Java Modeling Language (JML) Reference Manual

2nd edition

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

DRAFT February 15, 2024

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

Preface

This document defines the Java Modeling Language (JML), a language in which one can write formal behavioral specifications for Java programs. JML was first a vehicle for discussing theoretical and soundness issues in specification and verification of object-oriented software. It then also became a formal specification language used in education about verification, since Java was a commonly taught language in undergraduate curricula; it is also frequently a basis for master's theses and Ph.D. dissertations. Finally, JML is now being used to help verify, or at least increase confidence in, critical industrial software.

With this broadening of the scope of JML, the JML community, and in particular the participants in the more-or-less annual JML workshops, considered that the long-standing and evolving Draft JML Reference manual [46] should be rewritten, made more precise, and made to represent the current state of JML used in tools. In the process, many outstanding semantic and syntactic issues have been either resolved or clarified. This document, a 2nd edition of the JML Reference Manual, is the result of that collaborative effort. Accordingly this document is a completely revised, rewritten and expanded reference manual for JML, though it borrows much text from the original document.

The document does not do some other things in which the reader may be interested:

- This document does not describe tools that implement JML or how to use those tools. Some such tools are
  - OpenJML — `www.openjml.org` — with its user guide: `www.openjml.org/documentation/OpenJMLUserGuide.pdf`, and releases: `https:github.com/OpenJML/OpenJML/releases`
  - the KeY tool — `https://www.key-project.org/` — including a book about KeY: `https://www.key-project.org/thebook2/`
- This document is not a tutorial about writing specifications in JML. For such a tutorial, see `https://www.openjml.org/tutorial`.

You may also be interested in the JML project web site at

`http://www.jmlspecs.org`

and the GitHub project for this reference manual,
https://github.com/JavaModelingLanguage/RefMan,
whose Issues log includes ongoing discussion of JML.

Contents

General notes on things not to forget:

- 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

– other Java 6+ features

– Specification of subtypes - cf Clyde Ruby's dissertation and papers

– immutable types

Chapter 1

Introduction

JML is a *behavioral interface specification language* (BISL) that builds on the Larch approach [27] [28] and Eiffel [54] [55] (and other languages such as VDM [37] and APP [66]). In this style of specification, which might be called model-oriented [72], one specifies both the interface of a method or abstract data type and its behavior [40]. In particular JML builds on the work done by Leavens and others in Larch/C++ [44] [41] [42]. (Indeed, large parts of this manual are adapted wholesale from the Larch/C++ reference manual [42].) Much of JML's design was heavily influenced by the work of Leino and his collaborators [47] [48] [50], then subsequently by Cok's work on ESC/Java2 [22] and OpenJML [19], the work on the KeY tool [2], and by work on other specification languages such as Spec# [8], ACSL [10], SPARK [7], and Dafny [49]. 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`.

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 another logical predicate called the method's *precondition*. The set of locations the method is allowed to assign to is described by the method's *frame condition* [12]. 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 [31], as adapted by Meyer to the design-by-contract style of specification [53].

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 [18], 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 [30] [69] or Fresco [70].

1.2 A First Example

As a first example, consider the JML specification of a simple Java class `Counter` shown in Fig. 1.1 on the following page. (An explanation of the notation follows.)

