Java Modeling Language (JML)

Reference Manual

2nd edition

David R. Cok, Gary T. Leavens, and Mattias Ulbrich

DRAFT September 8, 2025

This draft is very much a work in progress with many points under discussion.

The most recent released version of this document is available at
https://www.openjml.org/documentation/JML_Reference_Manual.pdf

The LaTex source for the document is maintained and edited in Overleaf:
https://www.overleaf.com/project/5ceee26404c2854a1590029f

Contents

Chapter 1

Introduction

JML is a *behavioral interface specification language* (BISL) that builds on the Larch approach [?] [?] and Eiffel [?] [?] (and other languages such as VDM [?] and APP [?]). In this style of specification, which might be called model-oriented [?], one specifies both the interface of a method or abstract data type and its behavior [?]. In particular JML builds on the work done by Leavens and others in Larch/C++ [?] [?] [?]. (Indeed, large parts of this manual are adapted wholesale from the Larch/C++ reference manual [?].) Much of JML's design was heavily influenced by the work of Leino and his collaborators [?] [?] [?], then subsequently by Cok's work on ESC/Java2 [?] and OpenJML [?], the work on the KeY tool [?], and by work on other specification languages such as Spec# [?], ACSL/ACSL++ [?], SPARK [?], and Dafny [?, ?]. JML continues to be influenced by ongoing work in formal specification and verification. A collection of papers relating directly to JML and its design is found at http://www.jmlspecs.org/papers.shtml.

It is important to note that JML is intentionally a specification language for an existing, industrial-grade, heavily-used programming language — Java — that must be usable at scale for realistic software programs. To do this, JML and its tools must work with the large and continually evolving feature set of modern programming languages and the somewhat messy semantics of such languages and their standard libraries. This situation is a contrast to languages designed anew as both a specification programming language, such as Eiffel and Dafny, and in the case of Dafny in particular, designed expressly for proving programs. The latter can have more focused features and clear semantics attuned to proof, but may lack a broad user-base.

1.1 Behavioral Interface Specifications

The *interface* of a method or type (i.e., a Java class or interface) is the information needed to use it from other parts of a program. In the case of JML, this is the Java syntax and type information needed to call a method or use a field or type. For a

method, the interface includes such things as the name of the method, its modifiers (including its visibility and whether it is final), its number of arguments, its return type, what (checked) exceptions it may throw, and so on. For a field, the interface includes its name, type and modifiers. For a type, the interface includes its name, its modifiers, its package, whether it is a class or interface, its supertypes, and the interfaces of the fields and methods it declares and inherits. JML specifies all such interface information using Java's syntax.

A *behavior* of a method describes the possible state transformations that it performs when invoked. A behavior of a method is specified by describing

- a set of states for which calling the method is permitted, these are called the method's *pre-states*,
- the set of memory locations that the method is allowed to assign to (and hence may change), and
- the relation between each permitted pre-state and the post-state(s) that the method is supposed to achieve. These *post-states* may result from the method either (a) returning normally, (b) throwing an exception, or (c) not returning to the caller.

The states for which calling the method is defined are formally described by logical predicates called the method's *pre-condition*. The set of locations the method is allowed to assign to is described by the method's *frame condition* [**?**]. The post-states that are allowed to result from the method returning normally are specified by its *normal postcondition*. Similarly the relationships between the specified pre-states and the states that may result from throwing an exception are described by the method's *exceptional postcondition*. The pre-states for which the method need not return to the caller are described by the method's *divergence condition*. A method specification is thus a generalization of a Hoare triple [**?**], as adapted by Meyer to the design-by-contract style of specification [**?**].

The behavior of an abstract data type (ADT) is specified as a combination of the behavior of its methods (specified as described above) and by abstractly describing the states of its objects (and any static fields it may have). The *abstract state* of an object can be specified either by using JML's model and ghost fields [**?**], which are specification-only fields, or by using a shortcut (`spec_public` or `spec_protected`) that specifies that some fields used in the implementation are considered to have public or protected visibility for specification purposes. These declarations allow the specifier using JML to model an instance as a collection of abstract instance variables, in much the same way as other specification languages, such as Z [**?**] [**?**] or Fresco [**?**].

1.2 A First Example

As a first example, consider the JML specification of a simple Java class `Counter` shown in Fig. **??**. (An explanation of the notation follows.)

```
1  package org.jmlspecs.samples.jmlrefman;
2
3  /** A simple Counter. **/
4  public class Counter {
5
6      /** The counter's value. **/
7      /*@ spec_public @*/ private long count = 0;
8
9      //@ public invariant 0 <= count && count <= Long.MAX_VALUE;
10
11     /** Initialize this counter's value. **/
12     /*@ requires true;
13       @ ensures count == 0;
14       @*/
15     public Counter() {
16         count = 0;
17     }
18
19     /** Increment this counter's value. */
20     /*@ requires count < Long.MAX_VALUE;
21       @ assignable count;
22       @ ensures count == \old(count + 1);
23       @*/
24     public void inc() {
25         count++;
26     }
27
28     /** Return this counter's value. */
29     //@ ensures \result == count;
30     public /*@ spec_:pure @*/ long getCount() {
31         return count;
32     }
33 }
```

Figure 1.1: `Counter.java`, with Java code and a JML specification. The small line numbers to the left are only for the purpose of referring to lines in the text and are not part of the file.

The interface of this class consists of lines 4, 7, 15, 24, and 30.

Line 4 specifies the class name, `Counter` and the fact that the class is `public`. Line 7 declares the private field `count` and also that it is `spec_public`, which means that `count` can be treated as public for specification purposes.

Lines 15, 24, and 30 specify interfaces of the constructor (line 15) and two methods (lines 24 and 30). The methods `inc` and `getCount` are specified to be public and to have return types `void` and `long`, respectively.

The behavior of this class is specified in the JML annotations found in the special comments that have an at-sign (`@`) as their first character following the usual comment beginning. Such lines look like comments to Java, but are interpreted by JML and its tools. For example, the JML annotation on line 7 starts with an annotation comment marker of the form `/*@`, and this annotation continues until `*/` is seen. In such JML annotations, one can begin and end the comment with one or more at-signs, as in `@*/`. And, as in lines 12–14, at-signs at the beginnings of lines in a multi-line comment are also ignored by JML. The other form of such annotations can be seen on line 9, which is a JML annotation that starts with `//@` and continues to the end of that line. Note that there can be no space between the start of comment marker, either `//` or `/*`, and the first at-sign; thus `// @` starts a comment, not a JML annotation. (See §?? for more details about JML annotations.)

The first annotation, on line 7 of Fig. ?? specifies that the `count` field is `spec_public`, which means that it can be referred to in any (public) specification that has access to the class `Counter` (cf. §??). That is, as far as the JML specifications are concerned (but not for Java code), `count` can be used as if it were declared as `public`.

The `count` field is used on line 9 in the public invariant of the class. This invariant says that at the beginning and end of each public method, and at the end of the constructor, the assertion

```
0 <= count && count <= Long.MAX_VALUE
```

will be true. This can be regarded as an assumption at the beginning of each method and as an obligation to make true at the end of each method that might change the value of the field `count`. (See §?? for more about object and class invariants.)

In Fig. ??, the specification of each method and constructor precedes its interface declaration. This follows the usual convention of Java tools, such as `javadoc`, which put such descriptive information in front of the method. (See §?? for more details about method specifications.)

The specification of the constructor `Counter` is given on lines 12–13. The constructor's precondition is the predicate following the keyword `requires` (i.e., `true`), and it says that the constructor can be called in any state. Such trivial preconditions (and `requires` clauses) can be omitted. The constructor's postcondition follows the keyword `ensures`. It says that when the constructor returns, the value in the field `count` is 0. Note that the value 0 satisfies the specified invariant, as the specification dictates.

The specification of the method `inc` is given on lines 20–24. Its precondition is that `count` not be the largest value for a `long`, so that incrementing it does not cause its value to become negative, as that would violate the invariant. Its postcondition says that the final value of `count` is one more than the value of `count` in the state in which the method was invoked.

Note that in the postcondition, JML uses a keyword (`\old`) that starts with a backslash (\); this lexical convention is intended to avoid interfering with identifiers in the user's program. Another example of this convention is the keyword `\result` on line 29.

The frame condition expressed in the assignable clause on line 21 says that the method may assign to `count`, but also prohibits it from assigning to any other locations (i.e. fields of objects) that are visible outside the method and which existed before the method started execution. (See §?? for more details about method framing.)

The postcondition of the `getCount` method on line 29 says that the result returned by the method (`\result`) must be equal to the value of the field `count`.

The method `getCount` is specified using the JML modifier `spec_pure`. This modifier says that the method has no effects, so its assignable clause is implicitly

```
assignable \nothing;
```

and allows the method to be used in specification expressions, if desired.

1.3 What is JML Good For?

JML is a formal specification language tailored to Java. Its basic use is thus the formal specification of the behavior of Java program classes and interfaces. As it is a behavioral interface specification language, JML specifies how to use such classes and interfaces from *within* a Java program; hence JML is *not* primarily designed for specifying the behavior of an entire program, though it has been used for libraries and sizable sections of programs. So the question "what is JML good for?" really boils down to the following question: what good is formal specification for Java program classes and interfaces?

The two main benefits in using JML are:

- the precise, unambiguous description of the behavior of Java classes and interfaces, and documentation of Java code,
- the possibility of tool support [?].

Although we would like tools that would help with reasoning about the concurrent behavior of Java programs, the current version of JML focuses on the sequential behavior of Java code. While there has been work on extending JML to support concurrency [?], the current version of JML does not have features that specify how Java threads interact with each other. JML does not, for example, allow the specification of elaborate temporal properties, such as coordinated access to shared variables or

the absence of deadlock. Indeed, we assume, in the rest of this manual, that there is only one thread of execution in a Java program annotated with JML, and we focus on how the program manipulates object states. To summarize, JML is currently limited to sequential specification; we say that JML specifies the *sequential behavior* of Java classes and interfaces.

In terms of detailed design documentation, a JML specification can be a completely formal contract about an interface and its sequential behavior. Because it is an interface specification, one can record all the Java details about the interface, such as whether a method is `final`, `protected`, etc.; if one used a specification language such as OCL, VDM-SL, or Z, which is not tailored to Java, then one could not record such details of the interface, which could cause problems in code integration. For example, in JML one can specify the precise conditions under which certain exceptions may be thrown, something which is difficult in a specification language that is not tailored to Java and that does not model the notion of an exception.

When should JML documentation be written? That is up to you, the user. One goal of JML is to make the notation indifferent to the precise design or programming method used. One can use JML either before coding or as documentation of finished code. While we recommend doing some design, and JML specification of the design, before coding, JML can also be used for documentation after the code is written.

Reasons for formal documentation of interfaces and their behavior, using JML, include the following.

- One can ship the object code for a class library to customers, sending the JML specifications but not the source code. Customers would then have documentation that is precise, unambiguous, but not overly specific. Customers would not have the code, protecting proprietary rights. In addition, customers would not rely on details of the implementation of the library that they might otherwise glean from the code, easing the process of improving the code in future releases.
- One can use a formal specification to analyze certain properties of a design carefully or formally (see [**?**] and Chapter 7 of [**?**]). In general, the act of formally specifying a program module has salutary effects on the quality of the design.
- One can use the JML specification as an aid to careful reasoning about the correctness of code, or even for formal verification [**?**, **?**, **?**].
- JML specifications can be used by several tools that can help debug and improve the code [**?**].

There is one additional benefit from using JML. It is that JML allows one to record not just public interfaces and behavior, but also some detailed design decisions. That is, in JML, one can specify not just the public interface of a Java class, but also behavior of a class's protected and private interfaces. Formally documenting a base class's protected interface and its "subclassing contract" allows programmers to implement derived classes of such a base class without looking at its code [**?**, **?**].

Recording the private interface of a class may be helpful in program development or maintenance. Usually one would expect that the public interface of a class would be specified, and then separate, more refined specifications would be given for use by derived classes and for detailed implementation

The reader may also wish to consult the "Preliminary Design of JML" [?] for a discussion of the goals that are behind JML's design. Apart from the improved precision in the specifications and documentation of code, the main advantage of using a formal specification language, as opposed to informal natural language, is the ease and accuracy of tool support. One specific goal that has emerged over time is that JML should be able to unify several different tool-building efforts in the area of formal methods.

The most basic tool support for JML — simply parsing and type-checking specifications — is already useful. Whereas informal comments in code are typically not kept up to date as the code is changed, the simple act of running the typechecker will catch any JML assertions referring to parameter or field names that no longer exist, and other typos. Enforcing the visibility rules can also provide useful feedback; for example, a precondition of a `public` method which refers to a `private` field of an object is suspect.

Of course, there are more exciting forms of tool support than just parsing and type-checking. In particular JML is designed to support static analysis and formal verification, as in OpenJML's extended static checker (ESC) [?, ?, ?, ?], or the KeY tool [?].[1] Other tools for JML [?] include Daikon [?], which can infer some JML specifications from execution traces during testing, the runtime assertion checker (RAC) of OpenJML [?], the RAC found in AspectJML [?],[2], and documentation (as in JML's jmldoc tool). The paper by Burdy et al. [?] is a survey of tools for JML. The utility of these tools is the ultimate answer to the question of what JML is good for.

1.4 Purpose of this document

The purpose of this document is to define a standard for the syntax and formal semantics of JML as a language. The document also distinguishes core aspects of JML, which have proved to be the most used and most important specification elements.

This reference manual thus seeks to define a standard for JML that will be a common basis for tools and for discussion but does not mean to inhibit experimentation and proposals for change. Therefore we present a framework in which new tools and approaches can be defined such that a deviation of the semantics from this standard can be clearly stated.

To make JML a versatile specification vehicle, the meaning of its annotations must

[1] There have been other formal verification tools for JML, including the LOOP tool [?, ?].

[2] AspectJML is a further evolution of a previous RAC called ajmlc [?, ?]. There was also a RAC tool from Iowa State, called jmlc [?, ?, ?], that is no longer maintained.

be unambiguously clear. And *if* tools interpret a few language constructs differently, these differences must be easily and concise stated.

1.5 Previous JML Reference Manual

This reference manual builds on the previous draft JML Reference Manual [?], which evolved over many years and had many contributors. This current edition of the reference manual is a rewrite and update of the previous draft. Some sections, particularly introductory and overview material, are taken nearly verbatim from the previous JML draft reference manual [?]. However, the current version also incorporates the experience of building tools for JML by the OpenJML and KeY developers, many decisions about new features or deprecated features made at JML workshops, and discussions about JML on the JML mailing lists, on the JML Reference Manual GitHub site, and in personal communications. This edition of the reference manual includes features that are proposed enhancements or clarifications of the consensus language definition. It also includes rationale for non-obvious language features and discussion of points that are under current debate or require extended explanation.

JML changes as the current version of Java changes. The version of JML presented here corresponds to Java 21.

1.6 Historical Precedents and Antecedents

JML combines ideas from Eiffel [?] [?] [?] with ideas from model-based specification languages such as VDM [?] and the Larch family [?] [?] [?] [?]. It also adds some ideas from the refinement calculus [?] [?] [?] [?] [?]. In this section we describe the advantages and disadvantages of these approaches. Readers not interested in these historical precedents may skip this section.

Formal, model-based languages such as those typified by the Larch family build on ideas found originally in Hoare's work. Hoare used pre- and postconditions to describe the semantics of computer programs in his famous article [?]. Later Hoare adapted these axiomatic techniques to the specification and correctness proofs of abstract data types (ADTs) [?]. To specify an ADT, Hoare described a mathematical set of abstract values for the type, and then specified pre- and postconditions for each of the operations of the type in terms of how the abstract values of objects were affected. For example, one might specify a class `IntHeap` using abstract values of the form `empty` and `add(i,h)`, where `i` is an `int` and `h` is an `IntHeap`. These notations form a mathematical vocabulary used in the rest of the specification.

There are two advantages to writing specifications with mathematically-defined abstract values instead of directly using Java variables and data structures. The first is that by using abstract values, the specification does not have to be changed when the particular data structure used in the program is changed. This permits different implementations of the same specification to use different data structures. Therefore

the specification forms a contract between the rest of the program and the implementation, which ensures that the rest of the program is also independent of the particular data structures used [?] [?] [?] [?]. Second, it allows the specification to be written even when there are no implementation data structures, e.g., for a Java interface.

This idea of model-oriented specification has been followed in VDM [?], VDM-SL [?] [?], Z [?] [?], and the Larch family [?]. In the Larch approach, the essential elaboration of Hoare's original idea is that the abstract values also come with a set of operations. The operations on abstract values are used to precisely describe the set of abstract values and to make it possible to abbreviate interface specifications (i.e., pre- and postconditions for methods). In Z one builds abstract values using tuples, sets, relations, functions, sequences, and bags; these all come with pre-defined operations that can be used in assertions. In VDM one has a similar collection of mathematical tools to describe abstract values, and another set of pre-defined operations. In the Larch approach, there are some pre-defined kinds of abstract values (found in Guttag and Horning's LSL Handbook, Appendix A of [?]), but these can be extended if needed.

However, there is a problem with using mathematical notations for describing abstract values and their operations. The problem is that such mathematical notations are an extra burden on a programmer who is learning to use a specification language. The solution to this problem is the essential insight that JML takes from the Eiffel language [?] [?] [?]. Eiffel is a programming language with built-in specification constructs. It features pre- and postconditions, although it has no direct support for frame axioms. Eiffel programmers can easily read predicates in specifications, as these are written in Eiffel's own expression syntax. However, Eiffel does not provide support for specification-only variables, and it does not provide much explicit support for describing abstract values. Because of this, it is difficult to write specifications that are as mathematically complete in Eiffel as one can write in a language like VDM or a Larch-style BISL.

JML attempts to combine the good features of these approaches. From Eiffel we have taken the idea that assertions can be written in a language that is based on Java expressions. We also adopt the "**`old`**" notation from Eiffel, which appears in JML as `\old`, instead of the Larch-style annotation of names with state functions. To make it easy to write more complete specifications, however, we use various semantic ideas from model-based specification languages. In particular we use a variant of abstract value specifications, where one describes the abstract value of an object implicitly using several model fields. These specification-only fields allow one to implicitly partition the abstract value of an object into smaller chunks, which helps in stating frame axioms. More importantly, we hide the mathematical notation behind a facade of Java classes. This makes it so the operations on abstract values appear in familiar (although perhaps verbose) Java notation, and also insulates JML from the details of the particular mathematical logic used to do reasoning.

Despite this initial emphasis on Java-like syntax and semantics in JML, the experience of the JML community over the past couple of decades demonstrated that better

pure mathematical data types and structures were also needed. In particular, using heap-independent types simplifies the reasoning load on the underlying logic engines. Hence, this 2nd edition of JML incorporates more built-in mathematical types, although the syntax retains a Java flavor.

JML has also influenced later specification languages for other programming languages, such as Spec# [?] for C#, ACSL/ACSL++ [?] for C and C++, SPARK [?] for Ada, and Dafny [?, ?]. The experience with those languages has inspired specification approaches and features that have been added to JML in turn.

1.7 Status, Plans and Tools for JML

Even after 25 years from its inception, JML remains a strong, used base for specification language research. Its tools are the basis of ongoing research publications and student projects and theses. It is used modestly in non-academic verification projects. This language definition update, staying current with changes in Java, and the corresponding tool support point to JML's continuing evolution as a relevant specification language.

JML's goals remain the same:

- to be a usable platform for research and experimentation in specification and proof, including being a platform for student projects and theses;
- to be a useful educational tool for teaching Java and teaching formal methods (formal specification and proof);
- to be usable in industrial software specification settings, both for confidence in the software under study and for what such tasks teach about specification at scale.

To this end, the JML project's plans are these:

- update and maintain this language reference as a guide to researchers, users, tool builders and as a basis for discussion;
- continue to collaborate with tool builders in order to learn from both tool implementation and practical application experience;
- correct, clarify, and expand the tutorial and other educational material on JML (and its tools).

The currently active tools for JML are

- OpenJML [?]
- KeY [?]

Other tools are welcome.

1.8 Acknowledgments

This rewrite of the *JML Reference Manual* is largely the work of David R. Cok, Gary T. Leavens, and Mattias Ulbrich, building on the previous Draft Reference Manual [?] and discussions by the JML community.

Past contributions from David Cok have been supported in part by the National Science Foundation: This material is based upon work supported by the National Science Foundation under Grant No. ACI-1314674. David Cok has also been partially supported by industrial contracts from AWS and Goldman-Sachs.

The work of Leavens and his collaborators (in particular Clyde Ruby) was supported in part by a grant from Rockwell International Corporation and by NSF grant CCR-9503168. Work on JML by Leavens (and Ruby) was also supported in part by NSF grant CCR-9803843. Work on JML by Leavens (with Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and others) has been supported in part by NSF grants CCR-0097907, CCR-0113181, CCF-0428078, and CCF-0429567, CNS 08-08913, CNS 07-07874, CNS 07-07701, CNS 07-07885, CNS 07-08330, and CNS 07-09169. The work of Erik Poll was partly supported by the Information Society Technologies (IST) program of the European Union, as part of the VerifiCard project, IST-2000-26328.

Contributions of Mattias Ulbrich stem from his participation in the KeY project. Other members of that team, such as Alexander Weigl, also contributed comments, language suggestions and critiques.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or any other funding organization.

Thanks to Bart Jacobs, Rustan Leino, Peter Müller, Arnd Poetzsch-Heffter, and Joachim van den Berg, for many discussions about the semantics of JML specifications. Thanks to Raymie Stata for spearheading an effort at Compaq SRC to unify JML and ESC/-Java, and to Rustan and Raymie for many interesting ideas and discussions that have profoundly influenced JML.

See the "Preliminary Design of JML" [?] for more acknowledgments relating to the earlier history, design, and implementation of JML.

Chapter 2

Structure of this Manual

2.1 Organization

This document presents the syntax, grammar, and semantics of the Java Modeling Language (JML); all these aspects build on the corresponding aspects of Java. Like Java and other programming languages, JML source text is divided into syntactic tokens, (largely) independent of the grammar or semantics; the JML syntax is described in §??. JML's grammar is described throughout the manual in the form described below (§??), with common aspects of the grammar summarized in §??.

The semantics of JML is given informally, relying on the description of Java in the Java Language Specification (JLS).

Chapter 3 describes some fundamental concepts for JML and specification languages generally. Chapter 4 introduces JML syntax. The subsequent chapters describe the kinds of JML annotations used for various Java program elements. The final chapters include summary tables, descriptions of obsolete syntax, and the like.

2.2 Typographical conventions

The remaining chapters of this book follow some common typographical conventions.

The document has internal clickable hyperlinks: from section references to sections, from bibliography references in the text to the corresponding entry in the bibliography, from bibliography entries to the pages containing references, and from uses of grammar non-terminals to the definitions of those non-terminals.

This style of text is used for commentary on the JML language itself, such as outstanding issues or now-obsolete practice.

Java and JML program fragments are shown either as listed code, with line numbers for reference (the line numbers are not part of the code), as in

```
1 public class Example {
2 }
```

or as a boxed example

```
public class Example2 {
}
```

2.3 Grammar

The grammar of JML is intertwined with that of Java. The grammar is given in this Reference Manual as extensions of the Java grammar, using conventional BNF-style productions. The meta-symbols of the grammar are in slightly larger, normal-weight, mono-spaced font. The productions of the grammar use the following syntax:

- non-terminals are written in italics and enclosed in angle brackets: *<expression>*
- terminals, including punctuation as terminals, are written in bold font: **old ( ) .**
- parentheses express grouping: (...)
- an infix vertical bar expresses mutually-exclusive alternatives: ... | ... | ...
- repetitions of 0 or more and 1 or more and 0 or 1 (i.e., optional) elements use post-fixed symbols: * + ?
- square brackets enclose an optional element: [ ]
  (just like ( ) ? does)
- a post-fixed ... indicates a comma-separated list of 0 or more elements:
  <expression> **. . .**
  represents what would otherwise be written
  [*<expression>* (**,** *<expression>*) *]
- 1-or-more comma-separated elements is written as
  <expression> (**,** *<expression>*) *
- a production begins with: *<non-terminal>* **::=**
- non-terminals beginning with *java-* as in *<java-identifier>* refer to a purely Java non-terminal, as defined in the JLS; a prefix of *jml-* is used to emphasize a distinction from Java. For example, *<jml-expression>* includes both *<java-expression>* and various JML-specific kinds of expressions.

Uses of a non-terminal are clickable hyperlinks to their definitions.

Section §?? contains a list of definitions of common grammar non-terminals. Other grammar definitions are scattered throughout the document, in association with the description of the associated feature. All of the grammar definitions are collected in Appendix ??.

Chapter 3

JML concepts

This chapter describes some general design principles and concepts of the Java Modeling Language that are used throughout this manual. The chapter also presents the overall way that specifications are processed and used. Some of this discussion relies on syntactic and grammatical information presented in later chapters. Also, some major concepts are presented in chapters of their own.

JML specifications are declarative statements about the behavior and properties of Java entities, namely, packages, classes, methods, and fields. Typically JML does not make assertions about how a method or class is implemented, only about the net behavior of the implementation. However, to aid in proving assertions about the behavior of methods, JML does include statement and loop specifications (in the body of the implementation).

JML is a versatile specification vehicle. It can be used to add lightweight specifications (e.g., specifying ranges for integer values or when a field may hold null) to a program but also to formulate more heavyweight concepts (such as abstracting a linked list into a sequence of values).

JML annotations are not bound to a particular tool or approach, but can serve as input to a variety of tools that have different purposes, such as runtime assertion checking, test case generation, extended static checking, full deductive verification, and documentation generation.

In deductive verification, specifications and corresponding proof obligations may be considered at different levels of granularity. Deductive verification using JML is typically concerned with modular proofs at the level of Java methods. That is, a verification system will establish that each Java method of a program is consistent with its own specifications, presuming the specifications of all methods and classes it uses are correct. If this statement is true for all methods in the program, and all methods terminate, then the system as a whole is consistent with its specifications. [**?**]

3.1 JML and Java compilation units

A Java program is organized as a set of *compilation units* grouped into packages. The Java Language Specification does not stipulate a particular means of storing the Java program text that constitutes each compilation unit. However, the vast majority of systems supporting Java programs store each compilation unit as a separate file with a name that corresponds to the class or interface it contains; usually the files constituting a package are placed in a directory named the same as the last element of the package name, and these directories are organized into a hierarchy, with parent directories named by earlier components of a package name.

The simplest way of specifying a Java program with JML is to include the text of the JML specifications directly in the Java source text, as specially formatted comments. This was shown in Fig. **??**. By using specially formatted comments to express JML, any existing Java tools will ignore the JML text.

However, in some cases the source Java files are not permitted to be modified or it is preferable not to modify them or the source code may not be available at all. In these cases, the JML specifications must be expressed separate from the Java source program text in a way that the specifications of packages, classes, methods, and fields can be associated with the correct Java entity.

Therefore, JML tools permit specifications to be either stored (a) with the Java source or (b) separately. For Java language systems in which Java source material is stored in files, the JML specifications are either in the same `.java` file (case (a)) or in a separate `.jml` file (case (b)). In case (b), the separate file has a `.jml` suffix and the same root name as the corresponding Java source file (typically the name of the public class or interface in the compilation unit), the same package designation, and is stored in the file system's directory hierarchy according to its package and class name, in the same way as the Java compilation unit source files. For the rare case in which files are not the basis of Java compilation units, the JML tools must implement a means, not specified here, to recover JML text that is associated with Java source text to enable case (b).

The rules about the format of the text in `.jml` files are presented in §**??**.

3.2 Specification inheritance

Object-oriented programming with inheritance requires that derived classes satisfy the specifications of a parent class, a property known as *behavioral subtyping*[**?**, **?**, **?**, **?**, **?**, **?**]. Strong behavioral subtyping is a design principle in JML: any visible specification of a parent class is inherited by a derived class. Thus derived types inherit invariants from their parent types and methods inherit behaviors from supertype methods they override.

For example, suppose method `m` in derived class `C` overrides method `m` in parent class `P`. In a context where we call method `m` on an object `o` with static type `P`, we will expect the specifications for `P.m` to be obeyed. However, `o` may have dynamic type `C`.

Thus `C.m`, the method actually executed by the call `o.m()`, must obey all the specifications of `P.m`. `C.m` may have additional specifications, that is, additional behaviors, constraining its behavior further, but it may not relax any of the specifications given for `P.m`.

Specifications that are not visible in derived classes, such as those marked `private`, are not inherited, because a client cannot be expected to obey specifications that it cannot see. One additional exception to specification inheritance is method behaviors that are marked with the `code` modifier**??**. These behaviors apply only to the method of the class in which the behavior textually appears or to derived classes that do not override the parent class implementation.

3.3 JML modifiers and Java annotations

The Java Modeling Language was defined prior to the introduction of annotations in Java. Some of the features of JML can now be textually represented as Java annotations. Currently JML supports both the old and new syntactic forms.

3.3.1 Modifiers

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

Each such modifier has an equivalent Java annotation. For example

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

can be written equivalently as

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

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

```
import org.jmlspecs.annotation.Pure;
@Pure public int m(int i) { ... }
```

Note that in the second and third forms, the `@Pure` designation is now part of the *Java* program and so the import of the `org.jmlspecs.annotation` package must also be in the Java program, the package defining JML annotations must be available to the Java compiler when compiling the Java program. Consequently it is often easier and less intrusive on the Java program to use the non-annotation style modifiers.

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

3.3.2 Type modifiers

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

However, Java 1.8 allows Java annotations that are designated as applying to uses of types to be applied to types wherever type names may appear. For example

```
(@NonNull String)toUpper(s)
```

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

The only annotations defined in JML that are type annotations are `@NonNull` and `@Nullable`. Those are presented in the following section (§??).

3.4 Possibly null and non-null type annotations

With Java 8, Java annotations are permitted on types, not just on declarations. With this feature it is possible to implement non-null reference types within Java. Other tools, such as the Checker framework [?], have done so. Accordingly JML has adopted nullable and non-null annotations as type annotations as well. Here *nullable* means a reference is possibly null and *non-null* means it is never null.

3.4.1 Syntax

JML defines `@NonNull` and `@Nullable` in the package `org.jmlspecs.annotation` as Java *type annotations*. That is, these annotations may be applied to any use of a reference type. Equivalently `/*@ non_null */` and `/*@ nullable */` may be used with the same semantics in the same program text locations.

Generally speaking, type annotations are placed immediately prior to the unqualified type name that they modify. For example `@NonNull Object` denotes a type whose values are references to objects of class `Object`, but which are never `null`.

For details on using type annotations with generics and type variables, see the discussion in the JLS [?] and the more understandable explanations [?] and in the Checker framework [?], in the context of type inference and checking.

There are two syntactical complications to be aware of:

3.4.1.1 Qualified type names

In the text

```
@NonNull A.@Nullable B,
```

where `B` is a nested class of `A`, the `@NonNull` applies to `A` and the `@Nullable` applies to `B`. In JML one could also write

```
/*@ non_null*/ A./*@ nullable*/ B.
```

The complication is that `A` might be a package name; the nullness type annotations may not be applied to packages. Thus for example, one might write

```
org.lang.@NonNull Object
```

to mean the same as `@NonNull Object`, but `@NonNull java.lang.Object` is illegal syntax. However, for convenience, JML still allows

```
/*@ nullable*/ java.lang.Object
```

3.4.1.2 Array types

With an array declaration such as `Object[][] a`, one might want to declare that `a` itself is non-null, or that the elements of `a` are non-null, or that the elements of the elements of `a` are non-null. The syntax for these cases is

- `@NonNull Object[][]` – an array of arrays of non-null `Object`s
- `Object[] @NonNull []` – an array of non-null arrays of `Object`s
- `Object @NonNull [][]` - a non-null array of arrays of `Object`s

The non-backwards-compatible aspect here is that prior to type annotations, `@NonNull Object[] array;` would be a declaration of a non-null array; now it is a declaration of an array of non-null objects.

3.4.1.3 var

Type annotations may not be applied to declarations using type inference, that is `var`.

3.4.2 Defaults

JML defines that references are non-null by default. That default can be changed locally using the modifiers `non_null_by_default` and `nullable_by_default` (or `@NonNullByDefault` and `@NullableByDefault`). These modifiers may be applied to class (including interface, enum and record) declarations. Their effect is to set the default to non-null or nullable for all relevant declarations or type uses within the respective class declaration, unless overridden by another such modifier on an enclosed declaration.

Any given declaration may have at most one of these modifiers.

Tools may also provide capabilities (such as command-line or environment options) to set the global nullness default.

Methods used in a program specified with JML that do not have source code available and do not have any explicit JML specifications have slightly different defaults (cf. §??). To ensure soundness, a method's formal parameters of reference type are assumed to be non-null, but the return value, if of reference type, is assumed to be nullable.

3.4.3 Java and JML language features with type annotations

In the following

- an implicit nullness declaration is the default determined by the inner most enclosing method or class declaration that has one of the modifiers described in §??.
- an explicit nullness declaration is the presence of one of the type annotations `non_null`, `nullable`, `@NonNull`, `@Nullable` (possibly fully-qualified). Explicit annotations override any implicit defaults. At most one of these annotations may be explicitly applied to any given type name or declaration.

3.4.3.1 Field and ghost or model field declarations

The type of Java or JML field declaration is *non-null* if it is annotated as non-null or it is not annotated as nullable and the implicit default is non-null.

If the type is non-null, then the variable must be initialized with a non-null value and any assignments to the variable must assign non-null values.

3.4.3.2 Local and ghost local declarations

The type of Java or JML local declaration is *non-null* if it is annotated as non-null or it is not annotated as nullable and the implicit default is non-null.

If the type is non-null, then the variable must be initialized with a non-null value and any assignments to the variable must assign non-null values.

It is possible that in the future the nullity of the type of local and ghost local variables will be inferred, rather than set by default.

3.4.3.3 Method and model method declarations

- Formal parameters behave like local declarations: explicit or implicit nullness modifiers or annotations determine whether the formal parameter is permitted to be null or not.
- The return type may also have a type modifier that determines whether the return value is permitted to be null or not. The method return type is the one place that the type annotation may be placed with all the other modifiers. That is, one may write

  ```
  @NonNull public java.lang.Object m1()
  ```

  in addition to

  ```
  public java.lang.@NonNull Object m1()
  ```

  , and similarly using `/*@ non_null */`. *Careful - what about array types* .
- Types in the `throws` clause of a method declaration are permitted to have type annotations, but any nullness annotations are ignored: any thrown exception is always non-null.

3.4.3.4 Class declarations

- the name of the class (or interface or enum or record) being declared may not be annotated
- type names in `extends` or `implements` or `permits` clauses may be annotated, but these annotations are ignored

3.4.3.5 Type names in `instanceof` expressions

- The implicit default does not apply to `instanceof` expressions.
- For any type `T`, the expression `o instanceof T` behaves as in Java: the result is false if `o` is `null`.
- For any type `T`, the expression `o instanceof @NonNull T` is false if `o` is `null`.
- For any type `T`, the expression `o instanceof @Nullable T` is **true** if `o` is `null`.

3.4.3.6 Type names in cast expressions

- The implicit default does not apply to cast expressions.
- For any type `T`, the expression `(T)o` behaves as in Java: `o` may be `null`, and if so, the result is `null`.
- For any type `T`, the expression `(@Nullable T)o` behaves as in Java: `o` may be `null`, and if so, the result is `null`.
- For any type `T`, the expression `(@NonNull T)o` returns a non-null result; it is a verification failure if `o` cannot be proven to be non-null.

3.4.3.7 Type names in new expressions

Type names in `new` expressions (e.g. `new Object()`) may have nullness type annotations. However, such type annotations are ignored; the result of a `new` expression is always non-null (or throws an exception).

3.4.3.8 Type names in array allocation expressions

An array allocation produces a value and is not a declaration per se. Each level of the array is null or not depending on the initialization value for the array. For example `new String[3]` produces a value that is a (non-null) array of 3 `String` values, each of which is null. So this value could be used in an initialization such as this: `@Nullable String @NonNull [] a = new String[3];`

The type name following `new` need not have any type annotations. If it does those annotations must agree with the implicit or explicit initialization. For example, `new Object[5]` and `new @Nullable Object[5]` are equivalent, as are `new Object[]{3}` and `new @NonNull Object[]{3}`. But `new @NonNull Object[5]` and `new @Nullable Object[]{3}` are type-checking violations. *Problem with tt font sizing and also braces*

3.4.3.9 Type names in catch statements

Type names in `catch` statements (e.g. `catch (RuntimeException e)`) may have nullness type annotations. However, such type annotations are ignored; if the exception is of a type that the catch block is selected for execution, the value of the declared variable (e.g. `e`) is always non-null.

The above applies to multi-catch blocks as well, as in

```
catch (RuntimeException | AssertionError e)
```

3.4.3.10 For statements

The declaration in a `for` statement, if present, acts like a local declaration: the explicit or implicit (default) nullness modifiers and annotations apply. If the loop variable is declared non-null, then both the initialization and update expressions must be provably not null.

3.4.3.11 Enhanced for statements

In a `foreach` statement, the declared variable may be declared using a type that has type annotations. On each iteration, the variable must be initialized with a value (from the array or iterator) that has an appropriate type. For example, for an array `a` in `for (@NonNull String s : a)`, the elements of `a` must each be `@NonNull`.

3.4.3.12 Resource declarations in try statements

A declaration within the resource definition of a `try` statement is just like other local declarations: if the declaration type is explicitly or implicitly non-null, then the initialization expression must be provably non-null and the value of the declared identifier may be assumed to be non-null; if the variable declared non-null is assigned within the body of the try statement, the value assigned must be provably non-null.

3.4.3.13 Declarations in JML `old` method specification clauses

A declaration within a JML `old` clause (in a method specification) is just like other local declarations: if the declaration type is explicitly or implicitly non-null, then the initialization expression must be provably non-null and the value of the declared identifier may be assumed to be non-null.

3.4.3.14 Declarations in JML `\let` expressions

The declaration within a JML `\let` expression is just like other local declarations: if the declaration type is explicitly or implicitly non-null, then the initialization expression must be provably non-null and the value of the declared identifier may be assumed to be non-null. If the type is `@NonNull` but the initializer is not provably non-null, the expression is not well-defined.

3.4.3.15 Declarations in JML generalized quantified expressions

Local variable declarations in `\forall` and other quantified expressions behave like other local declarations: any explicit or implicit nullness modifiers or annotations apply.

For quantified expressions, if the variable is declared non-null, then the range specifically excludes the null value. Thus

```
\forall @NonNull Object o; range; value
```

is equivalent to

```
\forall @Nullable Object o; o == null || ( range ); value
```

and

```
\exists @NonNull Object o; range; value
```

is equivalent to

```
\exists @Nullable Object o; o != null && ( range ); value
```

3.4.3.16 Type names in JML `signals` clauses

Type annotations on type names in the exception declaration in a `signals` clause are parsed but ignored. The value of the exception variable within the expression of a `signals` clause is always non-null.

3.4.3.17 Type names in JML `signals_only` clauses

Type annotations on type names in `signals_only` clauses are parsed but ignored.

3.4.4 Generic types and type annotations

TODO

3.4.5 Interplay with other non-null annotations

There are other tools that also define the annotations `@NonNull` and `@Nullable`, albeit in different packages than JML's `org.jmlspecs.annotation`. A long list of such alternatives is given in the Checker project manual.[1] Note that the names and semantics of some of these alternate annotations can slightly differ from each other.

The semantics of JML's nullness annotations matches those of the Checker framework and of nearly all other nullness annotations. The one exception, as documented in detail by the Checker project[2] is the annotations in SpotBugs/FindBugs.

However, the defaults for non-null types are different between JML and other systems.

Tools implementing JML may also interpret specific other annotations as equivalent to JML's annotations.

[1] https://checkerframework.org/manual/#nullness-related-work
[2] https://checkerframework.org/manual/#findbugs-nullable

3.5 Model and Ghost

Declarations in JML are one of two types.

- Declarations that correspond a Java source declaration or the Java source declaration that produces a binary class, method or field. In a `.java` file this is just the Java declaration itself; in a `.jml` file it is a replication of a Java declaration. These declarations are not in JML annotations.
- Declarations that do not correspond to Java declarations. These are always contained within JML annotations. They are also always marked with either of the modifiers `model` or `ghost`. These declarations are only visible within JML specifications.

JML fields can be either `model` or `ghost`. `ghost` fields are just like Java fields, but only visible within JML annotations. The value of a `ghost` field is directly determined by its initialization or by a set-statement (§??). Runtime tools will insert these fields into the compiled, instrumented classes.

`model` fields do not hold values. Rather, a model field should be thought of as the abstraction of one or more non-model (i.e., Java or concrete) fields [?]. (By contrast, some authors refer to what JML calls model fields as *abstract fields* [?].) The value of a model field is determined by the concrete fields it abstracts from; in JML this relationship is specified by a `represents` clause (see §??). Thus the values of the model fields in an object determine its *abstract value* [?]. A model field also defines a data group [?], which collects model and concrete fields and is used to tell JML what concrete fields may be assigned by various methods (cf. §??).

Model methods are simply methods that we imagine that the program has, to help in a specification. They may have bodies, in which case a runtime-instrumentation tool will compile them into an executable program. But they may not have bodies, in which case they serve as *uninterpreted* methods. Whether there is a method body or not, the behavior of a method as seen from a client (caller) of the method is solely determined by its specification.

Model types are less commonly used, but are types written in JML annotations and used only by JML annotations. They may serve, for example, as an abstract data type useful in specifying other aspects of the program. A model type will be compiled into a runtime-instrumented program in so far as its contents are executable.

JML also has a model import directive, described in §??.

Although these model and ghost names are used only for specifications, JML uses the same namespace for such names as for normal Java names. Thus, one cannot declare a field to be both a model (or ghost) field and a normal Java field in the same class. Similarly, a method is either a model method or not. Furthermore, JML has no syntactic distinction between Java and JML field access or method calls.

3.6 Model methods and lemmas

In mathematical proofs, lemmas are useful intermediate steps that aid in creating and in understanding the resulting proof. Similarly in program proofs, lemmas can be useful.

Lemmas in JML are stated using model methods. For example,

```
1 //@ public normal_behavior
2 //@   ensures \forall int p; p >= 0; (p&1) ==  p%2;
3 //@ no_state
4 //@ model public void lemma1() {}
```

states a general equality between a bit-vector operation and an arithmetic operation. This lemma has an empty body. Because it has a body, albeit empty, a tool is obligated to try to prove the lemma.

A second way to state the same property is the following:

```
1 //@ public normal_behavior
2 //@   requires p >= 0;
3 //@   ensures (p&1) ==  p%2;
4 //@ no_state
5 //@ model public void lemma2(int p) {}
```

Again, having a body is an indication that the lemma should be proved. The body need not be empty. It may have a sequence of assertions or case splits that guide a proof tool to a proof of the lemma.

This second form does not have an explicit quantification; consequently it can be easier for the proof tool to prove and to apply, as it does not need to be concerned with appropriate triggers to indicate instantiating the quantification.

The lemma is used in this example.

```
1 //@ requires 0 <= k <= Integer.MAX_VALUE/2
2 public void m(int k) {
3     k = 2*k;
4     //@ set lemma2( (k+1) );
5     //@ assert ((k+1)&1) == 1;
6 }
```

The set command allows a general method call, in this case the lemma.

This particular example is simple enough that it may not need a lemma. However, it illustrates two points.

First, using a lemma with a formal parameter and then applying the lemma with an argument from the expression needing to be proved is a convenient form of manual instantiation. If we used `lemma1` in the above example, the prover would essentially see

```
1 assume \forall int p; p >= 0; (p&1) ==  p%2;
2 assert ((k+1)&1) == 1;
```

and it would need to figure out that it should substitute `(k+1)` for `p`. But using `lemma2`, the prover has the easier task:

```
1 assert \let int p = k+1; p >= 0;
2 assume \let int p = k+1; (p&1) ==  p%2;
3 assert ((k+1)&1) == 1;
```

Second, this particular example illustrates a larger point. Programs that are a mix of arithmetic and bit-vector operations can be difficult for current SMT solver technology. Using lemmas to separate the proof problems helps the solver. The same is true for any lemma that might be difficult to prove – it may be simpler in isolation.

3.7 Visibility

Java code that is not within a JML annotation uses the usual access control rules for determining visibility (or accessibility) of Java [**?**, **?**]. That is, a name declared in package P and type P.T may be referenced from outside P only if it is declared as public, or if it is declared as protected and the reference occurs within a subclass of P.T. This name may be referenced from within P but outside of P.T only if it is declared as public, default access, or protected. Such a name may always be referenced from within P.T, even if it is declared as private. See the Java language specification [**?**] for details on visibility rules applied to nested and inner classes.

Within annotations, JML imposes some extra rules in addition to the usual Java visibility rules [**?**, **?**]. These rules depend not just on the declaration of the name but also on the visibility level of the context that is referring to the name in question. For purposes of this section, the annotation context of a reference to a name is the smallest grammatical unit with an attached (or implicit) visibility. For example, this annotation context could be a method specification case, an invariant, a history constraint, or a field declaration. The visibility level of such an annotation context can be public, protected, private, or default (package) visibility. JML has two rules governing visibility that differ from Java. The first is that an annotation context cannot refer to names that are more hidden than the context's own visibility. That is, for a reference to a name `x` to be legal, the visibility of the annotation context that contains the reference to `x` must be at least as permissive as the declaration of `x` itself. The reason for this restriction is that the people who are allowed to see the annotation should be able to see each of the names used in that annotation [**?**], otherwise they might not understand it. For example, public clients should be able to see all the declarations of names in publicly visible annotations, hence public annotations should not contain protected, default access, or private names.

In more detail, suppose `x` is a name declared in package `P` and type `P.T`.

- An expression in a public annotation context (e.g., in a public method specification case) can refer to `x` only if `x` is declared as `public` (or `spec_public`).

- An expression in a protected annotation context (e.g., in a protected method specification) can refer to `x` only if `x` is declared as public or protected, and `x` must also be visible according to Java's rules (so if `x` is `protected`, or `spec_protected`, then the reference must either be from within `P` or, if it is from outside `P`, then the reference must occur in a subclass of `P.T`).
- An expression in a default (package) visibility annotation context (e.g., in a default visibility method specification) can refer to `x` only if `x` is declared as public, protected, or with default visibility, and `x` must also be visible according to Java's rules, So if `x` has default visibility, then the reference must be from within `P`.
- An expression in a private visibility annotation context (e.g., in a private method specification) can refer to `x` only if `x` is visible according to Java's rules (so if `x` has private visibility, then the reference must be from within `P.T`).

In the following example, the comments on the right show which uses of the various privacy level names are legal and illegal. Similar examples could be given for method specifications, history constraints, and so on.

```
1  public class PrivacyDemoLegalAndIllegal {
2      public int pub;
3      protected int prot;
4      int def;
5  }
6  private int priv;
7  //@ public invariant pub > 0;      // legal
8  //@ public invariant prot > 0;     // illegal!
9  //@ public invariant def > 0;      // illegal!
10 //@ public invariant priv < 0;     // illegal!
11
12 //@ protected invariant prot > 1;  // legal
13 //@ protected invariant def > 1;   // illegal!
14 //@ protected invariant priv < 1;  // illegal!
15 //@ invariant def > 1;             // legal
16 //@ invariant priv < 1;            // illegal!
17 //@ private invariant priv < 1;    // legal
```

Note that in a lightweight method specification (one without any variation of `behavior` keyword), the privacy level is assumed to be the same privacy level as the method itself. That is, a protected method with a lightweight method specification is considered to be a protected annotation context for purposes of checking proper visibility usage [?, ?]. See §?? for more about the various specification cases. [3]

The JML keywords `spec_public` and `spec_protected` provide a way to make a declaration that has different visibilities for Java and JML. For example, the following declaration declares an integer field that Java regards as private but JML regards as public.

