioned in an `\old` construct (in addition to those in modifies clauses). As a result, the static checking of the routine body proceeds correctly despite any errors in the modifies clauses.

5.9 Defaults for modifies clauses

ESC/Java had an implicit default for a missing modifies clause, namely that nothing was modified (`modifies \nothing;` in JML). JML defines the default instead to be `modifies \everything;`. ESC/Java2 has implemented this as the default. The implicit use of `modifies \nothing;` had the danger of missing or hiding many errors in usage, since unexpected changes to variables caused by calling a routine are a not unusual source of bugs. However, some outstanding issues are discussed in the following section.

There are three special cases of defaults for the modifies clause:

- Methods annotated as pure (including methods in pure classes) have a default modifies clause of `modifies \nothing;`.
- Constructors annotated as pure (including constructors in pure classes) have a defaults modifies clause of `modifies this.*;`. It may have non-default modifies clauses, but these may not allow more fields to be modified than are contained in `this.*`.
- A default constructor (that is not declared) has a default modifies clause that allows the same fields to be modified as does its superclass constructor.

5.10 modifies \everything

The use of `\nothing`, `\everything`, and `\not_specified` in a modifies clause are now implemented. `\not_specified` is equivalent to `\everything`, and `\everything` is the default. However there are two issues outstanding.

- With this default, any method that has no annotations is assumed to have the behavior `modifies \everything;`. There is very little reasoning that can be performed as a result. Thus any use of the ESC/Java2 tool will require having specification files for system classes and for many of the other classes in the user's body of code.
- The logic to reason about routine bodies that contain calls of methods that are specified with `modifies \everything;` has not yet been defined or implemented. A caution is issued in these cases. But to avoid breaking tests, this caution is not yet turned on.

5.11 Checking of modifies

5.12 \typeof

In ESC/Java, the `\typeof` operator applied only to arguments of reference type. In ESC/Java2, the arguments may also be primitive types and the static checker correctly equates the value with the corresponding type literal of the form `\type(T)`, where `T` is a primitive type name.

Status: not fully implemented

5.13 Use of pure routines in annotations

Status: This is allowed in ESC/Java2

5.14 Checking of model fields

The implementation here uses the Java parsing infrastructure to parse a type declaration, but within a model class ESC keywords are not enclosed in annotation comment symbols.

5.15 \not_modified

The `\not_modified` feature of JML is implemented in ESC/Java2 by replacing it with the predicate `x == \old(x)` for each argument `x`.

6 Incompatibilities

6.1 Major features of Java not implemented in ESC/Java2

6.1.1 Java 1.5

ESC/Java2 does not implement any of the new features in Java 1.5 or 1.6. Version 2.0b3 onwards will parse Java 1.5 bytecode using BCEL, so that ESC/Java2 can be run within a Java 1.5 JVM.

6.1.2 anonymous and block-level classes

These are supported, but the bodies of methods are not checked. See Section 3.19.8 [anonymous classes], page 49 and Section 3.19.9 [block-level class declarations], page 49.

6.1.3 serialization

Status: not implemented

6.1.4 most multi-threading considerations

Status: not implemented

6.1.5 Java generics

Status: Support for generic typing in Java (or any other features of Java 1.5 or 1.6) is not implemented in ESC/Java2.

6.2 Major features of JML not implemented in ESC/Java2

6.2.1 code_contract clauses

Status: Not implemented.

6.2.2 some aspects of store-ref expressions

Status: Not all aspects of store-ref expressions are implemented.

6.2.3 implies_that and for_example behavior

These sections of a routine's specification provide specifications that are implied by the regular specifications. They are parsed and typechecked, but no use of them is made in static checking; they are not used as an aid to reasoning, nor is it checked that they follow from the other specifications.

6.2.4 splitting of annotations across comments

JML technically does not permit annotations to be split across comments, though there is no definition of an unsplittable unit of an annotation. In practice, the JML tools allow some splitting and ESC/Java2 allows some splitting but they are not consistent. (See Section 3.2 [Format of annotations], page 8). When in doubt, use a multi-line comment.

6.3 Limitations of static checking

This is an open reseach question.