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

```
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 /*@ 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.

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 before the */, 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 §4 for more details about JML annotations.)

The first annotation, on line 7 of Fig. 1.1 on the previous page 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. §8.1.6). 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 §7.2 for more about object and class invariants.)

In Fig. 1.1 on the preceding page, 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 §8 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 locations (i.e. fields of objects) that are visible outside the method and which existed before the method started execution. (See §8 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 `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. 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 [13].

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 [65], 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 [29] and Chapter 7 of [27]). 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 [33, 35, 68].
- JML specifications can be used by several tools that can help debug and improve the code [13].

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 [67, 68].

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" [45] 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) [21, 23, 19, 20], or the KeY tool [11].[1] Other tools for JML [13] include Daikon [25], which can infer some JML specifications from execution traces during testing, the runtime assertion checker (RAC) of OpenJML [20], the RAC found in AspectJML [62],[2], and documentation (as in JML's jmldoc tool). The paper by Burdy et al. [13] 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 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 [46], which evolved over many years and had many contributors. This current edition of the ref-

[1] There have been other formal verification tools for JML, including the LOOP tool [33, 36].

[2] AspectJML is a further evolution of a previous RAC called ajmlc [64, 63]. There was also a RAC tool from Iowa State, called jmlc [15, 16, 17], that is no longer maintained.

erence manual is largely a rewrite of the previous draft. Some sections, particularly introductory and overview material, are taken nearly verbatim from the previous JML draft reference manual [46]. 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 and, more actively, on the JML Reference Manual GitHub site. 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 with changes to Java itself. The version of JML presented here corresponds to Java 21.

1.6 Historical Precedents and Antecedents

JML combines ideas from Eiffel [53] [54] [55] with ideas from model-based specification languages such as VDM [38] and the Larch family [27] [43] [71] [72]. It also adds some ideas from the refinement calculus [4] [5] [6] [58] [57]. 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 [31]. Later Hoare adapted these axiomatic techniques to the specification and correctness proofs of abstract data types (ADTs) [32]. 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 [52] [55] [53] [61]. 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 [38], VDM-SL [26] [60], Z [30] [69], and the Larch family [27]. 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 [27]), 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 [53] [54] [55]. 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, he experience of the JML community over the past couple of decades demonstrated that better pure mathematical data types and structures were also needed. Hence, this 2nd edition of JML incorporates more built-in mathematical types.

1.7 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 [46]

and discussions by the JML community.

Contributions from David Cok are 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 KeYproject. 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" [45] 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, the source text is divided into syntactic tokens, (largely) independent of the grammar or semantics; the JML syntax is described in §4. The grammar is described throughout the manual in the form described below (§2.3), with common aspects of the grammar summarized in §4.5.

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 entries to the page containing the reference, 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: [ ]
- 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 is defined in the JLS; a prefix of *jml-* is used to emphasize a distinction from Java.

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

Section §4.5 contains a list of definitions of common grammar non-terminals.

Chapter 3

JML concepts

This chapter describes some general design principles and concepts of the Java Modeling Language that are used throughout this manual and discuss 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 work 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]

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. 1.1 on page 4. 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; reasons for this include the Java source code not being available or being proprietary. 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 §17.

3.2 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.

3.3 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 behav-

iors that are marked with the `code` modifier8.1.4. 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.

3.4 JML modifiers and Java annotations

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

3.4.1 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 §A.

3.4.2 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 (§3.4.1) 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 (§??).

3.5 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.

3.5.1 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 JSR ?? https://docs.oracle.com/javase/specs/jls/se21/html/index.html and the more understandable explanation in the Checker framework ??, in the context of type inference and checking.

There are two syntactical complications to be aware of:

3.5.1.1 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
```

3.5.1.2 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

3.5.1.3 var

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

3.5.2 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. 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.

3.5.3 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 §3.5.2.
- 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 applied to any given type name or declaration.

3.5.3.1 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.

3.5.3.2 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.

3.5.3.3 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
  ```
  public @NonNull java.lang.Object m1()
  ```
  or
  ```
  @NonNull public java.lang.Object m1()
  ```
  in addition to
  ```
  public java.lang.@NonNull Object m1()
  ```
  , and similarly using `/*@ non_null */`.
- 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.

3.5.3.4 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

3.5.3.5 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`.

3.5.3.6 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.

3.5.3.7 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).

3.5.3.8 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, eavh of which is null. So this value could be use in an initialization such as this: `@Nullable String @NonNull [] a = new String[3];`

3.5.3.9 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)
```

3.5.3.10 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.

3.5.3.11 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`.

3.5.3.12 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.

3.5.3.13 Declarations in JML `old` method specification clauses

The 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.

3.5.3.14 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.

3.5.3.15 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 @NonNull Object o; o != null && ( range ); value
```

3.5.3.16 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.

3.5.3.17 Type names in JML `signals_only` clauses

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

3.5.4 Generic types and type annotations

TODO

3.5.5 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.

3.6 Visibility

To be written - note material written in Method Specifications section

3.7 Model and Ghost

To be written

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.

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

3.9 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 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.

3.9.0.1 Pure methods

Pure methods do not change the state and furthermore 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   //@ pure
3   abstract public int theInt();
4
5   public void test(A a) {
6     int x = theInt();
7     //@ assert theInt() == theInt();
```

```
    //@ assert x == theInt();
    int y = theInt();
    //@ assert x == y;
    //@ assert a == this ==> x == a.theInt();
  }
}
```

3.9.0.2 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.

```
abstract class A {
  //@ assigns \nothing;
  abstract public int theInt();

  public void test(A a) {
    int x = theInt();
    int y = theInt();
    //@ assert x == y;
    int z = a.theInt();
    //@ assert a == this ==> x == z;
  }
}
```

3.9.0.3 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.