[3] The ESC/Java2 system has the same visibility rules as described above. However, this was not true of the old version of ESC/Java [?].

```
private /*@ spec_public @*/ int length;
```

`length` in the above declaration could be used in a public method specification or invariant.

However, `spec_public` is more than just a way to change the visibility of a name for specification purposes. When applied to fields it can be considered to be shorthand for the declaration of a model field with a similar name. That is, the declaration of `length` above can be thought of as equivalent to the following declarations, together with a rewrite of the Java code to use `_length` instead of `length` (where we assume `_length` is fresh, i.e., not used elsewhere in the class).

```
1 //@ public model int length;
2 private int _length; //@ in length;
3 //@ private represents length = _length;
```

The above desugaring allows one to change the underlying field without affecting the readers of the specification. The desugaring of `spec_protected` is the same as for `spec_public`, except that one uses `protected` instead of `public` in the desugared form.

The second rule for visibility prohibits an annotation context from writing specifications in an annotation context that constrain fields that are visible to more clients than the specifications (see section 3 of [**?**]). In particular, this applies to invariants and history constraints. For example, a private invariant cannot mention a public field, since clients could see the public field without seeing the invariant, and thus would not know when they might violate the private invariant by assigning to the public field. Accordingly, the invariants in the following example are all illegal, since they constrain fields that are more visible than the invariant itself.

```
1  public class PrivacyDemoIllegal {
2     public int pub;
3     protected int prot;
4     int def;
5     private int priv;
6     //@ protected invariant pub > 1; // illegal!
7     //@ invariant pub > 1;           // illegal!
8     //@ invariant prot > 1;          // illegal!
9     //@ private invariant pub > 1;   // illegal!
10    //@ private invariant prot > 1;  // illegal!
11    //@ private invariant def > 1;   // illegal!
12 }
```

Do visibility modifiers work for classes, and do they work in RAC?

3.8 Static and Instance

In Java

- declared names are non-static unless explicitly declared `static`

- except for fields of interfaces, which are by default `static` (and `public`)

JML allows model fields within interfaces to be declared non-static using the JML modifier `instance`. This modifier may be used for fields within classes as well, but here it is not necessary because the default is already non-static.

3.9 Method side-effects and purity

In method specifications, pre- and postconditions garner the bulk of the attention, but it is actually frame conditions that pose the most difficulty. Frame conditions state what side-effects a method has, that is, what may have changed in the memory heap as a result of a method's action. A method without side-effects is called *pure*.

There are two reasons that pure methods are important. First, a pure method makes no change to its pre-state when it is called; this makes it easier to reason about than a method with side-effects. Second, methods without side-effects can be called within specifications, albeit with restrictions that are described in the remainder of this section.

There are four kinds of purity worth distinguishing.

3.9.1 `pure`

A `pure` method does not modify any memory locations of the pre-state. It may allocate new objects and set the fields of those new objects. In analyzing a programming language method, knowing it is `pure` simplifies reasoning because all pre-state values are the same in the post-state. However, `pure` methods may return newly allocated objects, so they are not necessarily deterministic: $p() == p()$ is not necessarily true for a pure method. Consequently, `pure` methods of this kind may be used in specifications only under two additional constraints:

- the constructor that allocates the object is itself `pure` – that is, it only sets the fields of the new object and does not change anything in the pre-state;
- if the method returns a *fresh* value then the result of the method may not be used in ways that are heap-exposing, namely
  - the result value may not be used in a object equality comparison (either == or !=)
  - the result value may not be used to call `hashCode` or `identityHashCode`.[4]

  Here, *fresh* means that the object has been allocated since the pre-state; in practice the restriction above applies unless the return value has a Java or JML primitive type or it must be possible to prove that the value is non-*fresh*.

[4] Technically, calls of `hashCode` could be permitted if they never (recursively) call `identityHashCode`, but this is difficult to determine.

3.9.2 spec_pure

A spec_pure method has a further restriction over pure methods: it must be deterministic. Thus it must return either a primitive (non-heap-based) value or the value returned must be a value taken from the pre-state heap, that is the returned value is *not fresh*. However, it is allowed to allocate and discard objects internally. Consequently, a spec_pure method may change the results of any subsequent heap-exposing operations, that is, any operations that may depend on the location of objects in memory, such as identityHashCode.

spec_pure methods may be used in specifications, subject to the well-definedness rules described in §??.

3.9.3 strictly_pure

A strictly_pure method makes no changes to the heap at all. It does not assign to any locations on the heap and it does not even have access to change any stack memory. The method does not allocate new objects internally, even ones that will be discarded. Nor does it return any newly allocated objects. The visible post-state of a strictly_pure method is precisely the same as the pre-state, and the method result is entirely deterministic, so long as the method does not read any non-deterministic data sources (such as the system clock).

It is not possible to determine (statically) the difference between a strictly_pure method and a spec_pure method without inspecting the implementation of the method. Thus the more conservative choice for a library method is spec_pure.

3.9.4 no_state

A no_state method is a strictly_pure method that also does not even read heap values, including this, nor the external environment. That is, its execution is totally independent of the heap and the environment. The value returned by such a method is completely a function of its arguments and is completely heap (state)-independent.

Although a no_state method is not required to be declared static, it is effectively static. Clear style suggests that such a method be explicitly declared static.

3.9.5 Well-definedness

One purpose of the purity modifiers is to mark those Java methods that may be used in a specification. For this purpose one other restriction is needed: the use of the method must be well-defined as defined by this restriction:

- in the context in which it is used in a specification, a spec_pure method must be provably normally-terminating. That is, it must terminate and must not terminate with an exception.

3.9.6 Manipulation of the heap

Part of the goal of `strictly_pure` methods is to know that the heap is untouched. In actuality, the JVM can run garbage collecting and can move objects at any time, effectively concurrently with any computation of a deterministic method. So the heap may well change independently of the actions of user code. Such system actions are visible by resource measurements (e.g., whether or not out-of-Memory errors occur) that are outside the bounds of JML. Changes in the heap may also be visible by differences in the result of `Objects.identityHashCode(o)` for a newly allocated `java.lang.Object` o, because that hash may depend on what other objects have been allocated or even on the memory address of the object. For this reason two executions of the same program may result in visibly different behavior of `strictly_pure` methods if the values of hashcodes are used in the program.

There is one other point to note. If the specification language can reason about non-reachable objects then differences will be visible between `strictly_pure` methods, which do not allocate any objects, and equivalent `spec_pure` methods that allocate but discard objects. The visible heap will be the same, but, say, a count of all allocations that have happened so far will be different. JML has not yet incorporated such features or settled on the meaning of unconstrained quantification over objects, as in `(\forall Object o; true; ...)`.

3.9.7 Purity summary

In conclusion, JML is changing from its traditional use of only `pure` to a hierarchy of four kinds of purity as follows:

- JML defines all four method modifiers `pure`, `spec_pure`, `strictly_pure` and `no_state` as described above.
- methods used in specifications must be either `spec_pure`, `strictly_pure` or `no_state`; methods marked `pure` may be used subject to proofs about freshness or restrictions on uses of the new object reference
- any uses of methods in proofs must be well-defined (§??).
- as before, a method inherits its purity level from a method it overrides (if any); it may strengthen but may not weaken that purity level
- as before, a method that does not inherit or is not marked with its own purity level is considered to have a given purity if it is contained (perhaps recursively) in a class that is marked with that purity

The proof obligations are these:

- methods of any purity are required to establish that no assignments are made to any pre-state memory location
- `no_state` methods are required to establish, by syntactic inspection, that no allocations or reads of the heap occur in the execution of the method

- `strictly_pure` methods are required to establish, by syntactic inspection, that no allocations occur in the execution of the method
- `spec_pure` methods are required to establish that the return value is a primitive type or was already allocated in the pre-state
- `pure` methods used in specifications are required to prove that the restrictions stated in §?? hold
- any use of a method in a specification is required to establish, in that context, that the method always terminates normally.

3.10 Program state and memory locations

In imperative programming languages, such as Java, actions of a program during execution act on a *program state*. In actual operation, the state of a program is stored in a computer's memory, with each action reading and writing various hardware memory locations. We can talk about the state of a program at each point of execution and about the states before (the *pre-state*) and after (the *post-state*) an action or series of actions. The state consists of a set of *memory locations* or, abstractly, just *locations*. These locations are either heap locations or stack locations. The program state can grow and shrink as the stack grows and shrinks and as new heap objects are allocated or become no longer reachable.

In Java memory, locations hold either primitive memory values or object references. Object references refer to objects that each have a set of defined *fields* or array elements, which are also memory locations. At any program execution point, the program state consists of (a) the `this` object, (b) locations on the stack (local variables), including formal parameters of the method being executed, (c) any field of a class (static fields), and (d) any field or array element, recursively, of a location in the program state.

In reasoning about the actions of a program, it is important to know, for each action, what locations it affects. In particular, it is very helpful to know that everything but some small set of locations is unaffected by a particular action.

For this purpose, JML has two important concepts: *storeref expressions* and *location sets*. Location sets describe sets of memory locations. JML has a first-class type for reasoning about locations sets, namely `\locset`, along with operations on values of that type, such as union and intersection; this type and its operations are described in §??.

Storeref expressions (storerefs for short), also described in §??, are a way to syntactically designate particular values of type `\locset`, that is particular location sets. For example, `this.a[*]` indicates the set of all array elements of the array referred to by the reference in the field `a` of the `this` object in the current scope.

Storerefs and location sets are used in *frame conditions*, which are JML's means to state properties of program actions and to reason about program state.

3.11 Location sets, Data groups and Dynamic Frames

To be written - see section in DRM on Data Groups

3.12 Determinism of method calls

Methods may be underspecified. An extreme case is a postcondition that is simply `true`:

```
1 //@ ensures true;
2 int theInt();
```

Such methods are allowed to return any value consistent with the type of the result and the postcondition — in this case, any `int` value at all.

A question then is, must two successive invocations of such a method from the same initial state yield the same result, or not. In some cases, such as a method that returns a different random value on each invocation, the answer would be no. But in most cases determinism is expected by the user.

It is possible to force determinism by using a ghost field, as in this example:

```
1 class A {
2   //@ spec_public
3   private int _theInt;
4
5   //@ assigns \nothing;
6   //@ ensures \result == _theInt;
7   public int theInt();
8 }
```

Now `theInt()` is specified to produce the same (unknown) value until a method call or assignment occurs that might assign to `_theInt`.

However, as nearly all methods are expected to be deterministic, it is inconvenient, extra boiler-plate to require such a specification and to only require an indication of non-determinism. Accordingly, JML presumes that

method invocations of the same method with the same arguments in the same program state produce the same result.

The following sections describe several different use cases related to determinism.

3.12.1 Pure methods

Pure methods do not change the state and `spec_pure` methods may be called within specifications. In this example, all the `assert` statements can be proved true, though the concrete value of `theInt()` is not known:

```
1  abstract class A {
2    //@ spec_pure
3    abstract public int theInt();
4
5    public void test(A a) {
6      int x = theInt();
7      //@ assert theInt() == theInt();
8      //@ assert x == theInt();
9      int y = theInt();
10     //@ assert x == y;
11     //@ assert a == this ==> x == a.theInt();
12   }
13 }
```

If `theInt()` is only `pure`, all the assertions still hold because the method is implicitly `spec_pure`, by the rules of §??. If `theInt()` returns a newly allocated object, `theInt()` is considered to have changed the program state, although all pre-state locations remain unchanged, and then the assertions cannot be proved

3.12.2 Effectively pure methods

Effectively pure methods are methods that do not change the state, but are not declared pure. These methods may not be called within specifications, but nevertheless are deterministic. Again, the asserts in the following example are all provable.

```
1  abstract class A {
2    //@ assigns \nothing;
3    abstract public int theInt();
4
5    public void test(A a) {
6      int x = theInt();
7      int y = theInt();
8      //@ assert x == y;
9      int z = a.theInt();
10     //@ assert a == this ==> x == z;
11   }
12 }
```

3.12.3 State-changing methods

A method that changes the program state (e.g., by assigning to some field) is not able to be called twice in the same program state. In the following example, a call of `nextInt()` changes the program state; thus `x` is not necessarily equal to `y`. In fact, as the effective frame condition is `assigns \everything;`, very little is provable at all.

```
1  abstract class A {
2    abstract public int nextInt();
3
```

```
4   public void test() {
5     int x = nextInt();
6     int y = nextInt();
7     //@ assert x == y; // NOT PROVABLE
8   }
9 }
```

3.12.4 Methods with reads clauses

Even though some state change operation has occurred, a method may not depend on those changes. The only way to know about a limited dependency is by a specification saying so. This is the purpose of a `reads` (or `accessible`) clause in a method specification — it states which memory locations the method at hand depends on.

Consider this example:

```
1 public class A {
2
3   public int myint;
4   public int count;
5
6   //@ reads myint;
7   //@ spec_pure
8   public int getInt() {
9     return myint;
10  }
11
12  //@ code_java_math
13  public void m() {
14    int i = getInt();
15    count++;
16    int j = getInt();
17    count++;
18    //@ assert i == getInt();
19    //@ assert i == j;
20  }
21 }
```

The three uses of `getInt` occur in three different states. Also, `getInt` is underspecified, so we do not actually know what it returns. But `getInt`'s `reads` clause states that it depends only on `myint` and so the result is not changed by the changes to `count`. So these assertions are provable. Without the `reads` clause they are not provable. Without the `reads` clause but with a postcondition like

```
ensures \result == myint;
```

the assertions are provable because the limited dependencies are now implicit in the fully specified postcondition.

3.12.5 Intentionally volatile methods

Java does not permit methods to be marked `volatile`, but the previous examples point to how such a method might be specified.

```
1  abstract class A {
2    //@ spec_public
3    private int _theInt;
4    //@ assigns _theInt; reads _theInt;
5    abstract public int randomInt();
6
7    public void test1() {
8      int x = randomInt();
9      int y = randomInt();
10     //@ assert x == y; // NOT PROVABLE
11   }
12   public void test2() {
13     int x = randomInt();
14     int y = randomInt();
15     //@ assert x != y; // NOT PROVABLE EITHER
16   }
17 }
```

The most conservative assumption about a method is that it is non-deterministic and to presume otherwise is not sound. However, the default specification of a method also includes `assignable \everything`. Thus any invocation of such a method is specified to modify the heap, and consequently the results of successive invocations of a method with default specifications are unrelated anyway. Any method specified to be `assignable \nothing` does not change the state and must be deterministic or it is specified incorrectly, as it must depend on some property outside the program.

3.13 Invariants

Invariants about the state of a system and of individual objects within a body of software are important to careful design, maintaining understanding, and correctly modifying a software system. Specification languages support this design paradigm by providing syntax to write invariants expressing properties expected to be true of classes and object instances.

However, it has been surprisingly difficult to express sound, usable semantics for invariants in an object-oriented system. A local invariant for a given object instance is easy to conceptualize. But thorny problems arise when there are reentrant callbacks, mutual references between different objects, and use of mutable data fields of one object in another object's invariant.

3.13.1 Kinds of invariants

It is conceptually useful to take an aside to differentiate two kinds of invariants: *strong* and *weak*.

Strong invariants are those which always hold; there is no program point at which they do not hold. The prototypical example is type constraints, such as that the value of a given memory location is never null, or that an integer value falls within a given range. Every time such a memory location is to be modified, it can be checked at that time that any new value satisfies the invariant constraint. And any use of the value of that memory location can be assured that the invariant holds.

In contrast, weak invariants may be allowed to be broken temporarily en route to being restored in a new program state. Now it is important to know when the invariant can be trusted.

Strong invariants are possible, and unproblematic, because they involve just one memory location. Weak invariants typically involve the relationship between multiple memory locations. In the absence of programming language mechanisms for simultaneous, concurrent update, such invariants must be temporarily broken en route to being restored.

3.13.2 Traditional JML

The semantics of invariants in the first version of JML were that all invariants of all objects in the system had to hold at each boundary between a calling routine and a called routine (both on entrance and exit from the called routine), whether the exit was normal or exceptional.

This in principle allows any routine to be assured that any invariant on which it relies does indeed hold at the begionning of execution of that routine's body. Within the body of a method an invariant can be broken while the code step-wise moves to a new state in which the invariants once again hold. However, this rule is impractical to enforce in actual tools.

First, although it solves the callback problem, it also makes the use of simple utility routines difficult. Even to use a library routine to do some mathematical computation requires reestablishing any invariants that are broken during the current work-in-progress.

Second, the possibility of non-local references permits situations in which an invariant of class A refers to fields in class B and other clients of B can change B's fields, breaking A's invariant, without even knowing that A exists. It is impractical to require that all invariants be reestablished without having modular mechanisms to compute which invariants need attention.

Third, one cannot actually recheck all of the invariants in a software system on each method call and return. In practice some heuristics are needed to determine which invariants might be at risk and check that those are valid. This is not an approach that ensures soundness.

3.13.3 Invariants in JML 2.0

Consequently, this version of JML is revising the semantics of invariants, based on our own experience with JML over the past 25 years and other work on invariants during that time. In particular, recent work [**?**], carefully analyzing invariants identifies the same problems and treats them with careful, sound rigor.

Our implementation in JML is constrained by needing to provide a specification language for an existing programming language, rather than designing facilities in a programming language designed for verification, such as Eiffel and Dafny. Nevertheless, we follow that work where possible.

The current approach to invariants has the following components. Note that this approach is under current implementation and research.

3.13.3.1 Local specification of needed invariants

First, rather than a global rule that all invariants must hold, each method is responsible to know and state which invariants must hold for the method to do its work soundly. In general, this list of invariants is stated in a new `invariants` (§**??**) clause that is part of a method's specification. However, most methods are served by an easy default clause: the method only requires that the static and instance invariants of the receiver (if the method is not static) and of each actual parameter to the method must hold.

This default serves for most[5] cases. In particular, it works for most library routines. Routines that may perform callbacks to objects already in the call stack will require explicit annotation.

One could argue that even this default is not needed in some cases. For example, consider a List capability, in which a client can insert and remove object references from the list. There is no need for the objects in the list to satisfy their own invariants: the list only manipulates the reference pointer (presuming there is no need to compute a Java-like equality or do sorting). However, allowing such behavior would require distinguishing lists containing invariant-satisfying objects from those with not-necessarily-invariant-satisfying objects, so that when a reference is extracted from the list, one knows whether it is self-consistent or not (as measured by its invariant). To make this distinction would require some kind of type annotation, akin to distinguishing lists holding possibly-null references and those holding only non-null references). We have not yet determined there is sufficient need for such a facility.

3.13.3.2 Managing non-local and mutual references

TODO

[5] the utility of this default is currently a topic of research

3.14 Preconditions, Exceptions and untrusted callers

3.14.1 Specifying failing preconditions

A common specification situation is having a method that has some restriction on the states in which it may be called, that is, it has a precondition. In the body of the method, the precondition is defensively checked and some error action taken if the precondition is false, such as throwing an exception. This leads to a method specification with two specification cases, as in this sketch of an example:

```
1 //@ public normal_behavior
2 //@   requires i >= 0;
3 //@   ...
4 //@ also public exceptional_behavior
5 //@   requires i < 0;
6 //@   signals_only IllegalArgumentException;
7 public void m(int i) {
8   if (i < 0) throw new IllegalArgumentException();
9   ...
10 }
```

Here, the first specification case states what happens if the precondition holds; the second case states that if the precondition is not true, then a specific exception is thrown.

This pattern is verbose and has a maintenance risk in that the two preconditions of the two specification cases have to be kept as negations of each other.

The pattern is so common that JML defines some syntactic sugar: the `requires-else` clause. This clause (cf. §??) contains the usual precondition and gives an exception type to be thrown if the precondition does not hold. The following illustrates how the clause is desugared:

```
1 /*@
2    <clauses-before>
3    requires P else E;
4    <clauses-after>
5 */
```

is desugared as

```
1 /*@
2    <clauses-before>
3    requires P;
4    <clauses-after>
5  also <visibility> exceptional_behavior
6    <clauses-before>
7    requires !P;
8    assignable \nothing;
```

```
9     signals_only E;
10 */
```

The clause list comprising the specification case is presumed to be normally ordered (§??) with all precondition clauses first.

A specification case may have multiple requires-else clauses, mixed with nested case sequences (cf. §??); the desugaring above is carried out for each requires-else clause and each nested case.

The desugaring laid out above may not always match the user's intent. For example, the exceptional behavior may in fact assign to some memory locations. If the desugaring is not a correct specification for the method, then the correct combination of specification cases will need to be written out, without benefit of the requires-else feature.

3.14.2 Untrusted callers

The desugaring of the requires-else clause may not match the intent of the programmer: the combination of the two specification cases states that it is perfectly fine to call the method whether the precondition holds or not. A common intent is actually that (a) the method should always be called in states in which the normal behavior precondition holds, but (b) just in case it is not, there is an error path to cleanly report the error.

To accomplish the intent (a), the method should just have the normal behavior specification case. Then any caller (or tool) that checks the precondition will be alerted if the caller does not satisfy the callee's precondition. However now the body of the program is verified under the sole precondition of the normal specification case. Consequently the error path code is dead code and not checked. This is unacceptable, especially since error paths in software, being less well tested, have a higher rate of bugs.

What we need is a specification case for the error path that is not seen as legitimate behavior by the caller but is used by the callee in verifying the method's body. It is a specification case solely to accommodate *unverified callers* and so is particularly important for library specifications.

The solution adopted by JML is that proposed in [?]. It consists of a variation on requires-else, named recommends-else.

Like requires-else, the recommends-else clause (cf. §??) is desugared into a two specification cases, one for the path where the precondition holds, and one where it does not. But there are these differences:

- All the recommends-else clauses of a specification case are processed together, producing one case where all the precondition predicates hold and one where at least one predicate does not.

- The recommends-else clauses are processed before any requires or old clauses (cf. §??).
- The exceptional behavior case produced from recommends-else clauses must be implemented by the callee, but is not an alternative for the caller.
- There is no ordering to the recommends clauses. Each precondition predicate must be well-defined on its own.

For example, given

```
/*@
   public normal_behavior
   recommends P else E1;
   recommends Q else E2;
   <other-clauses>
*/
```

the callee implementing this specification sees

```
/*@
   public normal_behavior
   requires P & Q;
   <other-clauses>
 also public exceptional_behavior
   requires !(P & Q);
   assignable \nothing;
   signals (E1) !P;
   signals (E2) !Q;
   signals_only E1, E2;
*/
```

but the caller only sees the first of these specification cases. Thus the caller must establish $P \wedge Q$ before calling the method.

3.15 Arithmetic modes

JML defines various *arithmetic modes*, separately for integer arithmetic and floating point arithmetic. These modes allow one to use mathematical numbers (integers and reals) or their machine representations (fixed-bit-width integers and IEEE floating point) in specifications and to enable or disable warnings about out of range computations.

The features for arithmetic modes are described in $??.

3.16 Redundant specifications

JML has some features that enable expressing redundant specification. In particular, there are keywords with a `..._redundantly` suffix, such as `requires` and

`requires_redundantly`. In each case the redundant predicate is expected to be provable from the other predicates for the corresponding root keyword.

Similarly, method specifications may have `implies_that` and `for_example` sections that express logical statements that are expected to be provable from the primary part of the method specification.

Redundant specifications are useful as alternate expressions of the primary specifications. They can serve as lemmas for the benefit of the reasoning engine or as restatements that make the import of the specification clearer to a human reader.

`_redundantly` suffixes are not widely used or implemented. They may be a candidate for deprecation if no good use cases are found.

3.17 Naming of JML constructs

Most JML constructs can be optionally named. The name is a Java identifier that is placed just after the keyword for the construct and is followed by a colon. For example

```
requires positive: i > 0;
```

and

```
public normal_behavior usual_case: .
```

These names are currently only for external reference. They have no type or scope; they are in a different namespace than any other Java or JML identifier. As they may be used by tools to distinguish various predicates, names should be unique for each method in the union of the names in the method and the names of predicates in the class (such as invariants). Because they are Java identifiers, they may not be Java keywords. Tools may use them in error messages or in tool directives as the tool sees fit.

In grammar productions, this optional name, with its colon, is indicated by *<opt-name>*.

Although currently these clause and specification case names have no meaning within JML, there are ideas for that to change.

- The name of a specification would have boolean type and be in scope in the body of the method and in any textually later specification cases (but not its own spec case). Its value would be the value of the conjoined precondition for that case, that is, true in those initial program states that the specification case applies, because its effective precondition is true.
- The name of a clause would be an identifier representing the value of that clause if that is meaningful, in the program state in which the identifier is used. So boolean for requires, ensures, invariant, assert etc. clauses, `\locset` for assignable clauses etc.
- In some cases it is useful to be able to refer, in a method body, to identifiers declared in `old` clauses in the method specification. For this purpose, using the name of a specification case as a state label in `\old` would be useful.

3.18 Specification inference

If no specifications are present for some program entity, JML presumes some defaults (§??). Alternatively, one could *infer* specifications based on the source code itself, on the uses of a particular method elsewhere in the overall program, or even based on external documentation. Inference of specifications would be very useful in reducing the amount of specification text a user would have to write. However, as specification inference is very much an area of research and JML does not want to presume any specific inference capability, the JML language defines specific defaults without presuming any inference.

Tools supporting JML may in fact implement useful inferences, saving the writing and reading of "obvious" specifications. We recommend that such inference be clearly identified, that there be options to enable and disable inference, and the inferred specifications be presented to the user for review and for possible explicit inclusion in the source code.

In addition, some caution is in order. If specifications are inferred based on the source code, the inferred specifications can presumably be verified with respect to that source code. That does not mean that that mutually consistent combination is correct when compared to some external requirements or the intent of the software. Thus inferred specifications should be reviewed by humans as well as being verified against the implementation.

3.19 org.jmlspecs.lang and org.jmlspecs.runtime

The `org.jmlspecs.lang` and `org.jmlspecs.runtime` packages are reserved for use by tools or future JML features.

3.20 Evaluation and well-formedness of JML expressions

JML text may be *syntactically incorrect*. Such errors are typically caught by the parser. Syntactically correct text may be *type-incorrect*. Such errors are typically caught during the name and type attribution phase of the compiler.

JML expressions must also be *well-defined*, that is have a logical meaning. For example, for integer values `i` and `j`, the expression `i/j` is well-defined only if it can be proven that `j` is never zero.

This means that predicates in JML are either true or false or not-well-defined — a three-value system. In JML it is considered a verification error if an expression cannot

be proven to be well-defined; it is a runtime error if an expression is not well-defined for a particular runtime execution of a program containing JML expressions.

The details of well-definedness are presented in §??.

3.21 Core JML

There is a tension in a language design project meant for several purposes: research, practical, and educational use. Language design research tends to add an assortment of experimental features; practical applications demand a robust but substantial and stable set of language capabilities; tool developers want a stable, smallish, easily maintainable but adequate feature set; and, educational use needs a small core that can be put to use in examples. In addition, sophisticated features may be needed to specify system libraries, which in turn are needed for educational use.

To help guide tool development, the features of JML are grouped into various categories, with *Core* features as the most basic. This categorization of JML features is enumerated in the table in Appendix **??**.

Chapter 4

JML Syntax

4.1 JML vs. Java syntax

In designing a syntax for a specification language, there is a tension between using bespoke syntax that immediately identifies specification constructs or to use Java syntax as much as possible. The former is the style used historically by JML and is typically more succinct; the latter makes it easier for Java parsing tools to be adapted to parsing JML.

A few examples of constructs will illustrate this point.

- JML defines a type `\bigint`, representing unbounded mathematical integers (§??); these could also be represented as a Java class, perhaps `org.jmlspecs.lang.internal.bigint`.
- JML defines arithmetic operations such as addition on `\bigint`s, by extending the regular Java operators, such as `a + b` (§??); in pure Java, one might write `a.add(b)`.
- JML defines a construct, an *informal comment* (§??), using the syntax `(* ... *)`, where the ellipsis is natural language text; a Java equivalent is the method `org.jmlspecs.lang.JML.informal(String s)`.

Most but not all JML constructs could be represented in pure Java; in fact the conversion between the two can sometimes be done prior to parsing by simple textual search and substitution.

An example that cannot be represented in pure Java is the JML `\result` expression (§??), which represents the value returned from a non-void method. Because the type of the expression depends on the type of method it is used within, there is no simple purely Java method to replace it; something like `JML.result()` would still need special attention in type attribution..

Any Java equivalents to JML constructs are not (yet) part os the JML language defini-

tion. However, this document indicates suggested Java equivalents when presenting JML syntax, so that tools can use a common set of Java constructs, should the tool choose to support them.

4.2 Textual form of JML specifications

Specifications in JML for a Java program are written either as specially formatted comments within the Java source text, described in this section, or in standalone `.jml` files, as described in §??. The `.jml` files are quite similar to `.java` files, just in a separate file.

4.2.1 Java lexical structure

The lexical structure of Java source text (typically, but not necessarily contained in files in the local file system) is described in the chapter on Lexical Structure of the JLS [?](Ch. 3).

Java source text is written in unicode using the UTF-16 encoding. It is permissible to represent unicode characters with *unicode escapes*, which use only ASCII characters and have the form `\u`*xxxx*. The source text is translated into a sequence of (Java) tokens using the following steps:

- The source text is converted to (unicode) character sequence lines, by abstracting the line ending characters used on various platforms into single line terminator tokens.
- Then, beginning at the beginning of the character sequence and continuing with the next token immediately after identifying the previous token, the character sequence is iteratively divided into Java tokens, which are
  - reserved words
  - identifiers
  - literals
  - operators
  - separators (i.e., punctuation)
  - white space
  - comments
  - line terminators
- For each token, character sequences are tokenized into the longest valid token, whether or not that token can be parsed as part of a legal Java program. Thus white space is needed to separate identifiers, which would otherwise be tokenized as a single longer identifier; similarly `--` is parsed as a single operator rather than two – operators, even if `--` cannot form a legal Java program whereas two – operators might. The one exception is that consecutive > characters, which by the longest token rule would be tokenized as >> or >>> shift operators, but in the context of closing generic type arguments are separated into separate > tokens, as in `List<List<Object>>`.

This tokenizing is inclusive enough that almost any sequence of characters can be translated to a sequence of Java tokens. The only errors in this process are from illegal characters such as `#`, `` ` ``, illegal escape sequences, illegal unicode characters, ill-formed floating-point literals, and un-closed string literals and comments.

The Java lexical analyzer then discards white space tokens, comment tokens, and line terminators to form the token sequence that is the input to the Java parser.

4.2.2 JML annotations within Java source

JML adjusts the above process in one small way. Java comments (by the rules of Java) are (by the rules of JML) identified as either *JML annotation comments* or as *plain Java comments*. The latter are discarded by both Java and JML. The former are still discarded by a Java parser (because they are Java comments), but retained by JML tools.

The *JML annotation text* is the content of a JML annotation comment without the beginning and ending comment markers, as defined below. The JML annotation text is tokenized into a sequence of JML tokens located at the position of the comment token in the Java token sequence.

Because JML annotation comments are Java comments, they do not affect the interpretation of Java source as seen by Java tools. It is an important rule that

a JML tool must semantically interpret the Java portion of Java source that includes JML annotation comments in precisely the same way as defined by the Java Language Specification, that is, as a Java compiler would.

A complementary rule is that

No text outside of a Java comment may be considered as part of JML annotation text.

Two examples demonstrate a bit of the intricacies. The text (as one complete text line)

```
/*@ ghost String s = "asd*/";*/
```

consists of a Java comment that is a JML annotation comment, namely

```
/*@ ghost String s = "asd*/
```

followed by four tokens, namely a quote, a semicolon, a star and a slash. Thus the JML annotation text is just `ghost String s = "asd`, which ends in an unclosed string literal. On first glance one might think that the JML annotation text should be

```
ghost String s = "asd*/";,
```

which would be a legitimate JML declaration, but that reading does not agree with the first rule above, which requires that the JML annotation comment end with the first occurrence of `*/`.

A second example is

```
1 public
2 //@ invariant a != null;
3 void mm() {}
```

Here a Java compiler would interpret `public` as a modifier of the method declaration that follows the comment. Consequently a JML tool may not interpret the `public` modifier as belonging to the invariant. To do so would violate the rule that the JML token sequence may only consist of tokens derived from text within JML annotations. In fact, in this case, the JML annotation text would be illegal because it is placed within a Java method declaration.

4.2.3 JML annotations

JML annotation comments are specially formatted Java comments. The determination of whether a Java comment is a JML annotation comment is made in the context of a globally-defined set of *keys*, each of which are Java identifier tokens; the keys are defined independent of the source text itself. JML tools may provide mechanisms to declare the set of keys defined for a particular invocation of the tool.

- A Java comment that begins with text matching the regular expression
  `/[/|*]([+|-]<java-identifier>)*@+`
  is a JML annotation comment if
  - (a) there are no *<java-identifier>* tokens (that is, the comment begins with either `//@` or `/*@` followed by zero or more `@` characters
  - or (b) (i) if there are any identifiers (in the regular expression above) preceded by a + sign, then at least one of them must be a key, and (ii) if there are any identifiers (in the regular expression above) preceded by a – sign, then none of them must be a key.
- Anything not matching the above regular expression or not meeting the rules on keys is not a JML annotation comment; it is a plain Java comment.
- Note that the permitted regular expression allows no white space.

Also note this terminology:

- JML annotation comments meeting condition (a) above are *unconditional JML annotation comments*.
- JML annotation comments meeting condition (b) above are *conditional JML annotation comments*, as they depend on the set of keys.
- JML annotation comments that are within Java line comments are *JML line annotation comments*.
- JML annotation comments that are within Java block comments are *JML block annotation comments*.

4.2.4 Unconditional JML annotations

By the definitions above, unconditional JML annotation comments either

- (a) begin with the characters `//@` and extend through the next line terminator or end-of-input, or
- (b) begin with the characters `/*@` and extend through the next occurrence of the characters `*/`, possibly spanning multiple lines.

Examples of unconditional JML annotation comments are

```
1 //@ requires a == b;
2
3 /*@@@ requires true;
4       ensures a == b;
5   @@@*/
6 }
```

4.2.5 Conditional JML annotation comments

If the identifiers `RAC` and `OPENJML` are declared as keys but `DEBUG` is not, then these are conditional JML annotation comments:

```
1 //+RAC@ requires true;
2 //+RAC-DEBUG@ requires true;
3 /*+OPENJML@@@ requires true; @@@*/
4 //-DEBUG@ requires true;
```

In lines 1 and 3, there is a key occurring with a + sign; in line 2, there is a key occurring with a + sign and there are no keys with a – sign; in line 4 there are no positive identifiers and the one negative identifier is not a key.

These are plain Java comments:

```
1 //-RAC@ requires true;
2 //+OPENJML-RAC@ requires true;
3 //+DEBUG@ requires true;
4 //+RAC @ requires true;
```

In lines 1 and 2, there is a key in the comment opening marker that has a – sign, so these are not JML annotation comments, despite the presence of a key with a + sign in line 2; in line 3 the identifier in the comment opening marker is not a key; and line 4 is a plain Java comment because of the white space between the `//` and the `@`.

4.2.6 Default keys

Tools should by default declare these identifiers as keys:

- `DEBUG` — not declared by default, but reserved
- `ESC` — by default, declared when static checking (deductive verification) is being performed by a tool, otherwise not

- `RAC` — by default, declared when runtime assertion checking is being performed by a tool, otherwise not
- `OPENJML` — reserved for use by the OpenJML tool and presumed to be defined when that tool is used and otherwise not
- `KEY` and `KeY`— reserved for use by the KeY tool and presumed to be defined when that tool is used and otherwise not

Other identifiers may be reserved for other tools. Keys are case-sensitive, but tools may relax that rule, so different identifiers used as keys should not intentionally be the same when compared case-insensitively. The tool-specific keys are intended to be used to include or exclude JML annotation text that contains tool-specific extensions or tool-specific unimplemented JML features, respectively.

4.2.7 Tokenizing JML annotations

The *JML annotation text* is obtained from a JML annotation comment by

- removing the opening comment marker as defined in §**??**
- removing the closing comment marker which is either the line terminator for a line comment or the characters `[@]*[*][/]` for a block comment (that is, the usual `*/` comment ending marker plus any number of consecutive preceding `@` characters

The JML annotation text resulting from the above is then tokenized in the same way as Java source text is tokenized, with the following additions:

- character sequences matching `[\]`<*java-identifier*> are valid identifiers in JML annotation text. Examples are `\result` and `\type` (in current practice, all such identifiers are all alphabetic after the backslash). These are defined as <*jml-identifier*>s.
- JML defines additional operators:
  `..` `==>` `<==>` `<=!=>` `<:` `<:=` `<#` `<#=`
- An integer literal followed by a period followed by a period followed by an integer literal (e.g., `1..2`) should, by the longest token rule, be tokenized as two floating-point literals (`1.` and `.2` in the above). JML however alters the rule in this case to tokenize such a character sequence as an integer literal, the JML `..` token, and an integer literal (as in `1 .. 2`).
- JML defines some additional two-character separators: `{|` and `|}`.
- JML defines an additional white space token: within a block annotation comment, the character sequence `[ \t]*[@]+` (that is, optional white space followed by one or more consecutive `@` characters) immediately following a line terminator is a white space token.

After being tokenized, any white space, plain Java comments, and line terminators are discarded; the result is the token string comprising the JML annotation.

For example, in

```
/*@@@@@@@@@@@@@@@@@@@@@@@@@@
  @@  requires x > 0;
  @@  ensures \result < 0;
  @@@@@@@@@@@@@@@@@@@@@@@@@@*/
```

none of the `@` characters is part of the JML annotation token string (after dropping white space tokens). But in this example

```
/*@@@@@@@@@@@@@@@@@@@@@@@@@@
  @   requires x > 0 @;   @  // invalid @ in and after text
  @ @ ensures \result < 0;   // second @ is invalid
  @@@@@@@@@@@@@@@@@@@@@@@@@@*/
```

the end-of-line comments identify some `@` tokens that are invalid.

4.2.8 Embedded comments in JML annotations

Because the text of Java comments is not tokenized, Java does not have embedded comments. JML, however, does tokenize the text of a JML annotation and that text may contain embedded Java comments. Those embedded Java comments are treated just like non-embedded Java comments: a determination is made as to whether the Java comment is a JML annotation comment; if so, the JML annotation text is tokenized and those tokens become part of the token stream of the enclosing JML annotation. This process can happen recursively.

Here are some pairs of example JML annotation text and corresponding JML token sequences (omitting white space, line terminator, and comment tokens)

- `//@ requires // comment`
  identifier token (`requires`)
- `//@ requires /* comment */ true;`
  identifier (`requires`), literal (`true`), semicolon
- `//@ requires /*@ true */ ;`
  identifier (`requires`), literal (`true`), semicolon
- `//@ requires //@ true ;`
  identifier (`requires`), literal (`true`), semicolon
- If the identifier `RAC` is a declared key
  `//@ requires //-RAC@ true ;`
  identifier (`requires`)
- If the identifier `RAC` is a declared key
  `//@ requires //+RAC@ true ;`
  identifier (`requires`), literal (`true`), semicolon

- If the identifier `RAC` is not a declared key
  `//@ requires //+RAC@ true ;`
  identifier (`requires`)

Note though that block comments embedded in line comments must begin and end within that line comment. Also block comments cannot be embedded in other block comments because the first `*/` will end the outer block comment, leaving the inner comment unclosed.

Overuse of embedded comments results in difficult to read text and poor style. The two principal use cases are these:

- adding plain Java comments inline, as in

```
1 /*@
2   @ requires true; // precondition
3   @ writes a; // frame condition
4   @ ensures a > 0; // postcondition
5   @*/
```

- conditionally discarding portions of a JML annotation for a particular situation, such as, commonly, to exclude non-executable JML features during runtime assertion checking:

```
1 /*@
2   @ requires true; // precondition
3   @ //-RAC@ writes a; // frame condition,
4   @                   // ignored during RAC
5   @ ensures a > 0; // postcondition
6   @*/
```

  A similar case is to include or exclude annotations particular to a given tool.

4.2.9 Compound JML annotation token sequences

A consecutive sequence of JML annotation comments in the source text is combined into a single JML annotation token sequence by concatenating the token strings from the individual JML annotation comments. The JML annotation comments in the sequence must be separated only by discarded Java tokens (white space, line terminators and plain Java comments). Note in particular that it is the *token strings* that are concatenated, not the text. Thus any token, such as a string literal or a Java text block, must still be contained within one JML annotation comment.

A common use case for this language feature is to write JML text such as

```
1 //@ requires a
2 //@        && b
3 //@        && c;
```

where `a`, `b`, and `c` are stand-ins for potentially long expressions that are best broken across lines. A block annotation comment could also be used here.

The JML annotation comments in the sequence may be any mix of line or block comments.

Obsolete syntax JML previously allowed JML text within Javadoc comments. This is no longer permitted or supported.

Issues with the JML textual format There are a few issues that can arise with the syntactical design of JML.

First, JML annotation token sequences are the concatenation of token sequences from individual JML annotation comments. These annotation comments may be separated by large blocks of discarded Java tokens, such as a large jmldoc comment. An error, say in terminating an expression, in an earlier JML annotation may not be recognized by the parser until a later annotation, leading to the parser issuing an error message quite far removed, textually, from where the correction is needed.

Second, other tools may also use the `@` symbol to designate comments that are special to that tool. If JML tools are trying to process files with such comments, the tools will interpret the comments as JML annotations, likely causing a myriad of parsing errors.

Third, Java uses the `@` sign to designate Java annotation interfaces. That in itself is not an ambiguity, but sometimes users will comment out such annotations with a simple preceding `//`, as in

```
//@MyAnnotation
```

This construction now looks like JML. The solution is to be sure there is whitespace between the `//` and the `@` when a Java comment is intended, but it may not always be possible for the user to perform such edits. Tools may provide other options or mechanisms to distinguish JML from other similar uses.

4.2.10 Java text blocks

Java 15 introduced *text blocks*, allowing writing multi-line String literals. Here is an example of a three-line String literal:

```
String s = """
    line 1
      line 2 with two spaces indentation
    line 3 with some quotes "very cool"
    """;
```

The rule for text blocks in Java is that the String literal consists of the lines between the opening and closing `"""`, but removing any common prefix of those included lines and the closing line that consists entirely of spaces and tabs, and removing any trailing spaces and tabs on each line. The prefixed sequence of spaces and tabs must be precisely equivalent (space for space, tab for tab), which can lead to subtle errors in code that has undisciplined mix of spaces and tabs. This removing of common pre-

fixes allows the String literal to be indented along with code, without the indentation becoming part of the String.

Now consider such a text block within JML comments. For JML block comments, it is common to prefix every line in a block of code with `@`, as in

```
/*@ ghost String s = """
  @    line 1
  @      line 2 with two spaces indentation
  @    line 3 with some quotes "very cool"
  @    """;
  @*/
```

Thus for JML block comments, JML removes any leading white space followed by one or more consecutive `@` characters prior to sending the text to Java to remove its indentation.

Similarly, using JML line comments, one might write

```
 //@ ghost String s = """
 //@    line 1
 //@      line 2 with two spaces indentation
 //@    line 3 with some quotes "very cool"
 //@    """;
 //@
```

Again, JML removes any leading white space followed by `//` followed by one-or-more consecutive `@` characters, and then passes the resulting text on to Java.

4.2.11 Terminating semicolons

Many JML clauses and statements end with a semicolon; this practice follows the syntax of Java. Such semicolons are not typically essential to parsing a program, though they are helpful in error recovery and in reading the program. In JML such a semicolon is optional if it immediately precedes the end of the JML comment (i.e., just before the terminating `*/` or end-of-line after removing any Java comments) and there is no immediately following JML annotation. The semicolon is required if the statement is succeeded by another statement within the same JML comment or in an immediately following JML annotation.

For example, in

```
//@ ensures true
public void m1() {}

//@ requires true;
//@ ensures true
public void m2() {}
```

the semicolons that would terminate the `ensures` clauses are optional, but the one terminating the `requires` clause is still required. Remember (§??) that consecutive

JML annotation comments are concatenated, so that a single clause could be continued from one JML annotation comment to the next.

4.3 Locations of JML annotations

A JML annotation's token string must conform to the grammatical rules presented throughout this document. The *placement* of JML annotation comments is also subject to various rules.

JML annotations fall into the following categories, each of which is described in detail in cross-referenced sections, along with a grammar for both the JML annotation and the location of the JML annotation within the Java source:

- modifiers (§??) — single words, like the Java modifiers `public` and `final`; these are placed as part of the declaration they modify, mixed in with Java modifiers. Examples are `pure` and `nullable`.
- file-level specifications (§??) – these are placed with Java top-level declarations, such as `import` statements or model class declarations. Examples are model import statements.
- type specification clauses (§??) — these are placed where Java places members of types, such as field and method declarations. Examples are `invariant` clauses.
- method specifications (§??) — these are placed in conjunction with the declaration of a method's signature. They in turn consist of
  - keywords
  - punctuation
  - clauses
- field specifications (§??) — these are placed in conjunction with a field declaration. Examples are `in` and `maps` clauses.
- statement specifications (§??) — these are placed like statements in a code block (a method or initializer body). Examples are `assert` and loop specification statements.

Thus all JML annotations consist of single-word tokens (modifiers and keywords), punctuation (one or more sequential non-alphanumeric characters), and clauses, which themselves begin with keywords.

JML annotations that are not in a prescribed location are errors (which tools should report).

4.4 JML identifiers and keywords vs. Java reserved words

As described in the previous section, JML annotations include, among other things, identifiers that have special JML meaning, as modifiers and keywords. Any Java identifier that is in scope for a JML annotation can potentially be used within a JML clause; consequently we want to be sure that there are no name clashes. There are a few aspects of JML design that intend to avoid possible name clashes. Again, these are presented more formally in later chapters.

- Java reserved words may not (by Java's rules) be used in Java expressions or declared as names in Java. These reserved words are also reserved in JML and may not be declared as new JML names, nor are they used as JML keywords. JML keywords are not reserved.
- Specialized JML identifiers used in expressions begin with a backslash, so they cannot be confused with Java identifiers. Examples are `\result` and `\old`.
- JML operators and punctuation (composed of non-alphanumeric characters) are either the same as in Java (e.g., `+`) or something not in Java (e.g., `<==>`). As authors of Java programs cannot add new operators or punctuation, there is no possibility of name clashes. There is a possibility of a backwards-compatibility clash if the Java language adds new operators in the future, such as perhaps `==>`, that clash with existing JML operators.
- JML modifiers, keywords, and the initial keywords of clauses are all regular Java identifiers. All JML modifiers and clauses begin with such a keyword and so can be recognized by that keyword. Thus on parsing a JML annotation, the parser considers the first token found, which, if not an operator or punctuation, must be an identifier, which then is either a standalone word (e.g., a modifier) or is the beginning of a clause. Importantly, these keywords are not reserved words and they are different from all of Java's reserved words[1]; however JML keywords may be Java or JML identifiers declared as program names. For example, `requires` is a keyword beginning a method precondition, as in `requires i >= 0;`. But `requires` could also be an identifier declared say as either a Java or JML field name. Thus it is possible to have a precondition `requires requires >= 0;`. If it is Java that declares `requires`, such a construct might be unavoidable; if JML does so it should likely be considered poor style owing to difficult readability.
- JML also uses class names that fall into conventional Java naming conventions but are in packages reserved for JML use. Such packages begin with either `org.jmlspecs` or `org.openjml`. It is conceivable but unlikely that Java users might define their own packages and classes that use this same name, in which

[1] More precisely, the JML keywords are all different from any of Java's reserved words that might start a declaration, notably built-in type names. The Java reserved word `assert` is also a JML keyword, but `assert` at the beginning of a JML clause is unambiguously the start of a JML clause.

case there would be an irreconcilable name conflict. However, the Java library itself would not use package names beginning with `org`.