6.4 Incompatibilities with ESC/Java

This section describes ESC/Java features that are not present or behave differently in ESC/Java2; the many additions to ESC/Java provided by ESC/Java2 are not discussed.

6.4.1 Error messages and warnings

ESC/Java2 has added error messages to conform with current JML semantics. Some old ESC/Java errors and warnings are no longer appropriate and have been removed. The overall organization of error messages and warnings is unchanged (see Section 4.1 [Error and warning messages], page 59).

6.4.2 also

The keywords `also_requires`, `also_ensures`, `also_exsures`, and `also_modifies` are no longer supported. Use `also` and note the change in semantics as described in Section 3.19.7 [Original also specifications], page 48.

6.4.3 inheritance of specifications

The combining of specifications of overridden and overriding methods to create a composite specification has been revised to match the semantics specified for JML, as described in Section 3.19.13 [Specifications and inheritance], page 51.

6.4.4 non_null on formal parameters and results of routines

The rules and meaning of `non_null` in the presence of inheritance is altered as described in Section 3.19.13.3 [Inheritance and non_null], page 54.

6.4.5 monitored_by

ESC/Java2 agrees with ESC/Java in the implementation of `monitored_by`. However, this keyword is deprecated in JML in favor of `monitors_for`. It may at some time be deprecated in ESC/Java2 as well.

6.4.6 readable_if

Status: Not implemented.

6.4.7 initially (old style)

ESC/Java1 had a field declaration annotation that was a modifier of the field declaration itself. JML has replaced that with a class level declaration (similar to the invariant) clause that states an assertion that must be true of objects immediately after construction. The old form is deprecated in JML but supported still in ESC/Java2.

6.4.8 semicolon termination

The original ESC/Java did not allow, and later tolerated, termination of clauses by semicolons. JML requires this. ESC/Java2 issues a caution if a required semicolon is not present (see [Terminating semicolons], page 10).

6.4.9 Routine bodies in spec files

The original ESC/Java allowed routines declared in specification files to have bodies, which were then checked. ESC/Java2 allows bodies for Java declarations only in .java files. Model methods, model constructors, and routines in model classes may have bodies in at most one specification file.

6.5 Non-JML features in ESC/Java2

There are some syntactic constructs accepted by ESC/Java and continue to be accepted by ESC/Java2 (or are additions in ESC/Java2) that are not part of JML.

6.5.1 annotation comments beginning with `//-@` or `/*-@`

ESC/Java2 allows annotation comments to begin with either `//-@` or `/*-@` to allow for experimental or old ESC/Java constructs that are not part of JML.

6.5.2 order of clauses

6.5.3 splitting of annotations

6.5.4 helper annotations

JML and ESC/Java2 differ in which sorts of routines are allowed to be `helper` routines.

6.5.5 \typeof applied to primitive types

6.5.6 unreachable

There is no `unreachable` annotation in JML.

6.5.7 \not_modified

ESC/Java2 allows arbitrary pure expressions as arguments to `\not_modified`; JML only allows store references.

6.5.8 specifications of default constructor

6.5.9 loop_predicate

6.5.10 skolem_constant

6.5.11 still_deferred

6.5.12 writable_deferred

6.5.13 writable_if

6.5.14 readable_if

6.5.15 monitored_by

ESC/Java2 still supports the `monitored_by` field annotation, as in ESC/Java, although this form has been deprecated in JML.

6.5.16 dttfsa

6.5.17 uninitialized

ESC/Java2 and ESC/Java include the `uninitialized` annotation. This is not present in JML.

6.5.18 placement of annotations

ESC/Java2 maintains some of ESC/Java's features with regard to alternate placement of annotations. The customary location for annotations is just prior to the declaration to which they apply (with the exception of the `in` and `maps` assertions for field declarations). ESC/Java2 allows in addition the following.