```
abstract class A {
  abstract public int nextInt();

  public void test() {
    int x = nextInt();
    int y = nextInt();
    //@ assert x == y; // NOT PROVABLE
  }
}
```

3.9.0.4 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 must be deterministic or it is specified incorrectly, as it must depend on some property outside the program.

3.10 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 referenes between different objects, and use of mutable data fields of one object in another object's invariant.

3.10.1 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 is an integer that 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.

3.10.2 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.

3.10.3 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 [56], 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.

3.10.3.1 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` (§7.2) clause that is part of a method's specifications. 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[3] 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 non-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). Do 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.

3.10.3.2 Managing non-local and mutual references

TODO

3.11 Location sets and Dynamic Frames

To be written - see section in DRM on Data Groups

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

3.12 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 §13.

3.13 Redundant specifications

JML has some features that enable expressing redundant specification. In particular, there are `..._redundantly` keywords, 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.

3.14 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; they may be duplicates of each other. 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 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 precondition is true.
- The name of a clauses 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.

3.15 Specification inference

If no specifications are present for some program entity, JML presumes some defaults (§10). 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 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.

3.16 org.jmlspecs.lang package

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

contains (at least[4]) these elements:

- `JML.informal(<string>)` : This method is a replacement for (and is equivalent to) the informal expression syntax (§12.20) `(* ... *)`. Both expressions return a boolean value, which is always `true`.
- TBD

More to write here

3.17 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 §12.5.

3.18 Core JML

Should this section come later? Or even perhaps as an appendix. I guess the Core attribute is a language definition property, but certainly note the discussion of tool support.

There is a tension in a language design project meant for several purposes: research, practical, and educational use. That is, language design research tends to add an assortment of experimental features; practical applications demand a robust but substantial set of language capabilities, but one for which tools are maintainable – that is, the feature set is reasonably stable; 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:

Core features should be supported by all tools and should be the focus of education,

[4] Tools implementing JML may add additional methods.

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

Tools may of course implement what they wish, but they are encouraged to follow the JML definition, the above categorization and, where tool-specific extensions to JML are implemented, to avoid conflicting with JML and to guard the use of extensions with tool-specific conditional annotations (§4.1.5).

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.

Move table to an appendix

The entries in the table have these meanings:

- **Core** — a Core construct
- **Dep.** — a Deprecated construct
- **Adv.** — an Advanced construct
- **Exp.** — an Experimental construct
- ¶ — a deprecated construct or feature
- **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
§1	`code`	**Adv.**	-	+	
§1	`code_bigint_math`	**Adv.**	-	+	
§1	`code_java_math`	**Adv.**	-	+	
§1	`code_safe_math`	**Adv.**	-	+	but is the default in **Core**
§1	`extract`	**Exp.**	-	-	
§1	`ghost`	**Core**	+	+	fields
§1	`heap_free`	**Adv.?**	+	+	trial OpenJML extension