- Declarations of fields, methods, and classes within JML cannot declare the same names as corresponding Java declarations in the same scope. For example, a declaration of a JML ghost field in a Java class may not have the same name as a Java field declaration. Simply put, if Java does not permit adding such a declaration (because of a duplicate declaration), then JML may not introduce the declaration either.
- There is a situation that is unavoidable. A Java class `Parent` may contain a declaration of a JML name `n` that is appropriately distinct from any potentially conflicting name in `Parent`. However, unknown to the specifier of `Parent`, a class `Child` can later be derived from `Parent` and the (Java) author of `Child`, not knowing about the JML specifications of `Parent`, may declare a name `n` in `Child`. In such a case, with a local Java entity and an inherited JML entity having the same name, what does the name refer to? In Java code, the name refers of course to the (one) Java declaration. In JML code the ambiguity is resolved in favor of the Java name. In this case the JML entity could be referred to in JML code within `Child` as `super.n` or `((Parent)x).n`.

4.5 JML Lexical Grammar

In the following grammar, the lexical syntax is defined using regular expressions, using the standard symbols: parentheses for grouping, square brackets for a choice of one character, `?`, `*`, `+` for 0 or 1, 0 or more and 1 or more repetitions. An identifier within angle brackets and in italics is a lexical non-terminal; terminal characters are in bold; backslash is used to escape characters with special meaning, but no escape is needed within square brackets. White space is included only where specifically indicated. The references to JLS are to the Java Language Specification, specifically the chapter on lexical structure [?].

<compound-jml-comment> ::= *<simple-jml-comment>*+

<simple-jml-comment> ::=
  <jml-line-comment> | *<jml-block-comment>*

<jml-line-comment> ::=
  //*<jml-comment-marker>*
  <jml-annotation-text>
  <line-terminator>

<jml-block-comment ::=
  /* *<jml-comment-marker>*
  <jml-annotation-text-no-blocks>
  <jml-block-comment-end>

<jml-comment-marker> ::=
  ([**+**|**-**]*<java-identifier>*)***@**+

in which the Java identifiers must satisfy the rules about keys stated in §**??**.

<jml-block-comment-end> ::= **@*/**

<plain-java-comment> is defined in §3.7 of the JLS, but excludes any character sequence matching a *<compound-jml-comment>*

<java-identifier> is defined in §3.8 of the JLS (and excludes any *<reserved-word>*)

```
<jml-annotation-text-no-blocks> ::=
        ( <identifier>
      | <reserved-word>
      | <literal>
      | <operator>
      | <separator>
      | <java-white-space>
      | <jml-line-comment>
      | <plain-java-comment>
      | <jml-line-terminator>
) *

<jml-line-terminator> ::=       | <line-terminator>
      | <jml-white-space>

<jml-annotation-text> ::=
      | <jml-annotation-text-no-blocks>
      | <jml-block-comment>[2]

<identifier> ::= <java-identifier> | <jml-identifier>

<jml-identifier> ::= [\]<java-identifier>
```

Note that users cannot define new *<jml-identifier>*s and all *<jml-identifier>*s currently defined in JML are purely alphabetic and ASCII after the backslash.

<reserved-word> is defined in §3.9 of the JLS

<literal> is defined in §3.10 of the JLS

```
<operator> ::= <java-operator> | <jml-operator>
```

<java-operator> is defined in §3.12 of the JLS

```
<jml-operator> ::= .. | ==> | <==> | <=!=> | <: | <:= | <# | <#=

<separator> ::= <java-separator> | <jml-separator>
```

<java-separator> is defined in §3.11 of the JLS

```
<jml-separator> ::= {| | |}
```

<java-white-space> is defined in §3.6 of the JLS

```
<jml-white-space> ::=
       <line-terminator> <java-white-space>? [@]+
```

within a *<jml-block-comment>*

<line-terminator> is defined in §3.4 of the JLS

[2] This *<jml-block-comment>* may not include any line terminators.

4.6 org.jmlspecs.lang.JML

JML introduces a variety of non-Java syntax. To promote tool interoperability, syntactic replacements for these are also defined, to enable writing JML expressions with purely Java syntax, allowing a JML program to be parsed by non-JML aware tools. Although this substitution would enable a JML-annotated program to be parsed and type-checked, understanding the semantics of JML would still be necessary to make use of the JML features. Also the goal of a complete replacement by Java equivalents is not yet achieved, as noted in the following subsections.

The replacements are defined in the class `org.jmlspecs.lang.JML`. For example the function `JML.implies` can be used instead of the operator `==>`.

4.6.1 JML modifiers

Modifiers can be replaced by the Java @-interface equivalents (e.g., `pure` by `@Pure`).

4.6.2 JML expressions

- `JML.informal(s)` – `(* ... *)`, where `s` is a string literal holding the content of the informal comment
- `JML.equivalence` – `<==>`
- `JML.inequivalence` – `<=!=>`
- `JML.implies` – `==>`
- `JML.subtypelt` – `<:`
- `JML.subtypeleq` – `<:=`
- `JML.locklt` – `<#`
- `JML.lockleq` – `<#=`
- `JML.bsZZZ` – for any JML expression beginning with a backslash, e.g., `JML.bsresult` for `\result`.
- Quantification expressions take two arguments: a `\set` or `\seq` as the range and a lambda function as the value. For example, `\exists int i; 0 <= i < 10; f(i))` can be written as `JML.bsexists(JML.seq(0,10), i-> f(i))`

4.6.3 JML statements

JML statements are represented in Java as methods returning `void`.

- `JML.assert`
- `JML.assume`
- `JML.unreachable`

- `JML.loop_invariant`
- `JML.assigning`
- `JML.decreases`
- `JML.begin`
- `JML.end`

TODO: JML primitive types, \TYPE \locset, JML datatype, JML tuples, model imports, type and method clauses, model fields/methods/classes, JML datagroup, set comprehension, operator chaining

local ghost variables, local model classes, ghost labels, set statement, block specifications,

state labels, scope issues for \old \type

store-ref expressions, line annotations(forbid, allow, ignore)

4.7 Definitions of common grammar symbols

This section assembles the definitions of a number of grammar non-terminals that are used throughout the manual. See §?? for information about how the productions of the grammar are written.

The grammar uses the syntactic tokens of §?? and the ones listed below as non-terminals. All other lexical tokens are presented as literal terminals in the grammar productions.

- *<java-identifier>* — a character sequence allowed as an identifier by Java. Note that *<java-identifier>* excludes Java reserved words, some of which are context-dependent.
- *<jml-identifier>* — an identifier with its preceding backslash. Only a specific set of such identifiers are legal in JML.
- *<string-literal>* — a traditional ("-delimited) string or a text block ("""-delimited)
- *<character-literal>* — as in Java
- *<integer-literal>* — as defined in Java. Note that multi-digit integer literals beginning with a `0` are octal numbers, those beginning with `0x` or `0X` are hexadecimal, those beginning with `0b` or `0B` are binary, and that these literals may have a trailing `l` or `L` and may include underscore characters.
- *<fp-literal>* — as in Java

Common grammar symbols. (Reminder: `...` means a comma-separated list of zero or more items.)

A possibly qualified name is a sequence of dot-separated identifiers:
<qualified-name> : := *<java-identifier>* (. *<java-identifier>*) *

A type name possibly has one or more type parameters:
<type-name> : := *<qualified-name>* [< *<type-name>* (, *<type-name>*) * >]

A predicate is just an expression whose type is boolean:
<predicate> : := *<expression>*

An expression is either a specifically JML expression or a Java expression that permits JML sub-expressions.
<expression> : := *<jml-expression>* | *<java-jml-expression>*

A *<java-jml-expression>* is a Java-like expression that may have JML subexpressions.

An optional name for a clause or specification case:
<opt-name> : := [*<java-identifier>* :]

Chapter 5

JML Types

To abstractly model program structures in specifications, specifiers need basic numeric and collection types, along with the ability to combine these into user-defined structures. All of the Java class and interface type names and all Java primitive type names are legal and useful in JML: `int short long byte char boolean double float`. In addition, JML defines some specification-only types, described in subsections below. There are several needs that JML addresses:

- Specifications are sometimes best written using infinite-precision mathematical types, rather than the fixed bit-width types of Java. Indeed, users typically prefer to (and intuitively do) think in terms of mathematical integers and reals, to the point of missing overflow and underflow bugs. JML's arithmetic modes (§**??**) allow choosing among various numerical precisions.
- Java's handling of class types only expresses erased types; JML adds a type and operations for expressing and reasoning about generic types.

With respect to reference types, note the following:

- Java's reference types are heap-based and so creation of and operations on these types may have side-effects on the heap.
- Though pure (side-effect free) methods on Java classes can reasonably be used in specifications, the `Object.equals` method cannot be pure without significantly restricting the set of programs that can be modeled.
- Side-effect-free types for specification should have value semantics, but classes constructed using Java syntax will still have a distinction between == and `.equals`.

Thus, although Java types can be named in specifications, types used for modeling are much more convenient if they are pure, value-based, heap-independent types that do not use non-pure methods of Java classes.

This leads us to consider types built-in and predefined in JML. At the cost of extra learning on the part of users, such types can have more natural syntax and clearly be primitive value types. Also, built-in types can be naturally mapped to types in SMT provers that have theories for them (e.g. the new string theory in SMT-LIBv2.6 [**?**]).

Specifically, JML's builtin types have the following properties:

- the type name begins with a backslash (e.g., `\seq`)
- if a builtin type takes type parameters, those parameters may be Java reference types or JML builtin types, but not Java primitive types
- they have value semantics, like Java primitive types: values are immutable and all operations produce new values
- there is no distinction between == and `.equals`. Where equality of values is needed, == is used and is defined as structural equality. For example, two sets are equal (==) iff they have the same elements. `.equals` is not used on value types.
- in JML terms, all operations
  - are *pure*, i.e., they do not modify the program state (`assignable \nothing;`)
  - are independent of the heap (i.e., `no_state` and `reads \nothing`)
  - are terminating (`diverges false;`)
  - are nonnull — there are no null values of value-semantics types
  - are immutable, i.e. any visible fields are `final`
- for the most part, the operations on builtin types follow Java syntax, though where the semantics is obvious some infix operations are defined as well. (Using Java syntax reduces the amount of parser customization required to support JML, even if that means a slightly more verbose and less mathematical syntax.)

There is no type hierarchy that subsumes all Java and JML types. All Java reference types are instances of the Java `Object` type, but the Java primitive types and all JML types (which are also primitive) are mutually independent. There is implicit conversion among Java and JML numeric types, as is present in Java.

5.1 Java reference types

Java reference types may be used in specifications, both library classes and user-defined classes. However, an important restriction applies: all operations on values of such types must be pure (§**??**), that is, they must not have side-effects and must be declared to be one of the kinds of purity, most simply `spec_pure`.

Consequently, allocation of new objects is restricted (cf. §**??**) in specification expressions and no operations change the current state. A significant implication of this rule is that methods such as `toString` and `equals` cannot generally be used in specifications. These methods of `Object` may be overridden by methods in arbitrary derived classes, and they may be implemented with side-effects. Accordingly, they cannot be (and are not) declared `pure` in `java.lang.Object` without severely restricting the implementers of other classes. Reference equality, inequality and comparisons

against `null` are all permitted.

A library or user-defined class that is `final` and has side-effect-free implementations for methods like `equals` and `toString` may declare them as some kind of `pure` and use them in specifications.

Java classes designed with mathematical, value semantics (an immutable class with all pure methods) can be used to model the behavior of a Java program. The methods of such a class would be defined in their own specifications using techniques such as an algebraic specification. Although some modeling types of this nature were supplied with earlier versions of JML tools, these are being replaced by built-in, heap-independent primitive value types with simpler, clearer, more mathematical semantics, as described in later sections of this chapter.

5.1.1 Java enums

An exception to the discussion of the previous section is Java enum types. As enums are immutable types, enum values and built-in operations on enums can be used in specifications. Also, enum values are never freshly allocated, so == comparisons can be used.

- == and != — equality and inequality of enum values are permitted in specifications

In addition Java defines several built-in methods for enums. Each of these has some implicit specifications. For a given enum type `E` the following hold (in the examples, `E` is presumed to have the three values `A`, `B`, `C`:

- the class `E` is `final`; that is, it may not be the parent class of any other class, except that the compiler creates anonymous classes for enum values that are customized with their own behavior. The following is true if `E` has no such customization.

```
//@ axiom \forall \TYPE t;; t <: \type(E) ==> t == \type(E);
```

- the class `E` extends `Enum<E>`, which extends `Object`; class `E` may implement interfaces

```
//@ axiom \forall \TYPE t;; \type(E) <: t ==>
        ( t == \type(Enum<T>) || t == \type(Object)
        || t is a superinterface);
```

- the declared values of `E` are each non-null, are all distinct from each other, and are final

```
1 //@ axiom \distinct(null, A, B, C);
```

- extensionality — any value of type `E` is either `null` or is one of the declared constants

```
//@ axiom \forall nullable E e;;
                e == null || e == A || e == B || e == C;
```

- the static method `values()` returns an array (`E[]`) of all the enum constants of `E`, in the textual declaration order

```
//@ public normal_behavior
//@   ensures \result.length == 3; // number of constants
//@   ensures \result[0] == A;
//@   ensures \result[1] == B;
//@   ensures \result[2] == C;
//@ pure
public static final E{} values();
```

- the static method `valueOf(String)` returns either the constant of type `E` with the given name or an exception is thrown.

```
//@ public normal_behavior
//@   requires n != null && (* n is e.name() for some E e *);
//@   ensures \result.name().equals(n); // FIXME - TODO
//@ also public exceptional_behavior
//@   requires n == null || !(* n is e.name() for some E e *);
//@   signals (NullPointerException) n == null;
//@   signals (IllegalArgumentException) !(* n is e.name() for some E e *);
//@   signals_only NullPointerException, IllegalArgumentException;
public static final /*@ non_null */ E valueOf(/*@ nullable */ String n);
```

- instance methods `name()` and `toString()` both return the name of an enum constant as given in its declaration
- instance method `ordinal()` returns an `int` giving the 0-based position of the enum constant in textual declaration order
- instance method `compareTo(E e)` compares enum constants according to their `ordinal` value:

```
//@ public normal_behavior
//@   requires e != null;
//@   ensures \result < 0 <==> this.ordinal() < e.ordinal();
//@   ensures \result == 0 <==> this.ordinal() == e.ordinal();
//@   ensures \result == 0 <==> this == e;
//@   ensures \result > 0 <==> this.ordinal() > e.ordinal();
//@ also public exceptional_behavior
//@   requires e == null;
//@   signals_only NullPointerException;
//@ pure
public int compareTo(nullable E e);
```

Java restricts the Java modifiers that can be applied to enum declarations. JML modifiers are allowed, but may be unnecessary. For example, the arithmetic mode modifiers are only useful if the enum declaration declares methods that do arithmetic computation.

5.1.2 Java records

Is this description duplicated in the Default specs section

Java records, like enums, are a special form of class declaration. A typical use of a record declaration is to create a class with private, final (i.e., readonly) data fields, all initialized by a constructor. Since the record is a class with automatically generated code, each instance of a record declaration has its own default specifications.

A record class automatically generates

- private final fields for each declared component of the record
- corresponding getter functions for each record component (named the same as the field)
- a public constructor that initializes all of the fields
- equals(), toString(), and hashCode() methods

The programmer can also add other methods and constructors. The specifications for both the generated and user-provided members can be placed in a `.jml` file. If there are only generated members, the following defaults are automatically provided.

Here are the default specifications, illustrated by example for a record declaration `record R(int x, Object o)  .`

The default constructor:

```
1 //@ public normal_behavior
2 //@   ensures this.x == x;
3 //@   ensures this.o == o;
4 //@ pure
5 R(int x, Object o);
```

Getter methods: For each generated field/getter method

```
1 //@ public normal_behavior
2 //@   ensures \result == this.x;
3 //@ strictly_pure
4 public int x();
```

equals(Object o) The default `equals` method is the conjunction of calling `equals` on each of its fields (or just == for fields with primitive type). Since `equals` is not necessarily pure, no default specification is presumed. The generated `equals()` method only has the purity (if any) of the least pure of the `equals()` methods of its reference-type arguments. If all of the arguments have primitive type, the record's `equals` can be considered `strictly_pure`.

toString(): The default `toString()` method produces a system-defined representation of the record. No default specification is presumed. The generated `toString()` method only has the purity (if any) of the least pure of the `toString()` methods of

its reference-type arguments. If all of the arguments have primitive type, the record's `toString()` can be considered `pure`.

hashCode(): The default `hashCode()` method produces a system-defined hashCode of the values of the record's fields. No default specification is presumed. The generated `hashCode()` method only has the purity (if any) of the least pure of the `hashCode()` methods of its reference-type arguments. If all of the arguments have primitive type, the record's `hashCode()` can be considered `spec_pure`.

5.1.3 Lambda functions

To be written.

5.1.4 Java Streams

To be written.

5.2 `boolean` type

The Java `boolean` type may be used as is in JML, along with the usual Java operators:

- `==` and `!=`
- `!` (not)
- `&` and `|` (and, or)
- `&&` and `||` (short-circuiting and, or)
- Java ternary operation (`? :`)

In addition JML adds these operations:

- `<==>` and `<=!=>` (§??)
- `==>` (implies operation, §??)

JML `strictly_pure` specifications may use auto-unboxing from `Boolean` to `boolean`; however, auto-boxing is allowed only in `pure` methods because that may allocate a new heap object. The Java defined constants `Boolean.TRUE` and `Boolean.FALSE` may be used in specifications.

In Java programs, `&&` and `||` are very commonly used in preference to the boolean `&` and `|` because the left operand may be necessary to avoid a runtime error in evaluating the right operand and because it may provide some performance benefit. For deductive verification, the short-circuit operations are still useful for well-definedness, but they are not for performance. In fact the non-short-circuit operations are simpler to encode and reason about and so are preferred over short-circuit operations when well-definedness is not an issue.

5.3 Java primitive integer and character types

The Java primitive integer and character types may be used as is in JML, along with all of the Java operations on those types, including casting among them. Depending on the arithmetic mode (§??), range checks may be performed on the results of operations.

Auto-boxing is allowed only in `pure` specification expressions.

5.4 `\bigint`

The `\bigint` type is the set of mathematical integers (i.e., $\mathbb{Z}$). Just as Java primitive integral types are implicitly converted (see *numeric promotion* in the JLS, Ch. 5) to `int` or `long`, all Java primitive integral types implicitly convert to `\bigint` where needed.

Within JML specifications, the `\bigint` type is treated as a primitive type. For example, == with two `\bigint` operands expresses equality of the represented integers, not (Java) identity of any underlying objects.

The familiar operators are defined on values of the `\bigint` type:

- unary: `+ - ~`
- binary: `+ - * / %`
- bit operations: `& | ^`
- equality: `== !=`
- comparisons: `< <= > >=`
- shifts: `>> <<` (unsigned right shift, `>>>`, is not permitted on `\bigint` values)
- casting: to and from primitive Java integral and character types
- JML defines `\bigint.zero` and `\bigint.one` as static final values holding `\bigint` constants `0` and `1`.

Also, these types can be used in quantified expressions and variables of these types can be declared as ghost or model variables.

Casting to lower precision types results in truncation of higher-order bits; in *safe java* arithmetic mode (§??) this may cause a verification warning.

Shift operations in Java can be surprising as the number of bits shifted is the right-hand value modulo 32 or 64 (for `int` or `long` left hand values). Shifts of `\bigint` do not limit the number of bits shifted. Also, for `\bigint`s, all shifts are signed; there is no `>>>` operator. The shift operations act like the numbers are infinite sequences of bits, so `-3 >> 1` is `-1`.

Like the shift operations, the bit operations on `\bigint` values act as operations on infinite sequences of bits. For example, `(-1)^(-2) == 1`.

The JML modulo operation on `\bigint` has the same behavior as in Java (and C). It satisfies $x == y*(x/y)+(x\%y)$; accordingly, as integer division is truncation (round-

ing toward 0), the sign of $x\%y$ is the same as the sign of the dividend (of x in these equations). For example, $-3\%2 == -1$.[1]

`\bigint` is the preferred type for writing specifications about integral values, instead of range-limited Java integral types.

The Java-equivalent methods are these, for `\bigint` values `b` and `b`:

- `b.longValue()`, `b.intValue()`, `b.shortValue()`, `b.charValue()` and `b.byteValue()` are equivalent to the corresponding type casts to primitive Java integral types
- `b.bigValue()` returns a `java.math.BigInteger` value corresponding to the `\bigint` value
- `b.eq(bb)` and `b.ne(bb)` are equivalent to == and `!=`
- `b.equals(bb)` is the same as `b.eq(bb)`.
- `b.lt(bb)`, `b.gt(bb)`, `b.le(bb)`, `b.ge(bb)` are equivalent to `b<bb`, `b>bb`, `b<=bb` and `b>=bb`
- `b.compareTo(bb)` has behavior corresponding to `.compareTo` in Java.
- `b.add(bb)` and `b.subtract(bb)` are equivalent to `r+bb` and `b-bb`
- `b.multiply(bb)`, `b.divide(bb)`, and `b.mod(bb)` are equivalent to `b*bb`, `b/bb` and `b\%bb`
- `b.negate()` and `b.comp()` are equivalent to `-b` and `~b`.
- `b.and(bb)`, `b.or(bb)`, `b.xor(bb)` are equivalent to `b&bb`, `b|bb` and `b^bb`
- `b.shiftLeft(i)` and `b.shiftRight(i)` and equivalent to `b<<i` and `b>>i`
- `\bigint.empty()` returns a `0` value
- `b.toString()` returns a `java.lang.String` representation of the `\real` value.
- `b.hashCode()` and `b.equals(Object o)` are defined so that `\real` values may be used in hashed sets and maps, but should not be otherwise used in Java+JML code

---

Current prover technology can take a long time to prove results about long (even 32- or 64-) bit sequences) and has difficulty mixing bit-operations with equivalent integral operations. Caution is recommended in using bit and integral operations together.

5.5 Java `double` and `float` types

The Java `double` and `float` types may be used as is in JML, along with their Java operations. However, extreme caution is needed: the Java operations on floating point values correspond to the IEEE standard [?] and do not correspond to common intuition based on real numbers or logic.

Java floating point numbers include NaN (not-a-number) values, positive and negative infinity, positive and negative zero, along with the usual positive or negative

[1] This is the *truncation* alternative as presented in https://en.wikipedia.org/wiki/Modulo.

double- or float- precision values. For example, if either a or b is a NaN, then both `a == b` and `a != b` are false, so `a != b` is not the same as `!(a == b)`. Similarly all comparisons among NaN values are false. Also, although `0.0 == -0.0` is true, `Double.valueOf(0.0).equals(Double.valueOf(-0.0))` is false. In a specification expression, these operations have the same semantics as in Java.

To aid in working with floating-point numbers in specifications, JML defines the model methods `Double.same(double x, double y)` and `Float.same(float x, float y)`. These define a logical equality among floating point values. That is, they return true iff either both operands are NaN, both are positive infinity, both are negative infinity, both are positive zero, both are negative zero, or they represent the same non-zero, finite floating point number.

Unboxing `Double` and `Float` values is allowed in `strictly_pure` specifications. Autoboxing is allowed in `pure` specifications.

5.6 \real

The `\real` type is the set of mathematical real numbers (i.e., $\mathbb{R}$). Just as the Java primitive type `float` is implicitly converted to `double`, both `float` and `double` values (and `\bigint` and integral Java primitive values) implicitly convert to `\real` where needed. When JML specifications are compiled for runtime checking, `\real` values are represented in some tool-dependent approximation. Within JML specifications, however, the `\real` type is treated as a primitive type.

The familiar operators are defined on values of the `\real` type:

- unary: + −
- binary: + − * / \%
- equality: == !=
- comparisons: < <= > >= `compareTo`
- implicit and explicit casts: to and from primitive Java numeric and character types, and `\bigint`
- to a `String` representation: `r.toString()`
- construction from another value: `\real.of(i)`, for all primitive numeric types and `\bigint`
- conversion to another type: `r.intValue()` etc. for all primitive numeric types and `r.bigValue()` to convert (by truncation toward zero) to a `\bigint`

The modulo operation on `\real` values, like that on `\bigint`, float, double and Java integral values, is truncated division and modulo. That is x/y is the nearest integer, rounding toward zero, and $x\%y == x - y(x/y)$, so that the sign of $x\%y$ is the sign of x.

`\real` can be used in quantified expressions and variables of type `\real` can be declared as ghost or model variables. There are no bit-operations on `\real` values.

It is not well-defined to attempt to cast a NaN or infinity value of a float or double to a `\real`. Both positive and negative zero cast to the 0 value of `\real`. Casting from `\real` to `float` or `double` may produce infinity values, but not NaN.

The following methods may be called on `\real` values `r` and `rr`:

- `r.eq(rr)` and `r.ne(rr)` are equivalent to == and `!=`
- `r.equals(rr)` is the same as `r.eq(rr)`.
- `r.lt(rr)`, `r.gt(rr)`, `r.le(rr)`, `r.ge(rr)` are equivalent to `r<rr`, `r>rr`, `r<=rr` and `r>=rr`
- `r.compareTo(rr)` has behavior corresponding to `.compareTo` in Java.
- `r.add(rr)` and `r.subtract(rr)` are equivalent to `r+rr` and `r-rr`
- `r.multiply(rr)`, `r.divide(rr)`, and `r.mod(rr)` are equivalent to `r*rr`, `r/rr` and `r\%rr`
- `r.negate()` is equivalent to `-r`.
- `r.toString()` returns a `java.lang.String` representation of the `\real` value.
- `r.hashCode()` and `r.equals(Object o)` are defined so that `\real` values may be used in hashed sets and maps, but should not be otherwise used in Java+JML code

5.7 \TYPE

The JML type `\TYPE` represents the type of Java expressions. Thus Java types are a first-class type within JML; there are values for various Java types and one can write expressions and reason about Java types. Values of `\TYPE` represent full generic types, not erased types as in runtime Java. Thus `\type(java.util.List<Integer>)` and `\type(java.util.List<Boolean>)` are different values and `\type(java.util.List)` is not well-defined.

`\TYPE` values are never null and only represent Java types, not JML types like `\bigint` and `\seq<Object>`. They are intended to be used to reason about the relationships among Java types within a Java program.

The following table summarizes the operations available on a `\TYPE` value and compares them with corresponding operations on Java `Class` values. Details are given in the following subsections.

- `n` represents a type name, like `int` or `Integer` or `Integer[]` or `List<String>`
- `t`, `tt` represent `\TYPE` values
- `c`, `cc` represent `Class` values
- `o` is a value of any reference type

	JML	Java
Type	`\TYPE`	`Class`
type literal (§??)	`\type(n)`	`n.class`
generic type constructor (§??)	`\TYPE.of(c,t,...)`	
array type constructor (§??)	`\arraytype(t)`	`Array.newInstance(c,0).getClass()`
type equality (§??	`== !=`	`== !=`
t subtype of tt (§??)	`t<:tt t<:=tt`	`cc.isAssignableFrom(c)`
dynamic type(§??)	`\typeof(o)`	`o.getClass()`
erased type (§??)	`\erasure(t)`	
type arguments (§??)	`t.typeargs()`	
is array type (§??)	`\isarray(t)`	`c.isArray()`
element type of array type (§??)	`\elemtype(t)`	`c.getComponentType()`
string representation	`t.toString()`	`c.toString()`

5.7.1 \TYPE literals

the `\type` syntax (§??) is the means to write literals of type `\TYPE`. The argument of `\type` is a (syntactic) type name. For example, these are all different values of type `\TYPE`:

- `\type(int)`
- `\type(\bigint)`
- `\type(Object)`
- `\type(Integer[]))`
- `\type(java.lang.Integer)`
- `\type(java.util.List<Integer>)` *Is this OK????*
- `\type(java.util.List<Boolean>)`
- `\type(T)`, where T is a type parameter

Non-literal parameterized `\TYPE` values can be constructed with the `of` method: `\TYPE.of(c,t,...)`, where c is a `Class` value (e.g., `List.class`) and the 0-or-more other arguments are `\TYPE` values that are the type arguments of the given class. The number of arguments must match the number of type arguments expected by the Java class.

5.7.2 \TYPE destructors: erasure and typeargs

`\TYPE` values are constructed from a Java `Class` value and type arguments. Correspondingly, a `\TYPE` value can be destructed into its components.

`\erasure(t)` gives the `java.lang.Class` value that is the erasure of `t`. For example `\erasure\type((java.util.List<Integer>)` equals `java.util.List.class`.

`t.typeargs()` for a `\TYPE` value `t` is a `\seq<\TYPE>` giving the `\TYPE` values of each of the type arguments of the parameterized type.

5.7.3 \TYPE equality

`\TYPE` values can be compared with == and `!=` as expected.

5.7.4 \TYPE subtype relationships

The <:= operator is the sub-type or equality operation and <: is the proper subtype relationship (subtype of, but not equal to). In JML V1, <: meant the improper comparison, but <:= is more naturally the improper subtype operation and <: the proper subtype operation.

The current definition is a backwards incompatible change.

5.7.5 \TYPE array values

`\TYPE` values can also represent array types.

- `\arraytype(t)` for a `\TYPE` value `t` returns a `\TYPE` value that represents a Java array of `t`.
- `\isarray(t)` (§??) returns `true` iff `t` is an array type.
  So `\isarray(\type(int[]))` is true, but `\isarray{\type(int))` is false.
- `\elemtype(t)` (§??) returns the element type of a `t` that is an array type. So then `\elemtype(\arrayOf(t))` is `t`.

5.7.6 Dynamic types

The `\typeof` function (§??) takes a JML expression and returns the `\TYPE` value corresponding to its *dynamic* type.

5.7.7 Type parameters

Within a class body in which a type variable, say `T`, is in scope, one can write `\type(T)`, whose value is the `\TYPE` with which `T` is instantiated. So in such a case, for example, the comparison `\type(T)== type(Integer)` is true only in the case that `T` is instantiated as an `Integer`. For example the asserted expression in the following example verifies as true.

```
1 class Value<T> {
2   public T value;
3
4   Value(T t) { value = t; }
5
6   void check() {
7     //@ assert \typeof(value) <: \type(T);
8   }
9 }
```

JML does not (yet) handle type parameters that extend types other than `Object`, nor `?` type parameters, nor `super` in type parameters, nor intersection type parameters.

5.8 \datagroup

A \datagroup is a pseudo-type that indicates that the name declared with this type is simply a static datagroup (§??). Such a declaration must always be model and must never have an initializer. Instead it is the target of in and maps clauses (§??). In addition, this type may only be used to declare model fields; it is not used for local declarations (within a method), formal parameters of methods, return types of methods, types in cast expressions, or as type arguments in generic types.

Here is an example of its use:

```
public class Demo {

  //@ model public \datagroup g;
  public int i; //@ in g;

  //@ assigns g;
  public void m() {
    i = 0;
  }
}
```

In one sense, this keyword is not needed because any model field is also a datagroup. So one could simply use an identifier declared Object. However, by using \datagroup, it is clear that the identifier is *only* a data group and does not have a value nor is it used in an expression. Datagroup identifiers may be used in frame clauses and \locset expressions like other implicit datagroups.

This type used to be called JMLDataGroup. That was the only type name in JML that did not begin with a backslash. That spelling is now removed and replaced by \datagroup.

It may be, when \locset is better defined, that a \datagroup is a kind of, and can be convcerted to, a \locset.

5.9 \locset

This section still needs work.

The \locset type is the type of *MU,DISCUSS: finite* sets of heap or stack locations. Stack locations are simple local variables. There are three kinds of heap locations:

1. static fields (ClassName.staticFieldName)
2. non-static fields in objects, considered as pairs (o,f) consisting of an object reference o and the name of a non-static field f.
3. reference to array indices (a,i) consisting of an object reference to an array object and a integer index into the array i.

Location sets are used in particular in `accessible` and `assignable` clauses.

In earlier versions of JML these clauses only took static lists of locations, but in order to reason about linked data structures, first-class expressions representing sets of locations are needed. *MU: Actually, with datagroups these were already dynamic, and also "o.f" could mean something different depending on the value of o. What is new is that are first-class cizizens and that they can be stored in entities.*

Syntactic designations of memory locations, also called *storerefs*, are described in §??). A location set can be constructed by

- `\locset()` - constructs an empty set
- `\locset(`*`<storeref>`* `...)` - constructs a set containing the designated locations
- *`obj`*`.*` - designates a location set that contains all fields (including inherited and private ones) of the given object *obj*
- *`ary`*`[*]` and *`ary`*`[`*`n`*`..`*`m`*`]` - designates a location set where all either all index positions of *ary* are included, or in the second case only the index position from *n* (inclusive) to *m* (inclusive).[2]
- `(\infinite_union` *`boundedvar`*`;` *`<guard>`*`,` *`<storeref>`*`)` - denotes an infinite union of the location sets, i.e.,

$$\bigcup_{boundedvar \wedge guard} storeref \tag{5.1}$$

These operations are associated with a `\locset`:

- `\union(`*`<expr>`* `...)` (also the binary operator `|`) - union of `\locset`s
- `\intersection(`*`<expr>`* `...)` (also the binary operator `&`) - intersection of `\locset`s
- `\disjoint(`*`<expr>`* `...)` - true iff the arguments are pair-wise disjoint
- `\subset(`*`<expr>`*`,` *`<expr>`*`)` (also the binary operators < and <=) - true iff the first expression evaluates to a (proper or improper) subset of the evaluation of the second
- `\setminus(`*`<expr>`*`,`*`<expr>`*`)` (also the binary operator –) - a `\locset` containing any elements that are in the value of the first argument but not in the value of the second

Note that there can be an ambiguity when expressing a location (say `x`) which is itself typed as a `\locset`: `\locset(x,y)`, where x has type \locset and `y`'s type is something else, represents a set of two locations; if you want the contents of `x` with the location of `y` added in, you write `\union(x, \locset(y))`.

[2]The syntax would be neater if `..` designated half-open intervals, but JML has historically used `..` for closed intervals.

TODO: Need to resolve the above with the KeY team. What about `\singleton` *and* `\storeref` *and* `\cup`*. What about binary operators for union, intersection, disjoint and setminus – e.g.* $|$ *or* $+$*,* $*$ *or &, ##,* $-$*.*

Conjectures (MU):

- `\locset(x,y,...)  := \union(\locset(x), \locset(y), ...)`
- `\locset(expr)` evalues to the same set as `expr` if the expression is of type `\locset`.
- *otherwise:* `\locset(o.f), \locset(a[i])` is the singleton set that contains the referenced heap location
- *otherwise* `\locset(expr)` is a syntax error.
- hence: locset(locset(x)) == x if not a syntax error.

Needs to be mentioned here or there: What is the meaning of storerefs in assignable (accessible) clauses?

(Weigl) Should locset not a specialization of a set?

(Weigl): Location set, syntax constructs from KeY: \emptyset(), \storeref(...), (\infinite_union <vars>; <guard>; <locset>), \locset (field, field, ...), \singleton(field), \union(<locset>,<locset> ,<locset> ...) \setminus(<locset>,<locset>) \disjoint(<locset>,<locset>, ...) \subset(<locset>,<locset>)

New primitive datatype `\locset` with the following operators: (Reification of datagroups / regions)

- `\nothing` only existing locations
- `\everything` all locations
- `\empty` no location at all: The empty set.
- `\union`(...) arbitrary arity
- `\intersect`(...) arbitrary arity
- `\minus`(·,·)
- `\subset`(·,·)
- `\disjoint`(...) pairwise disjointness
- `(\collect ...; ...; ...)` a variable binder in the sense of

$$\bigcup_{x|\varphi} locs(x) = \texttt{(\collect } T\ x\texttt{;}\ \varphi\texttt{;}\ locs(x)\texttt{)},$$

  e.g., `(\collect int i; 0<=i && 2*i<a.length; a[2*i])` is the set of all locations in $a[*]$ with even index.

  Often needed for things like `(\collect Person p; set.contains(p); p.footprint)`.

- `\new_elements_fresh(·)` with the meaning

  $\texttt{\textbackslash new\_elements\_fresh}(ls) := \forall l \in ls.l \in \texttt{\textbackslash old}(ls) \vee \texttt{\textbackslash fresh}(object(l))$

  . This is used to confine the extension of a location set in a postcondition to objects which have been recently created. This is important to guarantee framing in dynamic frame specifications. This is sometimes called the *swinging pivot* property. (Reasoning is usually: If ls_1 and ls_2 are disjoint before a method and both ls_1 is not touched and ls_2 grows only into fresh objects, then ls_1 and ls_2 are still disjoint after the method.)

5.10 Mathematical sets: `\set<T>`

The type `\set<T>` is a built-in type of finite sets of items of type `T`. `T` may be a Java reference type or a JML built-in type (but not a Java primitive type). Uniqueness of elements is determined by the == operation. There are no null values of `\set`.

The `\set` type has the following operations defined.

Constructors:

- `\set.<T>empty()` — creates an empty set of type `\set<T>`
- `\set.<T>of(T ... values)` – creates a value of type `\set<T>` containing the given elements. The argument is a varargs argument, so the elements may be listed individually or the argument may be a (Java) array. If the type `T` can be inferred from the arguments it need not be stated explicitly.

Operators

- == and != — equality and inequality. Two values of type `\set<T>` are equal iff they contain the same elements, determined by the operation == on the elements of type `T`.
- `|` — set union (binary operation): the result set contains all values of type `T` that are in either of the operands
- `&` – set intersection (binary operation): the result set contains all values of type `T` that are in both of the operands
- `-` – set difference (binary operation): the result set contains all values of type `T` that are in the left operand but not in the right operand
- < and <= – proper and improper subset (binary operation): the result is true iff all the elements of the left-hand operand are elements of the right-hand operand
- `[]` – element membership , that is `s[o]` returns true iff the `o` (of type `T`) is an element of `s` (of type `\set<T>`)

Functions All these functions have value semantics (they produce a result without modifying the operands or anything else).

- `s.size()` – returns the number of elements in the set (type `\bigint`)
- `s.has(T)` – returns true iff the argument is in the set
- `\set.<T>equals(\set<T>,\set<T>)` – returns true iff the two arguments have the same elements, as deterined by the Java == operation
- `s.add(T... )` – returns a new set with the given elements added (the arguments may already be elements of the set)
- `s.remove(T... )` – returns a new set with the given elements removed (the arguments need not be elements of the set)
- `\set.<T>disjoint(\set<T> ... args)` — the boolean result is true iff the arguments are all pair-wise disjoint. There must be at least two arguments.
- `\set.<T>subset(\set<T> s1, \set<T> s2)` — the result is true iff the first argument is a (possibly improper) subset of the second

5.11 Mathematical sequences: `\seq<T>`

The type `\seq<T>` is a built-in type of finite sequences of items of type `T`. `T` may be a Java reference type or a JML built-in type (but not a Java primitive type). These sequences have a non-negative, finite length. There are no null values of `\seq`.

The `\seq` type has the following operations defined.

Constructors:

- `\seq.<T>empty()` — creates an empty sequence of type `\seq<T>`
- `\seq.<T>of(T ... values)` – creates a value of type `\seq<T>` containing the given elements in the given order. The argument is a varargs argument, so the elements may be listed individually or the argument may be a (Java) array. If the type `T` can be inferred from the arguments, it need not be stated explicitly.

Operators

- == and != — equality and inequality. Two values of type `\seq<T>` are equal iff they contain the same elements in the same order, determined by the operation == on the elements of type `T`.
- `[]` – item value, that is `s[i]` is the `i`'th (0-based) element of sequence `s`, where `i` has `\bigint` type, `s` has type `\seq<T>`, and the result is type `T`. The expression is well-defined but the result is undefined if the index is out of range.

- `[i..j]` – the subsequence from `i` through `j` (inclusive). The argument may be any `\range` expression, as described in §??. (*range syntax is under discussion*)
- `+` — sequence concatenation (binary operation): the result sequence is the concatenation of the values of the two operands (hence this operator `+` is not commutative)
- `<` — sequence proper prefix (binary operation): `s < ss` is true iff `s` is equal to a proper prefix of `ss`
- `<=` — sequence improper prefix: `s <= ss` is true iff `s < ss` or `s == ss`

Functions All these functions have value semantics (they produce a result without modifying the operands or anything else). In the following `s` and `ss` are `\seq<T>`, `i` and `j` are integers (type `\bigint`), and `t` is a value of type `T`.

- `s.length()` or `s.length` — the length of the sequence (a `\bigint`)
- `s.isEmpty()` — returns true iff if `s` is empty, that is, has length 0.
- `s.eq(ss)` or `s.equals(ss)` – returns true iff `s` and `ss` are `\seq<T>`s having the same (==) elements in the same order
- `s.ne(ss)` – the negation of `.eq`.
- `s.get(i)` — the result is the element (type `T`) of the sequence at position `i` (0-based, with `0 <= i < s.length`.
- `s.contains(t)` — the boolean result is true iff `t` is an element of `s`.
- `s.put(i,t)` — returns a new sequence of type `\seq<T>` which is equal to `s` except that position `i` in the result sequence contains the value `t`, where `0 <= i <= s.length`.
- `s.insert(i,t)` — returns a new sequence of type `\seq<T>`, one element longer than `s`, which is equal to `s` with the value `t` inserted between old positions `i-1` and `i`, where `0 <= i <= s.length`.
- `s.prepend(t)` — returns a `\seq` that is `s` with `t` added onto the beginning, increasing the length by 1 (the same as `s.insert(0,t)`).
- `s.prepend(ss)` — the same as `ss.append(s)`
- `s.append(t)` — returns a `\seq<T>` that is `s` with `t` added onto the end, increasing the length by 1 (the same as `s.insert(s.length,t)`).
- `s.append(ss)` — returns a `\seq` that is the concatenation of `s` and `ss`
- `s.subseq(i,j)` – a sequence that is a subsequence of `s` of length `j-i` containing the elements from position `i` up to but not including position `j`, where `0 <= i <= j <= s.length`.
- `s.head()` – the first element (a `T`) of `s`, which must not be empty.

- `s.head(i)` – a sequence that is a subsequence of `s` containing the `i` elements from position `0` up to but not including position `i`, where `0 <= i <= s.length`.
- `s.tail()` – the same as `s.tail(1)`, where `s` must not be empty.
- `s.tail(i)` – a sequence that is a subsequence of `s` containing the elements from position `i` through the end of the sequence `s`, where `0 <= i <= s.length`.
- `ss.prefix(s)` — sequence proper prefix: `ss.prefix(s)` is true iff `s` is equal to a proper prefix of `ss`, that is `s < ss`.
- `s.toString()` — returns a `java.lang.String` representation of the sequence `s`.
- `s.hashCode()` and `s.equals(Object o)` are defined so that `\seq` values can be used in maps and sets, but should not be used directly in Java+JML code.

5.12 String and `\string`

The built-in type `\string` is equivalent to `\seq<char>`, though that type cannot be expressed as such because `char` is a Java primitive type. Nevertheless, `\string` has all the operations that `\seq` has and the additions listed below. As a built-in primitive value type, equality (`==`) of `\string` values means equality of the sequences of characters.

Constructors:

- `\string.empty()` — constructs an empty (zero-length) `\string` value.
- `\string.of(String s)` — constructs a `\string` value from a non-null instance of a `java.lang.String`.
- `\string.of(char... c)` — constructs a `\string` value from a non-null array of Java `char`s.
- `\string.of()` — constructs an empty (zero-length) `\string` value.

Operators:

- `==` and `!=` — equality and inequality. Two values of type `\string` are equal iff they contain the same characters in the same order.
- `[]` – item value, that is `s[i]` is the `i`'th (0-based) `char` of string `s`, where `i` has `\bigint` type, `s` has type `\seq<T>`, and the result is type `char`. The expression is well-defined but the result is undefined if the index is out of range.
- `[i..j]` – the substring from `i` through `j` (inclusive). The argument may be any `\range` expression, as described in §??. (*range syntax is under discussion*)
- `+` — string concatenation (binary operation)

- < — string proper prefix (binary operation): `s < ss` is true iff `s` is equal to a proper prefix of `ss`
- <= — string improper prefix: `s <= ss` is true if `s < ss` or `s == ss`
- `< <= > >=` — string comparison operators, with the same case-senstsivie ordering as Java Strings.

Functions All these functions have value semantics (they produce a result without modifying the operands or anything else). In the following `s` and `ss` are `\string`, `i` and `j` are integers (type `\bigint`), and `t` is a value of type `char`.

- `s.length()` — the length of the string (a `\bigint`)
- `s.isEmpty()` — returns true iff if `s` is empty, that is, has length 0.
- `s.eq(ss)` or `s.equals(ss)` – returns true iff `s` and `ss` are `\string`s having the same characters in the same order
- `s.ne(ss)` – the negation of `.eq`.
- `lt le gt ge` — string comparison functions. For example. `s < ss` is `s.lt(ss)`.
- `s.get(i)` — the result is the element (type `char`) of the sequence at position `i` (0-based, with `0 <= i < s.length`.
- `s.put(i,t)` — returns a new `\string` which is equal to `s` except that position `i` in the result character sequence contains the character `t`, where `0 <= i <= s.length`.
- `s.insert(i,t)` — returns a new `\string`, one element longer than `s`, which is equal to `s` with the value `t` inserted between old positions `i-1` and `i`, where `0 <= i <= s.length`.
- `s.remove(i)` — returns a new `\string`, one element shorter than `s`, which is equal to `s` with the character at position `i` removed, where `0 <= i < s.length`.
- `s.prepend(t)` — returns a `\string` that is `s` with `t` added onto the beginning, increasing the length by 1 (the same as `s.insert(0,t)`).
- `s.prepend(ss)` — the same as `ss.append(s)`
- `s.append(t)` — returns a `\string` that is `s` with `t` added onto the end, increasing the length by 1 (the same as `s.insert(s.length(),t)`).
- `s.append(ss)` — returns a `\string` that is the concatenation of `s` and `ss`
- `s.substring(i,j)` – a `\string` that is a substring of `s` of length `j-i` containing the characters from position `i` up to but not including position `j`, where `0 <= i <= j <= s.length`.
- `s.head()` – the first character (a `T`) of `s`, which must not be empty.
- `s.head(i)` – a character sequence that is a substring of `s` containing the `i` elements from position `0` up to but not including position `i`, where `0 <= i <= s.length` (that is, s.substring(0,i)).
- `s.tail()` – the same as `s.tail(1)`, where `s` must not be empty.
- `s.tail(i)` – a character sequence that is a substring of `s` containing the elements from position `i` through the end of the sequence `s`, where `0 <= i <= s.length` (that is, s.substring(i,s.length())).
- `ss.prefix(s)` — sequence proper prefix: `ss.prefix(s)` is true iff `s` is equal to a proper prefix of `ss`, that is `s < ss`.
- `s.toString()` — returns an equivalent `java.lang.String` object
- `s.hashCode()` and `s.equals(Object o)` are defined so that `\string` val-

ues can be used in maps and sets, but should not be used directly in Java+JML code.