- Annotations may be placed just prior to a routine's body - either just before the opening `{` or the ; marking an absent body.
  Example:
  ```
  public void m() //@ pure ensures true;
  {}
  ```
- Annotations on field declarations may be placed between the initializer for the field and the terminating semicolon. This syntax also applies to local variable declarations within routine bodies.
  Example:
  ```
  public Object o = new Object() /*@ non_null */;
  ```
- Annotations on formal parameters may be placed between the identifier and the following comma or right parenthesis.
  Example:
  ```
  public void m(Object o /*@ non_null*/, Object oo /*@ non_null */);
  ```

JML does not recognize this syntax, with the exception of specification clauses (but not modifiers such as `pure`, `non_null`, or `nullable`) placed just prior to the routine body. Consequently using this style of annotation writing is discouraged in ESC/Java2. Should we generate a caution?

6.5.19 semicolon termination

JML requires annotations to be terminated with semicolons. ESC/Java2 will warn about missing semicolons, but does not require them.

6.5.20 need for the field, method, constructor keywords

ESC/Java2 parses `non_null` and `nullable` in method declarations

6.5.21 omission of method bodies

ESC/Java2 does not require bodies of methods and constructors to be present, even in .java files. The JML checker does require such bodies in .java files, although not in specification files with other suffixes.

6.5.22 Errors and cautions

Some JML errors are reported as cautions by ESC/Java2. This enables as much checking to be performed as possible in cases in which the correction to the error seems obvious.

6.5.23 membership in lockset

The features associated with locks and the sets of locks associated with an object were part of the original ESC/Java and only recently added to JML. JML needs documentation of these features (\lockset, \max, and membership in a lockset); JML does not yet include the membership in a lockset operation.

6.6 JML features needing clarification

6.6.1 model programs

6.6.2 callable, accessible

6.6.3 when, measured_by

6.6.4 initializer, static_initializer

6.6.5 desugaring of forall

6.6.6 weakly

6.6.7 hence_by

6.6.8 use of \result in duration and working_space clauses

6.6.9 instance fields and this in constructor preconditions

6.6.10 rules about splitting annotations across comments

6.6.11 \typeof applied to primitive types

6.7 Desired extensions

This is a list of some discussed but not resolved potential extensions to current JML.

- Changing the order of suffixes so that either jml or spec comes before java.
- Allow modifies clauses outside of `{| |}` pairs; the clause would be distributed across the nested specifications just like requires is.
- Allow specifications on statements within a body of a routine.
- Definite definition of \elemtype applied to a non-array.

Appendix A Modifier Summary

This table summarizes which Java and JML modifiers may be used in various grammatical contexts.

Grammatical construct	Java modifiers	JML modifiers
All modifiers	`public` `protected` `private` `abstract` `static` `final` `synchronized` `transient` `volatile` `native` `strictfp`	`spec_public` `spec_protected` `model` `ghost` `pure` `instance` `helper` `non_null` `nullable` `non_null_by_default` `nullable_by_default`
Class declaration	`public` `final` `abstract` `strictfp`	`pure` `model`
Interface declaration	`public` `strictfp`	`pure` `model`
Nested Class declaration	`public` `protected` `private` `static` `final` `abstract` `strictfp`	`spec_public` `spec_protected` `model` `pure`
Nested interface declaration	`public` `protected` `private` `static` `strictfp`	`spec_public` `spec_protected` `model` `pure`
Local Class (and local model class) declaration	`final` `abstract` `strictfp`	`pure` `model`

Type specification (e.g. invariant)	`public`
	`protected`
	`private`
	`static`
		-

Field declaration	`public`
	`protected`
	`private final`
	`volatile`
	`transient`
	`static`
		`spec_public`
		`spec_protected`
		`non_null`
		`nullable`
		`instance`
		`monitored`

Ghost Field declaration	`public`
	`protected`
	`private`
	`static final`
		`non_null`
		`nullable`
		`instance`
		`monitored`

Model Field declaration	`public`
	`protected`
	`private`
	`static`
		`non_null`
		`nullable`
		`instance`

Method declaration	`public`
	`protected`
	`private`
	`abstract`
	`final static`
	`synchronized`
	`native`
	`strictfp`
		`spec_public`
		`spec_protected`
		`pure non_null`
		`nullable`
		`helper`

Constructor declaration	`public`
	`protected`
	`private`
		`spec_public`
		`spec_protected`
		`helper pure`