§1	`helper`	**Core**	+	+	
§1	`immutable`	**Adv.?**	-	+	extension?
§1	`instance`	**Core**	+	+	

	feature	Category	KeY	OpenJML	Comments
§1	`model`	**Core**	+	+	fields, methods
§1	`model`	**Adv.**	-	+	classes
§1	`monitored`	**Conc.**	**?**	-	
§1	`non_null`	**Core**	+	+	a type modifier
§1	`non_null_by_default`	**Core**	+	+	
§1	`no_state`	**Adv.?**	+	-	heap-independent model method
§1	`nullable`	**Core**	+	+	a type modifier
§1	`nullable_by_default`	**Core**	+	+	
§1	`public private protected`	**Core**	+	+	for clauses and contracts
§1	`peer rep read_only`	**Adv.**	(+)	-	
§1	`pure`	**Core**	+		
§1	`query secret`	**Exp.**	-	+	observational purity
§1	`spec_bigint_math`	**Adv.**	+	+	but is the default in **Core**
§1	`spec_java_math`	**Adv.**	+	+	
§1	`spec_protected`	**Adv.**	+	+	
§1	`spec_public`	**Core**	+	+	
§1	`spec_safe_math`	**Adv.**	-	+	
§1	`strictly_pure`	**Core?**	+	-	KeY
§1	`two_state`	**Adv.**	+	-	model method with access to `\old`
§1	`uninitialized`	**Adv.**	**?**	-	
§1	Java annotations instead of modifiers	**Adv.**	**?**	+	

File level features

	feature	Category	KeY	OpenJML	Comments
§1	model imports	**Core**	+	+	
§1	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.**	**?**	+	

feature	Category	KeY	OpenJML	Comments
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	**Adv.**	+	+	
for_example	**Adv.Exp.?**	-	-	semantics unclear
implies_that	**Adv.Exp.?**	-	-	semantics unclear
inline	**Ext.**	-	+	OpenJML: inlines method as its spec
measured_by	**Adv.Core?**	+	-	needs revision
normal_behavior	**Core**	+	+	
old	**Adv.**	**?**	+	
requires	**Core**	+	+	
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

feature	Category	KeY	OpenJML	Comments
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
§11.1	`assert`	**Core**	+	+	
§11.2	`assume`	**Core**	+	+	
§??	`debug`	**Dep.**	-	+	
§B.1.10	`hence_by`	**Dep.**	-	-	
§??	`reachable`	**Adv.**	-	+	
§11.8	`set`	**Adv.**	+	+	
§11.7	`unreachable`	**Adv.**	-	+	
§1	ghost label	**Core?**	?+		
§1	`loop_invariant`	**Core**	+	+	
§1	`loop_writes`	**Core**	+	+	
§1	(loop) `decreases`	**Core**	+	+	
§1	local ghost variables	**Core**	+	+	
§1	local model classes	**Adv.Exp.**	**?**	+	or perhaps forbid?
§1	block contracts	**Adv.**	+	+	
§1	`breaks`, `continues`, `returns`	**Adv.**	+	-	in block contracts
§1	begin-end markers	**Ext.Adv.?**	-	+	OpenJML
§??	`check`	**Adv.?**	-	+	OpenJML
§??	`havoc`	**Adv.?**	-	+	OpenJML
§??	`inline_loop`	**Adv.?**	-	+	OpenJML
§??	`show`	**Core?**	-	+	OpenJML
§??	`split`	**Ext.**	-	+	OpenJML

JML Types

feature	Category	KeY	OpenJML	Comments
`\bigint`	**Core**	+	+	
`\locset`	**Adv.**	+	+	builtin datatype
`\real`	**Adv.**	+	+	
`\TYPE`	**Adv.**	-	+	
built-in string , set, array, seq, map types	**?**			
`\seq`, `\map`	**?**	+	-	

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	**Core**	-	+	what does this do?
\duration	**Exp.**	?	-	
\elemtype	**?**	?	+	
\everything	**Core**	+	+	
\exception	**Ext.**	-	+	like \result
\exists	**Core**	+	+	
\forall	**Core**	+	+	
\fresh	**Core**	+	+	
\index	**Dep.?**	+	+	deprecated in favor of \count
\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.**	+	+	

feature	Category	KeY	OpenJML	Comments
`\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
§1	`//@`	**Core**	+	+	
§1	`/*@ @*/`	**Core**	+	+	
§1	`//` comments in specs	**Core**	+	+	
§1	conditional annotations	**Core?**	**?**	+	
§1	embedded annotations	**Adv.**	**?**	+	
§1	org.jmlspecs.lang	**Core**	**?**	+	package automatically imported
§1	`...redundantly`	**Adv.**	+	+	Typically implemented by ignoring the redundantly suffix
§1	`.jml` files	**Adv.?**	**?**	+	needed for library specs
§1	JML in Javadoc	**Dep.**	-	-	
§1	`nowarn`	**Dep.**	**?**	+	line annotation
	`\dl_`	**Ext.**	-	-	MU: or some other means of tool-spec exts.

Features to consider:

- recommends-else

- How to specify lambda functions
- naming and operations of `\locset`
- other primitive types

Chapter 4

JML Syntax

4.1 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 §17. The `.jml` files are quite similar to `.java` files, just in a separate file.

4.1.1 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 [1](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.

4.1.2 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.

4.1.3 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 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*.

4.1.4 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 }
```