TODO - to discuss – more, string-like operations? indexOf ?

Runtime equivalent The JML type `\string` is mapped to `java.lang.String` for runtime assertion checking.

5.13 Mathematical maps: `\map<K,V>`

The type `\map<K,V>` is a built-in type of finite maps with keys of type `K` and values of type `V`. `K` and `V` may be Java reference types or a JML built-in types (but not Java primitive types). There are no null values of `\map`.

The `\map` type has the following operations defined.

Constructors:

- `\map.<K,V>empty()` — creates an empty map of type `\map<K,V>`

Operators

- `==` and `!=` — equality and inequality. Two values of type `\map<K,V>` are equal iff they contain the same set of keys and each key maps to the same value, determined by the operation `==` on the elements of types `K` and `V`.
- `[]` – item value, that is, `m[k]` is the value (type `V`) in the map `m` (type `\map<K,V>` corresponding to the key `k` (type `K`). The expression is well-defined but the result undefined if the map has no association for the given key.

Functions All these functions have value semantics (they produce a result without modifying the operands or anything else). In the following `m` is a `\map<K,V>`, `k` is a value of type `K`, and `v` is a value of type `V`.

- `m.has(k)` — true iff `k` is in the domain of `m`, that is, `m.has(k)`
- `m.get(k)` or `m[k]` — the value of the map for the given key; the value is undefined if `m.has(k)` is false, but the expression is still well-defined.
- `m.size()` — returns the number of keys in a map (as a `\bigint`.
- `m.isEmpty()` – returns true iff the map has no keys
- `m.keys()` — a `\set<K>` containing exactly the keys of the map `m` (i.e. the domain of the map)
- `m.put(k,v)` — returns a new map of type `\map<K,V>` that includes `v` as the value for the key `k`, with the values for all keys not equal to `k` unchanged from those in `m`.

- `m.remove(k)` — returns a new map of type `\map<K,V>` that is unchanged except for removing the key-value pair with the key `k`.
- `m.putAll(mm)` – returns a new map that is a copy of `m` with all the -key-value associations from `mm` added in.
- `m.eq(mm)` and `m.ne(mm)` — equivalent to `m == mm` and `m != mm`
- `m.hashCode()` and `m.equals(mm)` — implemented only to support using JML maps in hashed structures; do not use these methods in user code
- `m.toString()` — returns a `java.lang.String` with a representation of the content of `m`.

5.14 Mathematical arrays: `\array<T>`

The type `\array<T>` is a built-in type of finite, fixed-length arrays with values of element type `T`, indexed by non-negative values of `\bigint`. `T` may be a Java reference type or a JML built-in type (but not Java primitive types). There are no null values of `\array`.

An `\array` differs from a `\seq` in that (a) there are no operations that produce new arrays with a changed length and (b) iteration over arrays is by indexing within a loop, rather than recursive uses of `head` and `tail`.

The `\array<T>` type has the following operations defined, for arrays `a` and `aa` of type `\array<T>` for some type `T`, value `t` of type `T` and values `i` and `j` of type `\bigint`:

- conversions – there are no implicit or explicit (cast) conversions to or from an `\array` value
- `\array.<T>empty()` — construct a new, zero-length array
- `\array.<T>of(T ... ta` — construct a new array with the given elements, given by either a Java array `T[]` or an explicit varargs list of element values `t`; the length of the new `\array` is the number of arguments or the length of the `T[]`.
- `a.length()` or `a.length` — the length of the `\array a`
- `a[i]` — the value (type `T`) at 0-based position `i`; the value is undefined (and possibly an exception at runtime) if `i` is outside the range `0 <= i < a.length`.
- `a.get(i)` — like `a[i]` except that the expression is not well-defined if `i` is out of range
- `a.put(i,t)` — returns a copy of `a` except that at position `i` the value is `t`; it is required that `0 <= i < a.length`.
- `a == aa`, `a.eq(aa)`, `a.equals(aa)` — returns true iff `a` and `aa` are equal element by element (equals elements using ==)
- `a != aa`, `a.ne(aa)` — the negation of `a == aa`
- `a.subarray(i,j) --- returns a new array of length \code{j-i` consisting of the elements of `a` from position `i` up to but excluding element `j`. It

must be that `0 <= i <= j <= a.length`.

- `a.toString()` — returns a `java.lang.String` representation of `a`
- `a.hashCode` and `a.equals(o) for any \code{Object o` — these are defined to allow `\array`s to be used in hashed sets and maps but are not to be used in user Java+JML code; in any case, casting a `\array<T>` to any reference type is not permitted.

Fill this in – do we need this in addition to sequences?

5.15 Inductive datatypes

This section is still in process

JMLv2 adds inductive datatypes to JML. As in other languages, these are value-based, recursive structures that are amenable to reasoning by induction.

The syntax for defining such a datatype is Java-like and reminiscent of Java enums and Java records. The syntax is defined by this grammar:

```
<jml-datatype> ::= <modifiers> datatype [ <typeargs> ] {
      <dt-constructor> ... [ ; <dt-method>+ ]
      }

<dt-constructor> ::=
        <ident>
      | <ident> ( <formal> ... )


<dt-method> ::=
```

Complete the grammar above and align it with other grammar definitions

Because datatypes are a JML feature, they must be declared `model` and within a JML annotation comment. They also belong to the package declared in the file containing the datatype declaration. Although syntactically, a `datatype` declaration is similar to a JML `model` class and a Java `enum`, datatypes are primitive, value-based types and do not inherit from or are the parent of other types.

For example, a classic List would be declared as

```
//@ model datatype List<T> { Nil, Cons(T head, List<T> tail) }
```

The constructors of the datatype are explicit. A data type also has implicit discriminators and destructors. A datatype may also have methods. And there is a `match` statement and `match` expression that are similar to Java `switch` statements.

Datatypes are `final`: no inheritance is permitted.

State restrictions on type arguments.

5.15.1 Constructors

Constructors create elements of a datatype that structurally include the arguments of the constructor. In a Java sense, the constructors are static members of the datatype. For example, `List.<Integer>Nil` returns an empty `List<Integer>` and `List.Cons(5, List.<Integer>Nil)` a one-element `List<Integer>`.

The constructors are all implicitly `public` and `static`. Constructor names are Java identifiers; they must be different and may not differ only in the case of the initial letter.

If a static import (`static import List.*`) is used, then the above can be shortened to `Cons(5, Nil)`.

The latter presumes some kind of type inference, to be worked out and explained.

Should a no-argument constructor still be required to be called with parentheses – `Nil()` in the above.

5.15.2 Discriminators

A value of a datatype has been constructed by one of the constructors; a discriminator tells which one. For each constructor, an implicit discriminator is defined, taking no arguments and returning a boolean. The name of the discriminator is formed by prepending `is` to the constructor name and changing the first letter of the constructor name to upper-case, if it is not already upper-case. So the discriminators for constructors named `nil` and `Nil` both are named `isNil`.

Thus in our example,

```
1 List<Integer> n = List.<Integer>Nil;
2 List<Integer> c = List.Cons(5, n);
3 n.isNil() // is true
4 c.isNil() // is false
5 n.isCons() // is false
6 c.isCons() // is true
```

5.15.3 Destructors

Destructors (or accessors) take apart a datatype value. The name of each formal parameter of each constructor becomes a no-argument method of the datatype. Consequently, the formal parameter names must be distinct across all of the constructors in the datatype. Continuing the example above,

```
1 c.head() // returns the Integer 5, presuming c.isCons() is true
2 c.tail() // returns n, presuming c.isCons() is true
```

A constructor with no arguments has no sub-structure so there is no corresponding destructor.

The application of a destructor must be well-defined: for a given value of a datatype, a given destructor may be called only if it can be proved that the discriminator for the corresponding constructor returns true.

5.15.4 Datatype equality

A datatype is a value-based type. There are no null values. Equality is structural equality: two datatype values (of the same type) are equal (==) iff they respond the same to all discriminators and destructors. Equivalently, they are equal iff they have both been constructed using the same constructor method with equal (==) arguments.

5.15.5 Structural comparison

For a datatype value `d` and other value `t`, `t < d` iff there is some value `v` that is the result of some destructor applied to `d` and either `t == v` or (if `v` is also a value of some datatype) `t < v`

For a datatype value `d` and other value `t`, `t <= d` iff `t == v` or `t < v`

5.15.6 Methods

A datatype may also define additional methods of its own. These are just like the methods of a class. They may use the datatype's constructors, discriminators, and destructors; they may be either static or instance (the default) in the Java sense; and they should have a JML specification.

At present a JML datatype may not declare additional fields, nor any nested classes or interfaces. Nor may the constructors be embellished with custom code (as Java enum constructors may).

5.15.7 Match expressions and statements

To be written – standard matches for datatype values

5.16 \range

TODO-range

5.17 Tuples

Experimental. Subject to change.

JML defines a simple, generic, value-based tuple type. A tuple can be thought of as an unnamed record with unnamed fields.

- A tuple value is written as a parenthesized list of two or more comma-separated values, such as `(1,2,3)`. There is no value of a 1-tuple, as a single expression in parentheses is just a parenthesized expression.
- The type of a tuple is written as a generic type with the corresponding number of type parameters, such as `\tuple<\bigint, \real, Object>`.
- The elements of a tuple value can be retrieved using array indexing syntax. For example, `(1,true)[0] == 1` and `(1,true)[1] == true`. Note that although array-index syntax is used, the indices must be integer literals; also the type of the expression may well differ depending on the value of the index.

Tuples are convenient when a method or lambda function needs to return multiple values within a specification expression.

An alternative to the array index notation is to use a numeric field notation. That is, `(1,true).0 ==1` and `(1,true).1 == true`. This syntax conveys that the indices are fixed literals and not computed; however, it requires additional parser customization.
It is conceivable to allow a 0-ary tuple, namely the single value `()`. However, then we would need to have types `\tuple0`, `\tuple2<T,U>`, `\tuple3<T,U,V>`, and so on. It is also possible to have the named tuple types just mentioned, with a common supertype, `\tuple`. However, use cases for this are yet to be developed.

5.18 Representation of JML types

This Java Modeling Language definition only states the semantics of the JML-specific types, not how that semantics is implemented in tools. Typically, however, in generating an executable program for runtime-assertion-checking, JML types are translated into some Java class that embodies the needed semantics. For example, a natural representation for `\string` is `java.lang.String`. A tool may even allow the choice of different representations.

For deductive verification with, for example, an SMT solver as the logical engine, a JML type may be mapped directly into a theory supported by the SMT solver. For example, `\seq` might be represented as an SMT `Array` sort. Alternatively, JML types can be represented simply as an uninterpreted SMT sort with uninterpreted functions that are given semantics by axioms.

Chapter 6

JML Specifications for Packages and Compilation Units

6.1 Package and Module specifications

There are no JML specifications at the package or module level. If there were, they would likely be written in the `package-info.java` file for the package or the `module-info.java` file for the module.

Specifications at the package and module level have been proposed. They would simply consist of JML annotation comments in the respective file that contain modifiers or Java @-annotations. Such modifiers would apply to all packages in the module or to all classes in a package. This would be a convenient way to state a global property. However, these `*-info.java` files are not heavily used in Java programs and annotations in them could well be silently not seen by users, causing confusion.

6.2 Compilation unit specifications

The only JML specifications that are defined at the file level, applying to all classes defined in the file, are model import statements (§??) and model classes (§??).

6.3 Model import statements

Java's import statements allow class and (with static import statements) field names to be used within a file without having to fully qualify them. The same import statements apply to names in JML annotations. In addition, JML allows *model import*

statements. The effect of a JML model import statement is the same as a Java import statement, except that the names imported by the JML statement are only visible within JML annotations. If the model import statement is within a `.jml` file, the imported names are visible only within annotations in the `.jml` file, and not outside JML annotations and not in a corresponding `.java` file. These import statements only affect name resolution within JML annotations and are ignored by Java. They have the form

```
//@ model <Java import statement>
```

Note that the Java import statement ends with a semicolon.

Note that both

```
model <Java import statement>
```

and

```
/*@ model */<Java import statement>
```

are invalid. The first is not within a JML comment and is illegal Java code. The second is a normal Java import with a comment in front of it that would have no additional effect in JML, even if JML recognized it (tools should warn about this erroneous use).

6.4 Default imports

The Java language stipulates that `java.lang.*` is automatically imported into every Java compilation unit. JML reserves the package `org.jmlspecs.lang` as one that might be automatically model-imported in the future. However, there are not yet any standard-defined contents of the `org.jmlspecs.lang` package.

6.5 Examples using model import statements

Consider two packages `pa` and `pb` each declaring a class `N`.

1)

```
import pa.N;
//@ model import pb.N;
```

JML behavior: In Java code `N` is `pa.N`; in JML code, `N` is ambiguous.

2)

```
import pa.N;
//@ model import pb.*;
```

JML behavior: In Java code `N` is `pa.N`; in JML code, `N` is `pa.N`.

3)

```
import pa.*;
//@ model import pb.N;
```

JML behavior: In Java code `N` is `pa.N`; in JML code, `N` is `pb.N`.

4)

```
import pa.*;
//@ model import pb.*;
```

JML behavior: In Java code `N` is `pa.N`; in JML code, `N` is ambiguous.

The behavior of model static imports is similar.

6.6 Model classes and interfaces

Just as a Java compilation unit (typically a file) may contain multiple class definitions, a compilation unit may also contain declarations of JML model classes and interfaces.

A model class declaration is very similar to a Java class declaration, with the following differences:

- the declaration is entirely contained within a (single) JML annotation comment
- the declaration has a `model` modifier
- if the compilation unit contains Java class or interface declarations, the model class or interface may not be the primary declaration (that is, the one with the `public` modifier)
- JML constructs within a JML model declaration need not be contained in (nested) JML annotation comments
- JML constructs within JML model classes or interfaces must not themselves be declared `model` or `ghost`

Though secondary model classes and interfaces are allowed, it is generally more convenient to declare such classes as primary classes in their own file or simply as Java classes that are included with a program when applying JML tools.

A JML model class or interfaces is only used in other JML specifications and not in Java code. Hence there is no need to distinguish Java from JML constructs within the model declaration. Consequently `ghost` and `model` modifiers on nested constructs are not permitted.

Chapter 7

Specifications for Java types in JML

By Java *types* in this reference manual we mean classes, interfaces, enums, and records, whether global, secondary, local, or anonymous. Some aspects of JML, such as the allowed modifiers, will depend on the kind of type being specified.

Specifications at the type level serve three primary purposes: specifications that are applied to all methods in the type, specifications that state properties of the data structures in the type, and declarations that help with information hiding.

Modifiers are placed just before the construct they modify. Example Java modifiers are `public` and `static`. JML modifiers may be in their own annotation comments or grouped with other modifiers, as shown in the following example code.

As discussed in §??, Java annotations from `org.jmlspecs.annotation.*` and placed in Java code can be used instead of modifiers.

```
//@ pure
public class C {...}

public /*@ pure nullable_by_default */ class D {...}
```

7.1 Modifiers for type declarations

7.1.1 `model`

As discussed in §??, classes within JML annotations can be declared `model`. Model classes are available to be used in specifications and compiled for runtime-checking.

Model classes may be nested inside Java classes. All the fields of a model class are intrinsically `ghost`; all the methods and nested classes are intrinsically `model`.

7.1.2 non_null_by_default, nullable_by_default, @NonNullByDefault, @NullableByDefault

The `non_null_by_default` and `nullable_by_default` modifiers or, equivalently, the `@NonNullByDefault` and `@NullableByDefault` Java annotations, specify the default nullity declaration within the class. Nullness is described in §??. The default applies to all typenames in declarations and in expressions (e.g. cast expressions), and recursively to any nested or inner classes that do not have default nullity declarations of their own.

These default nullity modifiers are not inherited by derived classes.

A class cannot be modified by both modifiers at once. If a class has no nullity modifier, it uses the nullity modifier of the enclosing class; the default for a top-level class is `non_null_by_default`. This top-level default may be altered by tools.

Because the specifications of a class depend in detail on the default nullness setting, it is best practice in any significantly sized project to declare the default class-by-class and not rely on the default set at the top-level.

7.1.3 Purity modifiers

The purity modifiers are `pure`, `@Pure`, `spec_pure`, `@SpecPure`, `strictly_pure`, and `@StrictlyPure`. At most one of these may be applied to a class declaration.

The different levels of purity are described in §??. The meaning of applying one of these modifiers to a class is that it states the default modifier for any method in the class, recursively, unless superseded by a declaration on a nested class or method. A purity modifier on a class is not inherited by derived classes, though purity modifiers on methods are.

There is no modifier to disable an enclosing purity specification.

7.1.4 @Options

The `@Options` modifier takes as argument either a String literal or an array of String literals, using the syntax `@Options({`*`s1 ...`*`})`, with each literal being just like a command-line argument, that is, it begins with one or two hyphens and possibly contains an = character with a value. These command-line options are applied to the processing (e.g., ESC or RAC) of each method within the class. The options may be augmented or disabled by corresponding `@Options` modifiers on nested methods or classes. In effect, the options that apply to a given class are the concatenation of the options given for each enclosing class, from the outermost in.

An Options modifier is not inherited by derived classes.

Not all command-line options can be applied to an individual method or class.

The `@Options` feature is inherently somewhat tool-dependent because it relates to the command-line options that a tool might implement. However, JML defines this feature so that tools will implement it with the same name and intent.

Using the `@Options` feature permits tool-defined command-line options that are applicable to a method to be applied without having to implement a new corresponding modifier in JML itself.

7.1.5 Arithmetic modes

Class declarations can take arithmetic mode modifiers as described in §??. Only one each of the `code...` and `spec...` modifiers may be applied at one time. The scope of the modifiers includes all of the declarations in the class, unless overridden by arithmetic modifiers on nested classes or methods.

7.1.6 `spec_public`, `spec_protected`, @SpecPublic, and @SpecProtected

These modifiers apply to classes declared in Java code, and not to classes declared in JML, such as model classes. They have the effect of replacing the Java visibility modifier for the class with an alternate, for the purposes of specification. For example, a Java declaration declared `private` in Java, but also `spec_public` in a JML modifier for the class, is treated as `public` in all specifications.

Visibility is discussed in detail in §??.

7.2 `invariant` clause

Grammar:

<invariant-clause> ::= **invariant** *<opt-name>* *<predicate>* ;

Type information: The *<predicate>* has boolean type.

Invariant clauses may have a visibility modifier and may be declared static

Type invariants are discussed in §??.

The `invariant` clause states a predicate which is expected to hold of the fields of the class. The program points at which invariants hold is a complex topic addressed in §??. However, the invariants of a class are assumed to hold at the entrance of non-helper methods and are required to hold again at the exit of non-helper methods. Also, non-static invariants do not apply to static methods.

Invariants clauses have a (Java) visibility. This affects which fields or methods may be mentioned in the clause's predicate. (cf. §??)

Any methods called in an invariant must, in addition to having some kind of purity, be `helper` methods. A non-helper method would require the invariant to hold before that non-helper method could be called, leading to an unsound, recursive situation.

Visibility modifiers needed in the grammar here and elsewhere. What about final invariants?

7.3 `constraint` clause

Grammar:

<constraint-clause> ::= **constraint** *<opt-name>* *<predicate>* ;

Type information: The *<predicate>* has boolean type and is evaluated in the post-state. It is a two-state predicate in which the pre-state is designated by the `\old` construct.

A `constraint` clause may have a Java visibility modifier and may be declared static. Non-static constraint clauses do not apply to static methods.

A `constraint` clause for a type is equivalent to an additional postcondition for each non-constructor method of the type, given that the constraint is visible from the method. That is, a `public` constraint applies to a `private` method, but a `private` constraint does not apply to a `public` method. Furthermore, the predicate of a `constraint` clause may only contain field and method references that have at least as much visibility as the clause; so, a `public` clause may not contain `private` fields.

A `constraint` clause is equivalent to adding `ensures` clauses (with the predicate stated by the `constraint` clause) to each specification behavior of each method in the type, subject to the visibility rules. Like an `ensures` clause, a `constraint` clause is evaluated in the post-state.

`Constraint` clauses are used only by methods of the class in which the clause appears. The clause is not "inherited" by derived classes. However, overriding methods in derived classes do inherit the method specifications from their parent methods, and these effectively have the `constraint` clauses included. *(This needs checking)*

A typical use of a `constraint` clause is to require some condition about the fields of a class to hold between the pre- and post-states of every method of the class. For example,

```
constraint count >= \old(count);
```

states that the field `count` never decreases when methods of the class are called.

There used to be syntax to list or exclude a set of methods

7.4 `initially` clause

Grammar:

<initially-clause> ::= **initially** *<opt-name>* *<predicate>* ;

Type information: The *<predicate>* has boolean type and is evaluated in the post-state of constructors.

An `initially` clause may have a Java visibility modifier. It may not be marked `static`.

An `initially` clause for a type is equivalent to an additional postcondition for each constructor method of the type, given that the clause is visible from the method. That is, a `public initially` clause applies to a `private` constructor, but a `private` clause does not apply to a `public` constructor. Furthermore, the predicate of an `initially` clause may only contain field and method references that have at least as much visibility as the clause; so, a `public` clause may not contain `private` fields.

A `initially` clause is equivalent to adding `ensures` clauses (with the predicate stated by the `initially` clause) to each specification behavior of each constructor in the type, subject to the visibility rules. Like an `ensures` clause, an `initially` clause is evaluated in the post-state. `Initially` clauses are used only by constructors of the class in which the clause appears. The clause is not "inherited" by derived classes because constructors are not overridden by derived classes.

7.5 `ghost` fields

Grammar:

<ghost-field-declaration> ::=
    ghost *<opt-name>* *<jml-field-declaration>*

A ghost field declaration has the same syntax as a Java declaration except that it contains the `ghost` modifier and is in a JML annotation. It declares a field that is visible only in specifications. Runtime-assertion-checking compilers would compile a ghost field like a normal Java field.

The type of a `ghost` field may be any JML or Java type. `Ghost` fields may have Java visibility modifiers, may be declared `static`, may have initializers, and may be declared `final`.

Although the grammar permits the `ghost` modifier to be included in any order with other modifiers, good style recommends that the `ghost` modifier is first (and this may be required in the future).

Note that all Java-like declarations (e.g. fields, methods, classes) must have either a `ghost` or `model` declaration (or be nested in a `ghost` or `model` declaration).

Are we sure about opt-names for ghost and model fields?

7.6 `model` fields

Grammar:

```
<model-field-declaration> ::=
        model <opt-name> <jml-field-declaration>
```

A model field declaration has the same syntax as a Java variable or field declaration except that it contains the `model` modifier and is in a JML annotation. However, a model field is not a "real" field in the sense that it is not compiled into an executable representation within its containing class, even for RAC compilation. Rather a model field designates some abstract property of its containing class. The value of that property may be completely uninterpreted, determined only by the constraints imposed by various other specifications. Alternately, the value of a model field may be given directly by a `represents` clause.

A model field is also implicitly a *datagroup* in that it designates a `\locset` of memory locations (store-refs), given by various `in` and `maps` clauses.

The type of a `model` field may be any JML or Java type.

Although the grammar permits the `model` modifier to be included in any order with other modifiers, good style recommends that the `model` modifier is first (and this may be required in the future).

Note that all Java-like declarations (e.g. fields, methods, classes) must have either a `ghost` or `model` declaration (or be nested in a `ghost` or `model` declaration).

7.7 `represents` clause

Grammar:

```
<represents-clause> ::= [ static ] <represents-keyword> <ident>
       ( = <jml-expression> ;
       | \such_that <predicate> ;
       )
<represents-keyword> ::= represents | represents_redundantly
```

Type information:

- The identifier named in the represents clause must be a model field declared in or inherited by the class containing the represents clause.
- the *<jml-expression>* in the first form must have a type assignable to the type of the given field (that is, *ident* = *expr* must be type-correct).
- the *<predicate>* in the second form must be a *<jml-expression>* with boolean type
- The visibility and static-ness of a represents clause is implicitly that of the field it is defining. A `static` modifier is optional but if present must match the field.
- In a static represents clause, only static elements may be referenced both in the left-hand side and the right-hand side. In addition, a static represents clause

must be declared in the type where the model field on the left-hand side is declared (and not in a parent class or interface).

The first form of a represents clause is called a functional abstraction. This form defines the value of the given identifier in a visible state as the value of the expression that follows the `=`. The represents clause for field *f* with expression *e* in class *C* is equivalent to assuming

```
forall non_null C c; c.f == e_c
```

where `e_c` is `e` with `c` replacing `this`.

The second form (with `\such_that`) is called a relational abstraction. This form constrains the value of the identifier in a visible state to satisfy the given predicate.

A represents clause does not take a visibility modifier. In essence, its visibility is that of the field whose representation it is defining. However, there is no restriction on the visibility of names on the right-hand-side. For example, the representation of a public model field may be an expression containing private concrete fields.

Note that represents clauses can be recursive. That is, a represents clause for a field `f` may contain a subexpression like `o.f` on its right hand side, where `o` has the same type but is a different object than `this`. It is the specifier's responsibility to make sure such definitions are well-defined. But such recursive `represents` clauses can be useful when dealing with recursive datatypes [**?**].

7.8 model methods

needs grammar

A JML annotation within a class or interface may contain a *model method* or *model constructor* declaration. Such a declaration is within a JML annotation comment and has a `model` modifier. It may use any Java or JML types, but is otherwise syntactically similar to Java method declarations. Methods and constructors declared within model classes and interfaces are also model methods, though they do not have a `model` modifier.

If a model method has a body it can be compiled and used during runtime checking; a model method's body must be consistent (as checked by verification tools) with the model method's specification.

A model method need not have a body. In this case it cannot be compiled for runtime checking. The semantics of the method are defined solely by its specification. The specification may be under-specified, in which case the method is (perhaps partially) uninterpreted. Even if uninterpreted, a model method is deterministic; that is, in the same state with the same arguments, the method always has the same effect and returns the same result.

Although model methods that have side-effects (that is, they are not any sort of pure) are permitted and might be useful in runtime checking, in practice, model methods

are nearly always pure and are intended for use in specifications.

For historical reasons, there are `model` methods but not `ghost` methods. An alternate nomenclature would call JML methods intended to be compiled for runtime checking (that is, having bodies) as `ghost` methods, while JML methods intended as abstractions (that is, without bodies) would be `model` methods.

7.9 nested model classes

needs grammar

A Java program may declare nested classes and interfaces. Similarly, a JML annotation within a Java class may contain a JML model class or interface declaration as a nested or inner class.

Like top-level model class and interface declarations, nested model declarations are used in stating (and proving) specifications and are also compiled for runtime-assertion checking.

The contents of a nested or inner model class declaration obey the same syntax rules as top-level model declarations and behave like nested or inner Java declarations.

7.10 `static_initializer`

Grammar:

<static-initializer-block> ::= *<specification-cases>* **static** *<block>*

<static-initializer> ::= *<specification-cases>* **static_initializer**

Type information: The *<specification-cases>* are type-checked in the static context of the class.

7.10.1 Simple static initialization

The process of class initialization defined by Java has these steps, omitting the complexities of locking:

- initialize all final static fields whose values are compile-time constant expressions
- initialize all other fields to zero-equivalent values
- initialize all super classes, if not already initialized
- initialize the fields that have non-compile-time-constant initializers and execute the static initializer blocks in textual order

Note that each static initializer block may have a specification, as if it were a method with no receiver or parameters, with a pre-state just before the execution of the block

and a post-state just after. The contents of the initializer block must satisfy that specification.

In addition, the class may have a JML `static_initializer` specification. This is a sequence of specification cases immediately preceding the `static_initializer` keyword. A class may have no more than one such `static_initializer`; it may be placed anywhere in the body of the method, as it is conceptually relocated to the end of the class body. This specification summarizes the entire static initialization.

The predicate `\isInitialized(C)` for a class name `C` is false until the class initialization is complete, and then it is (forever after) true. It is implicitly false in the pre-state of the static initializer and implicitly true in the post-state.

Need an example

7.10.2 Static initializers and static invariants

Static initialization is a one-time process. The values of (non-final) static fields can change after initialization. The process of creating a new instance of a class starts from the static state at the time of instance creation. What is known about the static state after initialization is captured by the class's *static invariants*.

The static invariants must be true in the post-state of the static initializer and they must be maintained by any method that possibly assigns to any field that the invariant depends on.

To be resolved:

There is still a conceptual problem here. A static invariant of class A might depend on non-constant fields of class B, which might be modified by method C.m(), which has no knowledge of class A and hence cannot be responsible for the maintenance of its invariant.

7.10.3 Default static initialization

Final static fields initialized by compile-time expressions do not change their values after initialization and their values can be computed independently of the program. For such fields, there is an obvious post-condition and an obvious static invariant: that the field equals its compile-time value. JML defines this postcondition and invariant to be implicit; it need not (but may, redundantly) be stated explicitly.

A common case is that a class's static fields are all final fields, initialized with compile-time constants. In this case both the specifications of the static initializer and the static invariants are obvious: they are a conjunction of conjuncts stating that each field equals its compile-time value.

However, final static fields not initialized by a compile-time expression do not always have this benefit. In some cases, an inline computation can compute the initialized

value of the field, but in others, such as where some method is called to compute the value, the initialized value may not be known by a static analyzer.

If the field is not final then the initialized value and the value stated by an invariant may well be different.

It is, however, an inconvenience to the user to need to write a static initializer, especially when the content seem obvious and repetitious. Therefore the default specification in the absence of a `static_initializer` is an inlining of the field initializations and static initializer blocks. If this is insufficient for reasoning about the program, a static initializer is required.

The proof rules for static initialization are still a matter of research.

7.10.4 Multi-class initialization

Static initialization is typically quite simple. It becomes complicated when classes refer to each other during initialization. For example, consider these interrelated classes:

```
1 class A {
2   static final int a = B.b;
3   static int aa = 42;
4 }
5 class B {
6   static final int b = A.aa;
7 }
```

If `A` begins initialization when B is not yet initialized then the following happens:

- `A.a` and `A.aa` are initialized to `0` (`aa` is not final and `a` does not have a compile-time constant initializer)
- `A.a` begins initialization, but that requires `B` to be initialized
- `B.b` is initialized to `0`
- `B.b` needs the value of `A.aa`; as `A` has already started initialization, that value is returned as `0` (since `A.aa` is not final it is not initialized as a compile-time constant expression).
- `B` has completed initialization
- `A.a` is initialized with the value `0`, which is the current value of `B.b`
- `A.aa` is initialized to `42`
- `A`'s initialization is complete

On the other hand, if `B` starts initialization before `A`, then

- `B.b` is initialized to `0`
- `B.b` computes its initializer, starting the initialization of `A`
- `A.a` gets the value of `0` for `B.b`
- `A.aa` is initialized to `42`, completing `A`'s initialization

- `B.b` gets the value of `42` for `A.aa`

So the value of `B.b` depends on the order of initialization, and it and the value of `A.a` may not be what the user intended.

Note that if `A.aa` were declared `final`, then `A.aa` would be initialized as a compile-time constant expression and all three fields would have the value `42` no matter which class started initialization first.

There are three important lessons:

- Inadvertently omitting a `final` modifier can change the semantics
- Inadvertently or intentionally having classes access other not-fully initialized classes (because of a dependency loop) can cause order-of-initialization dependent behavior.
- Dependency loops can be non-obvious: they may be mediated by chains of method calls for example.

7.11 (instance) `initializer`

Grammar:

<instance-initializer> ::= *<specification-cases>* **initializer**

Instance initialization blocks and instance field initializers are executed as part of constructors. In summary this sequence is followed for any constructor that begins. perhaps implicitly, with a `super` call.

- (Complete the static initialization for the class)
- Initialize all final instance fields with no initial values or compile-time initial values to the given values and all other instance fields to zero-equivalent values, in textual order.
- Execute the super-class constructor
- Execute the non-final or non-constant instance field initializers and instance initialization blocks in textual order
- Execute the body of the constructor

Constructors that begin with a `this` call simply delegate this process to a different constructor.

The state prior to the execution of the super-class constructor is independent of a constructor's arguments; it can be summarized with a JML `initializer` specification. For many classes, the initializer specification is simply the conjoining of the static initializer specification and the predicates stating the values of various final, constant fields.

Only one non-static `initializer` specification is permitted per class. Essentially the specification semantics of the constructor are the following:

- Assume the class's static invariants

- Assume the class's instance `initializer`
- Assume the constructor preconditions
- Assert the super-call preconditions
- Assume the super-call postconditions
- Symbolically execute the remainder of the constructor body
- Assert the constructor's postconditions
- Assert the class's `initially` clauses
- Assert the class's `instance` invariants
- Assert the class's `static` invariants

Is the order of the last three correct?

Need examples

Do initializers have visibility modifiers?

7.12 axiom

Grammar:

<axiom-clause> `::=` **`axiom`** *<opt-name>* *<predicate>* `;`

Type information: The *<predicate>* has boolean type. An axiom must be a state-independent, closed formula.

Axioms always have public visibility.

Axioms are assumptions introduced into the proof. An axiom must be a state-independent formula. Typically it might express a property of a mathematical type that is too difficult for an automated tool to prove.

As assumptions, axioms are a soundness risk for verification, unless they are separately proved.

7.13 readable if clause and writable if clause

Grammar:

<readable-if-clause> `::=` **`readable`** *<ident>* **`if`** *<jml-expression>* `;`
<writable-if-clause> `::=` **`writable`** *<ident>* **`if`** *<jml-expression>* `;`

Type information:

- the *<ident>* must name a field (possibly inherited) visible in the class containing the clause
- the *<jml-expression>* must have boolean type
- Any name used on the right-hand-side must be visible in any context in which the given *<ident>* is visible.

The `readable-if` clause states a condition that must be true at any program point at which the given field is read.

The `writable-if` clause states a condition that must be true at any program point at which the given field is written.

The visibility modifier of either of these clauses must match the visibility of the *<ident>* being specified.

Or - the visibility modifier is implicitly that of its ident

7.14 monitors_for clause

Grammar:

```
<monitors-for-clause> ::=
        monitors_for <ident> = <jml-expression> ... ;
```

Type information:

- the *<ident>* must name a field (possibly inherited) visible in the class containing the clause
- the *<jml-expression>*s must evaluate to a (possibly null) reference

A *<monitors-for-clause>* such as `monitors_for f = e1, e2;` specifies a relationship between the field, `f`, and a set of objects, denoted by a specification expression list `e1, e2`. The meaning of this declaration is that all of the (non-null) objects in the list, in this example, the objects denoted by `e1` and `e2`, must be locked at the program point at which the given field (`f` in the example) is read or written.

Note that the right-hand-side of the monitors-for-clause is not just a list of memory locations, but is in fact a list of expressions, where each expression evaluates to a reference to an object.

The visibility modifier of a `monitors_for` clause must match the visibility of the identifier being specified. *Or should the visibility be implicit*

The *<monitors-for-clause>* is adapted from ESC/Java [?] [?]. As it relates to synchronization locking, it is meant for future use in multi-threaded programs.

Chapter 8

JML Method specifications

Method specifications describe the intended behavior of the method with which they are associated. JML uses a modular specification methodology; that is, reasoning about method calls uses the called method's specification, not its code. Conversely, each method's code is verified to satisfy its specification. Calls of a method must satisfy the method's precondition, but the specification also tells callers what they may assume about the program's state when the method completes execution.

A method specification need not completely determine everything about the program's state after a call; that is one can use "under-specification." For example, a specification may simply say that a method always returns normally (that is, without throwing an exception), but give no constraints on the method's post-state (the program's state after the method terminates). The degree of precision needed will depend on the context and purpose of the specification effort.

8.1 Structure of JML method specifications

A JML method specification consists of a sequence of zero-or-more specification cases; each case has an optional behavior keyword followed by a sequence of clauses. The specification case may also have a Java visibility modifier.

```
<method-spec> ::= ( also )? <behavior-seq>
                      ( also implies_that <behavior-seq> )?
                      ( also for_example <behavior-seq> )?

<behavior-seq> ::= <behavior> ( also <behavior> )*

<behavior> ::=
     ( <java-visibility> ( code )? <behavior-id>)? <clause-seq>

<java-visibility> ::= ( public | protected | private )?

<behavior-id> ::=
       behavior | normal_behavior | exceptional_behavior
```

```
    | behaviour | normal_behaviour | exceptional_behaviour

<clause-seq> ::= ( <method-spec-clause> | <nested-clause> ) *

<method-spec-clause> ::=
        <invariants-clause>      §??
      | <requires-clause>        §??
      | <recommends-clause>      §??
      | <old-clause>             §??
      | <writes-clause>          §??
      | <reads-clause>           §??
      | <callable-clause>        §??
      | <ensures-clause>         §??
      | <signals-clause>         §??
      | <signals-only-clause>    §??
      | <diverges-clause>        §??
      | <measured-by-clause>     §??
      | <when-clause>            §??
      | <duration-clause>        §??
      | <working-space-clause>   §??
      | <captures-clause>        §??
      | <returns-clause>         §??
      | <throws-clause>          §??
      | <continues-clause>       §??
      | <breaks-clause>          §??
      | <model-program-block>    §??

 <nested-clause> ::=             §??
      {|  ( <clause-seq> ( also <clause-seq> ) * ) ? |}
```

Meta-parser rules:

- Each of the behavior keywords that uses the spelling "`behaviour`" is equivalent to the corresponding keyword with the spelling "`behavior`."
- A *<behavior>* beginning with `normal_behavior` may not contain a *<signals-clause>* or a *<signals-only-clause>*. It is a shorthand for a `behavior` that contains the clauses:

  ```
  signals (Exception e) false; signals_only \nothing;
  ```

  .
- A *<behavior>* beginning with `exceptional_behavior` may not contain an *<ensures-clause>*. It is a shorthand for a `behavior` that contains the clause `ensures false;`.
- The **also** that begins a *<method-spec>* is required if the method overrides a method in some parent class or interface and is forbidden if the method does not override any other method. It serves as a visual reminder that there are inherited specification clauses.
- If there is a **implies_that** or **for_example** section in the *<method-spec>*, then

the initial *<behavior-seq>* is required.

- A *<clause-seq>* may be empty. This is a convenience when some or all clauses might be conditionally excluded (cf. §??). A *<behavior>* with an empty *<clause-seq>* is not the same as an absent specification.
- The grammar allows clauses to appear in any order, including after a *<nested-clause-seq>*, which permits factoring out common subsequences of clauses. However, note that the scope of an `old` clause begins with the textual position of that clause. Also, there is a preferred order for clauses (§??) that should be used where possible to enhance readability.

Note that the vertical bars in the production for *nested-clause* are literals, not metasymbols.

8.1.1 Behaviors

The basic structure of JML method specifications is as a set of *behavior*s (or *specification case*s).

Each behavior contains a sequence of *clauses*. The various kinds of clauses are described in the subsequent sections of this chapter. Each kind of clause has a default that applies if the clause is textually absent from the behavior.

The order of *<behavior>*s within a *<behavior-seq>* is immaterial; this means that specification cases are not "executed" in some order, and thus each is independent of the others. For each behavior, if the method is called in a context in which that behavior's precondition (`recommends` and `requires` clauses) is true, then the method may be assumed to adhere to the constraints specified by the remaining clauses of the behavior. Only some of the behaviors need have preconditions that are true, but unless at least one behavior has a true precondition, the call is considered to violate the method's precondition, so the method is being called in a context in which its behavior is undefined. For example, a method with an integer argument may have a specification with two behaviors, one with a precondition that states that the method's argument is not zero and the other behavior with a precondition that states that the method's argument is zero. In this case, in any context, one or the other behavior will be active. If, however, the second behavior were not specified, then it would be a violation to call the method with an argument of zero. More than one behavior may be active (have its precondition true); every active behavior must be obeyed by the method independently. Where preconditions are not mutually exclusive, care must be taken that the behaviors themselves are not contradictory, or it will not be possible for any implementation to satisfy the combination of behaviors.

The *<behavior-id>* is a easily readable keyword that states an intention of the behavior:

- **behavior** – the general case
- **normal_behavior** – a behavior in which exiting the method by an exception is forbidden; the behavior implicitly contains `signals (Exception) false;`

and `signals_only \nothing;` clauses and may not contain explicit `signals` and `signals_only` clauses.

- **exceptional_behavior** – a behavior in which exiting the method is always by a thrown exception; no exit via a usual return statement may occur. The behavior contains an implicit `ensures false;` clause and no explicit `ensures` clause is permitted.
- no *<behavior-id>* – the same as `behavior` except that no visibility modifier is permitted; the implicit visibility is the same visibility as the method declaration. Historically, a behavior without a *<behavior-id>* was called a *lightweight specification*.

The keywords ending in *iour* mean precisely the same as the corresponding keyword ending in *ior*; it is just a different spelling.

8.1.2 Nested specification clauses

Nested specification clauses are syntactic shorthand for an expanded equivalent in which clauses are replicated. The nesting syntax simply allows common subsequences of clauses to be expressed without repetition, where that improves clarity.

In particular, referring to the grammar above, a *<behavior>* whose *<clause-seq>* contains a *<nested-clause>* is equivalent to a sequence of *<behavior>*s as follows:
if $\langle\textit{nested-clause}\rangle_A$ is a combination of n *<clause-seq>* as in
{| $\langle\textit{clause-seq}\rangle_{S1}$ **also** ... **also** $\langle\textit{clause-seq}\rangle_{Sn}$ |}
then a specification of the form
(*<java-visibility>*$_V$ (**code**)?$_W$ *<behavior-id>*$_X$)? *<clause>**$_D$ $\langle\textit{nested-clause}\rangle_A$ $\langle\textit{clause-seq}\rangle_E$
is equivalent to a sequence of n *<behavior>* constructions
(*<java-visibility>*$_V$ (**code**)?$_W$ *<behavior-id>*$_X$)? *<clause>**$_D$ $\langle\textit{clause-seq}\rangle_{S1}$ $\langle\textit{clause-seq}\rangle_E$
also
...
also
(*<java-visibility>*$_V$ (**code**)?$_W$ *<behavior-id>*$_X$)? *<clause>**$_D$ $\langle\textit{clause-seq}\rangle_{Sn}$ $\langle\textit{clause-seq}\rangle_E$

This desugaring may need to be repeated if there are multiple *<nested-clause»*s within the behavior.

8.1.3 Ordering of clauses

The clauses are defined to be in the following groups:

- recommends conditions (recommends-else clauses)
- preconditions (requires, requires-else, old clauses)
- when conditions (when clauses)
- read footprint (accessible clauses)
- frame conditions (assignable clauses)
- captures conditions (captures clauses)

- model program (model program block)
- postconditions (ensures clauses)
- exceptional postconditions (signals, signals_only clauses)
- diverges conditions (diverges clauses)
- resource conditions (working_space, duration clauses)
- termination conditions (measured_by clauses)

Need to put in invariants clauses

The clauses in a behavior can be sorted into a *normal clause order* by stably sorting the sequence of clauses so that the order of groups of clauses given above is adhered to, but not changing the order of clauses within a clause group.

Any method specification has the same semantics as a method specification with a set of behaviors formed by first denesting the specification to remove any *<nested-clause>*s and then (stably) sorting the clauses within each behavior. Good style suggests always writing clauses in normal order, in so far as any nesting being used permits. Within a clause group, the order of clauses may well be important, as described in the sections about those clause kinds.

8.1.4 Specification inheritance and the code modifier

The behaviors that apply to a method are those that are textually associated with the method (that is, they precede the method definition in the `.java` or `.jml` file) and those that apply to any methods overridden by the given method. In other words, method specifications are inherited (with exceptions given below), as was described in §??.

Specification inheritance has important consequences. A key one relates to preconditions. The composite precondition for a method is the *disjunction* of the preconditions for each behavior, including the behaviors of overridden methods. Thus, just looking at the behavior within a method, one might not immediately realize that other behaviors are permitted for which the precondition is more accepting.

There are a few cases in which behaviors are not inherited:

- Since static methods are not overridden, their behaviors are also not inherited.
- Since private methods are not overridden, their behaviors are also not inherited.
- A behavior with visibility *V* is inherited if and only if a Java declaration with that visibility would be visible within the derived class. For example, `private` behaviors in a parent class are not inherited by derived classes.
- A behavior with the `code` modifier is not inherited.

The `code` modifier is unique in that it applies to method behaviors and nowhere else in JML. It is specifically used to indicate that the behavior is not inherited by overriding methods. The `code` modifier is allowed but not necessary if the behavior would not be inherited anyway. The `code` modifier is not allowed if the method does

not have a body; so it cannot be used on an abstract method declaration unless that method is marked `default` (in Java) and has a body.

If a class `P` has method `m` with a behavior that has the `code` modifier and class `D` extends `P` but does not override `m`, then an invocation of `m` on an instance of `D` executes `P.m` and is subject to the specification of `P.m` even though `P.m` has the `code` modifier. If `D` declares a method `D.m` overriding `P.m`, then the `code` modifier applies and `D.m` is not subject to any part of `P.m`'s specification with the `code` modifier; this rule applies even if `D.m` does not declare any specification behaviors of its own—as it does not inherit any behaviors, it would be given a default behavior.

Java allows a class to extend multiple interfaces. More than one interface might declare behaviors for the same method. An implementation of that method inherits the behaviors from all of its interfaces (recursively).

8.1.5 Absent vs. empty behaviors

If an overriding method has no method specification at all, then its specification is only those behaviors inherited from its parent classes and interfaces.

However, if it has an *empty* specification, that is, a behavior keyword without any clauses, the meaning is different. In this case, the empty behavior is populated with default versions of each missing clause, such as `requires true;`. This default behavior is then combined with all inherited behaviors. Suppose the parent specification's precondition put some limitations on the arguments or state in which the method is called; the overriding method's precondition now includes `requires true;` and so there is no longer any limitation on the pre-state.

Simply having some JML modifiers for a method is not sufficient to create a default (empty) specification; even a purity modifier by itself is considered an absent specification. *Check this*

8.1.6 Visibility

The following discussion has some errors and needs fixing; also need to talk about spec_public, spec_protected

Duplicates material in Ch. 3?

Each method specification behavior has a *java-visibility* (cf. the discussion in §??). Any of the kinds of behavior keywords (`behavior`, `normal_behavior`, `exceptional_behavior`) may be prefixed by a Java visibility keyword (`public`, `protected`, `private`) or by a JML visibility keyword (`spec_public` or `spec_protected`); the absence of a visibility keyword indicates package-level visibility. A lightweight behavior (one without a behavior keyword) has the same visibility as that of its associated method.

The JML visibility keywords (`spec_public` and `spec_protected`) have the same meaning as the corresponding Java visibility keywords (`public` and `protected`)

for specifications, but do not affect visibility of the Java method or names mentioned.

The visibility of a behavior determines the names that may be referenced in the behavior. The general principle is that a client that has permission to see the behavior must have permission to see the entities in the behavior. Thus

A method specification that is visible to a client may contain only names (of a type, method or field) that are themselves visible to that client.