Model method	`public`
	`protected`
	`private`
	`abstract`
	`static final`
	`synchronized`
	`strictfp`
		`pure non_null`
		`nullable`
		`helper`

Model constructor	`public` `protected` `private`	`pure` `helper`
Java Initialization block	`static` ???	???
JML initializer and static_initializer annotation	???	???
Formal parameter	`final`	`non_null` `nullable`
Local variable and local ghost variable declaration	`final`	`ghost` `non_null` `nullable` `uninitialized`

Note that within interfaces, fields are implicitly public, static and final. Ghost and model fields are implicitly public and static, though they may be declared instance (i.e. not static).

Bibliography

[Flanagan-etal02]
C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, "Extended Static Checking for Java". Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. June, 2002.

[LeavensBakerRuby02]
G. T. Leavens, A. L. Baker, C. Ruby. "Preliminary Design of JML: A Behavioral Interface Specification Language for Java". Iowa State University, Department of Computer Science, TR 98-06t. December, 2002. Available from `ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.ps.gz`.

[Leavens-etal03]
G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, J. Kiniry. JML Reference Manual. Available from `http://www.jmlspecs.org`.

Concept Index

(

(* 42

*

*) 42
*/ 8

-

-bootclasspath 61
-class 61
-classpath 61
-da 61
-dir 62
-disableassertions 62
-ea 62
-eajava 62
-eajml 62
-enableassertions 62
-era 62
-erst 62
-f 62
-file 62
-gui 62
-help 62
-javaAssertions 62
-jmlAssertions 62
-list 62
-loop 62
-neverBinary 62
-neverSource 62
-noCautions 62
-nocheck 62
-nowarn 62
-package 62
-parsePlus 62
-preferBinary 63
-preferRecent 63
-preferSource 63
-quiet 63
-simplify 63
-source 63
-sourcepath 63
-specs 63
-v 63
-warn 63

/

/*+@ 8
/*@ 8
//+@ 8
//@ 8

<

< 39
<: 39
<= 39
<=!=> 39
<== 39
<==> 39
<esc> 8
<ESC> 8
<jml> 8
<JML> 8

=

==> 39

@

@*/ 8
@+*/ 8

\

\bigint 42
\duration 40
\elemtype 41
\everything 45
\exists 43
\forall 43
\fresh 40
\invariant_for 41
\is_initialized 41
\lblneg 41
\lblpos 41
\lockset 42
\LockSet 43
\lockset membership 42
\max 42, 43
\min 43
\nonnullelements 41
\not_modified 40
\nothing 45
\num_of 43
\old 40
\old 66
\product 43
\reach 40
\real 43
\result 40
\space 40
\sum 43
\type 41
\TYPE 42
\typeof 41

\working_space 40

1

1.3 63
1.4 63

A

accessible 29
also_ensures 24
also_exsures 24
also_modifies 24
also_requires 24
annotation markers 8
assert 63, 65
assignable 26
assignable_redundantly 26

C

callable 29
cautions 59, 62
classpath 5, 61
clause defaults 53
CM3 2
comments, annotation 8
constraint 66

D

defaults 53
diverges 28
diverges_redundantly 28
duration 28

E

error messages 59
ESC/Java 2
exsures 26
exsures_redundantly 26

F

fatal errors 59
file names 5
filenames 5
final-routine 31
for_example 69
format of annotations 8

I

implies_that 69
informal predicate 42
inheritance of specifications 53
initially 66

J

Java 1.5 69
Javafe 2
JML 2
JUnit 2

M

Mocha 2
model program statements 45
modifiable 26
modifiable_redundantly 26
modifies 26, 66
modifies_redundantly 26
most-refined compilation unit 6
MRCU 6

N

native 32

P

private 13
protected 13
public 13

Q

quantified expressions 43

R

refinement sequence 5, 6

S

set comprehension 43
signals 26
signals_redundantly 26
Simplify 2
sourcepath 5, 63
spec_protected 13
spec_public 13
static 31
strictfp 32
String 65
suffixes 5
synchronized 32

V

verbose 63

W

warnings 59

when 28
working_space 28