4.1.5 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 `@`.

4.1.6 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` — 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.

4.1.7 Tokenizing JML annotations

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

- removing the opening comment marker as defined in §4.1.3
- 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: `..`, `==>`, `<==>`, `<=!=>`, `<:`, `<:=`, `<#`, and `<#=` .
- 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.

4.1.8 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 asssertion 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.

4.1.9 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 parse 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 annotations. 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 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 (§1) — 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 declarations (§1) – these are placed with Java top-level declarations, such as `import` statements or model class declarations
- type specification clauses (§1) — these are placed where Java places members of types, such as field and method declarations
- method specifications (§1) — these are placed in conjunction with the declaration of a method's signature. They in turn consist of
  - keywords
  - punctuation
  - clauses
- field specifications (§1) — these are placed in conjunction with a field declaration
- statement specifications (§1) — these are placed like statements in a code block (a method or initializer body)

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).

4.3 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 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.
- 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

[1] More precisely, the JML keywords are all different from any of Java's reserved words that might start a declaration, notably 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.

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`.

4.4 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 [1].

<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 §4.1.3

<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>*
  | *<white-space>*
  | *<jml-line-comment>*

```
      | <plain-java-comment>
      | <line-terminator>

<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>` ::= **{|** | **|}**

```
<white-space> ::= <java-white-space> | <jml-white-space>
```

`<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-white-space is really a kind of line-terminator; make the names defined here match those introduced at the beginning of the chapter

4.5 Definitions of common grammar symbols

This section assembles the definitions of a number of grammar non-terminals that are used throughout the manual. See §2.3 for information about how the productions of the grammar are written.

The grammar uses these syntactic tokens 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.

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

- *<jml-identifier>* — an identifier with its preceding backslash
- *<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

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

A type name possibly has type parameters:
<type-name> **::=** *<qualified-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:
<java-jml-expression> **::=**
<java-identifier>
| *Do we replicate the Java grammar here, adjusted?*

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 (§13) 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 need to be pure, value-based 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 [9]).

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., `heap_free` 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

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 (§1), that is, they must not have side-effects and must be declared to be `pure`.

Consequently, no allocation of new objects is allowed in specification expressions and no operations that change an object's 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 `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. For consistency and convenience, some types of this nature are provided as built-in specification types in JML and are described later in this chapter. *Are they?*

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.

- `==` and `!=` — equality and inequality of enum values

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

```
//@ 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) );
```

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

```
//@ 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.

```
1 //@ public normal_behavior
2 //@   requires n != null && (* n is e.name() for some E e *);
3 //@   ensures \result.name().equals(n); // FIXME - TODO
4 //@ also public exceptional_behavior
5 //@   requires n == null || !(* n is e.name() for some E e *);
6 //@   signals (NullPointerException) n == null;
7 //@   signals (IllegalArgumentException) !(* n is e.name() for some E e *);
8 //@   signals_only NullPointerException, IllegalArgumentException;
9 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:

```
1 //@ public normal_behavior
2 //@   requires e != null;
3 //@   ensures \result < 0 <==> this.ordinal() < e.ordinal();
4 //@   ensures \result == 0 <==> this.ordinal() == e.ordinal();
5 //@   ensures \result > 0 <==> this.ordinal() > e.ordinal();
6 //@ also public exceptional_behavior
7 //@   requires e == null;
8 //@   signals_only NullPointerException;
9 //@ pure
10 public int compareTo(nullable E e);
```

getDeclaringClass vs getClass, equals, hashCode, clone

T note some restrictions on modifiers for enums

weigl: More particular, can Enum Constants carry JML modifiers. After JLS only annotations are allowed.

5.1.2 Java records

Write something

5.1.3 Java Streams

Discuss this – streams and other functional programming bits are handy within specifications

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 <=!=> (§12.9)
- ==> (implies operation, §12.8)

JML specifications may use auto-unboxing from `Boolean` to `boolean`; however, auto-boxing is not allowed 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 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 not allowed in a specification expression.

5.4 \bigint

weigl: Is there sort hierarchy? For example, can I quantify over all Java and JML objects?

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. When JML specifications are compiled for runtime checking, `\bigint` values are represented as `java.math.BigInteger` values. However, auto-boxing of `\bigint` (to `BigInteger`) is not allowed.

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 `BigInteger` objects.

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

- unary: `+ - ~`
- binary: `+ - * / %`
- bit operations: `& | ^`
- equality: `== !=`
- comparisons: `< <= > >=`
- shifts: `>> <<`
- casting: to and from primitive Java integral and character types

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 (§1) 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`.

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

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 [34] 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 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.

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 `\real` values may not be auto-boxed. 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: `< <= > >=`
- casting: to and from primitive Java integral and character types and `\bigint`

Also, `\real` can be used in quantified expressions and variables of type `\real` can be declared as ghost or model variables.

It is undefined 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 or negative 0.

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.

- the `\type` syntax (§12.21) 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(java.lang.Integer)`
  - `\type(java.util.List<Integer>)`
  - `\type(java.util.List<Boolean>)`

- `\TYPE` values are *not* erased types. Thus `\type(java.util.List<Integer>)` and `\type(java.util.List<Boolean>)` are different values and `\type(java.util.List)` is not well-defined.
- The `\typeof` function (§12.22) takes a JML expression and returns the `\TYPE` value corresponding to its *dynamic* type.
- `\TYPE` values can be compared with `==` and `!=` as expected.
- The `<:` operator is the sub-type or equality operation.[1] Note that this is subtype on non-erased (JML) type values.
- `t.typeargs()` for a `\TYPE` value `t` is a `\seq<\TYPE>` giving the `\TYPE` values of each of the type arguments of `t`.
- `t.erasure()` gives the `java.lang.Class` value that is the erasure of `t`. For example `\type(java.util.List<Integer>).erasure()` equals `java.util.List.class`
- `t.isArray()` returns `true` iff `t` is an array type. `\elemtype(t)` (§1) returns the element type of a `t` that is an array.
- `\arrayOf(t)` for a `\TYPE` value `t` returns a `\TYPE` value that is an array of `t`. So then `\elemtype(\arrayOf(t))` is `t`.
- 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 }
```

[1] By analogy with other comparison operators, this operator ought to be `<:=`, with `<:` denoting a proper subtype. But original JML included equality in `<:` and it would be highly backwards-incompatible to change that definition.