For example, a public behavior may contain only public names (including those declared as `spec_public`). A private behavior may contain any name visible to a client that can see those private names; this would include other private entities in the same or enclosing classes, any public name, any protected name from super classes, and any package or protected name from other classes in the same package. The visibility for protected and package behaviors is more complex. A protected behavior (including those that are declared as `spec_protected`), is visible to any client in the same class or in subclasses; since the subclasses may be in a different package, the protected behavior may contain other names with protected visibility only if they are visible in the behavior by virtue of inheritance, and not if they are visible only because of being in the same package. To be explicit, suppose we have class A, unrelated class B in the same package, class C a superclass of A in a different package, and class D derived from A but in a different package, with identifiers A.a, B.b, and C.c each with protected visibility. Only A.a and C.c are visible in class D; thus a protected behavior in class A, which is visible to D, may contain A.a and C.c but not B.b. Similarly a behavior with package visibility may only contain (public names and) names that are visible by virtue of being in the same package; names with protected visibility that are visible in a class by virtue of inheritance are not necessarily visible to clients who can see the package-visible behavior, and are thus not allowed in a behavior specification with package visibility.

The root of this complexity is that protected visibility is not transitive, whereas the other kinds of Java visibility are. Conceptually, protected visibility must be separated into two kinds of visibility: protected-by-inheritance and protected-by-package. Each of these is separately transitive. Then the visibility rules can be summarized in Table ??.

8.2 Method specifications as Annotations are Obsolete

Currently, only JML modifiers have equivalent Java annotations (e.g., `pure` has the equivalent annotation `@Pure`).

(Previously, some JML tools supported an experimental implementation of specification clauses written as string arguments to Java annotations, for example, `@Requires("requires true;")`.

Table 8.1: Visibility rules for method specification behaviors

Behaviors with this visibility	may contain names that are visible in the class because of this visibility
public	public, spec_public
protected	public, spec_public, protected (by inheritance), spec_protected
package (blank)	public, spec_public, protected (by package), spec_protected (by package), package (blank)
private	any

However, this use of Java annotations is no longer defined in JML or in current tools.)

8.3 Common JML method specification clauses

There are quite a few kinds of method specification clauses. Those clauses described in this section are the more commonly used ones. The following section (??) describes the others, some of which are still research ideas.

8.3.1 `requires` clause

Grammar:

```
<requires-clause> ::= requires <opt-name> <jml-expression>
        [ else <qual-ident> ];
```

Type information: The *<jml-expression>* in a *<requires-clause>* must have `boolean` type. Names in the *<jml-expression>* are resolved as if the expression appeared as the first expression in the body of the method; that is, the formal and type parameters of the method and anything visible in the body of the enclosing class are all in scope. Also, any name declared in an *<old-clause>* (§??) prior to the *<requires-clause>* in the same specification case is also in scope and hides any other names. If an `else` suffix is present, the *<qual-ident>* must name a Java class derived from `java.lang.Exception`.

The disjunction of the effective preconditions from each of the behaviors (including any inherited ones) is the *effective precondition* of the entire method specification; the effective precondition must be true when the method is called (so at least one of the behaviors must have a true precondition); when verifying the body of a method, the effective precondition is assumed to be true at the beginning of the method body.

The default requires clause is `requires true;`, which puts no requirements on the caller of the method. Thus, the requires clause may be omitted entirely when the precondition is true.

For a given specification case (behavior) with multiple requires clauses, the effective

precondition of that specification case is the conjunction of the predicates from each `requires` clause, in their order of specification. The order of these requires clauses within a specification case is significant in the same way that the order of terms in a short-circuit boolean expression is significant: earlier *<requires-clause>* expressions may state conditions that enable later ones to be well-defined. In addition the order of `old` clauses with respect to `requires` clauses is significant. However, any requires-else clause just contributes `true` to the specification case's effective precondition. Also, any `recommends` clause (§??) contributes its predicate as a conjunct in the specification case's effective precondition, but only for the caller of the method (not for inheritance).

If a `else` suffix is present, the clause is desugared as described in §??.

8.3.2 `ensures` clause

Grammar:

<ensures-clause> `::=` **`ensures`** *<opt-name>* *<jml-expression>* `;`

Type information: The *<jml-expression>* in a *<ensures-clause>* must have `boolean` type. Names in the *<jml-expression>* are resolved as if the expression appeared as the first expression in the body of the method; that is, the formal and type parameters of the method and anything visible in the body of the enclosing class are all in scope. Also, any name declared in any *<old-clause>* in the same specification case is also in scope and hides any other names. Ensures clauses may also use the `\result` expression (cf. §??) and `\old` expressions (§??).

An `ensures` clause states a postcondition for a method. That is, the given predicate must be true just at any `return` statement in the method body (with any returned value substituted for `\result`) and may be assumed by the caller after the call. Note that the semantics is that *if the method returns normally, then the postcondition holds* (which assumes that the method has been verified). The converse, (namely, "if the postcondition holds then the method terminated normally"), is not necessarily true.

The conjunction of implications, with each behavior's effective precondition implying the corresponding effective postcondition) from each of the behaviors (including any inherited ones) is the *effective postcondition* of the entire method specification; the effective postcondition may be assumed to hold after a method call. That is, a caller may pick the specification cases that will be used in the call, and by making those preconditions true, the caller may assume that all of the corresponding postconditions hold after the call. When verifying the body of a method, the effective postcondition must hold whenever the method terminates.

As an example, consider the specification (shown in Fig. ??) of a method `safeDiv`, which refers to a `spec_public` but Java private field `ans` declared in the surrounding class.

The effective postcondition of the `safeDiv` method is the following JML expression:

```
/*@
    requires y != 0;
    ensures ans == x/y;
  also
    requires y == 0;
    ensures ans == \old(ans);
@*/
void safeDiv(int x, int y);
```

Figure 8.1: A method specification with two behaviors

```
((y != 0) ==> ans == x/y) && ((y == 0) ==> ans == \old(ans))
```

There may be more than one `ensures` clause in a specification case. The effective postcondition of a specification case with multiple ensures clauses is the conjunction of the *<jml-expression>*s in the order in which they were specified. The order of `ensures` clauses in a specification case is thus significant in the same way that the order of terms in a short-circuit boolean expression is significant: earlier *<jml-expression>*s may state conditions that enable later ones to be well-defined. The effective precondition for a specification case can also be used to make the expressions in a postcondition be well-defined.

The default ensures clause is `ensures true;`, which puts no requirements on the body of the method.

8.3.3 `assignable` clause

Grammar:
TODO

8.3.4 `signals` clause

Grammar:

<signals-clause> ::=
    signals *<opt-name>* (*<name>* [*<ident>*]) *<jml-expression>* ;

Type conditions: The *<name>* in the parentheses must be the name of a class derived from `java.lang.Exception`. The *<jml-expression>* must be a boolean expression. The identifier is declared to have the type of the exception and is in scope only within the *<jml-expression>*. The identifier may be omitted if it is not needed in that expression.

A `signals` clause states a condition that must be true if a method exits by throwing an exception of the given type (or a subtype of that type, such as an exception derived from it).

The semantics is that *if the method terminates by throwing an exception of that type, then the expression must be true*. The converse, (namely, "if the predicate is true then the method terminates with the given exception"), is not necessarily true. Furthermore, a given signals clause says nothing about what other exceptions may be thrown. To state a limitation on what exceptions that may be thrown, use a `signals_only` clause.

There may be more than one `signals` clause in a specification case. There is no significance to their order. However, when one type of exception specified is a subtype of another, then one must consider that an exception that has the subtype will need to satisfy both signals clauses. For example, suppose exception class `DE` is a subtype of `CE`, and consider the method `barf` shown in Fig. **??**. To verify an implementation

```
/*@ signals(CE e) e.f > 5;
    signals(DE g) g.f > 3; @*/
void barf() throws CE;
```

Figure 8.2: A specification of a method with two signals clauses. In this example assume that `DE` is a subtype of `CE`.

of this specification that throws an exception of the subtype `DE`, the exception object would need to have a field `f` with a value strictly greater than 5, since that exception object would also have the supertype `CE`.

The default `signals` clause is `signals (Exception) true;`, which puts no requirements on the body of the method.

8.3.5 `signals_only` clause

Grammar:

```
<signals-only-clause> ::=
    signals_only <opt-name> ( \nothing | <name> ( , <name> )* );
<name> ::= <ident> ( . <ident> )*
```

Type information: The possibly-qualified names in the clause must denote (resolve to) Java types derived from `java.lang.Exception`. The names are resolved just like any other type name in a Java program, using names in scope at the point of the method declaration.

A *<signals-only-clause>* specifies that, when the effective precondition of that specification case holds, only the listed Java Exceptions (or subtypes of those exception types) may be thrown. That is, if the method terminates with an exception, the thrown exception must have one of the listed exception types or be derived from one of the listed exception types. The token `\nothing` denotes an empty list (no exceptions may be thrown). In contrast to the Java throws list, if an exception type derived from `java.lang.RuntimeException` may be thrown by the specified method, then it must be explicitly listed.

There is no point to listing checked exception types in a *<signals-only-clause>* that are not (subtypes of those types) listed in the Java `throws` clause, as the Java compiler will complain about them if they are actually thrown by the code. On the other hand, the *<signals-only-clause>* allows specifying fewer exceptions or none at all for a given specification case. For example, a method may be expected to terminate normally (i.e., with the specification `signals_only \nothing;`) under one set of preconditions, while terminating with an exception under other preconditions.

The default *<signals-only-clause>* lists all the exceptions that are in the Java method declaration's throws clause plus `java.lang.RuntimeException`.

Note that the exceptions listed in a *<signals-only-clause>* have an effect on the use of `allow` and `forbid` annotations (§??).

8.4 Advanced JML method specification clauses

These clauses are less commonly used and may be less-well-supported by tools.

8.4.1 `invariants` clause

TODO

8.4.2 `recommends` clause

Grammar:

```
<recommends-clause> ::= requires <opt-name> <jml-expression>
       else <qual-ident> ;
```

Type information: The *<jml-expression>* in a *<recommends-clause>* must have `boolean` type.

Names in the *<jml-expression>* are resolved as if the expression appeared as the first expression in the body of the method; that is, the formal and type parameters of the method and anything visible in the body of the enclosing class are all in scope. The `else` suffix is required; its *<qual-ident>* must name a Java class derived from `java.lang.Exception`. If a specification-case has more than one `recommends` clause, the predicate in each clause must be well-defined independently and independent of any `requires` clause. Also, `recommends` clauses may not use any names declared in `old` clauses.

The motivation and use of the `recommends` clause is described in §??.

8.4.3 `accessible` clause

Grammar:

TODO

8.4.4 `diverges` clause

Grammar:

<diverges-clause> ::= **diverges** *<opt-name>* *<jml-expression>* ;

Type information: The *<jml-expression>* in a *<diverges-clause>* must have `boolean` type. Names in the *<jml-expression>* are resolved as if the expression appeared as the first expression in the body of the method; that is, the formal and type parameters of the method and anything visible in the body of the enclosing class are all in scope. Also. any name declared in an *<old-clause>*(§??) in the same specification case is also in scope and hides any other names. The *<jml-expression>* is evaluated in the method's pre-state.

When a `diverge` clause is omitted in a specification case, a default clause is used; the default `diverges` condition is `false`. Thus by default, specification cases give total correctness specifications [?]. Explicitly writing a diverges clause allows one to obtain a partial correctness specification [?].

As an example of the use of `diverges`, consider the `abort` method in the following example. This example is simplified from the specification of Java's `System.exit` method. This specification says that the method can always be called (the implicit precondition is true), is always allowed to not return to the caller (i.e., diverge), may never return normally, and may never throw an exception. Thus the only thing the method can legally do, aside from causing a JVM error, is to not return to its caller.

```
package org.jmlspecs.samples.jmlrefman;
public abstract class Diverges {

    /*@ public behavior
      @    diverges true;
      @    assignable \nothing;
      @    ensures false;
      @    signals (Exception) false;
      @*/
    public static void abort();
}
```

The diverges clause is useful to specify things like methods that are supposed to abort the program when certain conditions occur, although such behavior is not really good practice in Java. In general, it is most useful for examples like the one given above, when you want to say when a method cannot return to its caller.

Having the default diverges clause be `diverges true` instead of `false` would be the most conservative and the more sound choice. With the chosen default there is the risk that a library function that does not terminate is incorrectly presumed to do. However, `diverges false` is by far the more common behavior and so is chosen as the default.

8.4.5 `measured_by` clause

Grammar:

```
<measured-by-clause> ::=
        <measured-by-keyword> <opt-name> <jml-expression> ;
<measured-by-keyword> ::= measured_by
```

Type information: The *<jml-expression>* in a *<measured-by-clause>* must have `\bigint` type but must always be non-negative. Names in the *<jml-expression>* are resolved as if the expression appeared as the first expression in the body of the method; that is, the formal and type parameters of the method and anything visible in the body of the enclosing class are all in scope. Also, any name declared in an *<old-clause>*(§??) in the same specification case is also in scope and hides any other names. The *<jml-expression>* is evaluated in the method's pre-state.

The default measured-by clause for a method which has integer parameters is the sum of those parameter values. If a method does not have any integer parameters, then the measure defaults to `0`.

The *<measured-by-clause>* is used to prove termination of recursions. This is done by decorating the activation records for calls with the measure specified in the *<jml-expression>* specified. Each call of method generates an *activation record* which is part of the context of a program that records the measures for each method that has been called. A call to a method `m` must either not be a recursive call (if there is no measure for `m` in the current context) or it must strictly decrease `m`'s measure compared to its measure in the current context. For example, if a method `f` calls itself, then its recursive calls must strictly decrease the value of the measured-by expression computed at the beginning of `f`'s body without making the measured-by expression for the call become negative. When several methods are mutually recursive, then calls to each method must decrease the measure of the method being called; one way to do this is to derive these measures from data that is passed from one such call to another.

A recursive method must terminate in each specification case for which that behavior's diverges clause is false. (Note that pure methods are not allowed to have a diverges clause.)

The measures in JML must be non-negative integers. However, this is not sufficiently expressive for all methods.
[[[We should add another clause, say `decreasing`, that is more like what is in Dafny...]]]

8.4.6 `when` clause

Grammar:

```
<when-clause> ::= <when-keyword> <opt-name> <jml-expression> ;
<when-keyword> ::= when  | when_redundantly
```

Type information: The *<jml-expression>* in a *<when-clause>* must have `boolean` type.

Names in the *<jml-expression>* are resolved as if the expression appeared as the first expression in the body of the method; that is, the formal and type parameters of the method and anything visible in the body of the enclosing class are all in scope. Also, any name declared in an *<old-clause>*(§??) in the same specification case is also in scope and hides any other names. The *<jml-expression>* is evaluated in the method's pre-state.

The `when` clause allows concurrency aspects of a method or constructor to be specified [?, ?]. In a program with concurrent executions, a caller of a method may be delayed, for example, by a locking condition. What is checked is that the method does not proceed to its commit point, which is the start of execution of a statement with the label `commit`, until the given predicate is true.

When a `when` clause is omitted in a specification case, a default clause is used, in which the *<jml-expression>* is `true`.

See [?] for more about the `when` clause. This clause is meant to be used in multi-threaded programs, which JML does not currently support.

8.4.7 `old` clause

Grammar:

<old-clause> `::=` **`old`** *<jml-var-decl>*

Type conditions: The clause declares and initializes a variable. The initializer must evaluate to a value of a type that can be assigned to the newly declared name. The initializer is evaluated in the method's pre-state.

An **old** clause declares and initializes a single named constant. The initializer for the declared constant is evaluated in the method's pre-state. The scope of the newly declared name is the remainder of the *<behavior>* or *<nested-clause-seq>* in which the declaration occurs. Like Java declarations, the new variable name hides variables of the same name, including instances of those names in the new variable's initializer; however, the new variable may not be used in the initializer, because it is not yet initialized.

Any declaration in the *<old-clause>* clauses must textually precede any uses of the declared variables.

The order of the *<old-clause>* with respect to any *<requires-clause>*s in the same *<behavior>* is significant: the initializer of the *<old-clause>* must be well-defined given that any textually preceding `requires` clauses hold.

The purpose of the clause is to capture and name the value of a subexpression that is used more than once in a behavior, or just to abbreviate long expressions or to make them clearer.

8.4.8 `duration` clause

Grammar:

```
<duration-clause> ::= <duration-keyword> <opt-name> <expression>
        [ if <predicate> ] ;
<duration-keyword> ::= duration | duration_redundantly
```

Type information: The *<expression>* in the `duration` clause has type `\bigint`; the optional *<predicate>* has boolean type.

A duration clause is used to specify the maximum (i.e., worst case) time needed to process a method call in a particular specification case. This is adapted from the work of Krone, Ogden, and Sitaraman on RESOLVE [?].

The expression is to be understood in units of the JVM instruction that takes the least time to execute, which may be thought of as the JVM's cycle time. The time it takes the JVM to execute such an instruction can be multiplied by the number of such cycles to arrive at the clock time needed to execute the method in the given specification case. This time should also be understood as not counting garbage collection time.

The expression in a `duration` clause is evaluated in the post state and thus may use `\old`, `\result`, and other JML operators appropriate for postconditions.

In any specification case, an omitted `duration` clause means the same as a duration clause giving an infinite amount of time.

See §?? for information about the `\duration` expression, which can be used in the duration clause to specify the duration of other methods.

8.4.9 `working_space` clause

Grammar:

```
<working-space-clause> ::=
    <working-space-keyword> <opt-name> <predicate>
        [ if <expression> ] ;
<working-space-keyword> ::= working_space
      | working_space_redundantly
```

Type information: The expression in a working space clause must have type `\bigint`, in units of bytes. The optional *<predicate>* has boolean type.

A *<working-space-clause>* can be used to specify the maximum amount of heap space used by a method, over and above that used by its callers. The clause applies only to the particular specification case it is in. This is adapted from the work of Krone, Ogden, and Sitaraman on RESOLVE [?].

The expression is evaluated in the post-state and thus may use `\old`, `\result`, and other JML operators appropriate for postconditions. In some cases this space may depend on the `\result`, exceptions thrown (`\exception`), or other post-state values.

An omitted working space clause makes no guarantees of the amount of space used; it is equivalent to a clause specifying an infinite number of bytes.

See §?? for information about the `\working_space` expression that can be used to describe the working space needed by a method call. See §??, for information about the `\space` expression that can be used to describe the heap space occupied by an object.

8.4.10 `callable` clause

Grammar:

```
<callable-clause> ::= callable <opt-name>
      ( \nothing
      | <method-signature> ( , <method-signature> ) +
      ) ;
<method-signature> ::= [ <type-name>. ] <java-identifier>
      [ ( <type-name> ... ) ]
```

Type information: Each *<method-signature>* must name a unique method; that is, a method in a particular class. If no *<type-name>* is given, the *<identifier>* names a method in the enclosing class; otherwise it must name a method in the named class. If the method name is not unique in its class, then the types of its arguments must be listed in exact correspondence to the declaration of the method.

The semantics is that every direct call appearing in a method must be to one of the methods methods listed in the clause. The term `\nothing` denotes an empty list of method signatures.

What about generic methods? [[[The syntax for method names shoudl be adjusted to allow for (possibly overloaded) methods of generics.]]]

I expect that just as model fields and locsets are needs to abstract sets of modified memory locations, so kind of abstraction grouping called methods is needed to reason about method call sets. [[[These callable clauses could be restricted to only appear in code contracts, but you might still be right, and if so, then we would need a way to specify such sets of method names.]]]

8.4.11 `captures` clause

Grammar:

```
<captures-clause> ::=
    <captures-keyword> <opt-name> <expression> ... ;
<captures-keyword> ::= captures | captures_redundantly
```

TODO

8.4.12 `returns` clause

Grammar:

```
<returns-clause> ::= returns [ <jml-expression> ];
```

Static checks:

- A *<returns-clause>* may only be used within a *<block-specification>* (cf. §??).
- The expression if present must be well-defined. and have boolean type.

The default for an absent expression is `true`. If there is no *<returns-clause>* at all, the default clause is `returns false;`.

The meaning of a *<returns-clause>* is that if the statement that implements the statement (block) specification executes a `return`, then the given predicate (if any) must hold in the state after the return value is evaluated and before the return itself is executed. The expression `\result` may be used in the given expression if the method enclosing this specification returns a value.

8.4.13 `throws` clause

Grammar:

```
<throws-clause> ::= throws [ <jml-expression> ];
```

Static checks:

- A *<throws-clause>* may only be used within a *<block-specification>* (cf. §??).

Type information: If an expression is present, it must be well-defined and have type boolean.

The default for an absent expression is `true`. If there is no *<throws-clause>* at all, the default clause is `throws false;`.

The meaning of a *<throws-clause>* is that if a statement that implements a block statement specification executes a `throw`, then the given predicate (if any) must hold in the state after the thrown object is evaluated and before the throw statement itself is executed. The expression `\exception` may be used in the *<jml-expression>*.

8.4.14 `continues` clause

Grammar:

```
<continues-clause> ::=
       continues [ <ident> :  ] [ <jml-expression> ];
```

Static checks:

- A *<continues-clause>* may only be used within a *<block-specification>* (cf. §??).
- The expression if present must be well-defined and have boolean type.

- The target label must be outside the scope of the statement that the enclosing the block-specification applies to.

The target for an absent target label is the innermost loop enclosing the block-statement being specified.

The default for an absent expression is `true`. If there is no *<continues-clause>* for a given label `L`, the default clause is `continues L: false;`.

The meaning of a *<continues-clause>* for a label `L` is that if the statement that implements the statement (block) specification executes a `continue L`, then the given predicate (if any) must hold in the state just before the continue is executed.

There can be a `continues` clause for each label in scope; these are all independent. If there is no explicit clause for a given label, then the default applies, meaning that control flow will never continue to that label.

8.4.15 `breaks` clause

Grammar:

<breaks-clause> ::=
breaks [*<ident>* :] [*<jml-expression>*] ;

Static checks:

- A *<breaks-clause>* may only be used within a *<block-specification>* (cf. §??).
- The expression if present must be well-defined and have boolean type.
- The target label must be outside the scope of the statement the block-specification applies to.

The target for an absent target label is the innermost loop enclosing the block-statement being specified.

The default for an absent expression is `true`. If there is no *<breaks-clause>* for a given label `L`, the default clause is `breaks L: false;`.

The meaning of a *<breaks-clause>* for a label `L` is that if the statement that implements the statement (block) specification executes a `break L`, then the given predicate (if any) must hold in the state just before the continue is executed.

There can be a `breaks` clause for each label in scope; these are all independent. If there is no explicit clause for a given label, then the default applies, meaning that control flow will never break to that label.

8.5 Modifiers for method specifications

8.5.1 Purity modifiers

The purity modifiers are `pure`, `spec_pure`, `strictly_pure`, `no_state`, `@Pure`, `@SpecPure`, `@StrictlyPure`, and `@NoState`. At most one of these may be applied to a method declaration. If none are applied to a method declaration, the method inherits its purity from its super classes and interfaces; if none of those designate purity, the method gets its purity designation from the innermost enclosing class that does have one. If a method overrides a some ancestor's method, the method must have a purity designator at least as strong as that of the overridden class.

The different levels of purity are described in §??. All the purity designations require that the method not assign to any field in the pre-state of the call. That is purity implies `assignable \nothing;` in all specification cases.

8.5.2 `non_null`, `nullable`, `@NonNull`, and `@Nullable`

The types of methods can be annotated as `non_null` or `nullable`. Because these are *type annotations*, the modifier is attached to the type, not to the declaration per se. This complicates the syntax a bit. See §?? and §?? for details.

8.5.3 `non_null_by_default`, `nullable_by_default`, `@NonNullByDefault`, `@NullableByDefault`

The `non_null_by_default` and `nullable_by_default` modifiers or, equivalently, the `@NonNullByDefault` and `@NullableByDefault` Java annotations, specify the default nullity declaration within the method. Nullness is described in §??. The default applies to all typenames in declarations and in expressions (e.g. cast expressions), and recursively to any local classes in the method that do not have default nullity declarations of their own.

These default nullity modifiers are not inherited by overriding methods.

A method cannot be modified by both modifiers at once. If a method has no default nullity modifier, it uses the corresponding modifiers of the enclosing class; the default for a top-level class is `non_null_by_default`. This top-level default may be altered by tools.

8.5.4 `model` and `@Model`

This modifier identifies a method declared within a JML annotation as a *model method*. A method may not have a `ghost` modifier. Model methods are discussed in §??.

8.5.5 `spec_public`, `spec_protected`, @SpecPublic, and @SpecProtected

These modifiers apply only to methods declared in Java code, and not to methods declared in JML, such as model methods. They have the effect of replacing the Java visibility modifier for the method with an alternate, for the purposes of specification. For example, a Java declaration declared `private` in Java, but also `spec_public` in a JML modifier for the method, is treated as `public` in all specifications.

8.5.6 `helper` and @Helper

The `helper` modifier states that a method may not assume that the invariants, constraints, and initially clauses of the containing class hold and it need not establish that they are true at the end of the body of the method. See the discussion of when invariants are required to hold, in §??.

Typically `helper` methods are internal utility methods and are often (though not necessarily) private.

May helper methods be overridden, if they remain helper

8.5.7 `no_state` and @NoState

A method marked as `no_state` is one whose result is independent of the state of the heap (a claim that tools should check). A `no_state` method is implicitly also `strictly_pure`. A `no_state` method is one that computes some result based only on its inputs and using just Java primitive or JML types. Such a method may also use Java static final constants whose types are Java primitive types; if the method uses a static final value that is not a compile-time constant, the method has an implicit precondition that the class containing that value has completed initialization.

The value of such methods is precisely that they are independent of the state of the heap and thus are much easier to reason about.

This duplicates discussion of purity, now that no_state is a kind of purity.

It seems that values of immutable types, such as Java Strings or enums or records, could also be considered as heap-independent. One would need to check that any final (constant) references are not just a container for mutable references, and also that their values are only used once initialization is complete.

8.5.8 Arithmetic modes: `code_java_math`, `spec_java_math`, `code_bigint_math`, `spec_bigint_math`, `code_safe_math`, `spec_safe_math`, @CodeJavaMath, @CodeSafeMath, @CodeBigintMath, @SpecJavaMath, @SpecSafeMath, @SpecBigintMath

In JML, arithmetic can be analyzed in three different modes: java mode, safe mode, and bigint mode, as described in detail in §??. The arithmetic mode can be different for specifications and Java code. These method modifiers permit setting the analysis mode for the body and specifications of the method they modify. They apply to the whole body, though the arithmetic mode scoping operators (§??) may change the mode for subexpressions.

8.5.9 `skip_esc`, `skip_rac`, @SkipEsc, and SkipRac

These modifiers apply only to methods with bodies.

When these modifiers are applied to a method or constructor, static checking (respectively, runtime checking) is not performed on that method. In the case of RAC, the method will be compiled normally, without inserted checks. These modifiers are a convenient way to exclude a method from being processed without needing to remember to use the correct command-line arguments.

8.5.10 `@Options`

This is perhaps too tool-specific

I think this duplicates discussion elswhere. This Java annotation applies to class or method declarations. It is available only as a Java annotation (not as a JML modifier).

The annotation takes either a string literal or a { }-enclosed list of string literals as its argument. The literals are interpreted as individual command-line arguments, optionally with a = and a value, which set options used just for processing the class or method declaration that the annotation modifies. Not all command-line arguments are applicable to individual classes or methods.

This is useful when there is not a built-in modifier for a particular option. For example, one could write

```
1 public void TestOption {
2   @org.jmlspecs.annotation.Options({"-progress","-timeout=1"})
3   public void m() {
4     //@ assert false;
5   }
6 }
```

Here method `m` is processed with the given timeout and verboseness level, despite the settings used elsewhere. In the first case, the strings are enclosed in braces,

while in the second case, the single string does not need enclosing braces. Note that the prefix `org.jmlspecs.annotation.` may be omitted if the appropriate import is used (e.g., `import org.jmlspecs.annotation.Options;` or `import org.jmlspecs.annotation.*;`. The `@Options` annotation is in Java code, so a library containing `org.jmlspecs.annotation` must be on the classpath when a class using `Option` is compiled or executed.

8.5.11 `extract` and @Extract

The `extract` modifier for methods relates to specification inference for model programs, as described in [?].

This modifier applies only to methods with bodies.

Much more needs to be said somewhere.

8.5.12 `peer`, `rep`, and `readonly`

These modifiers relate to Universe types, which are described in §??.

8.5.13 `inline`

The `inline` modifier is an experimental feature. When applied to a method declaration, it indicates to tools that, for static reasoning purposes, the method body should be inlined where it is used. Use of `inline` on methods that are neither `final` nor `private` is disallowed. (That is, inlined methods may not be overridable.) Note that a method may be declared `final` in JML even if it is not so declared in Java.

The advantage of inlining small methods is that the method then needs not have any specifications, saving the work of writing them. Any specifications it does have must still be checked at the call site along with the inlined code.

Are inline constructors allowed

8.5.14 `query`, `secret`, @Query, and @Secret

The `secret` modifiers on fields are used with the `query` modifiers on methods in specifying observational purity. This is an experimental and not yet settled feature.

8.6 TODO Somewhere

`<::` token

lots more backslash tokens

Chapter 9

Field and Variable Specifications

Fields may have various modifiers, each of which states a restriction on how the field may be used. Fields may be part of *data groups*, which allow specifying frame conditions on fields that may not be visible because of the Java visibility rules. Also, a specification may introduce *ghost* or *model* fields that are used in the specification but are not present in the Java program.

Local variables and formal parameters, typically declared and used in method bodies, have properties similar to fields and many of the same modifiers.

9.1 Field and Variable Modifiers

The modifiers permitted on a field, variable, or formal parameter declaration are shown in a table in the Appendix (§??).

9.1.1 `final`

The `final` modifier is a Java modifier used to indicate that a field or variable will not be assigned to after it is initialized. This is helpful information in both reading code and in reasoning about it as one knows the value does not change through subsequent state changes in the program's execution.

However, programmers rarely notate a field or variable as `final` unless it is in a situation where Java requires it. So it is useful to be able to so as part of the JML specifications, as in `/*@ final */ public int x;`.

The effect of a `final` modifier (whether in Java or JML) on a field, variable or formal parameter is that outside of initializers and constructor bodies

- any assignment to the field is a JML error;
- the memory location may not be listed in any frame condition clause or expression;
- the field location is not part of wild-card storerefs, like `this.*` and `\everything`.

If the field is `static`, then its value is fixed at the end of static initialization; the above rules about frame conditions also apply.

9.1.2 `non_null` and `nullable` (`@NonNull`, `@Nullable`)

The `non_null` and `nullable` *type* modifiers, and equivalent `@NonNull` and `@Nullable` annotations, specify whether or not a field, variable, or parameter may hold a null value. The modifiers are valid only when the type of the modified construct is either a reference or array type, not a primitive type.

Note that because nullness modifiers are actually type modifiers, in a declaration like `non_null Object x, y;`, both `x` and `y` have `non_null Object` type.

More detailed discussion of nullness in given in §??.

9.1.3 `spec_public` and `spec_protected` (`@SpecPublic`, `@SpecProtected`)

These modifiers are used to change the visibility of a Java field when viewed from a JML construct. A construct labeled `spec_public` has `public` visibility in a JML specification, even if the Java visibility is less than public; similarly, a construct labeled `spec_protected` has `protected` visibility in a JML specification, even if the Java visibility is less than protected. §?? contains a detailed discussion of the effect of information hiding using Java visibility on JML specifications.

Listing 9.1: Use of spec_public

```
1 private /*@ spec_public */ int value;
2
3 //@ ensures value == i;
4 public setValue(int i) {
5   value = i;
6 }
```

For example, Listing ?? shows a simple setter method that assigns its argument to a private field named `value`. The visibility rules require that the specifications of a public method (`setValue`) may reference only public entities. In particular, it may not mention `value`, since `value` is private. One solution is to declare, in JML, that `value` is `spec_public`, as shown in the Listing.

9.1.4 `ghost` and `@Ghost`

Ghost fields and variables are described in §??. Formal parameters may not be `ghost`.

9.1.5 `model` and `@Model`

Model fields are described in §??. Variables and formal parameters may not be `model`.

9.1.6 `uninitialized` and `@Uninitialized`

The `uninitialized` modifier may be used on a field or local variable declaration to say that despite an initializer, the location declared is to be considered uninitialized. Thus, the field should be assigned in each path before it is read. [?]

Java requires that locations be assigned before they are read, to avoid uninitialized memory bugs. Consequently the Java compiler does a static analysis and will complain if it finds code paths on which a variable is read before being written. It may well be that such a path is indeed not feasible, but it will take a more complex reasoning and perhaps additional JML annotations to prove that fact. This modifier allows the Java program to have an initializer in the declaration in order to satisfy the Java compiler, but still requires a JML-based proof that the variable is always written before being read.

9.1.7 `instance` and `@Instance`

The JML `instance` modifier is the opposite of the Java `static` modifier; that is, an `instance` entity is an object instance of a class (with a different entity for each object instance), whereas a `static` entity is a member of the class (and is the same entity for all object instances of that class).

It does no harm to declare a non-static JML field as `instance`, but the only time it is necessary is in an interface, as fields are by default static in an interface. It is common, however, to declare some instance model fields in an interface that are used by specifications in the interface and inherited by derived classes.

Obviously, it is a type error to declare a field both `instance` and `static`.

```
1 public interface MyCollection {
2   //@ model instance int size; // a public instance JML field
3   final int MAX = 100;  // a public static Java field
4 }
```

9.1.8 `monitored` and `@Monitored`

The `monitored` modifier may be used on a non-model field declaration to say that a thread must hold the lock on the object that contains the field (i.e., the `this` object containing the field) before it may read or write the field [?].

9.1.9 `peer`, `rep`, `readonly`, `@Peer`, `@Rep`, `@Readonly`

These modifiers are part of the Universe type system (cf. §??). Their interaction with the rest of JML has yet to be fully worked out.

Check readonly vs. read_only, Readonly vs. ReadOnly TODO

9.1.10 `query`, `secret` and `@Query`, `@Secret`

The `secret` modifiers on fields are used with the `query` modifiers on methods in specifying observational purity. This is an experimental and not yet fully fleshed out feature.

TODO

9.2 Ghost fields

Ghost fields are in all respects like Java fields, except that they are not compiled into the Java program (because the declarations are in JML, which are Java comments). However they are compiled into the output programs for runtime-assertion checking. They can also be reasoned about in static checking just like any Java field.

Within a program, ghost fields are assigned to in `set` statements (§??).

- a JML field must be one of either `ghost` or `model`, and not both
- a ghost field in an interface must be static

Ghost fields are discussed in §??.

9.3 Model fields

Model fields are described in §??.

- a JML field must be one of either ghost or model, and not both
- a non-final model field may not have an initializer. Rather, the value of a model field is constrained by specifications, including `represents` clauses — §??.

9.4 Datagroups: `in` and `maps` clauses

There are two JML clauses specifically associated with field declarations; the `in` and `maps` clauses. These clauses must immediately follow the declaration of the field to which they apply.

The `monitor_for` and `represents` clauses also relate to specific fields, but those clauses may appear where any declaration within a class body appears.

9.4.1 `in` clause

<in-clause> ::= **in** *<ident>* ... ;

Type information: The *<ident>*s listed in the clause must name datagroups (which includes model fields) that are members of the containing class or any of its superclasses or interfaces. The listed datagroups must be visible to any client for whom the field associated with the clause is visible.

The `in` clause states that its associated field belongs to the datagroups listed in the clause. A typical use is to declare a public datagroup and then add to it various private fields of the class. Then various frame conditions can mention the datagroup name, which then includes all the private memory locations which would not otherwise be visible.

Here is some example code.

```
class A {
  //@ public \datagroup state;

  private int k; //@ in state;
}
```

9.4.2 `maps` clause

<maps-clause> `::=` **`maps`** *<storeref>* `\into` *<identifier>* `...` `;`

Type information:

- Each *<identifier>* must be a datagroup (including model fields).

The `maps` clause states that the given *<storeref>* is a member of each of the given datagroups.

Allow a list of storerefs?

Why is this associated with a declaration?

Allow null dereferences in the storerefs

Add examples - particularly using array elements

Chapter 10

Default specifications and specification inference

10.1 Static invariants

Some, if not most, of the static fields of a class are static final fields with constant initializers. The values of these fields are invariant after the completion of static initialization. As it is tedious, duplicative and error prone to write an explicit invariant stating these facts, JML infers such a specification.

Each class has a static invariant consisting of a conjunction of the equalities *field* == *value* for each *static, final* field that is initialized with a compile-time constant. If necessary there is one such invariant for each level of visibility (usually only a public one is needed).

Note that if a field is not declared `final` it is not included; if the field is actually final but not declared so in Java, it can be declared `final` in JML. Also, if the field does not have a compile-time constant initializer, it is not included in the invariant.

Either of these omissions can lead to a field being silently omitted from an initializer and thereby causing confusion, because it is not readily clear why an 'obvious' fact — that the field has its initializer as its value — is not known by the prover.

10.2 Instance invariants

Just like for static initializers, a set of instance initializers (one for each visibility) is created by conjoining equalities for each instance field that is `final` and has a compile-time-constant initializer.

10.3 Method specifications

A default specification for a Java method is assumed wherever that method (a) is a library method with no source code or specification file, (b) has Java source code but no explicit specifications, or (c) is an implicit (compiler constructed) method.

10.3.1 Non-overridden methods

A method that does not override any methods of parent classes and does not specify any behavior and is not marked `pure` has this default behavior: *This mention of pure here differs from the behavior described in §??).*

```
requires true;
accessible \everything;
assignable \everything;
captures \everything;
callable \everything;
ensures true;
signals (Exception e) true;
signals_only RuntimeException + the Java throws declaration;
diverges true;
when true;
measured_by <very large number>;
duration <very large number>;
working_space <very large number>
```

In addition, the method is by default `volatile`. *Make reference to discussion of determinism.*

If the method has a `pure` modifier then the default is

```
requires true;
accessible \everything;
assignable \nothing;
captures \everything;
callable \everything;
ensures true;
signals (Exception e) true;
signals_only RuntimeException + the Java throws declaration;
diverges true;
when true;
measured_by <very large number>;
duration <very large number>;
working_space <very large number>
```

In addition, the method is by default not `volatile`. *Make reference to discussion of determinism. Is this default an unsoundness.*

The visibility of these default behaviors is the same as the method itself.

Such behaviors are about as conservative as it is possible to be, with just the exception that the specification only allows a `java.lang.RuntimeException` or any checked excption in the method's `throws` clause to be thrown, and not any other kind of unchecked `Throwable`, such as a `java.lang.Error` (including `java.lang.AssertionError`. The rationale for this restriction is that JML make no guarantees about a program's behavior (whatever verification was successfully performed) if an `Error` is thrown — most `Error` exceptions are program faults (e.g. out of memory or stack overflow) from which it is difficult to perform meaningful recovery action These default behaviors are sound (baring program `Error`s) but are too general to be useful. Any method implementation at all can be verified against these postconditions, but no method that called such a method (and relied on its behavior) could be verified to do anything. Consequently, users are advised to provide actual specifications for any method that is called.

Tools may help by (a) warning about methods without specifications or (b) inferring better specifications (§??) or (c) providing options that enable more useful if unsound defaults.

10.3.2 Overriding methods

A method that overrides a method from a parent class or interface inherits all the behaviors (recursively) from its superclasses and interfaces. There will be at least the default behavior of the top-most method in the overriding hierarchy. If the method does not have any specification clauses of its own, it does not add any behaviors to those it inherits; if it has behaviors of its own, those are concatenated with the inherited behaviors.

In addition, an overriding method inherits any purity modifier (§??) if any method it overrides is marked with a purity modifier.

A method may add its own modifiers (e.g., `spec_public`, `spec_protected`, `helper`) independently of its inheriting behaviors.

10.3.3 Library methods

To ensures soundness, the defaults for library methods without either source code or explicit specifications are these:

```
1  requires true;
2  accessible \everything;
3  assignable \everything;
4  captures \everything;
5  callable \everything;
6  ensures true;
7  signals (Exception e) true;
8  signals_only RuntimeException + the Java throws declaration;
9  diverges true;
10 when true;
11 measured_by <very large number>;
```

```
12 duration <very large number>;
13 working_space <very large number>
```

In addition, any formal parameters of reference type are `non_null`, but any return value of reference type is `nullable`, independent of any global nullness defaults.

These conservative default behaviors are somewhat onerous for library methods. Many of these methods are `pure` or at least have no side effects outside their own receiver. The user will likely need to provide some specifications for the library methods that are being used. Again, tools may be able to provide some help here, as well as efforts to specify more of the Java standard library.

10.4 Object()

As the `java.lang.Object` class has no superclass, its default constructor has a simple default specification:

```
1  requires true;
2  accessible \nothing;
3  assignable \nothing;
4  captures \nothing;
5  callable \nothing;
6  ensures true;
7  signals (Exception e) false;
8  signals_only \nothing;
9  diverges false;
10 when true;
11 measured_by 0;
12 duration <very large number>;
13 working_space <very large number>
```

10.5 Constructors

Constructors have a method specification like non-constructor methods, but with a few differences and additional considerations.

When a constructor is called, the following sequence of operations takes place:

- Static initialization of the class happens, if it has not already occurred
- All instance fields are initialized to zero-equivalent values (including final instance fields with compile-time-constant initializers)
- The parent class constructor is called (per the explicit or implicit super call)
- The instance fields are initialized and the instance initialization blocks are executed in textual order

- The body of the constructor is executed.

The pre-state of the constructor specification is the state after static initialization but before any instance initialization is started. Thus any instance fields have undefined values and the object being constructed is not yet allocated.

Accordingly, `this` may not be used in the preconditions or the frame conditions (or any specification clause that is evaluated in the pre-state). Because the object being constructed is not part of the pre-state, any instance fields that are initialized by the constructor need not be in the frame conditions. Indeed they may not be because that would require an implicit reference to `this`.

A constructor marked `pure` or `assigns \nothing;` may initialize the object's instance fields and may assign only to those fields.

It is (unfortunately) the case in Java that a parent class constructor can downcast `this` to get access to a derived class object before it is initialized. The result is that in an example like the following the asserted expression is true.

```
1  class P {
2    public P() {
3      //@ assert (this instanceof Con) ==> ((Con)this).f == 0;
4    }
5  }
6
7  public class Con extends P {
8    public int f = 1;
9    {
10     f = 2;
11   }
12 }
```

10.6 Default constructors

A default constructor is a zero-argument constructor generated by the compiler when a user writes no constructors. Its implementation (per Java) is just to call the zero-argument constructor of its parent class.

10.6.1 Specification in .jml file

If there is a .jml file containing the specification of the class of the constructor in question, then the specification of the default constructor can be put in that .jml file, whether or not there is a corresponding .java file.

```
1  // (portion of) .jml file
2  class A {
3    //@ pure
4    public A() {}
5  }
```

10.6.2 Specification in .java file

If there is a source `.java` file and no `.jml` file, then a specification of the default constructor can be put in the `.java` file along with an implementation of the default constructor:

```
// (portion of) .java file
class A {
  //@ pure
  public A() {}
}
```

10.6.3 Default specification

If there is no specification of the default constructor in either a `.java` or a `.jml` file for the class in question, then a default specification is assumed for the default constructor. That default specification is a copy of the specification cases of the parent class's default constructor's specification, omitting any specification cases that are not visible in the child class.

For example, the specification of the constructor `Object()` is just

```
    /*@ public normal_behavior
      @   assignable \nothing;
      @   reads \nothing;
      @*/
    public /*@ pure @*/ Object();
```

Any class that is derived directly from `java.lang.Object` and has a default constructor would have this same specification for that default constructor, unless the user supplied a different one.

10.7 Enums

Write this. Refer to/do not duplicate earlier discussion

10.8 Records

Refer to/do not duplicate earlier discussion

A Java record declaration is a class declaration with much of the body of the class automatically generated. For example, the declaration
`record Rectangle(double length, double width)`
creates a class with

- One `private final` field for each formal argument
- A public constructor with a signature corresponding to the declaration

- public getter methods for each field
- default `equals`, `hashCode` and `toString` methods

The class is immutable.

The default specifications for such a class are these:

- The class has the `immutable` modifier
- Each generated private field has the modifier `spec_public`
- The constructor has a `public normal_behavior` specification case with a simple postcondition in which each field is set to the value of the corresponding formal argument. The constructor has the `pure` modifier. *Which kind of pure*
- Each getter function has a `public normal_behavior` specification case with the simple postcondition that the result of the method is the value of the corresponding field. *Which kind of pure*
- The generated `equals` method has a `public behavior` specification case in which the ensures postcondition calls == to compare each primitive value and `.equals()` for reference values. The record's `.equals()` method is `pure` if all of the component types have `pure .equals()` methods. *other clauses Which kind of pure*
- The generated `hashCode` method has a `public behavior` with an `ensures true;` postcondition. The record's `.hashCode()` method is `pure` if all of the component types have `pure .hashCode()` methods. *other clauses Which kind of pure*
- The generated `toString` method has a `public behavior` specification case in which the ensures postcondition is `ensures true;`. The record's `.toString()` method is `pure` if all of the component types have `pure .toString()` methods. *Which kind of pure other clauses*

Thus for an example declaration
`record Count(int number, /*@ nullable */ T value);`
we have the specification

```
1  final class Count {
2      //@ spec_public
3      final private int number;
4
5      //@ spec_public
6      final private /*@ nullable */ T value;
7
8      //@ public normal_behavior
9      //@   ensures this.count == count && this.value == value;
10     //@ pure
11     public Count(int number, T value);
12
13     //@ public normal_behavior
```

```
14      //@   ensures \result == number;
15      //@ pure
16      public int number();
17
18      //@ public normal_behavior
19      //@   ensures \result == value;
20      //@ pure
21      public int value();
22
23      //@ public behavior
24      //@   ensures true;
25      public int hashCode();
26
27      //@ public behavior
28      //@   ensures true;
29      public String toString();
30
31      //public behavior
32      //@   requires o instanceof Count;
33      //@   ensures \result == (
34      //@        ((Count)o).number == this.number &&
35      //@         Objects.equals(((Count )o).value, this.value));
36      public boolean equals(Object o);
37  }
```

Need to say what all other clauses are; conditions under which methods are pure and under which they are 'signals false' and what exceptions might be thrown

Record declarations can include customizations and may include explicit declarations of the fields and methods that are typically implicit. If there is any customization then no default specification is generated; the user is expected to supply a complete specification.

10.9 Lambda functions

Write this

Chapter 11

JML Statements

JML statements are JML constructs that appear as statements within the body of a Java method or initializer. Some are standalone statements, while others are specifications for loops or blocks or statements that follow.

The body of a method is not part of its interface—it is the implementation. Hence, JML statements within the method body are not part of the method's specification. Rather they are generally statements that aid in the verification of the implementation or help to debug it. Consequently, JML includes just a few specification statements that are commonly used. Individual tools supporting JML are likely to add other specification statements to aid or debug the proof.

Grammar:

```
<jml-statement> ::=
        <jml-assert-statement>          §??
      | <jml-assume-statement>          §??
      | <jml-local-variable>            §??
      | <jml-local-class>               §??
      | <jml-ghost-label>               §??
      | <jml-unreachable-statement>     §??
      | <jml-set-statement>             §??
      | <jml-loop-specification>        §??
      | <jml-refining-specification>    §??
```

11.1 `assert` statement and Java assert statement

Grammar:

```
<jml-assert-statement> ::=
       <assert-keyword> <opt-name> <jml-expression> ;
<assert-keyword> ::= assert  | assert_redundantly
```

Type checking requirements:

- the *<jml-expression>* must be boolean

The `assert` statement requires that the given expression be `true` at that point in the program. A static checking tool is expected to require a proof that the asserted expression is true and to issue a verification failure if the expression is not provable. A runtime assertion checking tool is expected to check whether the asserted expression is true and to issue a warning message if it is not true in the given execution of the program.

In static-checking, after an assert statement, the asserted predicate is assumed to be true. For example, in

```
1 // c possibly null \\
2 //@ assert c != null;\\
3 //@ int i = c.value;
```

if `c` is null prior to this code snippet, then the assert statement will trigger a verification failure, but no warning should be given on `c.value` since `c != null` is implicitly assumed after the assert.[1]

By default, JML will interpret a Java assert statement in the same way as it does a JML assert statement — attempting to prove that the asserted predicate is true and issuing a verification error if not. This proof attempt happens whether or not Java assertions are enabled (via the Java `-ea` option).

In executing a Java program, when assertion checking is enabled, a Java assert statement will result in a `AssertionError` at runtime if the corresponding assertion evaluates to false; if assertion checking is disabled (the default), a Java assert statement is ignored. Runtime assertion checking tools may implement JML assert statements as Java assert statements or may issue unconditional warnings or exceptions.

11.2 `assume` statement

Grammar:

<jml-assume-statement> ::=
    <assume-keyword> *<opt-name>* *<jml-expression>* ;
<assume-keyword> ::= **assume** | **assume_redundantly**

Type checking requirements:

- the *<jml-expression>* must be boolean

The `assume` statement adds an assumption that the given expression is `true` at that point in the program.

Static analysis tools may assume the given expression to be true. Runtime assertion checking tools may choose to check or not to check that the assume statement is actually true.

[1] The OpenJML tool defines the `check` statement, which also checks that a given expression is true, but makes no assumption that it is true after the check.

An `assume` statement might be used to state an axiom or fact that is not easily proved. However, `assume` statements should be used with caution. Because they are assumed but not necessarily proven, if they are not actually true an unsoundness will be introduced into the program. For example, the statement `assume false;` will render any code following the statement silently infeasible. Even this may be useful, since, during debugging, it may be helpful to shut off consideration of certain branches of the program.

11.3 Local ghost variable declarations

Grammar:

```
<jml-local-variable> ::=
        ghost <modifier>* <decl-type> <identifier>
                [ = <jml-expression> ] ;
```

A ghost local declaration serves the same purpose as a Java local declaration: it introduces a local variable into the body of a method. A ghost declaration may be initialized only with a (side-effect-free) JML expression. The type in the ghost declaration may be either a Java or a JML type.

The only modifiers allowed for a ghost declaration, in addition to `ghost`, are

- `final` — as for Java declarations, this modifier means the variable's value will not be changed after initialization.
- `uninitialized` — indicates that JML should consider the variable uninitialized even though there is a Java initializer (cf. §??).
- `non_null`, `nullable` - these may modify the *<decl-type>* in the declaration, if it is a Java reference type. The current nullness default for the method will implicitly state a `non_null` or `nullable` type annotation for the declaration.
- Java annotations

Variables declared in such a ghost declaration may be used in subsequent JML expressions and they may be assigned values in `set` statements (§??).

Any other JML modifiers?

Grammar needs to permit array initializers

11.4 Local model class declarations

```
<jml-local-class> ::= model <class-declaration>
```

Java permits local class declarations as method body statements. Similarly, JML permits the declaration of a local model class as a specification statement. The syntactic rules for a local JML model class are the same as for a local Java class, such as restrictions on scope and that all local variables used within the class definition are final. However a local JML model class may use other JML constructs, such as JML ghost

variables and fields. Furthermore the methods of a local JML model class need not have an implementation.

The declaration of a JML local model class must be contained in just one JML annotation. JML constructs within the model class declaration, such as method specification clauses, do not need to be contained in embedded JML annotations because they are already in an outer JML annotation, as shown in the following code snippet.

```
1  public void m(int i) {
2    int k = i*i;
3    //@ ghost final int g = k;
4    /*@ model class Helper {
5      @    requires k == i*i;
6      @    ensures \result == k*k;
7      @    pure
8      @    int helper(int x);
9      @ }
10     @*/
11 }
```

11.5 Ghost statement label

Grammar:

```
<jml-ghost-label> ::= <java-identifier> :[ { } | ; ]
```

Java allows statement labels to be placed before statements; they serve as targets of `break` and `continue` statements. JML also uses such labels as targets of `\old` and `\fresh` expressions.

Consequently there is sometimes a need to add a label for JML purposes that can be referred to by `\old` and `\fresh`. The JML ghost-label does that.

A ghost-label may be placed anywhere in a block immediately preceding a Java or JML statement. If a statement must be introduced as the target of the label or the statement label needs to be disambiguated from names on other JML constructs, the optional forms `//@ label:{}` or `//@ label:;` can be used.

Any Java identifier may be used for the label if it would be permitted to be a Java label at that location, which means it may not be the name used to label an enclosing labeled statement. A label name may shadow the name of a previously labeled statement that is not enclosing. However, this is not recommended as it may cause a misreading if a reader does not notice the need to disambiguate identically-named labeled statements.

11.6 Built-in state labels

The `\old` (§??) and `\fresh` (§??) expressions can refer to the program state at a particular statement label. JML also allows inserting statement labels into the source

code (§??).

In addition, JML provides some built-in state labels:

- `Pre` — the pre-state of the containing method (even in block contracts)
- `Old`
  - in method or block contracts: the pre-state of that contract
  - in specification statements: the pre-state of the innermost enclosing contract (either a block contract, or if, there are no enclosing block contracts, the pre-state of the enclosing method)
- `Here`
  - in specification statements, the program state just prior to the statement using the label
  - in clauses evaluated in the pre-state of a contract, that pre-state
  - in clauses evaluated in the post-state of a contract, that post-state

Uses of these built-in labels always refer to the corresponding program state, even if there is an explicit Java or ghost-label with the same name, as illustrated in this example:

Need example

A possible alternative is to allow escaped label names (e.g., `\Pre`) to always refer to the built-in label, despite any explicit labels.

Possible other built-in labels are `Post` (post-state of contract), `Init` (after static initialization), `LoopEntry` (just after loop initialization), `LoopCurrent` (beginning of current loop iteration). (cf. ACSL)

11.7 `unreachable` statement

Grammar:

```
<jml-unreachable-statement> ::=
        unreachable <opt-name> [ ; ]
```

The `unreachable` statement asserts that no feasible execution path will ever reach this statement. Runtime-checking can only check that no `unreachable` statement is executed in the current execution of a program.

It has been common practice to insert `assert false;` statements to check whether a given program point is infeasible. The `unreachable` statement accomplishes the same purpose with clearer syntax.

11.8 set statement

Grammar:

<jml-set-statement> ::=
    set <opt-name> <java-statement>

The java-statement in the grammar is not quite right since the statements can include ghost variables.

Type checking requirements:

- the *<java-statement>* may be any single executable Java statement, including a block statement

If the *<java-statement>* ends in a semicolon, that semicolon is required and may not be omitted just because it occurs at the end of a JML comment.

A `set` statement marks a statement that is executed during runtime assertion checking or symbolically executed during static checking, commonly called *ghost code*. As such, the statement must be fully executable and may have side effects on ghost code; also it may contain references and assignments to local ghost variables and ghost fields, and calls of model methods and classes that have executable implementations. The primary motivation for a `set` statement is to assign values to ghost variables, but it can be used to execute any statement.

JML previously contained a `debug` statement that was semantically equivalent to

`//+DEBUG@ set` *statement*

11.9 Loop specifications

Grammar:

<loop-specification> ::= (*<loop-clause>*) *
<loop-clause> ::= *<loop-invariant>* | *<loop-variant>* | *<loop-frame>*
<loop-invariant> ::= *<loop-invariant-keyword>* *<jml-expression>* ;
<loop-invariant-keyword> ::= **loop_invariant** | **maintaining**
    | **loop_invariant_redundantly**
    | **maintaining_redundantly**
<loop-variant> ::= *<loop-variant-keyword>* *<jml-expression>* ;
<loop-variant-keyword> ::= **decreases** | **decreasing** | **loop_decreases**
    | **decreases_redundantly**
    | **decreasing_redundantly**
<loop-frame> ::=
    <loop-frame-keyword> (**nothing** | *<location-set>* (, *<location-set>*) *) ;
<loop-frame-keyword> ::= **assigning** | **loop_writes** | **loop_modifies**

Type checking requirements:

- the *<jml-expression>* in a *<loop-invariant>* must be a boolean expression
- the *<jml-expression>* in a *<loop-variant>* must be a `\bigint` expression

- the *<location-set>*s in a *<loop-assignable>* clause may contain local variables that are in scope at the program location of the loop
- a *<loop-specification>* may only appear immediately prior to a Java loop statement
- the variable scope for the clauses of a *<loop-specification>* includes the declaration statement within a `for` loop, as if the *<loop-specification>* were textually located after the declaration and before the loop body

Fix the grammar for the frame item

A special and common case of statement specifications is specifications for loops. In many static checking tools loop specifications, either explicit or inferred, are essential to automatic checks of implementations.

A loop typically has four kinds of specification clauses:

- §??: a loop invariant that constrains the value of the loop index (or `\count` value §??)
- §??: a loop invariant that gives the inductive predicate stating what the loop is accomplishing
- §??: a `loop_writes` clause that states what memory locations are assigned in the loop body
- §??: a `loop_decreases` clause needed to demonstrate loop termination

11.9.1 Loop invariants

A loop invariant states a property that is maintained by the execution of the loop body. Specifying a loop invariant implies two proof obligations:

- After any loop initialization (for a for-loop) but before the loop test or execution of the loop body, the loop invariant must be true.
- Assuming the loop invariant is true after the loop test but before beginning execution of the loop body, the loop invariant must again be true after executing the loop body and (just after) the loop update statement, but before the loop test is performed. This includes any execution paths from `continue` statements and `continue` statements with labels in enclosed loops. The loop invariant is not checked for any `break`, `throws` or `return` statement that exits this loop.

It is important to realize that each iteration of the loop is checked independently, as is the normal exit from the loop when the loop test is false. Thus anything that needs to be known from a previous iteration must be present in an invariant. Here is an example that illustrates a very common pattern for loop specifications.

```
1 // a is a non-null array to be initialized
2 //@ loop_invariant 0 <= k <= a.length;
```

```
3 //@ loop_invariant \forall int j; 0<=j<k; a[j] == j*j;
4 //@ loop_writes a[*];
5 //@ loop_decreases a.length-k;
6 for (int k=0; k<a.length; ++k) {
7     a[k] = k*k;
8 }
9 //@ assert \forall int j; 0<=j<a.length; a[j] == j*j;
```

There are two loop invariants here.

- The first one simply, but importantly, restricts the range of the loop index: `k` may take any value from `0` to `a.length` inclusive (`k` equals `a.length` at the end of the last iteration, when, just like after all iterations, the loop invariant must hold).
- The second iteration states what has been accomplished by the loop iterations so far. Nothing from previous iterations is "remembered". What is known is that all array values up to but not including `k` have been initialized. Then, given the execution of the loop body and the update to the loop index `k`, the loop invariant is again true for one more element of the array.

If `k` is `a.length`, as it is on exit from the loop, then, given the loop invariants, the `assert` statement on line 9 is true.

11.9.2 Loop variants

The loop invariants alone do not determine whether a loop terminates. For that we need a well-defined measure that counts down to an end-point. JML implements this with integers. The `decreasing` clause gives an expression that must be non-negative at the beginning of each loop iteration (after the loop test) and is smaller after the loop update and prior to the loop test after the conclusion of the loop body (including control flows from `continue` statements, but not `break`, `throws` or `return` statements). Because the variant expression begins as some finite value, is always non-negative, and decreases on each iteration, we can infer that the loop will terminate in some finite number of iterations.

The example above shows a very typical loop variant expression.

11.9.3 Loop frame conditions

The loop frame specification states which values are assigned to (that is, might possibly change) in the loop. The frame clause implicitly includes the loop index and the `\count` value. In the example above, the frame condition states `a[*]`, that is that all elements of the array may change during all the loop iterations, not just `a[k]`, that a particular element is changed during a particular iteration.

The logical encoding of the loop presumes that any memory location in the frame condition has an unknown value at the start of the loop. It is therefore important

that the loop invariants state the values of any memory locations that are listed in the loop frame condition.

11.9.4 Inferring loop specifications

Loop specifications are not part of a method interface. The necessity of loop specifications is a result of the current state of specification technology, namely, that inferring the loop specifications from arbitrary source code is an unsolved problem. However, in many common cases the loop specifications can be inferred. In the example above a tool might readily infer lines 2, 4, and 5, and possibly also line 3 for simple loops.

Depending on the tool used, it may not be necessary to explicitly state each of these loop specification statements. However tools should be clear about any loop specifications that are implicitly used.

11.10 Statement (block) specification

Grammar:
`<statement-specification>` `::=` **`refining`** `<behavior-seq>`

The semantics of a *block specification*[2] are very similar to those of a method specification (cf. §??). However, note that:

- The `\result` expression may only be used in a `returns` clause,
- the `\exception` expression may only be used in a `throws` clause, and
- `recommends` clauses may not be used.
- There is no inheritance of specification cases as there might be for method specifications.

On the other hand, the `returns` (§??),
`throws` (§??), `continues` (§??, and
`breaks` (§??) specification clauses are permitted in a block specification. Furthermore, in the pre- and postconditions of block specifications, any local (including ghost) variables that are in scope may be used in the clause expressions.

Furthermore, there are a few conceptual differences between method and block specifications. A method specification states preconditions on the legal states in which a method may be called and gives postconditions stating the effects of a method's execution, including comparisons between the pre-state (before the method call) and the post-state (after the method completion). Similarly, a block specification makes

[2] We use the term *block specification* (or *block contract*) even though the specification can apply to a single statement because a block of statements is the usual case and because *statement specification* is easily confused with *specification statement* as used in §??.

assertions about the execution of the (block) statement that follows the block specification, or, if a **begin** JML statement (§??) immediately follows the statement specification, then the specification applies to the sequence of statements within that **begin-end** block. It also must be that:

- For at least one of the *<behavior>*s in the *<behavior-seq>*, all of the `requires` clauses in that *<behavior>* must be true at the code location of the statement specification.
- For any *<behavior>* for which all of the `requires` clauses are true, all other clauses in that *<behavior>* must be satisfied in the statement's post-state, that is after execution of the following (block) statement.

The motivation for a block specification is that it summarizes the behavior of the subsequent Java statement, which might be a begin-end block. Thus one application of this specification idiom is to check the behavior of a section of a method's implementation. It also allows the remainder of the method body to be checked just using the block specification as a summary without needing to use the summarized portion of the implementation.

For example, some block of code may implement a complicated algorithm. The implementation writer may encapsulate that code in a syntactic block and include a specification that describes the effects of the algorithm. Then a tool may separate its static checking task into two parts:

- checking that the implementation in the block (along with any preceding code in the method body) does indeed have the effect described by the specification;
- checking that the surrounding method satisfies the method's specification when, within its body, the encapsulated block of code is replaced by its block specification.

It might be a reasonable critique that such a block of code should be extracted into its own method. However, when one is specifying existing, unalterable code, such refactoring is not an option.

11.11 begin-end statement groups

Grammar:

<begin-end> `::=` **`begin`** | **`end`**

Pairs of JML `begin` and `end` statements may be used to define a block of Java statements, just as using opening and closing braces might in Java. However `begin` and `end` do not introduce a local scope and can be inserted in the code without modifying the Java code per se.

Begin-end statement pairs may be nested. The `end` corresponding to a given `begin` must be in the same scope as the `begin`, and not in a nested or containing scope.

These begin-end blocks are only useful with block specifications as described in §??.

11.12 Experimental statement specifications

The following JML statements are purely experimental. They are not currently part of JML, though they may be used in tools.

- `check` statement – like the `assert` statement, but does a soft assert. That is, the asserted predicate is not assumed to be true after the statement.
- `show` statement – emits values of program variables as part of a counterexample when a proof fails or during runtime assertion checking
- `halt` statement – limits checking of assertions just up to the location of the statement
- `split` statement – enables breaking up a single method into multiple proofs at the locations of branching statements (e.g., if, switch, loops).
- `havoc` statement – gives listed arguments arbitrary (though type-consistent) new values
- `reachable` – tests whether a program location is reachable from some legal pre-state. This is part of feasibility testing.
- `use` statement – introduces a lemma
- `inline_loop` – used to specify actions of methods that implicitly perform iterations over arbitrary actions
- `comment` statement – used to insert comments into translated code

11.13 Experimental line specifications: allow, forbid, ignore

TODO

Chapter 12

JML Expressions

Grammar:

```
<jml-expression> ::=
        <conditional-expression>                §??
      | <quantified-expression>                 §??
      | <lambda-expression>                     §??
      | <assignment-expression>                 §??
      | <jml-infix-expression>                  §??

<jml-infix-expression> ::=
        <jml-binary-expression>                 §??
      | <jml-prefix-expression>                 §??

<jml-prefix-expression> ::=
        <jml-unary-expression>                  §??
      | <jml-cast-expression>                   §??
      | <jml-postfix-expression>

<jml-postfix-expression> ::=
        <dot-expression>                        §??
      | <array-expression>                      §??
      | <jml-primary-expression>

<jml-primary-expression> ::=
```

<result-expression> §??
| <exception-expression> §??
| <informal-expression> §??
| <old-expression> §??
| <nonnullelements-expression> §??
| <fresh-expression> §??
| <type-expression> §??
| <erasure-expression> §??
| <typeof-expression> §??
| <arraytype-expression> §??
| <isarray-expression> §??
| <elemtype-expression> §??
| <invariant-for-expression> §??
| <static-invariant-for-expression> §??
| <is-initialized-expression> §??
Missing some - check the list
| <java-math-expression> §??
| <safe-math-expression> §??
| <bigint-math-expression> §??
| <duration-expression> §??
| <working-space-expression> §??
| <space-expression> §??

The right-most expressions in quantified expression, conditional expression, assignment expressions and range (`..`) expressions all extend up to a closing punctuation mark (`, ) ] ; : or }`).

shift bit logical dot cast new methodcall ops

Need sections on \values

12.1 Syntax

JML expressions may include most of the operations defined in Java and additional operations defined only in JML. JML operations are one of five types:

- infix operations that use non-alphanumeric ASCII symbols (e.g., <==>)
- identifiers or expressions that begin with a backslash (e.g., `\result`, `\forall`)
- identifiers that begin with a backslash but have a functional form (e.g., `\old`)
- some special-purpose syntax (e.g. `(* ... *)`
- methods defined in JML whose syntax is Java-like (e.g., `JML.informal(...)`)

The Java-like forms replicate some of the special-purpose syntax and backslash forms. The backslash forms are traditional JML and more concise. However, Java-like syntax can be directly parsed by Java tools.

12.2 Well-defined expressions

An expression used in a JML construct must be well-defined, in addition to being syntactically and type-correct. This requirement disallows the use of functions with argument values for which the result of the function is undefined. For example, the expression `(x/0) == (x/0)` is considered in JML to be not well-defined (that is, undefined), rather than true by identity. An expression like `(x/y) == (x/y)` (for integer x and y) is true if it can be proved that y is not 0, but undefined if y is possibly 0. For example, `y != 0 ==> ((x/y) == (x/y))` is well-defined and true.

The well-definedness rules for operators are given in the section describing that operator. They presume that the expressions are type correct. The [[ ]] notation denotes true or false according to whether the enclosed expression is or is not well-defined.

Distribute these to the sections

(literals and names)			**true**
(parenthesis)	$[\![(e)]\!]$	$\equiv$	$[\![e]\!]$
(dot access)	$[\![e.f]\!]$	$\equiv$	$[\![e]\!]$ & $e \neq null$, where f is a field of the type of e
(array element)	$[\![e[e_1]]\!]$	$\equiv$	$[\![e]\!]$ & $[\![e_1]\!]$ & $e \neq null$ & $0 \leq e_1 < e.length$
(cast)	$[\![(T)e]\!]$	$\equiv$	$[\![e]\!]$, for a type name T (the cast is well-defined even if it causes an overflow warning)
(unboxing)	$[\![(T)e]\!]$	$\equiv$	$[\![e]\!]$ & $e \neq null$, for a type name T, including implicit unboxing to primitive values
(boxing)	$[\![(T)e]\!]$	$\equiv$	**true**, for a type name T, including implicit boxing of primitive values
(boolean negation)	$[\![\,!e]\!]$	$\equiv$	$[\![e]\!]$
(complement)	$[\![\sim e]\!]$	$\equiv$	$[\![e]\!]$
(string +)	$[\![e_1 + e_2]\!]$	$\equiv$	$[\![e_1]\!]$ & $[\![e_2]\!]$
(non-short-circuit binary operations)	$[\![e_1 \; op \; e_2]\!]$	$\equiv$	$[\![e_1]\!]$ & $[\![e_2]\!]$, for operators `&` `|` `^` `<=` `<` `==` `!=` `>` `>=`
(short-circuit `&&`)	$[\![e_1 \;\&\&\; e_2]\!]$	$\equiv$	$[\![e_1]\!]$ & $(e_1 \Rightarrow [\![e_2]\!])$
(short-circuit `||`)	$[\![e_1 \;\|\|\; e_2]\!]$	$\equiv$	$[\![e_1]\!]$ & $(\neg e_1 \Rightarrow [\![e_2]\!])$
(short-circuit ==>)	$[\![e_1 \;==>\; e_2]\!]$	$\equiv$	$[\![e_1]\!]$ & $(e_1 \Rightarrow [\![e_2]\!])$
(arithmetic operations)	$[\![e_1 \; op \; e_2]\!]$	$\equiv$	$[\![e_1]\!]$ & $[\![e_2]\!]$, for operators `+` `-` `*` (these are well-defined even if they overflow)
(divide)	$[\![e_1 \,/\, e_2]\!]$	$\equiv$	$[\![e_1]\!]$ & $[\![e_2]\!]$ & $e_2 \neq 0$
(modulo)	$[\![e_1 \,\%\, e_2]\!]$	$\equiv$	$[\![e_1]\!]$ & $[\![e_2]\!]$ & $e_2 \neq 0$
(conditional)	$[\![e_1 \;?\; e_2 \;:\; e_3]\!]$	$\equiv$	$[\![e_1]\!]$ & $(e_1 \Rightarrow [\![e_2]\!])$ & $(\neg e_1 \Rightarrow [\![e_3]\!])$

- method calls: well-defined iff (a) the receiver and all arguments are well-defined and (b) if the method is not static, the receiver is not null and (c) the method's precondition and invariants are true and (d) the method can be shown to not throw any Exceptions in the context in which it is called
- new operator: well-defined iff (a) all arguments to the constructor call are well-defined, (b) the preconditions and static invariants of the constructor are satisfied by the arguments, and (c) the constructor does not throw any Exceptions in the context in which it is called
- shift operators (`<<` `>>` `>>>`): well-defined iff all operands are well-defined. Note that Java defines the shift operations for any value of the right-hand operand; the value is trimmed to 5 or 6 bits by a modulo operation appropri-

ate to the bit-width of the left-hand operand. JML tools may choose to raise a warning if the value of the right-hand operand is outside the 'expected' range, but an out-of-range right-hand value does not make the operation undefined.

floating point operations?

12.3 Purity (no side-effects)

Specification expressions must not have side effects. During run-time assertion checking, the execution of specifications may not change the state of the program under test. Even for static checking, the presence of side-effects in specification expressions would complicate their semantics.

Thus some Java operators are not permitted in JML expressions:

- allowed: `+ - * / % == != <= >= < > .^ & | && || << >> >>> ?:`
- prohibited: `++ -- = += -= *= /= %= &= |= ^= <<= >>= >>>=`

12.4 Java expressions

The expressions described here are defined in Java and are unchanged in JML except that they can take *<jml-expression>*s as operands.

12.4.1 Conditional expression

<conditional-expression> ::=
    <jml-infix-expression> **?** *<jml-expression>* **:** *<jml-expression>*

Well-definedness: $[[c?t:e]] \equiv [[c]] \wedge ((c \implies [[t]]) \wedge (!c \implies [[e]]))$

Type information:

- the first expression must have boolean type
- the middle and last expressions must be implicitly convertible to a common type
- the type of the expression is that common type of the latter two expressions

This operation is unchanged from Java, other than that the consitutent expressions may be JML expressions.

12.4.2 Assignment expression

<assignment-expression> ::=
    <jml-postfix-expression> *<assign-op>* *<jml-expression>*
<assign-op> ::= **=** | **+=** | **-=** | ***=** | **/=** | **%=**
| **&=** | **|=** | **≙**

Fix presentation of shifts and xor

Well-definedness: $[[lhs\ op\ rhs]] \equiv [[lhs]] \wedge [[rhs]]$
with this additional condition: for `/=` and `%=` : (rhs != 0)

Type information:

- except for the shift operations, the right-hand-side must be implicitly convertible to the type of the left-hand-side
- the type of the expression is the type of the left-hand-side

This operation is unchanged from Java, other than that the consituent expressions may be JML expressions. As in Java, the value of the expression is the new value of the left-hand-side.

Assignment operations may be used in specifications only in statement specifications that allow side-effects, such as the `set` statement or initializers of ghost declarations. In these cases the side-effects may only affect ghost memory locations.

12.4.3 Unary Expression

<unary-expression> `::=` [`!` | `-` | ~] *<unary-expression>*

Well-definedness: $[[op\ arg]] \equiv [[arg]]$

There are three unary operators in Java; JML adds no additional unary operators. In each case the operation is well-defined if the operand is: $[[op\ x]] \equiv [[x]]$.

- `-` Numeric negation can be applied to a value of any numeric type, including `\bigint` and `\real`. The type of the expression is the same as the type of the argument, except that `char`, `short` and `byte` operands are promoted to `int`.
- ~ Bit-vector complement can be applied to values of any integral type. `char`, `short` and `byte` operands are converted to `int`. Complement of a `\bigint` value x yields $-1-x$.
- `!` Boolean 'not' can be applied to boolean values, returning a boolean value.

12.4.4 Cast expression

<cast-expression> `::=` `(` *<type-name>* `)` *<jml-expression>*

Type information:

- The result type is the named type, *<type-name>*.
- If the *<type-name>* is a reference type and the type of the *<jml-expression>* is a primitive type, the operation is a *boxing* expression; the named type must be the boxed type corresponding to the primitive type of the operand.
- If the *<type-name>* is a primitive type and the type of the *<jml-expression>* is a reference type, the operation is a *unboxing* expression; the named type must be the unboxed type corresponding to the reference type of the operand. *Conversions allowed?*

- If the *<type-name>* is a reference type, then the dynamic type of the *<jml-expression>* must be a sub- or supertype of the named type.
- If the *<type-name>* is a primitive type, including a JML primitive type, then the type of the *<jml-expression>* must be one that is convertible to the named type. Which type pairs are convertible is defined in the discussions of those types (§??).

12.4.5 Prefix ++ and −− operations

The ++ and −− prefix operations are Java operations. Because they change their operand, they my not be used in JML specifications.

12.4.6 Postfix ++ and −− operations

The ++ and −− postfix operations are Java operations. Because they change their operand, they may not be used in JML specifications.

12.4.7 Dot selection operation

WRITE THIS - needs grammar

Type information: The expression before the dot must name a type or be an expression having some reference type. The name after the dot must name a static member of the named type or name a member (method or field) of the static type of the expression. Two special cases are the suffixes `.length` for array expressions and `.class` for type names.

Well-definedness: *Fillin*

This operation is unchanged from Java, other than allowing, in specifications, using JML names, expressions and types.

12.4.8 Array element operation

<array-expression> ::= *<postfix-expression>* **[** *<expression>* **]**

Well-defined: $[\![\, a[i] \,]\!] \equiv [\![\, a \,]\!] \wedge (a \neq null) \wedge [\![\, i \,]\!] \wedge 0 \leq i \wedge i < a.length$,
where $a.length$ is the length of the array a, as in Java.

Type information:

- the first operand must have type 'array of T' for some type T
- the second operand must have integral type
- the result has type T

This operation is the same in JML and Java. It extracts a particular element from a Java array.

Some built-in types also allow this syntax to mean element selection (cf. §?? §?? §??).

12.4.9 Method call

Write more

12.4.10 Parenthesized expression

```
<parenthesized-expression> ::= ( <jml-expression> )
```

Type Information: The type of the result is the same as the type of the argument. The argument may have any non-void type.

Well-definedness: [[(*<jmlexpression>*)]] ≡ [[*<jml-expression>*]]

The value of the *<parenthesized-expression>* is the value of its operand.

12.5 JML expressions

The expressions described in this section are unique to JML and may only be used within JML annotation comments.

12.5.1 Quantified expressions

Grammar:

```
<quantified-expression> ::=
        <quantifier> <type-name> <java-identifier> ;
         [  [  <jml-expression>  ] ;  ]  <jml-expression>


<quantifier> ::=
        \forall  |  \exists  |  \choose
       |\num_of  |  \sum  |  \product  |  \max  |  \min
```

The first expression (called the range expression, $R(x)$) is optional. If omitted, its default value is `true`. The second expression is called the *value expression*, $V(x)$.

The scope of the declared variables is only the bodies of the two subexpressions; the declared variables shadow any variable or fields with the same name in the scope containing the quantified expression.

Well-definedness: The quantified-expression is well-defined for a quantifier `Q` as stated here:

```
[[Q T x; R(x); V(x) ]] ≡
        (∀ Tx; [[ R(x) ]] )
        ∧(∀ Tx; R(x)) ⟹ [[V(x)]] )
```

$\wedge(\exists\, T\, x; R(x))$

with the last conjunct only present for `\choose`, `\min` and `\max`.

Type information:
A *<quantified-expression>* declares a new local variable whose scope is only the two expressions within the quantified-expression. The variable name hides any identical names in enclosing scopes. The optional range expression must be boolean. The type of the value expression and of the whole quantified-expression depend on the quantifier, as shown in the following table (**T** is the type of the declared local variable), with details discussed in the subsections below.

Quantifier	Value expression	Entire expression
`\forall`	boolean	boolean
`\exists`	boolean	boolean
`\choose`	boolean	**T**
`\num_of`	boolean	`\bigint`
`\sum`	`\bigint` or `\real`	same as value expression
`\product`	`\bigint` or `\real`	same as value expression
`\max`	**N**	same as value expression
`\min`	**N**	same as value expression

Here **N** is any Java or JML numeric type

Although, the range expression is optional, runtime-assertion checking tools may use its form to infer a constrained range over which to iterate in order to compute the value of the quantified expression. Thus an appropriately written range expression may improve the runtime performance of a compiled program, or even make executing the program possible at all.

12.5.1.1 `\forall`, `\exists`

The `\forall` and `\exists` quantifiers correspond to the universal and existential quantifiers of first-order predicate logic.

- the universally quantified expression `(\forall T x; R(x); V(x))` is true iff $R(x) \implies V(x)$ is true for every x of type T.
- the existentially quantified expression `(\exists T x; R(x); V(x))` is true iff $R(x) \wedge V(x)$ is true for some x of type T.

For non-primitive types. $R(x)$ should be some predicate such as x is contained in a given collection, so that the domain of the quantification is unambiguous.

The phrase "for every x of type T is clear for primitive types T. But its meaning is not clear for reference types. What set of reference values are being considered? – all conceivable objects? all allocated objects? all reachable objects? For example, if T is Java's `java.lang.String` type it is easy to envision a quantification over all possible strings, whether they exist in a program or not. But even for such an immutable type, there can be different objects (that is, two separate allocations) that contain the same internal data (e.g. character sequence). Should these be counted as two separate instances in the quantification (such as in the `\num_of` quantifier below) or just one?
For non-immutable objects, the concept of all possible such objects seems problematic; then the question of whether (and how) to consider just allocated, or alternatively, reachable, objects needs more consideration.

12.5.1.2 `\choose`

Whereas the `\exists` quantifier tells whether there is some value that satisfies a given predicate, the `\choose` expression yields an arbitrary one such value. Thus the `\choose` expression is well-defined only if such a value exists.

The value of a *choose-expression* is any value that satisfies its range and value predicates; its result is deterministic but arbitrary if there is more than one such value. Any logical expressions that depend on the value of the choose-expression are valid only if they are valid no matter which choice is made. In logic-speak, the choice is *demonic*, as if a demon were making the choice, always seeking to invalidate your proof.

For example, in

```
1 //@ set int x = (\choose int k; 1 <= k <= 2);
2 //@ assert x == 1 || x == 2;  // valid
3 //@ assert x == 1; // invalid
```

the first assert is valid – no matter what allowed value `\choose` produces the assertion is true. But the second is not always true, and hence is invalid. In runtime-checking the second may be reported true or false non-deterministically.

Note also that two separate instances of the same `\choose` expression produce the same result (so that logically, an identity axiom holds). That is, if, for all x

$$(R(x) \wedge V(x)) = (R'(x) \wedge V'(x))$$

then

`(\choose T x; R(x); V(x)) == (\choose T x; R'(x); V'(x))` .

However, it would take a more-than-currently-capable logical reasoning engine to prove such equalities. Even when `R` and `V` are syntactically identical to `R'` and `V'`. tools may not be able to prove the equality. [1]

[1] The choose operator implements Hilbert's choice operator ε. The property last explained is called extensionality. All generalised quantifiers in JML are extensional in the sense that whenever R(x) and V(x) are replaced by a semantically equivalent R' and V', the expression yields the same value.

12.5.1.3 \one_of, \sum, \product, \max, \min

These generalized quantifiers perform various (commutative and associative) operations over the set of values specified by the range and value expressions: for each operation, that operation is applied to all the values $V(x)$ for which $R(x)$ is true.

- \one_of: this operation yields the number of values for which $R(x) \wedge V(x)$ is true, with the result type being \bigint. If $R(x)$ is not true for any x, the value of the \num_of expression is 0.
- \sum: this operation yields the sum of integer or real values, with $V(x)$ being promoted to either \bigint or \real and the result being of the same type. If $R(x)$ is not true for any x, the value of the \sum expression is 0.
- \product: this operation yields the product of integer or real values, with $V(x)$ being promoted to either \bigint or \real and the result being of the same type. If $R(x)$ is not true for any x, the value of the \product expression is 1.
- \max: this operation yields the maximum of all the values $V(x)$ for which $R(x)$ is true, with the result being of the same type as $V(x)$. There is no default if the range is empty; instead the expression is not-well-defined. Note that \max is overloaded with the max-locset expression (§??).
- \min: this operation yields the minimum of all the values $V(x)$ for which $R(x)$ is true, with the result being of the same type as $V(x)$. There is no default if the range is empty; instead the expression is not-well-defined.

I'd prefer that max and min be undefined if the range is always false. Otherwise the expression cannot be generalized to other data types. For example, to anything for which a (pure) total-order comparison function is supplied

---

One can conceive of operations like \sum and \product for other commutative and associative binary operations over other domains. Similarly, \max and \min could be extended to other total orders over other domains. However, there would need to be both a need and a reasoning mechanism to justify such additions. The current operations pose plenty of problems of their own.

12.5.2 Set comprehension

Text needed

12.5.3 Lambda expression

TODO

12.5.4 Binary (infix) expressions

```
<jml-binary-expression> ::=
<jml-prefix-expression> ( <binop> <jml-prefix-expression> )*
```

Table 12.1: Java and JML precedence. Note that postfix and prefix ++ and – have the same precedence as other postfix and prefix operations, but are not allowed in JML expressions.
(cf. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html)

Java operator	JML operator	
highest precedence		associativity
literals, names, parenthesis, new		
postfix: `.` `[]` method calls		left
postfix: (Java only) `++` `--`		left
prefix: unary `+` `-` `!` `~` cast		right
prefix: unary (Java only) `++` `--`		right
`*` `/` `%`		left
binary `+` `-`		left
`<<` `>>` `>>>`		left
`<=` `<` `>=` `>` `instanceof`	`<:` `<:=` `<#` `<#=`	chainable
`==` `!=`		left
`&`		left
`^`		left
`|`		left
`&&`		left
`||`		left
	`==>`	right
	`<==>` `<=!=>`	left
`?:`		right
	quantified	right
`..`		none
assignment, assign-op (Java only)		right
lowest precedence		

A sequence of alternating expressions and binary operators is parsed as a tree of binary operations according to the precedence and associativity of the various operators, as is customary in programming languages. The precedence and associativity are given in Table **??**. Precedence and associativity are purely syntactic; the results of parsing do not depend on the types of the expressions.

Note that, just as in Java, the bit-operations have precedence lower than equality. So `(a & b == 1)` is `a & (b == 1)`. For clarity's sake, always use parentheses around bit operations: `((a & b) == 1)`.

Well-definedness: typically $[[x\ op\ y]] \equiv [[x]] \wedge [[y]]$, with some special rules given for certain operators.

Edit this

implicit conversion. The operands are individually converted to potentially larger data types as follows:

- if either operand is `\real`, the other is converted to `\real`,
- else if one operand is `\bigint` and the other either `double` or `float`, they both are converted to `\real`,
- else if either operand is `double`, the other is converted to `double`,
- else if either operand is `float`, the other is converted to `float`,
- else if either operand is `\bigint`, the other is converted to `\bigint`,
- else if either operand is `long`, the other is converted to `long`,
- else both operands are converted to `int`.

12.5.5 Chaining of comparison operators

Grammar:

```
<chained-expression> ::=
        <expression> ( [< | <=] <expression> )+
      | <expression> ( [> | >=] <expression> )+
```

Well-defined:

$$[[\, e_1 \; op \; e_2 \; op \; \ldots \; op \; e_n]] \equiv \forall i \; [[e_i]]$$

Type information:

All the e_i must have numeric type; the result is boolean

In Java, an expression like `a < b < c` with `a`, `b`, and `c` having integer types is type-incorrect because `(a < b)` is a boolean and booleans and integers cannot be compared, and there is no implicit conversion between them, as in C.

However, JML allows such chains as a boolean operation that means `(a < b) & (b < c)`. The operators < and <= may be mixed in a chain, as may > and >=. The equality operators are not chainable[2] because the equality operators have different precedence than relational operators. In addition, `a < b == c < d` is meaningful in Java, as `(a < b) == (c < d)`. Chaining, were it supported, would give it a different meaning in JML: `(a < b) & (b == c) & (c < d)`.

Note that the desugaring of the chain is written with non-short-circuit operators. This emphasizes that all the operands must be independently well-defined. Also it allows static-checkers to optimize reasoning (non-short-circuit operators have simpler semantics than short-circuit ones). Runtime assertions checks are welcome to evaluate the expression in equivalent short-circuit fashion.

Though less commonly used, the subtype operators (`<:` `<:=`) and the lock comparison operators (`<\#` `<\#=`) also chain.

[2] They are chainable in Dafny, by comparison.

12.5.6 Implies operator: ==>

Grammar:

<implies-expression> ::= *<expression>* **==>** *<expression>*

Well-defined:

$$[\![\, e_1 \texttt{ ==> } e_2 \,]\!] \equiv [\![\, e_1 \,]\!] \;\&\; (e_1 \implies [\![\, e_2 \,]\!])$$

Type information:

- two arguments, each an expression of boolean type
- result is boolean

The ==> operator denotes implication and is a short-circuit operator. Its value is true if the left-hand operand is false or the right-hand operand is true; if the left operand is false, the right operand is not evaluated and may be undefined. The operation

<left> ==> *<right>*

is equivalent to

(! *<left>*) || *<right>* .

The ==> operator is right associative: `P ==> Q ==> R` is parenthesized as `P ==> (Q ==> R)`. This is the natural association from logic: `(P ==> Q) ==> R` is equivalent to `(P && !Q) || R`, whereas `P ==> (Q ==> R)` is equivalent to `!P || !Q || R`.

Obsolete syntax: The reverse implication operation <== is no longer supported.

12.5.7 Equivalence and inequivalence: <==> <=!=>

Grammar:

<equiv-expression> ::=
<expression> **<==>** *<expression>*
| *<expression>* **<=!=>** *<expression>*

Well-defined:

$$[\![\, e_1 \texttt{ <==> } e_2 \,]\!] \equiv [\![\, e_1 \,]\!] \;\&\; [\![\, e_2 \,]\!]$$
$$[\![\, e_1 \texttt{ <=!=> } e_2 \,]\!] \equiv [\![\, e_1 \,]\!] \;\&\; [\![\, e_2 \,]\!]$$

Type information:

- two arguments, each an expression of boolean type
- result is boolean

The <==> operator denotes equivalence: its value is true iff both operands are true or both are false. It is equivalent to equality (==), except that it is lower precedence. For example, `P && Q <==> R || S` is `(P && Q) <==> (R || S)`, whereas `P && Q == R || S` is `(P && (Q == R)) || S`.

The <=!=> operator denotes inequivalence: its value is true iff one operand is true and the other false. It is equivalent to inequality (!=), except that it is lower precedence.

For example, `P && Q <=!=> R || S` is `(P && Q) <=!=> (R || S)`, whereas `P && Q != R || S` is `(P && (Q != R)) || S`.

Both of these operators are associative and commutative. Accordingly left- and right-associativity are equivalent. The operators are not chained: `P <==> Q <==> R` is `(P <==> Q) <==> R`, not `(P <==> Q) && (Q <==> R)`; for example, `P <==> Q <==> R` is true if P is true and Q and R are false. Similarly `P <=!=> Q <=!=> R` is `(P <=!=> Q) <=!=> R` and is true if P is true and Q and R are false.

12.5.8 JML subtype: <: <:=

Grammar:

<subtype-expression> ::=
    <expression> **<:=** *<expression>*
  | *<expression>* **<:** *<expression>*

Well-defined:

$$[\![\, e_1 \texttt{ <:= } e_2 \,]\!] \equiv [\![\, e_1 \,]\!] \;\&\; [\![\, e_2 \,]\!]$$
$$[\![\, e_1 \texttt{ <: } e_2 \,]\!] \equiv [\![\, e_1 \,]\!] \;\&\; [\![\, e_2 \,]\!]$$

Type information:

- two arguments, each of type `\TYPE`
- result is boolean

The `<:=` operator denotes JML subtyping: the result is true if the left operand is a subtype of or the same type as the right operand. Note that the argument types are `\TYPE`, that is JML types (cf. §??). *Say more about relationship to Java subtyping*

JML also has the operator `<:`. In JMLv1 this also meant improper subtype. However the syntax is more indicative of proper subtype (subtype but not the same type). Accordingly, `<:` is deprecated and will be reintroduced later to mean proper subtype.

12.5.9 Lock ordering: <# <#=

Grammar:

<lockorder-expression> ::=
    <expression> **<#=** *<expression>*
  | *<expression>* **<#** *<expression>*

Well-defined:

$$[\![\, e_1 \texttt{ <\#= } e_2 \,]\!] \equiv [\![\, e_1 \,]\!] \wedge [\![\, e_2 \,]\!] \wedge e_1 \neq \mathit{null} \wedge e_2 \neq \mathit{null}$$
$$[\![\, e_1 \texttt{ <\# } e_2 \,]\!] \equiv [\![\, e_1 \,]\!] \wedge [\![\, e_2 \,]\!] \wedge e_1 \neq \mathit{null} \wedge e_2 \neq \mathit{null}$$

Type information:

- two arguments, each of reference type
- result is boolean

It is useful to establish an ordering of locks. If lock A is always acquired before lock B (when both locks are needed) then the system cannot deadlock by having one thread own A and ask for B while another thread holds B and is requesting A. Specifications may specify an intended ordering using axioms and then check that the ordering is adhered to in preconditions or assert statements. Neither Java nor JML defines any ordering on locks; the user must define an intended ordering with some axioms or invariants.

The `<#` operator is the 'less-than' operator on locks; `<#=` is the 'less-than-or-equal' version. That is

```
a <#= b ≡ ( a <# b | a == b )
```

Previously in JML, the lock ordering operators were just the < and <= comparison operators. However, with the advent of auto-boxing and unboxing (implicit conversion between primitive types and reference types) these operators became ambiguous. For example, if a and b are `Integer` values, then $a < b$ could have been either a lock-ordering comparison or an integer comparison after unboxing a and b. Since the lock ordering is only a JML operator and not Java operator, the semantics of the comparison could be different in JML and Java. To avoid this ambiguity, the syntax of the lock ordering operator was changed and the old form deprecated.

12.5.10 \result

Grammar:

<result-expression> `::= \result`

Well-defined:

$$[\![\ \backslash\textbf{result}\ ]\!] \equiv \textbf{true}$$

Type information:

- no arguments
- result type is the return type of the method in whose specification the expression appears
- may only be used in method specification clauses that are evaluated in a normally-terminating post-state.

The `\result` expression denotes the value returned by a method. The expression is only permitted in clauses of the method's specification that state properties of the state of a method after a normal exit. It is a type-error to use `\result` in the specification of a constructor or a method whose return type is `void`.

12.5.11 \exception

\exception is an OpenJML extension

Grammar:
<exception-expression> ::= \exception

Well-defined:
[[\exception]] ≡ true

Type information:

- no arguments
- the expression type is the type of the exception given in the signals clause; it is java.lang.Exception in duration and working_space clauses
- only permitted in method specification clauses that are evaluated in terminating-with-exception post-states.

The \exception expression denotes the exception object in the case a method exits throwing an exception. Using this expression is an alternative form to using a variable declared in the signals clauses's declaration. For example, the following two constructions are equivalent:

```
//@ signals (RuntimeException e) ...  e ...  ;
//@ signals (RuntimeException) ...  \exception ...  ;
```

Duration and workingspace clauses do not have a exception variable declaration and consequently need to use \exception.

12.5.12 \count

Grammar:
<count-expression> ::= \count | \index

Well-defined:
[[\count]] ≡ true

Type information: This expression is valid only in the body and specifications of a loop. It has type \bigint.

The value of this term is the number of times the loop body has been completed. If there are nested loops, it refers to the innermost loop that contains the expression. For a simple loop, like for (int i=0; i<10; i++) ... , \count is the same as the loop index i. In a more complex loop, like
for (int i=1; i<10; i*=2) ... , then some equality, such as $i == 2^{\backslash count}$ for this example, holds and using \count might be more useful.

In the for (var v: list) ... style of loop, there is no loop index. Then \count is equivalent to a ghost variable as in

```
1 //@ ghost count = 0;
2 for (var v: list) {
3    ...
4    count++;
5 }
```

The spelling of this term is `\count`, replacing `\index`, which has been removed from JML. The rationale is that `\count` connotes the count of the number of times the loop body has been executed, rather than the value of a loop index variable (which is the connotation of `\index`), though often those are the same.

12.5.13 `\values`

Grammar:

```
<values-expression> ::= \values
```

Well-definedness: $[[\ \backslash values\ ]] \equiv true$

Type information: The `\values` expression is permitted inside a loop body or loop specifications. Its type is `\seq<T>`, where `T` is the type of the declared loop index.

The value of the `values` expression is the sequence of values taken on by the loop index in the innermost enclosing loop.

The `values` expression is particularly useful in Java's foreach statements. For example, one might write, for some Collection c,

```
Collection<Integer> c = ...
int sum = 0;
//@ loop_invariant k == \values[\count];
//@ loop_invariant sum == (\sum int j; 0 <= j < \count; \values[k]);
//@ decreases c.size() - \count;
for (Integer k: c) {
   sum = sum + k;
}
```

12.5.14 `\old`, `\pre`, and `\past`

Grammar:

```
<old-expression> ::=
        ( \old( <expression> ( , <label> )? )
        ( \pre( <expression> )
        ( \past( <expression> )
<label> ::= <id>
```

Well-definedness: The expression is well-defined if the first argument is well-defined and any label argument names either a built-in label (§??) or an in-scope Java or JML ghost label (S??).