5.8 \locset

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 §1). 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

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

- `\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))`.

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 data-groups / 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

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

  . 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.)

5.9 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.

Do we want to define the non-Java syntax \seq(T ...) as a value constructor? – DRC: Probably `of` is clear and concise enough to not have to use non-Java syntax

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

  AW: Maybe we should consider the binary operations. intersection "&", union "|"
- `-` – 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>`)

syntax for disjoint

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

Do we use a special JML function syntax like `\disjoint` or a Java method syntax like `set.disjoint(...)` – DRC: I suggest sticking with Java syntax

Do we have methods that duplicate the operations above so that we can use them as Java method references in lambda operations? – DRC: Yes - simplifies implementation for both ESC and RAC, while giving concise operator syntax as well

- `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
- `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

Runtime equivalent The JML type `\set<T>` is mapped to *org.jmlspecs.lang.set or org.jmlspecs.lang.Set or org.jmlspecs.runtime.set or org.jmlspecs.runtime.Set* for runtime assertion checking.

5.10 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 §1. (*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)

DRC: I'm in favor of the two [] operations above – we already have them in JML in other contexts

DO we permit `s[i]=v;` as an update operation, equivalent to `s = s.put(i,v);` – DRC: not sure – nice syntax but possibly confused with mutable operations on Java arrays

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

- s.length() — the length of the sequence (a \bigint)
- \seq.<T>equals(\seq<T>,\seq<T>) – returns true iff the two arguments have the same elements in the same order

- `s.get(i)` — the boolean result is the element (type `T`) of the sequence at position `i` (0-based, with `0 <= i < s.length`.
- `s.put(i,t)` — returns a new sequence of type `\seq<T>` which is equal to `s` except that position `i` in the sequence now contains the value `t`, where `0 <= i < s.length`.
- `s.has(t)` — the boolean result is true iff `t` is an element of `s`.
- `s.add(t)` — returns a `\seq` that is `s` with `t` added onto the end
- `s.prepend(t)` — returns a `\seq` that is `s` with `t` added onto the begining
- `s.concat(ss)` — returns a `\seq` that is `s` concatenated with `ss`
- `s.sub(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(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(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`.

5.11 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.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.

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 §1. (*range syntax is under discussion*)
- `+` — string concatenation (binary operation)

Functions All the functions defined for `\seq` are defined for `\string` as well, with these additions:

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

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

5.12 Mathematical maps: `\map<T,U>`

The type `\map<T,V>` is a built-in type of finite maps with keys of type `T` and values of type `V`. `T` 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.<T,V>empty()` — creates an empty map of type `\map<T,V>`

Operators

- `==` and `!=` — equality and inequality. Two values of type `\map<T,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 `T` and `V`.
- `[]` – item value, that is `s[t]` is the value in the map of type `V` corresponding to the key `t` (of type `T`). 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<T,V>`, `t` is a value of type `T`, and `v` is a value of type `V`.

- `m.keys()` — a `\set<T>` containing exactly the keys of the map `m` (i.e. the domain of the map)
- `m.values()` — a `\set<V>` containing exactly the values of the map `m` (i.e. the range of the map). Note that the cardinality of the range may be less than that of the domain because different keys may map to the same value.
- `m.get(t)` or `m[t]` — the value of the map for the given key; the value is undefined if `t` is not an element of `m.keys()`, but the expression is still well-defined.

- `m.put(t,v)` — returns a new map of type `\map<T,V>` that includes `v` as the value for the key `t`, with the values for all keys not equal to `t` unchanged from those in `m`.

5.13 Mathematical arrays: `\array<T>`

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

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

Fill this in

Chapter 6

JML Specifications for Packages and Compilation Units

There are no JML specifications at the package level. If there were, they would likely be written in the `package-info.java` file for the package. 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. Model classes are discussed in §6.4.

6.1 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 are import statements that 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).

6.2 Default imports

The Java language stipulates that `java.lang.*` is automatically imported into every Java compilation unit. Similarly in JML there is an automatic model import of `org.jmlspecs.lang.*`. However, there are not yet any standard-defined contents of the `org.jmlspecs.lang` package.

Write a placeholder for this somewhere

6.3 Issues with model import statements

As of this writing, no tools distinguish between Java import statements and JML import statements. Such implementations may resolve names in Java code differently than the Java compiler does. Consider two packages `pa` and `pb` each declaring a class `N`.

1)

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

Correct behavior: In Java code `N` is `pa.N`; in JML code, `N` is ambiguous.
Non-conforming behavior: JML tools consider `N` in Java code to be ambiguous.