Type information: The type of the expression is the type of the first argument. Note though that the expression may be evaluated in a different state than the current state and different variable names may be in scope.

The scope of a label is the remainder of the block in which a label is defined, including any nested blocks. Note that in Java, a nested block is not allowed to reuse an

identifier as a label that labels an enclosing block. However, a label may be used subsequent to a block that a previous use labeled; it then hides the name of the earlier use, as shown in the following example.

```
public void m(int i) {
  a: {
     a: {} // forbidden nested use
     b: {}
  }
  a: {} // permitted subsequent use
  //@ assert \old(i,a) ==m ... // refers to the most recent use of a
  //@ assert \old(i,b) ==m ... // Error - b is out of scope
}
```

12.5.14.1 \old

The \old expression enables referring to the value of an expression in a previous program state. An \old expression without a label argument implicitly refers to the Old state (cf. §??). The value of the \old expression is the result of evaluating the first argument in the state designated by the second argument. Note that identifiers in the given argument are resolved and type-checked in the given state. Thus they may refer to different variables (with perhaps different types) than in the current state. The following example shows how different variables can have the same name.

```
public class Old {

  public boolean k;

  //@ requires k;
  public void m() {

    //@ assert k; // k is this.k, a boolean

    int k = 0;
    //@ assert k >= 0; // k is the local k, an int
    //@ assert \old(k); // k is this.k, a boolean
  }
}
```

12.5.14.2 \pre

The \pre expression is simply an abbreviation for \old with the built-in label Pre (cf. §??).

12.5.14.3 \past

The \past expression is similar to \old, but with slightly different semantics. It was proposed at the Shonan Workshop as a way to have field access operations within the method specifications carried out in a previous program state. It is currently

not adopted as a feature in JML, but the syntax is reserved for potential future use.
\past is an extension

Text needed

12.5.15 \fresh

Grammar:

```
<fresh-expression> ::=
        \fresh( <expression> [ , <java-identifier> ])
```

Well-definedness: The argument must be well-defined and non-null. The second argument, if present, must be an identifier corresponding to an in-scope label or a built-in label.

Type information:

- the first argument is an expression of reference type
- the optional second argument is an identifier, which must be the name of either a pre-defined label (§??) or a Java statement label or a JML ghost label (§??). If omitted, the built-in label `Old` is implicit.
- expression type is boolean
- `\fresh` may be used only in postcondition clauses or statement specifications

The arguments of the `\fresh` expression must be expressions that evaluate to non-null references. The `\fresh` expression is true iff the argument is a reference to an object that was not allocated in the state indicated by the given label.

12.5.16 \nonnullelements

Grammar:

```
<nonnullelements-expression> ::=
        \nonnullelements ( <expression> )
```

Well-definedness: the expression is well-defined iff the argument is well-defined (the argument is permitted to be null)

Type information:

- a single argument that is an expression of either Java array type, a Java iterable (which includes Java collections) or the `\seq`, `\set` or `\map` built-in types
- expression type is boolean

The `\nonnullelements` expression is true iff the argument is non-null and each element of the argument's value is not null.

Do we need a separate recursive version?

12.5.17 Arithmetic mode scope

Grammar:

```
<arithmetic-mode-expression> ::=
        <arithmetic-mode-name> ( <expression> )
<arithmetic-mode-name> ::= \java_math | \safe_math | \bigint_math
```

Well-definedness: The expression is well-defined iff the argument is well-defined.

Type information: The expression has the same type as the argument.

The function-like expressions `\java_math`, `\safe_math`, and `\bigint_math` may be used in specifications to change the arithmetic mode (§??) for the enclosed expression. Such arithmetic mode contexts override the default set for the program or the specific method and may be nested.

12.5.18 informal expression: (* . . . *)

Grammar:

```
<informal-expression> ::= (*  .*  *)
```

Well-defined:

$[\![\textbf{(*} .* \textbf{*)}]\!] \equiv \textbf{true}$
$[\![\textbf{JML.informal(}e\textbf{)}]\!] \equiv [\![e]\!]$

Type information:

- special syntax
- the argument of `JML.informal` is a string literal
- expression type is boolean; value is always true

The syntax of the informal expression is

```
(* ...  *),
```

where the ... denotes any sequence of characters not including the two-character sequence `*)`. An alternate form is

```
JML.informal(<expression>)
```

where *<expression>* is a String literal. The character sequence and the string expression are natural language text that may be ignored by JML tools; the intent is to convey to the reader some natural language specification that will not be checked by automated tools.

In the second form, the argument is type checked and must have type `java.lang.String`; it is not evaluated. It is generally a string literal.

The expression always has the value true.

Examples:

```
//@ ensures (* data structure is self-consistent *);
//@ ensures JML.informal("data structure is OK");
public void m()  ...
```

12.5.19 \type

Grammar:

<type-expression> ::=
    \type(*<jml-type-expression>* **)**

Well-defined:

[[**\type(***<jml-type-expression>***)**]] ≡ **true**

Type information:

- one argument, a type name
- result type is `\TYPE`

This expression is a type literal. The argument is the name of a type as might be used in a declaration; the type may be a primitive type, a non-generic reference type, a generic type with type arguments or an array type. The value of the expression is the JML type value corresponding to the given type. It is analogous to `.class` in Java, which converts a type name to a value of type `Class`. The type name is resolved like any other type name, with respect to whatever type names are in scope.

Generic types must be fully parameterized; no wild card designations are permitted. However type variables that are in scope are permitted as either stand-alone types or as type parameters of a generic type.

For more discussion of JML types and their relationships to Java types, see §??.

Examples: (T is an in-scope type variable)

```
1 //@ ... \type(int) ...
2 //@ ... \type(Integer) ...
3 //@ ... \type(java.lang.Integer) ...
4 //@ ... \type(java.util.LinkedList<String>) ...
5 //@ ... \type(java.util.LinkedList<String>[]) ...
6 //@ ... \type(T) ...
7 //@ ... \type(java.util.LinkedList<T>) ...
```

12.5.20 \typeof

Grammar:

<typeof-expression> ::=
    \typeof (*<expression>* **)**

Well-defined:

$$[[\ \backslash\textbf{typeof}(e)\ ]] \equiv [[\ e\ ]] \wedge e \neq \mathit{null}$$

Type information:

- one expression argument, of any type
- result type is `\TYPE`

The `\typeof` expression returns the dynamic type of the expression that is its argument. In run-time checking this may require evaluating the argument. This operation returns a JML type (`\TYPE`); it is analogous to the Java method `.getClass()`, which returns a Java type value (of type `Class`).

Expressions having primitive Java type are allowed as arguments, but JML-typed expressions are not allowed.

Verify that JML types are not allowed

Examples:

```
Object o = new Integer(5); \\
// o has static type Object, but dynamic type Integer
//@ assert \typeof(o) == \type(Integer); // - true
//@ assert \typeof(o) == \type(Object); // - false
//@ assert \typeof(5) == \type(int); // - true
```

12.5.21 \arraytype

Grammar:

<arraytype-expression> ::=
    \arraytype (<expression>)

Well-defined:

$$[[\ \backslash\textbf{arraytype}(e)\ ]] \equiv [[\ e\ ]]$$

Type information:

- one argument, of JML type `\TYPE`
- expression has type `\TYPE`

This operator takes a type argument and returns a type value representing a Java array of the argument type. So for any `\TYPE t`, `\isarray(\arraytype(t))` is true and `\elemtype(\arraytype(t))` is `t`.

12.5.22 \isarray

Grammar:

<isarray-expression> ::=
    \isarray (<expression>)

Well-defined:

$$[[\ \backslash\textbf{isarray(}e\textbf{)}\ ]] \equiv\ [[\ e\ ]]\ \&\ (e \neq \mathit{null} \text{ if e has } \texttt{Class} \text{ type})$$

Type information:

- one argument, either of JML type `\TYPE` or is non-null and has Java type `Class`
- expression has type `boolean`

This operator takes a type argument and returns true iff the argument is an array type. Note that the argument is a type value not a general Java reference value. So one must often write `\isarray(\typeof(o))`.

12.5.23 \elemtype

Grammar:

```
<elemtype-expression> ::=
        \elemtype ( <expression> )
```

Well-defined:

$[[\ \backslash\textbf{elemtype(}e\textbf{)}\ ]] \equiv\ [[\ e\ ]]\ \&\ \backslash\mathit{isarray}(e)$, if `e` has type `\TYPE`

$[[\ \backslash\textbf{elemtype(}e\textbf{)}\ ]] \equiv\ [[\ e\ ]]\ \&\ e \neq \mathit{null}\ \&\ \backslash\mathit{isarray}(e.\mathit{getClass}())$, if `e` has type `Object`

Type information:

- one argument, of type `\TYPE` or a general Java reference value that is non-null and has an array type
- expression has type `\TYPE`

This operator returns the static element type of an array type.

Canonically, the argument of `\elemtype` is a `\TYPE` value that is an array type. For example, though not very usefully, `\elemtype(\type(Integer[]))`. Note that the expression is not well-defined if the argument is not an array type; whether it is an array type or not can be determined using the `\isarray` operator.

A common application of this operator is `\elemtype(\typeof(a))`, where `a` is some non-null array value; the result is then the *dynamic* element type of the array. This expression is so common that JML allows writing `\elemtype(a)` (for a non-null Java expression `a` with some array type) as shorthand for `\elemtype(\typeof(a))`.

Examples:

```
1 //@ assert \elemtype(\type(int[])) == \type(int);
2 Object o = new Integer[2];
3 //@ assert \elemtype(o) == \type(Integer);
```

One could allow any Java reference value as the argument of `\elemtype`, returning `null` if the argument was not an array. However, there is no `null` value of `\TYPE`.

12.5.24 \erasure

Grammar:

```
<erasure-expression> ::=
        \erasure ( <expression> )
```

Well-defined:

$$[[\ \textbf{\textbackslash erasure}(e)\ ]] \equiv\ [[\ e\ ]]$$

Type information:

- one argument, of JML type `\TYPE`
- expression has Java type `Class`

This operator takes a type argument and returns a value of the Java type `Class` that is the erasure of the JML type. So `\erasure(\type(List<Integer))` equals `Integer.class` and in general `\erasure(\TYPE.of(t)) == t`.

12.5.25 \is_initialized

Grammar:

```
<is-initialized-expression> ::=
        \is_initialized ( <type-name> ... )
      | \is_initialized ( )
```

Type information: The argument must be a list of names of reference types.

The value of this expression is true iff the classes named as arguments have all completed their static initialization. The expression has value true if it has no arguments.

12.5.26 \invariant_for

Grammar:

```
<invariant-for-expression> ::=
        \invariant_for ( <expression> )
```

Well-definedness: The expression is well-defined if the argument is. The argument may be null.

Type information: The expression takes one argument, which is a possibly-null-valued expression of any reference type. The result has boolean type.

The `invariant_for` expression is equivalent to the conjunction of the non-static invariants in the static type of the receiver and all its super classes and interfaces (recursively), with the argument as the receiver for the invariants.

If the value of the argument is `null`, the value of the expression is `true`.

Questions: Should this be the conjunction of invariants of the dynamic type?

Does visibility matter?

Does the order of the conjunctions matter? A natural order would be: the order of invariants is (1) that invariants of super classes and interfaces occur before derived classes and interfaces, (2) Object is first and the named type is last, and (3) within a type, invariants occur in textual order.

The rationale for permitting null-valued arguments is that then it is easy to write the conjunction of a series of invariants for variables: `\invariant_for(a) && \invariant_for(b) && ...` without having to repeatedly check for null values. A quite convenient extension would be to allow multiple arguments to this function, with the meaning being the conjunction of the individual invariants.

12.5.27 \static_invariant_for

Grammar:

```
<static-invariant-for-expression> ::=
        \static_invariant_for ( <type-name> )
```

Well-definedness: [[`\static_invariant_for` (*<type-name>*)]] ≡ *true*

Type information: The argument is a syntactic type name (not a typed expression) that is the name of a Java or JML (that is, a model) class or interface, and not a primitive type. The value of the expression is boolean.

This expression returns the conjunction of the static invariants of the given type. It does not include invariants of super- or sub-types (either classes or interfaces).

If the type named in the argument is a Java generic type, any type parameters are optional. Recall that in Java type variables may not be used in static contexts; a declaration of a static invariant is a static context, so type variables may not be used in static invariants. Thus any concrete type given as a type parameter is irrelevant to the invariant. For example
`\static_invariant_for(java.util.List)` and
`\static_invariant_for(java.util.List<Integer>)` mean the same thing, while `\static_invariant_for(T)`, where `T` is a type variable, is illegal.

Open questions: does visibility matter? Do we exclude invariants of super-types? Does order of conjoining matter?

The comment given in §?? applies here as well.

12.5.28 \not_modified

Grammar:

```
<not-modified-expression> ::=
        \not_modified ( <store-ref-expression> ... )
      | \not_modified ( )
```

Well-definedness: well-defined iff the arguments are well-defined

Type information:

- any number of arguments, each expression of any type other than void
- result type is `boolean`

A `\not_modified` expression is a two-state expression that may occur only in post-condition and statement specification clauses. It satisfies this equivalence:

```
\not_modified(o) == ( \old(o) == (o) )
```

The argument may be null.

A `\not_modified` expression with multiple arguments is the conjunction of the corresponding terms each with one argument; if `\not_modified` has no arguments, its value is true.

If an argument has `\locset` type, the meaning is that no memory location in the `\locset` is modified.

The expression is true iff the values of the memory locations in the locations sets designated by the *<store-ref-expressions>* are the same in the post-state as in the pre-state; for example, `\not_modified(xval,yval[*])` says that the field `xval` and each of the array elements of `yval` have the same value in the pre- and post-states (in the sense of the equals method for their types).

`\not_modified` allows one to specify benevolent side-effects, as one can name `x.f` (or a data group in which it participates) in an assignable clause, but use `\not_modified(x.f)` in the postcondition.

Use == or equals?

Does the predicate refer to the actual value – in which case we should stipulate expressions not store-ref-expressions – or all the data group (which would be more consistent with similar features)?

12.5.29 \not_assigned

Grammar:

<not-assigned-expression> ::=
    \not_assigned (*<store-ref-expression>* ...)

Well-definedness: The expression is well-defined iff each argument is well-defined.

Type information: Each argument must be properly typed. The expression has boolean type.

This expression may be used only in (method or block specification) postconditions or in specification statements. In a postcondition of a contract, the expression is true if none of the arguments have been assigned to in the body of code that the contract specifies (i.e., a method body or a block). In a specification statement (e.g., an `assert` statement), the expression is true if none of the arguments have been assigned to since the beginning of the inner-most block contract, or of the method

body if there is no enclosing block contract, and up to the position of the containing statement.

The `\not_assigned`, `\not_modified` and various `\only_ZZZ` operators are two-state operators comparing the current program state to `Old`. There is no syntax to use a different program label.

12.5.30 `\only_assigned`, `\only_accessed`, `\only_captured`

Grammar:

```
<only-assigned-expression> ::=
        \only_assigned ( <store-ref-expression> ... )
<only-accessed-expression> ::=
        \only_accessed ( <store-ref-expression> ... )
<only-captured-expression> ::=
        \only_captured ( <store-ref-expression> ... )
```

Well-definedness: The expression is well-defined iff each argument is well-defined.

Type information: Each argument must be properly typed. The expression has boolean type.

The argument list of this expression denotes a `\locset`, as described in §??.

These expressions may be used only in postconditions or in specification statements. In a postcondition of a contract, the expression is true iff the set of locations that have been assigned to, accessed, or captured, respectively, in the body of code that the contract specifies (i.e., a method body or a block) is a subset of the argument. In a specification statement (e.g., an `assert` statement), the expression is true if the set of locations that have been assigned to, accessed, or captured, respectively, since the beginning of the inner-most block contract, or of the method body if there is no enclosing block contract, and up to the position of the specification statement containing the expressionis a subset of the argument.

A predicate such as `\only_ZZZ(x.f)` refers to the entire data group named by `x.f` not just to the location `x.f` itself. These operators may be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that the (concrete) locations in that model field's data group are permitted to be assigned/accessed/captured during the method's execution.

`\only_assigned` The JML operator `\only_assigned` is true iff the method's execution only assigned to a subset of the data groups named by the given fields. For example, `\only_assigned(xval,yval)` says that no fields outside of the data groups of `xval` and `yval` were assigned by the method. This includes both direct assignments in the body of the method and assignments during calls that were made by the method (and methods those methods called, etc.).

\only_accessed The JML operator `\only_accessed` is true iff the containing method's execution only reads from a subset of the data groups named by the given fields. For example, `\only_accessed(xval,yval)` says that no fields outside of the data groups of `xval` and `yval` were read by the method. This includes both direct reads in the body of the method, and reads during calls that were made by the method (and methods those methods called, etc.).

\only_captured The JML operator `\only_captured` is true iff the containing method's execution only captured references from a subset of the data groups named by the given fields. A reference is captured when it is stored into a field (as opposed to a local variable). Typically a method captures a formal parameter (or a reference stored in a static field) by assigning it to a field in the method's receiver (the `this` object), a field in some object (or to an array element), or to a static field. A predicate such as `\only_captured(x.f)` refers to the references stored in the entire data group named by `x.f` in the pre-state, not just to those stored in the location `x.f` itself. However, since the references being captured are usually found in formal parameters, the complications of data groups can usually be ignored.

12.5.31 \only_called

Grammar:

<only-called-expression> ::=
    \only_called (*<method-signature>* ...)

Type information: The arguments are not typed. The expression has boolean type.

Well-definedness: The expression is always well-defined (given that all the arguments are type-correct, as defined in §??).

The argument list of this expression denotes a set of methods, as described in §??. If there are no arguments, the set of methods is empty.

This expression may be used only in postconditions or in specification statements. In a postcondition of a contract, the expression is true if the set of methods that have been called in the body of code that the contract specifies (i.e., a method body or a block) is a subset of the argument. In a specification statement (e.g., an `assert` statement), the expression is true if the set of methods that have been called since the beginning of the inner-most block contract, or of the method body if there is no enclosing block contract, and up to the position of the specification statement containing the expression is a subset of the argument.

Although this feature is part of JML, it has been used only very little. It is expected to need adjustment based on experimentation.

12.5.32 \lockset and \max

Grammar:

```
<locset-expression> ::=
        \lockset
      | \max ( <expression> )
```

Type information:

- The type of `\lockset` is `\set<Object>`
- The type of the argument of `\max` must be `\set<Object>`; the result type of the `\max` expression is `Object`.

Well-definedness:

$$[\![\ \texttt{\textbackslash lockset}\ ]\!] \equiv \mathit{true}$$
$$[\![\ \texttt{\textbackslash max}(\textit{<expression>})]\!] \equiv [\![\ \textit{<expression>}\ ]\!] \wedge\ \textit{<expression>} \neq \mathit{null}$$

The value of `\lockset` is a set of `Object`s that have locks. The value of `\max` is the element of such a set that has the largest lock value, as defined by axioms on the `<#` operator; the value of `\max` is `null` if the argument is an empty set.

12.5.33 \reach

Grammar:

```
<reach-expression> ::= \reach ( <jml-expression> )
```

Type information:

- The construct takes just one argument of any reference type. The argument may be null.
- The result type is `\set<Object>`.

Well-definedness: $[\![\ \textit{\textbackslash reach}(\textit{<expression>})\ ]\!] \equiv [\![\ \textit{<expression>}\ ]\!]$

The `\reach` expression allows one to refer to the set of objects reachable from some particular object. The syntax `\reach(x)` denotes the smallest set containing the object denoted by x, if any, and all objects accessible through all fields (of any visibility) of objects in this set. That is, if x is null, then this set is empty otherwise it contains x, all objects accessible through all fields of x, all objects accessible through all fields of these objects, and so on, recursively. If x denotes a model field (or data group), then `\reach(x)` denotes the smallest set containing the objects reachable from x or reachable from the objects referenced by fields in that data group. Objects reachable from an object include those reachable through static fields and through super-class fields.

Some use cases are needed for this feature. For example, if the need is to know all objects that are not eligible for garbage collection, then `\reach` would include fields of superclasses that cannot be accessed (because of visibility) from a derived class. But if the need is just to know what objects can be accessed through some expression, then another rule is needed. Also, what about (non-fresh) objects that are returned by reachable method calls?

There has been no work with this JML feature to validate how tools should implement reasoning tasks. Perhaps it should be put in an experimental category until more experience is gained.

12.5.34 Store-ref expressions

Grammar:

<store-ref-expression> **::=**
  <non-wild-store-ref-expression>
| *<non-wild-store-ref-expression>* **.** ***** (a wild-field store-ref)
| *<type-name>* **.** ***** (a static wild-field store-ref)

<non-wild-store-ref-expression> **::=**
  <java-identifier> (a field or local variable store-ref)
| *<type-name>* **.** *<java-identifier>* (a static field store-ref)
| *<non-wild-store-ref-expression>* **.** *<java-identifier>* (a field store-ref)
| *<non-wild-store-ref-expression>* **[** *<expression>* **]** (an array element store-ref)
| *<non-wild-store-ref-expression>* **[** [*<expression>*] **..** [*<expression>*] **]** (an inclusive array-range store-ref)
| *<non-wild-store-ref-expression>* **[** [*<expression>*] **:** [*<expression>*] **]** (an exclusive array-range store-ref)
| *<non-wild-store-ref-expression>* **[** ***** **]** (a wild-array store-ref)

Type information: For expression o and a type-name T,

- A *<store-ref-expression>* has type `\locset`. Each particular syntax identifies a set of memory locations that then constitute the designated `\locset`.
- In $o.f$, f must name a field of the type of o, which must have reference type and not be null.
- In $o.*$, o must be a non-null value of a reference type.
- In $T.f$, T must name a reference type, f must name a static field of T.
- In $T.*$, T must be a valid type name of a reference type
- In $o[i]$, o must be an array type, i must be (convertible to) `\bigint`
- In $o[i..j]$ and $o[i:j]$, o must be an array type, i and j must be (convertible to) `\bigint`

- In $o[*]$, o must be a non-null array object

Well-definedness:
$[[\ o.f\ ]] \equiv [[o]] \wedge o \neq null$
$[[\ o.*\ ]] \equiv [[o]] \wedge o \neq null$
$[[\ T.f\ ]] \equiv true$
$[[\ T.*\ ]] \equiv true$
$[[\ a[i]\ ]] \equiv [[a]] \wedge a \neq null \wedge [[\ i\ ]] \wedge 0 \leq i < a.length$
$[[\ a[i..j]\ ]] \equiv [[a]] \wedge a \neq null \wedge [[\ i\ ]] \wedge [[\ j\ ]] \wedge 0 \leq i \wedge j < a.length$
$[[\ a[i:j]\ ]] \equiv [[a]] \wedge a \neq null \wedge [[\ i\ ]] \wedge [[\ j\ ]] \wedge 0 \leq i \wedge j \leq a.length$
$[[\ a[*]\ ]] \equiv [[a]] \wedge a \neq null$

A store-ref expression denotes a set of memory locations, that is a `\locset` (§??).

- A local variable store-ref (`v`) denotes the location of that (stack) variable
- A field store ref (`o.f` or `f`, where the `f` names a field, that is, `this.f`) denotes the location of the named field of the object; if the named field is a model field, the expression denotes the set of all locations that are contained in that model field
- A static field store ref (`T.f`) denotes the location of the named static field of the class; if the named field is a model field, the expression denotes the set of all locations that are contained in that model field
- A wild-field store-ref (`o.*` denotes the set of locations of all the fields of the given object, including all fields of suprtypes, but not static fields.
- A static wild-field store-ref (`T.*`) denotes the set of locations of all the static fields of the given type (but not static fields of supertypes).
- An array element store-ref ($a[i]$) denotes the one array-element of the given array
- An array range store-ref ($a[i..j]$) denotes the locations of the array elements from indices i to j *inclusive*; if $j < i$ the set is empty
- An array range store-ref ($a[i : j]$) denotes the locations of the array elements from indices i to j *exclusive*; if $j \leq i$ the set is empty
- The expressions on either side of **..** and **:** are optional. A missing left operand is 0 and a missing right operand means the range extends through the end of the array.
- A wild-array store-ref ($a[*]$) denotes the locations of all of the array elements of the given array

Store-ref expressions serve as literals for `\locset`s. In some contexts an expression can be interpreted either as a store-ref expression or as a regular expression with `\locset` type. Such situations are disambiguated as a regular expression. If the store-ref expression is desired (that is, the memory location is intended, not the value

of what is in the memory location), then the store-ref expression must be wrapped in `\locset( )`.

The wild-card syntax, `o.*`, can be thought of this way: the `*` designates a model field containing all the fields of the object including super-class fields. Thought of this way it is natural that all fields are included, whatever their visibility.

Note that the grammar does not allow constructions like $o.*.*$, but it does allow $a[*].f$ and $a[*][*]$.

Allowing the beginning and ending expressions to be optional is syntactic sugar that avoids having to write out expressions like `a[0..a.length-1]`. It does mean that `a[*]`, `a[..]`, and `a[:]` all mean the same thing. They are all retained for historical backwards-compatibility.

JML has always had the `..` syntax, which has always meant an inclusive range. This is unfortunate because a half-exclusive range is easier to read and write: a range is divided into subranges by `a[0:i]`, `a[i:j]`, `a[j:a.length]`, rather than having to insert `-1` in various places. But rather than change or deprecate, `..`, JMLv2 adds the `:` half-exclusive range designator.

Questions: Does T. include supertype fields? Should there be a syntax for all static fields including supertype fields? Does o.* include fields of the dynamic type?*

Two aspects of the store-ref semantics may need to await further experience. First, is there a need to write a store-ref expression meaning all the static fields of a class, including the static fields of supertypes (perhaps `T.^*`).
Second, is there a need to express the collection of all the fields of an object, including any fields of its dynamic type?

12.6 JML resource expressions

The following expressions enable statements about use of resources — time and memory space — in JML specifications. They are used in conjunction with the `duration` (§??) and `working_space` (§??) clauses. They have had very little use and are expected to evolve with more experimentation.

12.6.1 `\duration`

Grammar:

```
<duration-expression> ::= \duration ( <expression> )
```

Type information:

- one argument, an expression of any type, including void
- well-defined iff the argument is well-defined
- expression has type `\bigint`

This is feature has not yet had a great deal of use.

The value of a `\duration` expression is the maximum number of virtual machine cycles needed to evaluate the argument. The argument is not actually executed and need not be pure. However, reasoning about assertions containing `\duration` expressions is based on the specifications of method calls within the expression, not on their implementation. Consequently, for a `\duration` expression to be useful, any methods or constructors within its argument must have a `duration` expression as part of their method specifications.

The argument must be an executable expression because different expressions (e.g., method calls with different arguments) may consume different numbers of machine cycles during execution.

One of many issues here is how to abstract the duration of computations. A simple approach might be to just count the number of operations. The approach would depend on whether one just needs an abstract measure of duration or something close to actual time. How abstract is a 'virtual machine cycle'?

What about runtime assertion checking

12.6.2 \working_space

Grammar:

<working-space-expression> ::= \working_space (<expression>)

Type information:

- one argument, of any type, including void
- expression has type `\bigint`

Well-definedness: [[`\working_space`(*<expression>*)]] ≡ [[*<expression>*]]

This is feature has not yet had a great deal of use.

The result of the `\working_space` expression is the number of bytes of heap space that would be required to evaluate the argument, if it were executed. The argument is not actually executed and may contain side-effects. That is, if `\working_space`(*expr*) free bytes are available in the system and there are no other concurrent processes or threads executing, then evaluating *expr* will not cause an `OutOfMemory` error. *Is this last sentence true?*

The argument must be an executable expression because different expressions (e.g., method calls with different arguments) may consume different amounts of memory space during execution.

12.6.3 \space

Grammar:

<space-expression> ::= \space (<expression>)

Type information:

- one argument, of any reference type
- expression has type `\bigint`

Well-definedness: [[`\space`(*<expression>*)]] ≡ [[*<expression>*]]

This feature has not yet had a great deal of use.

The result of a `\space` expression is the number of bytes of heap space occupied by the argument. This is a shallow measure of space: it does not include the space required by objects that are referred to by members of the object, just the space to hold the references themselves and any primitive values that are members of the argument.

One of many issues here is how to account for padding for alignment, which would make the analysis platform dependent.

Chapter 13

Arithmetic modes

Programming languages use integral and floating-point values of various ranges and precisions. However, often specifications are written and understood as mathematical integer and real values. JML's arithmetic modes allow the choice of using mathematical or machine-precision types for integers and floating-point numbers in specifications. They also allow enabling and disabling warnings about out-of-range operations.

In JML, the type of mathematical integers is expressed as `\bigint`; the type of mathematical reals is `\real`.

13.1 Integer arithmetic

13.1.1 Integer arithmetic modes

Chalin [**?**] surveyed programmer expectations and desires and identified three useful integer arithmetic modes:

- Java mode: values belong to one of Java's fixed-bit-length data types; overflows and underflows either occur silently or result in undefined values according to the rules of Java arithmetic.
- Safe mode: values belong to one of Java's fixed-bit-length data types; overflows and underflows cause static or dynamic warnings.
- Math ('bigint') mode: numeric values are promoted to mathematical types prior to arithmetic operations, so arithmetic operations do not result in overflow or underflow warnings; warnings may be issued when values are assigned or explicitly cast back into fixed-bit-length variables.

Consider this example:

```
1 //@ ensures \result == a + b;
2 public int add(int a, int b) {
3   return a+b;
4 }
```

If both the specification and implementation are interpreted in 'java' mode, then the specification is consistent with the implementation, as the writer might expect. But any overflow error will be missed and might be a bug.

An alternate choice is to have the implementation use 'safe' mode, in which case a warning is issued if overflow occurs. To match that, the specification would use 'math' mode. This better identifies errors, but may lead to confusion as the specification as given is not consistent with the implementation. It may not be intuitive to the specifier that a precondition is needed:

```
1 //@ requires Integer.MIN_VALUE <= a + b <= Integer.MAX_VALUE;
2 //@ ensures \result == a + b;
3 public int add(int a, int b) {
4   return a+b;
5 }
```

The fact that the specification is in 'Math' mode makes the precondition readily intuitive, which it would not be if the specification used 'java' or 'safe' mode (in which case the precondition would be a tautology).

Chalin proposed that most of the time, programmers would like Safe mode semantics for programming language operations and Math mode for specification expressions. JML has adopted these modes as the default

JML contains a number of modifiers (§??) and pseudo-functions (§??) to control which mode is operational for a given sub-expression. As would be expected, the innermost mode indicator in scope for a given expression overrides enclosing arithmetic mode indicators. The arithmetic mode can be set separately for the Java source code and the JML specifications.

- the class and method modifiers `code_java_math`, `code_safe_math`, and `code_bigint_math`, and corresponding annotation types `@CodeJavaMath`, `@CodeSafeMath`, and `@CodeBigintMath`, set the default arithmetic mode for all expressions in Java source code within the class or method (unless overridden by a nested mode indicator).
- the class and method modifiers `spec_java_math`, `spec_safe_math`, and `spec_bigint_math`, and corresponding annotation types `@SpecJavaMath`, `@SpecSafeMath`, and `@SpecBigintMath`, set the default arithmetic mode to be used within JML specifications, within the respective class or method.
- Within specification expressions, the operators `\java_math`, `\safe_math`, and `\bigint_math` can be used to locally alter the arithmetic mode. These take one argument, an expression, and set the arithmetic mode for evaluating

that expression (unless overridden by a nested arithmetic mode operator); the result of these operators has the type and value of its argument, adjusted for the arithmetic mode.

- the default arithmetic modes for the whole static or dynamic analysis are set by the tool in use (e.g., by command-line options); in the absence of any other setting, the default modes should be safe math for Java code and bigint math for specifications.

The arithmetic mode affects the semantics of these operators:

- arithmetic: unary plus, unary minus, and binary `+ - * / %`
- shift operations: `<< >> >>>`
- cast operations

The semantics of these operations in each mode are described in the following sections.

13.1.2 Semantics of Java math mode

Java defines several fixed-precision integral and floating-point data types. In addition JML allows the `\bigint` and `\real` data types. The arithmetic and shift operators act on these data types as follows:

- implicit conversion. The operands are individually converted to potentially larger data types as follows:
  - if either operand is `\real`, the other is converted to `\real`,
  - else if one operand is `\bigint` and the other either `double` or `float`, they both are converted to `\real`,
  - else if either operand is `double`, the other is converted to `double`,
  - else if either operand is `float`, the other is converted to `float`,
  - else if either operand is `\bigint`, the other is converted to `\bigint`,
  - else if either operand is `long`, the other is converted to `long`,
  - else both operands are converted to `int`.
- the result type of each arithmetic operator is the same as that of its implicitly converted operands
- the result type of a shift operator is the same as its left-hand operand
- the argument of unary `+` or `-`, if of `char`, `short` or `byte` type, is implicitly converted to `int`, and is otherwise unchanged
- `double` and `float` operators behave as defined by the IEEE standard
- the unary plus operation simply returns its operand (after implicit conversion)

- the unary minus operation, when applied to the least `int` or `long` value will overflow, returning the value of the operand
- binary add, subtract, and multiply operations on `int` or `long` values may overflow or underflow; the result is truncated to the number of bits of the result type
- the binary divide operation will overflow when the least value of the type is divided by -1. The result is the least value of the result type.
- the binary modulo operation does not overflow. Note that the sign of the result is the same as the sign of the *dividend*, and that it is always true that $x == (x/y) * y + (x\%y)$ for x and y both `int` or both `long`.[1]
- the shift operators apply only to integral values. Note that in Java, $x << y == x << (y\&n)$ where n is 31 when x is an `int` and 63 if x is a `long`. However, no such adjustment to the shift amount happens when the type is `\bigint`.
- In narrowing cast operations, the value of the operand is truncated to the number of bits of the given type.
- A divide or modulo operation with the right operand of 0 produces a divide-by-zero error

13.1.3 Semantics of Safe math mode

The result of an operation in safe math mode is the same as in Java math mode, except that any out of range value causes a verification error in static or dynamic checking. These warnings are produced in these cases:

- a unary minus applied to the least value of the int or long type
- a binary plus or minus or multiply of integral values where the mathematical result would lie outside the range of the data type
- a divide on integral values where the numerator is the least value of the type and the denominator is -1
- a shift operation in which the right-hand value is negative or is larger than 31 for int values or 63 for long values
- narrowing cast operations on integral values in which the result is not equal to the argument (because of truncation).
- a divide or modulo operation with the right operand of 0

There is one additional nuance of safe math mode. The value of `\sum`, `\prod`, and `\num_of` quantifiers is computed in `bigint` mode and then the result is cast to the type of the quantifier expression; if there is an overflow on that cast, a verification warning is given. The result is the same as if the expression were computed in java

[1] https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.17.3

math mode. No overflow warning happens if intermediate results overflow but the final result is in range. Note though that the default arithmetic mode for specifications is `bigint` mode, so this situation rarely arises.

13.1.4 Semantics of Bigint math mode

In `bigint` math mode, all reasoning is performed with each integral value promoted to an infinite-precision mathematical value. Thus there are no warnings issued on arithmetic operations (except divide or modulo by 0). Warnings may be issued when a mathematical value is cast to a fixed-precision programming language type or assigned to a variable of a fixed-precision type.

13.1.5 Arithmetic modes and Java code

Java programs are executed using what in JML is called java-math mode. However, when analyzing a program using JML, safe-math mode is assumed for Java code so that any arithmetic overflows are discovered. Though there are situations in which overflows are intended, that is ordinarily not the case.

JML allows math-mode (bigint-mode) to be stipulated for analysing Java code as well. However the semantics of this mode are not yet defined.

Question: In math mode is it just the operations that are on math types and then casts or writes to variables might trigger warnings; or are all integral data types implicitly bigint and real? - this latter can work for local declarations but not for formal parameters of callees

13.2 Real arithmetic modes

Operations using real numbers are quite straightforward; there are only limited cases (such as divide by zero) for which the results are undefined.

In contrast, floating point (FP) operations are rather complex. JML presumes that FP arithmetic follows the IEEE-754 standard. Results of operations are rounded, so using `==` is perilous. There are also a positive and negative zero, a positive and negative infinity, and NaN (not-a-number). Furthermore, operations on NaNs are unusual in that `NaN == NaN` and `NaN != NaN` are both false and the `equals` operation on `Double` and `Float` values is different than the `==` operation on `double` and `float` values.

It is tempting to treat all Java `double` and `float` quantities as real numbers. However, the `Double` and `Float` classes define constants for NaN and positive and negative infinity, so some accommodation must be made for these aspects of FP numbers.

Automated reasoning about floating-point and real values is very much an area of research. Encoding such operations in SMT is fairly recent and logical solvers still have difficulty with non-linear arithmetic.

JML defines two floating-point arithmetic modes: `fp_strict` and `fp_real`.

Note that Java once had an `strictfp` modifier. This modifier is deprecated because FP arithmetic must always follow IEEE standard rules. The modifier is not supported by JML.

13.2.1 fp_strict mode

In `fp_strict` mode, all operations among `double` and `float` quantities have the results that are stipulated by the IEEE-754 standard. Conversions between floating point values and real values are permitted.

When converting from a real value to a floating-point value, the result is the nearest representable floating-point value, or positive or negative infinity if the real value is outside the range of representable FP numbers. Negative zero and NaN are never produced.

In practice, static verification in `fp_strict` requires a backend SMT solver that supports reasoning about floating-point arithmetic.

what about: rounding modes, conversion to integer values

13.2.2 fp_real mode

The semantics of this mode need more work

In `fp_real` mode, a FP number is modeled as either a NaN, positive infinity, negative infinity, negative zero, or a real number. Most of the time, operations on FP numbers are just operations on corresponding reals, with only the special cases of operations with undefined results needing the special floating point values.

The results of operations using `fp_real` are a bit more intuitive than when using precise floating-point, but they are not the same. For example, the product of two finite real numbers will always produce another finite real number; but the product of two FP numbers may overflow and yield an infinity value.

Safe mode - i.e. overflow and NaN warning

Local change of mode

TBD -reference what is done in ACSL

Chapter 14

Specification and verification of lambda functions

TODO: to be written

Chapter 15

Universe types

TODO: To be written

Chapter 16

Model Programs

This chapter discusses JML's model programs, which are adapted from the refinement calculus [?] [?] [?] [?] [?]. Details of JML's design and semantics for model program specifications are described in a paper by Shaner, Leavens, and Naumann [?].

The material for this chapter is a somewhat edited and revised version of the Model Programs description in the Draft Reference Manual.

16.1 Ideas Behind Model Programs

The basic idea of a model program is that it is a specification that is written as an abstract algorithm. Such an abstract algorithm specifies a method in the sense that the method's execution should be a refinement of the model program.

JML adopts ideas from Büchi and Weck's "grey-box approach" to specification [?] [?]. However, JML structurally restricts the notion of refinement by not permitting all implementations with behavior that refines the model program, but only allowing implementations that syntactically match the model program [?]. The initial JML notion of matching used refining-statements. This was a simple and easy to understand technique for specifying and verifying both higher-order features and callbacks.

Consider the following example (from a survey on behavioral subtyping by Leavens and Dhara [?]). In this example, both the methods are specified using model programs, which are explained below. [*This example uses the syntax described in the paper and in the previous version of JML; the current syntax is described in §*??.]

```
1  package org.jmlspecs.samples.dirobserver;
2
3  //@ model import org.jmlspecs.models.JMLString;
4  //@ model import org.jmlspecs.models.JMLObjectSetEnumerator;
5  /** Directories that can be both read and written. */
6  public interface Directory extends RODirectory {
7    /** Add a mapping from the given string
8     *  to the given file to this directory.
9     */
10   /*@ public model_program {
11     @   normal_behavior
12     @     requires !in_notifier && n != null && n != "" && f != null;
13     @     assignable entries;
14     @     ensures entries != null
15     @        && entries.equals(\old(entries.extend(
16     @                                        new JMLString(n), f)));
17     @   maintaining !in_notifier && n != null && n != "" && f != null
18     @               && e != null;
19     @   decreasing e.uniteratedElems.size();
20     @   for (JMLObjectSetEnumerator e = listeners.elements();
21     @        e.hasMoreElements(); ) {
22     @     set in_notifier = true;
23     @     ((DirObserver)e.nextElement()).addNotification(this, n);
24     @     set in_notifier = false;
25     @   }
26     @ }
27     @*/
28   public void addEntry(String n, File f);
29
30   /** Remove the entry with the given name from this directory. */
31   /*@ public model_program {
32     @   normal_behavior
33     @     requires !in_notifier && n != null && n != "";
34     @     assignable entries;
35     @     ensures entries != null
36     @        && entries.equals
37     @               (\old(entries.removeDomainElement(
38     @                                              new JMLString(n))));
39     @   maintaining !in_notifier && n != null && n != "" && e != null;
40     @   decreasing e.uniteratedElems.size();
41     @   for (JMLObjectSetEnumerator e = listeners.elements();
42     @        e.hasMoreElements(); ) {
43     @     set in_notifier = true;
44     @     ((DirObserver)e.nextElement()).removeNotification(this, n);
45     @     set in_notifier = false;
46     @   }
47     @ }
48     @*/
49   public void removeEntry(String n);
50 }
```

Both model programs in the above example are formed from a specification statement, which begins with the keyword `normal_behavior` in these examples, and a for-loop. The key event in the for loop bodies is a method call to a notification method (in the examples this is eithr `addNotification` or `removeNotification`). These calls must occur in a state equivalent to the one reached in the model program for the implementation to satisfy the specification.

The specification statements abstract away part of a correct implementation. The `normal_behavior` statements in these examples both have a precondition, a frame (an assignable clause), and a postcondition. These mean that the statements that they abstract away from must be able to, in any state satisfying the precondition, finish in a state satisfying the postcondition, while only assigning to the locations (and their dependees) named in the frame. For example, the first specification statement says that whenever `in_notifier` is false, `n` is not null and not empty, and `f` is not null, then this part of the method can assign to entries something that isn't null and that is equal to the old value of entries extended with a pair consisting of the string `n` and the file `f`.

The model field entries, of type `JMLValueToObjectMap`, is declared in the supertype `RODirectory` [**?**].

Implementations of model programs must match each specification statement in a model program with a corresponding refining statement. In the matching refining statement, the specification part must be textually equal to the specification statement. The body of the refining statement must thus implement the given specification for that statement.

This research on model programs also introduced an **extract** keyword that signaled a tool to automatically extract a model program from the implementation. An easy implementation of such extraction would be to simply inline the whole implementation, which might be appropriate for particularly simple methods.

16.2 Model programs in practice

There are two uses of specifications with model programs:

First, the model program is a specification that summarizes the implementation. To verify that the implementation matches the specification a proof tool will need to prove that the model program is behaviorally equivalent to the implementation. This can be a difficult proof task. Hence tools may place restrictions on the structure of the model program compared to the implementation, but any such restrictions are a limitation of the tool rather than imposed by JML itself.

Second, a specification containing a model program is used in the proof of a method that calls a callee method that has such a specification. In this case, any method specification clauses are used or proved as in a non-model-program and the model program statements are inlined between asserting the preconditions and assuming the postconditions.

Model programs are particularly useful for abstract or library methods that have no implementation and have some implicit loop behavior. See the examples in ??.

16.3 Syntax of model programs

Grammar:

```
<model-program-block> ::= { <model-program-statement>* }

<model-program-statement> ::=
        <java-statement>
      | <jml-statement>
      | <jml-model-statement>

<jml-model-statement> ::=
        <choose-statement>
        <repeat-statement>
```

Restrictions: A *<model-program-block>* may not appear as the first clause in an unnamed (lightweight) specification case (where it would be mistaken for an initialization block). At most one *<model-program-block>* may be present in any specification case.

Statements with specifications (cf. §??) may occur within a model program, just like other JML statements. However, within the model program, such a statement being specified may simply be an empty block. Here the specification attached to the empty block statement is used to verify the calling program. When verifying the implementation of the method with such a model program, the empty block would be matched to a non-empty block within the body of the method.

Other points to discuss: need examples using statement specs

extract needs text somewhere

This design of the model program syntax removes the `model_program` specification case and instead places the model program as a block in any of the other named behavior cases. This design simplifies parsing and reading such specifications.

16.4 choose statement

Grammar:

```
<choose-statement> ::=
        <guarded-block> ( or <guarded-block> )* ( else <or-block> )?

<guarded-block> ::= [ <jml-expression> -> ] <or-block>

<or-block> ::= <model-program-block>
```

Well-definedness: Each guard must be independently well-defined. There is no ordering to the guards and each guard expression must be pure. A guard may be presumed to be true within the body of its block.

Type information:

- If a guarded block contains the optional expression, the expression must have boolean type. The default value for an absent guard is `true`.
- The statements within a *<model-program-block>* may be any Java or JML statements, including JML statements that are permitted only within a model program (such as this *<choose-statement>*); the statements may have side-effects.
- If there is no else block, then it cannot be that all the or-blocks have guards and all guards are false.

The execution of a choose statement executes exactly one of the or-blocks that has either no guard or a guard that evaluates to true. The choice of which or-block to execute is completely non-deterministic, with no expectation of fairness on repeated executions. If all guards are false, then if there is an else block, it is executed; if all guards are false and there is no else block, the statement is not well-defined.

The intention of the `choose` statement is that it offers non-determinism. If a comparable deterministic statement is needed, use a Java `if` or `switch` statement.

There seems no reason to restrict a choose statement to a model program. It could just as well be used within the body of a method like any other JML statement. Loosening this restriction would also simplify the grammar.

Should there be a JML non-deterministic choose expression?

This design of the `choose` statement merges the JMLV1 `choose` and `choose_if` statements.

16.5 repeat statement

Grammar:

```
<repeat-statement> ::=
        [ <loop-specification> ] <guarded-block> ( or <guarded-block> ) *
```

Well-definedness: Each guard must be independently well-defined. There is no ordering to the guards and each guard expression must be pure. A guard may be presumed to be true within the body of its block.

Type information:

- The guarding expression must have boolean type.

- The statements within a *<model-program-block>* may be any Java or JML statements, including JML statements that are permitted only within a model program (such as this *<choose-statement>*); the statements may have side-effects.
- All or-blocks must have guards.

The execution of a `repeat` statement repeatedly chooses an or-block with a guard that evaluates to true and then executes it, reevaluating the guards and potentially executing a different or-block on each iteration, until an iteration in which all the guards evaluate to false. The choice of which or-block to execute on each iteration is completely non-deterministic; there are no assurances of fairness.

Like other kinds of loops, a `repeat` statement likely needs a loop specification to be able to prove any assertions about its behavior or to prove termination.

A `repeat` statement can be written as a `while` loop containing a `choose` statement with all of the `repeat` statement's or-blocks and with an else-block that is just a `break` statement. The loop specifications apply to the `repeat` statement as if it were this `while` loop.

The intention of the `repeat` statement is that it offers non-determinism. If a comparable deterministic statement is needed, use a Java `while` statement combined with a Java `if` or `switch` statement.

16.6 inlined_loop clause

Grammar:
<inlined-loop-clause> **::= inlined_loop ;**

Consider a method like `java.util.stream.Stream.forEachOrdered` in the Java library. The method repeatedly applies a method, typically a lambda method, to each element of a `Stream` in turn. As the method being applied can have any side-effect, and the effects can depend on the order of elements in the Stream, it is difficult to specify the method without essentially inlining the implementing loop. This is an example that is appropriate for model methods.