2)

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

Correct behavior: In Java code `N` is `pa.N`; in JML code, `N` is `pa.N`.
Non-conforming behavior: non-conforming JML tools will act correctly in this case.

3)

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

Correct behavior: In Java code `N` is `pa.N`; in JML code, `N` is `pb.N`.
Non-conforming behavior: JML tools consider `N` in Java code to be `pb.N`.

4)

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

Correct behavior: In Java code `N` is `pa.N`; in JML code, `N` is ambiguous.
Non-conforming behavior: JML tools consider `N` in Java code to be ambiguous.

6.4 Model classes and interfaces

Just as a Java compilation unit (e.g., 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
- 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 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 *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.

7.1 Modifiers for type declarations

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 §3.4, 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 {...}
```

7.1.1 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 §1. 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 specfications 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.

7.1.2 pure and @Pure

Specifying that a class is *pure* means that each method and nested class within the class is specified as pure. The `pure` modifier on a class is not inherited by derived classes, though `pure` modifiers on methods are.

There is no modifier to disable an enclosing `pure` specification.

7.1.3 @Options

The `@Options` modifier takes as argument either a String literal or an array of String literals (with 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 containing 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.

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.

7.2 `invariant` clause

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

TODO

visibility modifiers?

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 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 an adding `ensures` clauses (with the predicate stated by the `constraint` clause) to each specification behaviors 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.

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.

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.

An `innitially` 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 an adding `ensures` clauses (with the predicate stated by the `constraint` clause) to each specification behaviors of each constructor in the type, subject to the visibility rules. Like an `ensures` clause, a `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.

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).

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 in 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
- A *<represents-clause>* may be declared as `static`. 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.
- A non-static represents clause must not have a static model field in its left-hand side.

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 [59].

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 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 effects (that is, are not *pure*) are permitted and might be useful in runtime checking, in practice, model methods are nearly always `pure`.

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.

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 model class declaration obey the same syntax rules as top-level model declarations and behave like nested 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 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`
- `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 `B.bb` were declared `final`, then `B.bb` 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 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

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.

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 righthand-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.

The *<monitors-for-clause>* is adapted from ESC/Java [50] [65]. 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 behavior of the method. JML is a modular specification methodology, with the Java method being the fundamental unit of modularity. Method specifications constrain the implementation of a method, in that the implementation must do what is stated by the specification; method specifications constrain callers of methods in that they constrain the states in which the method may be called and what may be assumed about the state when the method completes execution.

The specifications may under-specify a method. For example, the specifications may simply say that the method always returns normally (that is, without throwing an exception), but give no constraints on the value returned by the method. The degree of precision needed will depend on the context.

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 may also contain Java visibility modifiers.

```
<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> ( code )? <model-program>

<java-visibility> ::= ( public | protected | private )?

<behavior-id> ::=
```

```
      behavior | normal_behavior | exceptional_behavior
    | behaviour | normal_behaviour | exceptional_behaviour

<clause-seq> ::= ( <clause> | <nested-clause> )*

<clause> ::=
       <requires-clause>          §8.4.1
     | <old-clause>               §8.5.5
     | <writes-clause>            §8.4.3
     | <reads-clause>             §8.5.1
     | <callable-clause>          §8.5.8
     | <ensures-clause>           §8.4.2
     | <signals-clause>           §8.4.4
     | <signals-only-clause>      §8.4.5
     | <diverges-clause>          §8.5.2
     | <measured-by-clause>       §8.5.3
     | <when-clause>              §8.5.4
     | <duration-clause>          §8.5.6
     | <working-space-clause>     §8.5.7
     | <captures-clause>          §8.5.9
     | <method-program-block>