A reasonable specification for this method might include this model program:

```
1   /*@ public normal_behavior
2     requires consumer != null;
3     {
4       //@ loop_invariant i == \count && 0 <= i <= values.length;
5       //@ decreases values.length - i;
6       for (\bigint i=0; i < values.length; i++) {
7         consumer.accept(values[i]);
8       }
9     }
10    @*/
11    void forEachOrdered(Consumer<? super T> consumer);
```

Here `\count` (cf. §??) is the number of times the loop has been executed, and `values` is a model field of `Stream` holding the sequence of values in the stream. The method `consumer.accept` might have any effect at all.

Now consider the specification of a calling program. Here is an example code snippet:

```
1  public int[] arr = new int[5];
2
3  static public void putAtI(Integer v) {
4    arr[ii] = v;
5    ii++;
6  }
7
8  public void mm() {
9    ...
10   Stream<Integer> st = Stream.<Integer>of(1,2,3,4,5);
11   //@ assume arr.length == 5;
12
13   ii = 0;
14   //@ loop_invariant Test.ii == \count;
15   //@ loop_invariant (\forall int j; j>=0 && j<\count; arr[j] == j+1);
16   //@ loop_modifies Test.ii, Test.arr[*];
17   //@ inlined_loop;
18   st.forEachOrdered(v -> putAtI(v));
19         ...
20 }
```

The specification challenge is that the needed loop specifications uses variables some of which are in scope for `forEachOrdered` and others of which are in scope in the calling program. Thus the loop specifications must be split between the two locations. Those in the calling program are attached to the `st.forEachOrdered` method call, which is not a loop. Instead, the loop specifications are attached to a special specification statement, `inlined_loop`, which indicates that the loop for the preceding specification is actually in a model program that is part of the specification of the method being called.

The `inlined_loop` statement is still experimental and not officially part of JML, but appears to be essential to verifying functional programs such as the example above.

Chapter 17

Specification `.jml` files

Specifications for a Java class and its members can be placed inline within the Java source file for that class or they can be placed in a parallel specification file. Such a specification file has a `.jml` extension.

17.1 Locating .jml files

A `.jml` file has the same package designation as its corresponding class. It is up to tools supporting JML to determine where `.jml` files are stored and how they are retrieved. Typically, however, `.jml` files are stored in a folder hierarchy corresponding to the package hierarchy, in the same way that `.java` source files are stored in a file system, with the only difference being the filename extension. The `.jml` specification files may be stored mixed in with the `.java` source files or may be stored underneath a different set of package roots. Tools supporting JML will provide means to designate where the specification files are located.

JML allows at most one `.jml` file per Java class.[1]

17.2 Rules applying to declarations in .jml files

A `.jml` file is syntactically similar to the corresponding `.java` file. The form follows the following rules. Every `.jml` file has a corresponding `.java` or `.class` file; where no `.java` file is available, the `.jml` file is similar to the `.java` file that would have been compiled to produce the `.class` file.

The principle present throughout these rules is that a declaration in a `.jml` file either (1) corresponds to a declaration in the Java file, having the same name, types, non-

[1] In original JML, a sequence of specification files was allowed, each one further refining its predecessor. There were complicated rules about how to combine these specifications. That system is now obsolete and no longer supported; it was complex and not used.

JML modifiers and annotations, or (2) does not correspond to a Java declaration, in which case it must declare a different name. Declarations that correspond to a Java declaration must not be in JML annotations and must not be marked `ghost` or `model`; JML declarations that do not correspond to Java declarations must be in JML annotations and must be marked `ghost` or `model`.

File-level rules

- The `.jml` file has the same package declaration as the `.java` file.
- The `.jml` file may have a different set of import statements and may, in addition, include `model import` statements (§??).
- The `.jml` file must include a declaration of the public type (i.e., class or interface) declared in the `.java` file. It may but need not have JML declarations of non-public types present in the `.java` class. Any type declared in the `.jml` file that is not present in the `.java` file must be in a JML annotation and must have a `model` modifier.

Class declarations

- The JML declaration of a class and the corresponding Java declaration must extend the same superclass, implement the same set of interfaces, and have the same set of Java modifiers and Java annotations. The JML declaration may add additional JML modifiers and annotations.
- Nested and inner class declarations within an enclosing non-model JML class declaration must follow the same rules as file-level class declarations: they must either correspond in name and properties to a corresponding nested or inner Java class declaration or be a model class.
- JML model classes need not have full implementations, as if they were Java declarations. However, if runtime-assertion checking tools are expected to check or use a model class, it must have a compilable and executable declaration.

Interface declarations

- The JML declaration of a interface and the corresponding Java declaration must extend the extend the same set of interfaces and have the same set of Java modifiers and Java annotations. The JML declaration may add additional JML modifiers and annotations.
- In Java, fields declared in an interface are always public and static. JML declarations of model fields within an interface may be non-static; the JML `instance` modifier designates a non-static field (§??).

Method declarations

- Methods declared in a non-model JML type declaration must either correspond precisely to a method declared in the corresponding Java type declaration or be a model method. *Correspond precisely* means having the same name, same

type arguments with the same names, exactly the same argument and return types, and the same set of declared exceptions.

- Methods that correspond to Java methods must not be declared model and must not have a body. They must have the same set of Java modifiers and annotations as the Java declaration, but may add additional JML modifiers, JML annotations, and specifications.
- A Java method of a class or interface need not have a JML declaration (in which case various default specifications might apply).
- Methods that do not correspond to Java declarations must have names that would be permitted if they were Java declarations, must be `model`, and must be within JML annotations. Such model methods may or may not have bodies; if they are expected to be executed by RAC, they must have executable bodies.

Field declarations

- Fields declared in a `.jml` file in a non-model type declaration must either correspond precisely to a field declared in the corresponding Java type declaration or be a model or ghost field. *Correspond precisely* means having the same name and type and Java modifiers and annotations. The JML declaration may add additional JML modifiers and annotations.
- A JML field declaration that corresponds to a Java field declaration may not be in a JML annotation, may not be `model` or `ghost` and must not have an initializer.
- A JML field declaration that does not correspond to a Java field declaration must be in a JML annotation and must be either `ghost` or `model`.
- `ghost` field declarations have the same grammatical form as Java declarations, except that they may use JML types and operators and may refer to names declared in other `ghost` or `model` declarations.
- `model` field declarations have the same grammatical form as Java declarations, except that they may use JML types; they may not have initializers. *May they have initializers as short-hand for represents clauses?*
- A Java field of a class or interface need not have a JML declaration (in which case various default specifications might apply).

Initializer declarations

- A Java class may contain declarations of static or instance initializers. A JML redeclaration of a Java class may not have any initializers.
- A JML model class may have Java initializer blocks.

17.3 Combining Java and JML files

The specifications for the Java declarations within a Java compilation unit are determined as follows.

- If there is a `.java` file and no corresponding `.jml` file, then the specifications are those present in the `.java` file.
- If there is a `.java` file and a corresponding `.jml` file, then the JML specification present in the `.jml` file supersedes all of the JML specifications in the `.java` file, except those within a method body; class, method interface and field specifications in the `.java` file are ignored, even where there is no method declared in the `.jml` file corresponding to a method in the `.java` file.
- If there is no `.java` file, but there is a `.class` file and a corresponding `.jml` file, then the specifications are those present in the `.jml` file.
- If there is no `.java` file and no `.jml` file, only a `.class` file, then default specifications are used (cf. §??).

When there is a `.jml` file processing proceeds as follows to match declarations in JML to those in Java. First all matches among type declarations are established recursively:

- Top-level types in each file are matched by package and name. The type-checking pass checks that the modifiers, superclass and super interfaces match. JML classes that match are not model and are not in JML annotations; JML classes that do not match must be model and must be in JML annotations. Not all Java declarations need have a match in JML; those that have no match will have default specifications.
- Model types contain their own specifications and are not subject to further matching.
- For each non-model type, matches are established for the nested and inner type declarations in the `.jml` and `.java` declarations by the same process, recursively.

Then for each pair of matching JML and Java class or interface declarations, matches are established for method and field declarations.

- Field declarations are matched by name. Type-checking assures that declarations with the same name have the same type, Java modifiers and Java annotations.
- Method declarations are matched by name and signature. This requires that all the processing of import statements and type declarations is complete so that type names can be properly resolved.

For each pair of matching declarations, the JML specifications present in the `.jml` file give the specifications for the Java entity being declared. If there is a `.jml` file but no match for a particular Java declaration in the corresponding `.java` file, then that

declaration uses default specifications, even if the .java file contains specifications. The contents of the .jml file supersede all the JML contents of the .java file; there is no merging of the files' contents.[2]

17.4 Specifications in method bodies

Specification statements in method bodies are necessarily stated in the .java source file, even if there is a .jml file. Specification statements in method bodies are there to aid the proof of the method's specification and are not part of the method's interface or its specification.

17.5 Obsolete syntax

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

In addition, the .jml file is now sought before seeking the .java file; if any corresponding .jml file is found, then any specifications in the .java file are ignored (except those within method bodies). This is a different search algorithm than was previously used.

[2] Previous definitions of JML did require merging of specifications from multiple files; this requirement added complexity without appreciable benefit. The current design is simpler for tools, with the one drawback that the JML contents of a .java file is silently ignored when a .jml file is present, even if that .jml file does not contain a declaration of a particular entity.

Chapter 18

Interaction with other tools

18.1 Interaction with the Checker framework

To be written

Chapter 19

Future topics

There are other topics that are under discussion for JML but are not ready to be “standardized" in this reference manual. Some of them are presented briefly here.

19.1 Observational purity

Consider a method that executes a lengthy computation, producing a result that is solely dependent on its input. For better performance, the output value is cached (for the given input) in a private cache. Although this method is not pure, because it changes its internal memory state, it does not change the program state in any way that the rest of the program can detect. This is called observational purity.

Specifying such methods in a way that allows them to be used as pure functions is a matter of research. The topic is mentioned here as a placeholder for future features in JML.

19.2 Termination

Though loop termination is handled well in JML, specifications that enable proofs of termination of recursive calls has yet to be worked out. This relates to the `callable` and `measured_by` clauses, but most importantly, JML needs a workable well-founded metric to use in termination proofs. The sequence-of-values approach used by Dafny would be good inspiration. But also needed is a means of assembling a sufficient call-graph without needing global analysis and without a verbose listing of calls made by each method.

19.3 Arithmetic modes for floating point

Java and object-oriented programming are less applied to float-point computations than languages like C and C++. Accordingly, reasoning about floating point and real arithmetic is also not as advanced. Work is needed on how to do such reasoning in the context of JML, JML tools and SMT solvers. Note that SMT solvers have added the native ability to reason about floating point calculations.

19.4 Race condition and deadlock detection

Race condition and deadlock detection are aspects of multi-threaded programs. JML does not (yet) have the features to specify such programs.

Appendix A

Summary of Modifiers

The tables on the following pages summarize where the various Java and JML modifiers may be used, with references to sections in this document.

Note that `final` modifiers can occur in either Java text or JML text. This allows a specification to declare a Java variable as `final`, when appropriate, even if the Java program text does not.

JML Keyword	Java annotation	class	interface	method	field	variable	Status
code_bigint_math	@CodeBigintMath	??	??	??			JML
code_java_math	@CodeJavaMath	??	??	??			JML
code_safe_math	@CodeSafeMath	??	??	??			JML
extract	@Extract			??			Advanced
final	@Final				??	??	Advanced
ghost	@Ghost				??	??	JML
helper	@Helper			??			JML
inline	@Inline			??			Experimental
instance	@Instance				??		JML
model	@Model	??	??	??	??		JML
monitored	@Monitored				??		Concurrent
no_state	@NoState	?? ??	??	??			JML
non_null	@NonNull			??	??	??	JML
non_null_by_default	@NonNullByDefault	??	??	??			JML
nullable	@Nullable			??	??	??	JML
nullable_by_default	@NullableByDefault	??	??	??			JML
	@Options(...)	??	??	??			Experimental
peer	@Peer						Advanced
pure	@Pure	?? ??	??	??			JML
query	@Query			??			Experimental
readonly	@Readonly						Advanced
rep	@Rep						Advanced
secret	@Secret				??		Experimental
spec_bigint_math	@SpecBigintMath	??	??	??			JML
spec_java_math	@SpecJavaMath	??	??	??			JML
spec_protected	@SpecProtected	??	??	??	??		JML
spec_public	@SpecPublic	??	??	??	??		JML
spec_pure	@SpecPure	?? ??	??	??			JML
spec_safe_math	@SpecSafeMath	??	??	??			JML
strictly_pure	@StrictlyPure	?? ??	??	??			JML
uninitialized	@Uninitialized				??	??	JML

Table A.1: Summary of JML modifiers. All Java annotations are in the `org.jmlspecs.annotation` package.

Grammatical construct	Java modifiers	JML modifiers
All modifiers	`public protected private abstract static final synchronized transient volatile native`	`spec_public spec_protected model ghost pure instance helper heap_free non_null_by_default nullable_by_default monitored uninitialized final` Type annotations `non_null` and `nullable` apply to any type names.
Class declaration	`public final abstract`	`pure model spec_public spec_protected non_null_by_default nullable_by_default` *arithmetic modes*
Interface declaration	`public`	`pure model spec_public spec_protected non_null_by_default nullable_by_default` *arithmetic modes*
Nested Class declaration	`public protected private static final abstract`	`pure model final spec_public spec_protected non_null_by_default nullable_by_default` *arithmetic modes*
Nested interface declaration	`public protected private static`	`pure model spec_public spec_protected non_null_by_default nullable_by_default` *arithmetic modes*
Local Class (and local model class) declaration	`final abstract`	`pure model final non_null_by_default nullable_by_default` *arithmetic modes*

Grammatical construct	Java modifiers	JML modifiers
Type specification (e.g. invariant)	`public protected private static`	
Field declaration	`public protected private final volatile transient static uninitialized`	`final instance monitored spec_public spec_protected`
Ghost Field declaration	`public protected private static final`	`final instance monitored uninitialized`
Model Field declaration	`public protected private static`	`final instance`
Method declaration in a class	`public protected private abstract final static synchronized native final`	`spec_public spec_protected pure helper extract heap_free non_null_by_default nullable_by_default` *arithmetic modes*
Method declaration in an interface	`public abstract`	`spec_public spec_protected pure extract helper heap_free non_null_by_default nullable_by_default` *arithmetic modes*
Constructor declaration	`public protected private`	`spec_public spec_protected helper pure extract heap_free non_null_by_default nullable_by_default` *arithmetic modes*

Grammatical construct	Java modifiers	JML modifiers
Model method (in a class or interface)	`public protected private abstract static final synchronized`	`pure helper extract final heap_free non_null_by_default nullable_by_default` *arithmetic modes*
Model constructor	`public protected private`	`pure helper extract heap_free non_null_by_default nullable_by_default` *arithmetic modes*
Java initialization block	`static`	`non_null_by_default nullable_by_default` *arithmetic modes*
JML `initializer` and `static_initializer` annotation	-	`non_null_by_default nullable_by_default` *arithmetic modes*
Formal parameter	`final`	`final`
Local variable and local ghost variable declaration	`final`	`ghost final uninitialized`

Appendix B

Core JML

To help guide tool development, the features of JML are grouped into various categories:

Core features should be supported by all tools and should be the focus of education,

Advanced features are those needed for practical use and to specify the system library,

Experimental features are the result of research or represent research in progress; they are defined so that all tools will use the same syntax for them, but may well evolve as more experience is gained in their use, and

Concurrent features to support reasoning about concurrency, which is not yet a capability of JML

The following table states the category for each language feature. As context for the reader, the table also lists which features are supported by the two most prominent JML tools (at the time of writing). Note that not all features are necessarily executable in RAC; more details on limitations of tools can be found in the tools' respective documentation. Tools will typically parse and ignore unsupported features.

These tables are being edited and are not (yet) settled. Compare with categories in table A.1

The entries in the table have these meanings:

- **Core** — a Core construct
- **Dep.** — a Deprecated construct
- **Adv.** — an Advanced construct
- **Exp.** — an Experimental construct
- **Conc.** — a construct for concurrency
- **Ext.** — an extension to JML (not defined as standard)
- - — not supported by a given tool

- + — supported by a given tool,

These tables are being edited and are not (yet) settled

Modifiers

	feature	Category	KeY	OpenJML	Comments		
§??	`code`			**Adv.**	-	+	
§??	`code_bigint_math`			**Adv.**	-	+	
§??	`code_java_math`			**Adv.**	-	+	
§??	`code_safe_math`			**Adv.**	-	+	but is the default in **Core**
§??	`extract`			**Exp.**	-	-	
§??	`ghost`			**Core**	+	+	fields
§??	`heap_free`			**Adv.?**	+	+	trial OpenJML extension
§??	`helper`			**Core**	+	+	
§??	`immutable`			**Adv.?**	-	+	extension?
§??	`inline`			**Ext.**	-	+	OpenJML: inlines method as its spec
§??	`instance`			**Core**	+	+	
§??	`model`			**Core**	+	+	fields, methods
§??	`model`			**Adv.**	-	+	classes
§??	`monitored`			**Conc.**	?	-	
§??	`non_null`			**Core**	+	+	a type modifier
§??	`non_null_by_default`			**Core**	+	+	
§??	`no_state`			**Adv.?**	+	-	heap-independent model method
§??	`nullable`			**Core**	+	+	a type modifier
§??	`nullable_by_default`			**Core**	+	+	
§??	`public private protected`			**Core**	+	+	for clauses and contracts
§??	`peer rep read_only`			**Adv.**	(+)	-	
§??	`pure`			**Core**	+		
§??	`query secret`			**Exp.**	-	+	observational purity
§??	`spec_bigint_math`			**Adv.**	+	+	but is the default in **Core**
§??	`spec_java_math`			**Adv.**	+	+	
§??	`spec_protected`			**Adv.**	+	+	
§??	`spec_public`			**Core**	+	+	
§??	`spec_safe_math`			**Adv.**	-	+	
§??	`strictly_pure`			**Core?**	+	-	KeY
§??	`two_state`			**Adv.**	+	-	model method with access to `\old`
§??	`uninitialized`			**Adv.**	?	-	
§??	Java annotations instead of modifiers			**Adv.**	?	+	

File level features

	feature	Category	KeY	OpenJML	Comments
§??	model imports	**Core**	+	+	
§??	model classes	**Adv.**	+	+	

Class- and field-level features

	feature	Category	KeY	OpenJML	Comments
§??	ghost fields	**Core**	+	+	
§??	model fields	**Adv.**	+	+	
§??	datagroups	**Adv.**	+	+	
§??	model methods	**Adv.**	+	+	
§??	`axiom`	**Adv.**	+	+	
§??	`constraint`	**Adv.**	+	+	
§??	`in`	**Adv.**	?	+	
§??	`initially`	**Adv.**	?	+	
§??	`initializer`	**Adv.**	?	+	
§??	`invariant`	**Core**	+	+	
§??	`maps`	**Adv.**	-	+	
§??	`monitors_for`	**Conc.**	-	-	
§??	`readable if`	**Adv.**	-	+	
§??	`represents`	**Adv.**	+	+	
§??	`static_initializer`	**Adv.**	?	+	
§??	`writable if`	**Adv.**	-	+	

Method specifications

	feature	Category	KeY	OpenJML	Comments	
§??	`accessible`	**Adv.**	+	+		or 'reads'
§??	`also`	**Core**	+	+		
§??	`assignable`	**Core**	+	+		KeY: also for loops; OpenJML uses loop_writes for loops
§??	`behavior`	**Core**	+	+		
§??	`callable`	**Adv.**	-	+		or 'calls'?
§??	`captures`	**Adv.**	-	+		
§??	`diverges`	**Adv.**	+	+		
§??	`determines`	**Ext.**	+	-		information flow
§??	`duration`	**Exp.**	?	-		
§??	`ensures`	**Core**	+			
§??	`exceptional_behavior`	**Core**	+			
§??	`forall`	**Dep.**	+	-		
§??	`for_example`	**Adv.Exp.?**	-	-		semantics unclear
§??	`implies_that`	**Adv.Exp.?**	-	-		semantics unclear
§??	`measured_by`	**Adv.Core?**	+	-		needs revision
§??	`normal_behavior`	**Core**	+	+		

	feature	Category	KeY	OpenJML	Comments		
§??	`old`			**Adv.**	?	+	
§??	`recomments ... else`			**Adv.**	-	+	
§??	`requires`			**Core**	+	+	
§??	`requires ... else`			**Adv.**	-	+	
§??	`signals`			**Core**	+	+	
§??	`signals_only`			**Core**	+	+	
§??	`when`			**Conc.**	-	-	
§??	`working_space`			**Exp.**	-	-	
§??	`XXX_free`			**Adv.**	+	-	specification elements w/o justification
§??	model program			**Adv.**	-	-	needs discussion
§??	model program block			**Exp.**	-	+	needs discussion
§??	model program clauses: `choose` `choose_if` `extract` `or` `returns`			**Adv.**	-	-	
§??	`continues` `breaks` `{| ... |}`			**Adv.**	?	+	(nested specs)
§??	JML in Javadoc			**Dep.**	-	-	

Statement specifications

	feature	Category	KeY	OpenJML	Comments		
§??	`assert`			**Core**	+	+	
§??	`assume`			**Core**	+	+	
§??	`debug`			**Dep.**	-	+	
§??	`hence_by`			**Dep.**	-	-	
§??	`reachable`			**Adv.**	-	+	
§??	`set`			**Adv.**	+	+	
§??	`unreachable`			**Adv.**	-	+	
§??	ghost label			**Core?**	?+		
§??	`loop_invariant`			**Core**	+	+	
§??	`loop_writes`			**Core**	+	+	
§??	(loop) `decreases`			**Core**	+	+	
§??	local ghost variables			**Core**	+	+	
§??	local model classes			**Adv.Exp.**	?	+	or perhaps forbid?
§??	block contracts			**Adv.**	+	+	
§??	`breaks`, `continues`, `returns`			**Adv.**	+	-	in block contracts
§??	begin-end markers			**Ext.Adv.?**	-	+	OpenJML
§??	`check`			**Adv.?**	-	+	OpenJML
§??	`havoc`			**Adv.?**	-	+	OpenJML
§??	`inline_loop`			**Adv.?**	-	+	OpenJML
§??	`show`			**Core?**	-	+	OpenJML
§??	`split`			**Ext.**	-	+	OpenJML

	feature	Category	KeY	OpenJML	Comments	
§??	halt		**Ext.**	-	+	OpenJML
§??	use		**Ext.**	-	+	OpenJML
§??	comment		**Ext.**	-	+	OpenJML

JML Types

	feature	Category	KeY	OpenJML	Comments
§??	\bigint	**Core**	+	+	
§??	\real	**Adv.**	+	+	
§??	\TYPE	**Adv.**	-	+	
§??	\string	**Adv.**	-	+	
§??	\range	**Adv.**	-	+	
§??	\datagroup	**Adv.**	-	+	
§??	\seq<E>	**?**	+	-	
§??	\map<K,V>	**?**	+	-	
§??	\set<E>	**?**	+	-	
§??	\array<E>	**?**	+	-	
§??	\locset	**Adv.**	+	+	builtin datatype

Operators and Expressions

feature	Category	KeY	OpenJML	Comments	
<==>		**Core**	+	+	
<=!=>		**Core**	+	+	
==>		**Core**	+	+	
<==		**Dep.**	**?**	**?**	
..		**Core**	+	+	in storeref indexing only
<:		**Core**	+	+	
<# <#=		**Conc.**	**?**	-	
(* *)		**Adv.**	+	+	
operator chaining		**Core**	+	+	Only < <= and > >=
\bsum		**Adv.?**	+	**?**	
\bigint_math		**Adv.**	**?**	+	
\count \index		**Core**	-	+	\index deprecated in favor of \count
\duration		**Exp.**	**?**	-	
\elemtype		**?**	**?**	+	
\everything		**Core**	+	+	
\exception		**Ext.**	-	+	like \result
\exists		**Core**	+	+	
\forall		**Core**	+	+	

feature	Category	KeY	OpenJML	Comments	
\fresh	**Core**	+	+		
\invariant_for	**Core**	+	+		
\is_initialized	**Adv.**	**?**	-		
\java_math	**Adv.**	**?**	+		
\lbl	**Adv.**	**?**	+		
\lblpos	**Dep.**	**?**	+		
\lblneg	**Dep.**	**?**	+		
\lockset	**Conc.**	**?**	+		
\max (locks)	**Conc.**	**?**	-		
\max	**Adv.**	+	+		
\min	**Adv.**	+	+		
\new_elems_fresh	**Adv.?**	+	**?**	needed for dyn frames	
\nonnullelements	**Core**	+	+		
\nothing	**Core**	+	+		
\not_assigned	**Adv.**	**?**	-	never used, I think	
\not_modified	**Adv.**	**?**	+		
\num_of	**Adv.**	+	+		
\old	**Core**	+	+	w/o label	
\old	**Core**	+	+	w/ label	
builtin labels	**?**	**?**	+	\LoopInit etc.	
\only_accessed	**Adv.**	**?**	-	never used, I think	
\only_assigned	**Adv.**	**?**	-	never used, I think	
\only_called	**Adv.**	**?**	-	never used, I think	
\only_captured	**Adv.**	**?**	-	never used, I think	
\past	**?**	**?**	-		
\pre	**Core**	**?**	+		
\product	**Adv.**	+	+		
\reach	**?**	**?**	-	is this still in JML?	
\result	**Core**	+	+		
\safe_math	**Adv.**	**?**	+		
\space	**Exp.**	**?**	-		
\static_invariant_for	**Adv.**	+	-		
\strictly_nothing	**Ext.?**	+	-	KeY	
\sum	**Adv.**	+			
\type	**Adv.**	-	+		
\typeof	**Core**	-	+		
\values	**Core**	+	+		
\working_space	**Exp.**	**?**	-		
set comprehension	**Adv.**	**?**	-		

Miscellaneous features

	feature	Category	KeY	OpenJML	Comments	
§??	`//@`	**Core**	+	+		
§??	`/*@ @*/`	**Core**	+	+		
§??	`//` comments in specs	**Core**	+	+		
§??	conditional annotations	**Core?**	?	+		
§??	embedded annotations	**Adv.**	?	+		
§??	org.jmlspecs.lang	**Core**	?	+	package automatically imported	
§??	`...redundantly`	**Adv.**	+	+	Typically implemented by ignoring the redundantly suffix	
§??	`.jml` files	**Adv.?**	?	+	needed for library specs	
§??	JML in Javadoc	**Dep.**	-	-		
§??	`nowarn`	**Dep.**	?	+	line annotation	
	`\dl_`	**Ext.**	+	-	MU: or some other means of tool-spec exts.	

allow, forbid, ignore

Appendix C

Deprecated Syntax

The sections below briefly describe the deprecated and replaced features of JML. A feature is deprecated if it is supported in the current release, but slated to be removed from a subsequent release, or if it was commonly known but is now removed altogether. Such features should not be used.

A feature that was formerly deprecated is replaced if it has been removed from JML in favor of some other feature or features. While we do not describe all replaced syntax in this appendix, we do mention a few of the more interesting or important features that were replaced, especially those discussed in earlier papers on JML.

Deprecated syntax might be supported with a deprecation warning by some tools.

C.1 Deprecated Annotation Markers

The following lexical syntax for annotation markers is deprecated.

<annotation-marker> `::=`

```
    //+@ [ @ ] ...
  | /*+@ [ @ ] ...
  | //-@ [ @ ] ...
  | /*-@ [ @ ] ...
```

The +-style of JML annotations, that is, JML annotations beginning with `//+@` or `/*+@`, has been replaced by the annotation-key feature described in §??.

C.2 Represents clause syntax using <-

The following syntax for a functional represents-clause is deprecated.

<represents-clause> : : = *<represents-keyword>* *<store-ref-expression>* **<-** *<spec-expression>* ;

Instead of using the <–, one should use = in such a *<represents-clause>*. See §?? for relevant discussion.

C.3 `monitors_for` clause syntax

The following syntax for the monitors-for-clause is deprecated.

<monitors-for-clause> : : = **monitors_for** *<ident>* **<-** *<spec-expression-list>* ;

Instead of using the <-, one should use = in such a monitors-for-clause. See §?? for the supported syntax.

C.4 Filename suffixes

The set of file name suffixes supported by JML tools has been simplified. The suffixes '.refines-java', '.refines-spec', '.refines-jml', '.spec', '.java-refined', '.spec-refined', and '.jml-refined' are no longer supported. Instead, one should write specifications in files with the suffixes '.java' and '.jml'. See §?? and §?? for details on the use of file names with JML tools.

C.5 Deprecated `weakly` modifier

The `weakly` modifier is not longer supported.

C.6 `refine` prefix

The following syntax involving the refine-prefix is deprecated.

<compilation-unit> ::=
<package-declaration>?
<refine-prefix>
<import-declaration>⋆
<top-level-declaration>⋆

<refine-prefix> : : = *<refine-keyword>* *<string-literal>* ;

<refine-keyword> : : = **refine** | **refines**

Instead of using the `refine-prefix` in a compilation unit, modern JML tools just use a .jml file that contains any specifications not in the .java file. See §?? and §?? for details.

C.7 reverse-implication (<==) token

The <== token and the reverse-implication expression are deprecated. It was rarely used and a bit confusing. Just reverse the order of the operands and use the ==> operator instead.

C.8 <: in favor of <:=

The <: operator used to represent the improper subtype operation on \TYPE values. But a better spelling is that improper subtype (that is, subtype or equals) be represented by <:= and proper subtype by <:, just like the <= and < comparisons on numbers.

Consequently, <: is deprecated in favor of <:= for improper subtype; <: will be reintroduced later to mean proper subtype.

C.9 < as lock-ordering operator

Previously in JML, the lock ordering operators were just the < and <= comparison operators. However, with the advent of auto-boxing and unboxing (implicit conversion between primitive types and reference types) these operators became ambiguous. For example, if a and b are `Integer` values, then $a < b$ could have been either a lock-ordering comparison or an integer comparison after unboxing a and b. Since the lock ordering is only a JML operator and not Java operator, the semantics of the comparison could be different in JML and Java. To avoid this ambiguity, the syntax of the lock ordering operator was changed and the old form deprecated. The new operators are <# and <#=.

C.10 Deprecated \index in favor of \count

The expression \count is the number of times a loop has been completed. \count is preferred to \index because the latter was confused with the value of the loop index.

C.11 \not_specified token

The \not_specified token used as an alternative to a predicate in many clauses is deprecated.

C.12 `nowarn` line annotation and `\nowarn_op` and `\warn_op` functions

The `nowarn` annotation was used to suppress warnings on the line on which it occurred. Similarly, `\nowarn_op` and `\warn_op` suppressed or unsuppressed warnings within subexpressions. These were rarely used and created unsound implicit assumptions.

C.13 `hence_by` statement

The `hence_by` statement specification is deprecated. The same purpose is provided by a `assume` statement. This deprecation is in line with avoiding having proof-guiding information in JML, leaving that to tools.

C.14 `debug` statement

The `debug` statement was similar to the `set` statement (§??), but only executed when the `DEBUG` key was active (§??). It is deprecated because it is entirely equivalent to `//+DEBUG@ set ...`

C.15 `forall` method specification clause

The `forall` method specification clause is deprecated. It had little to no use and no compelling use cases. Any use one might make of it can be accomplished with an `old` method specification declaration (§??) initialized with a `\choose` expression (§??), as in

```
old T e = (\choose T e; true);
```

C.16 constructor, method and field keywords

The `constructor`, `method`, and `field` keywords were intended to help with parsing. However, they are not needed and, in fact, complicate parsing. Accordingly, they have been removed.

C.17 `\lblpos` and `\lblneg`

These two expressions were rarely used. In addition they are in the category of proof debugging aids rather than program specification per se. Hence they are removed from JML.

C.18 JMLDataGroup

The `JMLDataGroup` typename has been replaced by `\datagroup` (§??).

C.19 Java annotations for specifications

The only Java annotations used in JML are the JML modifiers (e.g. `pure` and `@Pure`). Java annotations for clauses, such as `@Requires`, are removed from JML.

C.20 Specifications in JavaDoc

Earlier JML and tools allowed JML specifications to be written inside Java javadoc (`/**`) comments. This is no longer permitted.

C.21 subclassing_contract

As a note for readers of older papers, the keyword `subclassing_contract` was replaced with `code_contract`, which is now removed. Instead, one should use a heavyweight specification case with the keyword `code` just before the behavior keyword, and a precondition of `\same`.

C.22 depends clause

The `depends` clause has been replaced by the mechanism of data groups and the 'in' and 'maps' clauses of variable declarations.

Appendix D

Grammar Summary

Automatic collection of all of the grammar productions listed elsewhere in the document

```
<compound-jml-comment> ::= <simple-jml-comment>+

<simple-jml-comment> ::=
        <jml-line-comment> | <jml-block-comment>

<jml-line-comment> ::=
        //<jml-comment-marker>
        <jml-annotation-text>
        <line-terminator>

<jml-block-comment ::=
        /* <jml-comment-marker>
        <jml-annotation-text-no-blocks>
        <jml-block-comment-end>

<jml-comment-marker> ::=
        ([+|-]<java-identifier>)*@+
```

in which the Java identifiers must satisfy the rules about keys stated in §??.

```
<jml-block-comment-end> ::= @*/
```

<plain-java-comment> is defined in §3.7 of the JLS, but excludes any character sequence matching a <*compound-jml-comment*>

<java-identifier> is defined in §3.8 of the JLS (and excludes any <*reserved-word*>)

```
<jml-annotation-text-no-blocks> ::=
         ( <identifier>
       | <reserved-word>
       | <literal>
       | <operator>
       | <separator>
       | <java-white-space>
       | <jml-line-comment>
```

```
      | <plain-java-comment>
      | <jml-line-terminator>
) *

<jml-line-terminator> ::=       | <line-terminator>
      | <jml-white-space>

<jml-annotation-text> ::=
      | <jml-annotation-text-no-blocks>
      | <jml-block-comment>[1]

<identifier> ::= <java-identifier> | <jml-identifier>

<jml-identifier> ::= [\]<java-identifier>
```

Note that users cannot define new <*jml-identifier*>s and all <*jml-identifier*>s currently defined in JML are purely alphabetic and ASCII after the backslash.

`<reserved-word>` is defined in §3.9 of the JLS

`<literal>` is defined in §3.10 of the JLS

```
<operator> ::= <java-operator> | <jml-operator>
```

`<java-operator>` is defined in §3.12 of the JLS

```
<jml-operator> ::= .. | ==> | <==> | <=!=> | <: | <:= | <# | <#=

<separator> ::= <java-separator> | <jml-separator>
```

`<java-separator>` is defined in §3.11 of the JLS

```
<jml-separator> ::= {| | |}

<java-white-space> is defined in §3.6 of the JLS

<jml-white-space> ::=
        <line-terminator> <java-white-space>? [@]+
```

within a <*jml-block-comment*>

`<line-terminator>` is defined in §3.4 of the JLS

```
<jml-datatype> ::= <modifiers> datatype [ <typeargs> ] {
      <dt-constructor> ... [ ; <dt-method>+ ]
      }

<dt-constructor> ::=
       <ident>
      | <ident> ( <formal> ... )


<dt-method> ::=

<invariant-clause> ::= invariant <opt-name> <predicate> ;

<constraint-clause> ::= constraint <opt-name> <predicate> ;

<initially-clause> ::= initially <opt-name> <predicate> ;
```

[1] This <*jml-block-comment*> may not include any line terminators.

```
<ghost-field-declaration> ::=
       ghost <opt-name> <jml-field-declaration>

<model-field-declaration> ::=
       model <opt-name> <jml-field-declaration>

<represents-clause> ::= [ static ] <represents-keyword> <ident>
      ( = <jml-expression> ;
      | \such_that <predicate> ;
      )
<represents-keyword> ::= represents | represents_redundantly

<static-initializer-block> ::= <specification-cases> static <block>
<static-initializer> ::= <specification-cases> static_initializer

<instance-initializer> ::= <specification-cases> initializer

<axiom-clause> ::= axiom <opt-name> <predicate> ;

<readable-if-clause> ::= readable <ident> if <jml-expression> ;
<writable-if-clause> ::= writable <ident> if <jml-expression> ;

<monitors-for-clause> ::=
       monitors_for <ident> = <jml-expression> ... ;

<method-spec> ::= ( also )? <behavior-seq>
                      ( also implies_that <behavior-seq> )?
                      ( also for_example <behavior-seq> )?

<behavior-seq> ::= <behavior> ( also <behavior> )*

<behavior> ::=
     ( <java-visibility> ( code )? <behavior-id>)? <clause-seq>

<java-visibility> ::= ( public | protected | private )?

<behavior-id> ::=
       behavior | normal_behavior | exceptional_behavior
     | behaviour | normal_behaviour | exceptional_behaviour

<clause-seq> ::= ( <method-spec-clause> | <nested-clause> )*
```

```
<method-spec-clause> ::=
         <invariants-clause>      §??
       | <requires-clause>        §??
       | <recommends-clause>      §??
       | <old-clause>             §??
       | <writes-clause>          §??
       | <reads-clause>           §??
       | <callable-clause>        §??
       | <ensures-clause>         §??
       | <signals-clause>         §??
       | <signals-only-clause>    §??
       | <diverges-clause>        §??
       | <measured-by-clause>     §??
       | <when-clause>            §??
       | <duration-clause>        §??
       | <working-space-clause>   §??
       | <captures-clause>        §??
       | <returns-clause>         §??
       | <throws-clause>          §??
       | <continues-clause>       §??
       | <breaks-clause>          §??
       | <model-program-block>    §??

 <nested-clause> ::=              §??
       {|  ( <clause-seq> ( also <clause-seq> )* )? |}

<requires-clause> ::= requires <opt-name> <jml-expression>
       [ else <qual-ident> ];

<ensures-clause> ::= ensures <opt-name> <jml-expression> ;

<signals-clause> ::=
    signals <opt-name> ( <name> [ <ident> ] ) <jml-expression> ;

<signals-only-clause> ::=
    signals_only <opt-name> ( \nothing | <name> ( , <name> )* );
<name> ::= <ident> ( .  <ident> )*

<recommends-clause> ::= requires <opt-name> <jml-expression>
       else <qual-ident> ;

<diverges-clause> ::= diverges <opt-name> <jml-expression> ;

<measured-by-clause> ::=
       <measured-by-keyword> <opt-name> <jml-expression> ;
<measured-by-keyword> ::= measured_by

<when-clause> ::= <when-keyword> <opt-name> <jml-expression> ;
<when-keyword> ::= when | when_redundantly

<old-clause> ::= old <jml-var-decl>
```

```
<duration-clause> ::= <duration-keyword> <opt-name> <expression>
       [ if <predicate> ] ;
<duration-keyword> ::= duration |duration_redundantly

<working-space-clause> ::=
   <working-space-keyword> <opt-name> <predicate>
       [ if <expression> ] ;
<working-space-keyword> ::= working_space
      |working_space_redundantly

<callable-clause> ::= callable <opt-name>
      ( \nothing
      |<method-signature> ( , <method-signature> )+
      );
<method-signature> ::= [ <type-name>. ]<java-identifier>
      [ ( <type-name> ... ) ]

<captures-clause> ::=
    <captures-keyword> <opt-name> <expression> ... ;
<captures-keyword> ::= captures |captures_redundantly

<returns-clause> ::= returns [ <jml-expression> ];

<throws-clause> ::= throws [ <jml-expression> ];

<continues-clause> ::=
       continues [ <ident> : ] [ <jml-expression> ];

<breaks-clause> ::=
       breaks [ <ident> : ] [ <jml-expression> ];

<in-clause> ::= in <ident> ... ;

<maps-clause> ::= maps <storeref> \into <identifier> ... ;

 <jml-statement> ::=
        <jml-assert-statement>          §??
       |<jml-assume-statement>          §??
       |<jml-local-variable>            §??
       |<jml-local-class>               §??
       |<jml-ghost-label>               §??
       |<jml-unreachable-statement>     §??
       |<jml-set-statement>             §??
       |<jml-loop-specification>        §??
       |<jml-refining-specification>    §??

<jml-assert-statement> ::=
       <assert-keyword> <opt-name> <jml-expression> ;
<assert-keyword> ::= assert |assert_redundantly

<jml-assume-statement> ::=
       <assume-keyword> <opt-name> <jml-expression> ;
```

```
<assume-keyword> ::= assume |assume_redundantly

<jml-local-variable> ::=
        ghost <modifier>* <decl-type> <identifier>
                [ = <jml-expression> ];

<jml-local-class> ::= model <class-declaration>

<jml-ghost-label> ::= <java-identifier> :[ { } | ; ]

<jml-unreachable-statement> ::=
        unreachable <opt-name> [ ; ]

<jml-set-statement> ::=
        set <opt-name> <java-statement>

<statement-specification> ::= refining <behavior-seq>

<begin-end> ::= begin | end

<jml-expression> ::=
          <conditional-expression>                §??
        | <quantified-expression>                 §??
        | <lambda-expression>                     §??
        | <assignment-expression>                 §??
        | <jml-infix-expression>                  §??

<jml-infix-expression> ::=
          <jml-binary-expression>                 §??
        | <jml-prefix-expression>                 §??

<jml-prefix-expression> ::=
          <jml-unary-expression>                  §??
        | <jml-cast-expression>                   §??
        | <jml-postfix-expression>

<jml-postfix-expression> ::=
          <dot-expression>                        §??
        | <array-expression>                      §??
        | <jml-primary-expression>

<jml-primary-expression> ::=
```

```
        <result-expression>                          §??
      | <exception-expression>                       §??
      | <informal-expression>                        §??
      | <old-expression>                             §??
      | <nonnullelements-expression>                 §??
      | <fresh-expression>                           §??
      | <type-expression>                            §??
      | <erasure-expression>                         §??
      | <typeof-expression>                          §??
      | <arraytype-expression>                       §??
      | <isarray-expression>                         §??
      | <elemtype-expression>                        §??
      | <invariant-for-expression>                   §??
      | <static-invariant-for-expression>            §??
      | <is-initialized-expression>                  §??
 Missing some - check the list
      | <java-math-expression>                       §??
      | <safe-math-expression>                       §??
      | <bigint-math-expression>                     §??
      | <duration-expression>                        §??
      | <working-space-expression>                   §??
      | <space-expression>                           §??

<conditional-expression> ::=
       <jml-infix-expression> ?  <jml-expression> :  <jml-expression>

<assignment-expression> ::=
       <jml-postfix-expression> <assign-op> <jml-expression>
<assign-op> ::= = | += | -= | *= | /= | %=
              | &= | |= | ≙

<unary-expression> ::= [ ! | - | ~ ] <unary-expression>

<cast-expression> ::= ( <type-name> ) <jml-expression>

<array-expression> ::= <postfix-expression> [ <expression> ]

<parenthesized-expression> ::= ( <jml-expression> )

<quantified-expression> ::=
       <quantifier> <type-name> <java-identifier> ;
       [ [ <jml-expression> ]; ] <jml-expression>


<quantifier> ::=
       \forall | \exists | \choose
      |\num_of | \sum | \product | \max | \min

<jml-binary-expression> ::=
<jml-prefix-expression> ( <binop> <jml-prefix-expression> )*
```

```
<chained-expression> ::=
        <expression> ( [<|<=] <expression> )+
      | <expression> ( [>|>=] <expression> )+

<implies-expression> ::= <expression> ==> <expression>

<equiv-expression> ::=
        <expression> <==> <expression>
      | <expression> <=!=> <expression>

<subtype-expression> ::=
        <expression> <:= <expression>
      | <expression> <:  <expression>

<lockorder-expression> ::=
        <expression> <#= <expression>
      | <expression> <# <expression>

<result-expression> ::= \result

<exception-expression> ::= \exception

<count-expression> ::= \count | \index

<values-expression> ::= \values

<old-expression> ::=
        ( \old( <expression> ( , <label> )? )
        ( \pre( <expression> )
        ( \past( <expression> )
<label> ::= <id>

<fresh-expression> ::=
        \fresh( <expression> [ , <java-identifier> ])

<nonnullelements-expression> ::=
        \nonnullelements ( <expression> )

<arithmetic-mode-expression> ::=
        <arithmetic-mode-name> ( <expression> )
<arithmetic-mode-name> ::= \java_math | \safe_math | \bigint_math

<informal-expression> ::= (*  .*  *)

<type-expression> ::=
        \type( <jml-type-expression> )

<typeof-expression> ::=
        \typeof ( <expression> )
```

```
<arraytype-expression> ::=
        \arraytype ( <expression> )

<isarray-expression> ::=
        \isarray ( <expression> )

<elemtype-expression> ::=
        \elemtype ( <expression> )

<erasure-expression> ::=
        \erasure ( <expression> )

<is-initialized-expression> ::=
        \is_initialized ( <type-name> ...  )
      | \is_initialized ( )

<invariant-for-expression> ::=
        \invariant_for ( <expression> )

<static-invariant-for-expression> ::=
        \static_invariant_for ( <type-name> )

<not-modified-expression> ::=
        \not_modified ( <store-ref-expression> ... )
      | \not_modified ( )

<not-assigned-expression> ::=
        \not_assigned ( <store-ref-expression> ... )

<only-assigned-expression> ::=
        \only_assigned ( <store-ref-expression> ... )
<only-accessed-expression> ::=
        \only_accessed ( <store-ref-expression> ... )
<only-captured-expression> ::=
        \only_captured ( <store-ref-expression> ... )

<only-called-expression> ::=
        \only_called ( <method-signature> ... )

<locset-expression> ::=
        \lockset
      | \max ( <expression> )

<reach-expression> ::= \reach ( <jml-expression> )

<duration-expression> ::= \duration ( <expression> )

<working-space-expression> ::= \working_space ( <expression> )

<space-expression> ::= \space ( <expression> )

<model-program-block> ::= { <model-program-statement>* }
```

```
<model-program-statement> ::=
        <java-statement>
      | <jml-statement>
      | <jml-model-statement>

<jml-model-statement> ::=
        <choose-statement>
        <repeat-statement>

<choose-statement> ::=
        <guarded-block> ( or <guarded-block> )* ( else <or-block> )?

<guarded-block> ::= [ <jml-expression> -> ] <or-block>

<or-block> ::= <model-program-block>

<repeat-statement> ::=
        [ <loop-specification> ] <guarded-block> ( or <guarded-block> )*

<key-expression> ::= \key( <string-literal> )

<lbl-expression> ::=
           ( ( \lbl <id> <expression> ) )
        | ( ( \lblpos <id> <expression> ) )
        | ( ( \lblneg <id> <expression> ) )
```

Appendix E

TODO

Be sure (or perhaps) to talk about

- `\same`
- Throwables in signals clauses, particularly AssertionError
- switch statements with strings
- switch expressions
- yield statements
- modules and specifications
- var declarations (type inference)
- Java 17 pattern matching switch statements
- pattern matching instanceof
- text blocks
- records
- sealed and hidden classes ?
- JEP 390: Warnings for Value-Based Classes
- enum types
- default specs for binary classes
- datagroups, JML.* utility functions, @Requires-style annotations. arithmetic modes, universe types
- visibility in JML
- Sorted First-order-logic

- individual subexpressions; optional expression form; optimization; usefulness for tracing
- RAC vs. ESC
- nomenclature
- lambda expressions
- Specification of subtypes - cf Clyde Ruby's dissertation and papers
- immutable types
- recommends-else
- How to specify lambda functions
- naming and operations of `\locset`
- other primitive types

Index