 <nested-clause> ::=              §8.1.2
      {|  ( <clause-seq> ( also <clause-seq> )* )? |}
```

Meta-parser rules:

- Each of the behavior keywords spelled `behaviour` is equivalent to the corresponding keyword spelled `behavior`.
- A behavior beginning with `normal_behavior` may not contain a *<signals-clause>* or a *<signals-only-clause>*. It implicitly contains the clauses
  `signals (Exception e) false; signals_only \nothing;` .
- A behavior beginning with `exceptional_behavior` may not contain a *<ensures-clause>*. It implicitly 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. §4.1.5). 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>*. This 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 (§8.1.3) that should be used where possible to enhance readability.

Note that the vertical bars in the production for *nested-clause* are literals, not meta-symbols.

8.1.1 Behaviors

The basic structure of JML method specifications is as a set of *behavior*s (or *specification case*s). The order of *<behavior>*s within a *<behavior-seq>* is immaterial.

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.

For each behavior, if the method is called in a context in which the behavior's precondition (`requires` clause) is true, then the method must adhere to the constraints specified by the remaining clauses of the behavior. Only some of the behaviors need have preconditions that are true; unless at least one behavior has a true precondition, the method is being called in a context in which its behavior is undefined. For example, a method's specification may have two behaviors, one with a precondition that states that the method's argument is not null and the other behavior with a precondition that states that the method's argument is null. 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 in any context other than those in which the first precondition, that the argument is not null, is true. 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.

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** $\langle\textit{clause-seq}\rangle_{Si}$) $*$ |}
then
($\langle\textit{java-visibility}\rangle_V$ (**code**)? $_W$ $\langle\textit{behavior-id}\rangle_X$)? $\langle\textit{clause}\rangle*_D \langle\textit{nested-clause}\rangle_A \langle\textit{clause-seq}\rangle_E$
is equivalent to a sequence of n *<behavior>* constructions
($\langle\textit{java-visibility}\rangle_V$ (**code**)? $_W$ $\langle\textit{behavior-id}\rangle_X$)? $\langle\textit{clause}\rangle*_D \langle\textit{clause-seq}\rangle_{S1} \langle\textit{clause-seq}\rangle_E$
also

. . .
also
(*<java-visibility>*$_V$(**code**)?$_W$ *<behavior-id>*$_X$)? *<clause>** $_D$*<clause-seq>*$_{Sn}$*<clause-seq>*$_E$

Is there a better way to describe this desugaring? and a better way to format it?

8.1.3 Ordering of clauses

The clauses are defined to be in the following groups:

- preconditions (requires, old clauses)
- read footprint (accessible clauses)
- frame conditions (assignable clauses)
- call conditions (callable 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 when, captures, recommends 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 methods overridden by the given method. In other words, method specifications are inherited (with exceptions given below), as was described in §3.3.

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 is not 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 `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 the 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.

8.1.6 Visibility

The following discussion has some errors and needs fixing; also need to talk about spec_public, spec_protected

Each method specification behavior has a *java-visibility* (cf. the discussion in §1). Any of the kinds of behavior keywords (`behavior`, `normal_behavior`, `exceptional_behavior`) may be prefixed by a Java visibility keyword (`public`,

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
protected	public, protected-by-inheritance
package	public, protected-by-package, package
private	any

`protected`, `private`); the absence of a visibility keyword indicates package-level visibility. A lightweight behavior (one without a behavior keyword) has the visibility of its associated method.

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

any method specification that is visible to a client may contain only names (of a type, method or field) that are themselves visible to the client.

For example, a public behavior may contain only public names. A private behavior may contain any name visible to a client that can see the 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 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 names that are visible by virtue of being in the same package (and public names); 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.

The root of the 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 8.1.

8.1.7 Grammar of method specifications

Fillin – remember lightwieght, behavior, normal_behavior, exceptional_behavior, examples, implies_that, visibility, model program behaviors, `also`, nested behaviors

Do we relax the ordering and the constraints on nesting that are in the current RefMan

8.2 Method specifications as Annotations

At one time, there was an experimental implementation of method and other specifications written as string arguments of Java annotations, for example, `@Requires("requires true;")`. Such use of Java annotations is no longer defined in JML. Only JML modifiers (which do not have arguments) have equivalent Java annotations (e.g., `pure` and `@Pure`).

8.3 Modifiers for methods

TODO

8.4 Common JML method specification clauses

TODO

There are quite a few kinds of method specification clauses. Those described in this section are the more commonly used. The following section (8.5) describes the others, some of which are still research ideas.

8.4.1 `requires` clause

Grammar:
<requires-clause> `::=` **`requires`** *<opt-name>* *<jml-expression>* `;`

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>* (§8.5.5) prior to the *<requires-clause>* in the same specification case is also in scope and hides any other names.

A `requires` clause states a precondition for a method. That is, if the given predicate is true at the point of the associated method, then the rest of the clauses in the behavior must also hold.

The disjunction of the preconditions from each of the behaviors (including any inherited ones) is the *effective precondition* of the method: it must be true at any point the method is called (so at least one of the behaviors has a true precondition); when

reasoning about the body of a method, the effective precondition is assumed to be true at the beginning of the method body.

There may be more than one `requires` clause in a specification case. The order of `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.

The default requires clause is `requires true;`, which puts no requirements on the caller of the method.

8.4.2 `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. §12.12).

An `ensures` clause states a postcondition for a method. That is, the given predicate must be true just after any `return` statement in the method body and may be assumed by the caller at the call point. Note that the semantics is that *if the method returns normally, then the postconditions are true* (if the program is verified). The converse, namely, *if the postcondition is true then the method terminates normally*, is not necessarily true.

There may be more than one `ensures` clause in a specification case. The order of `ensures` clauses in a specification case is significant in the same way that the order of terms in a short-circuit boolean expression is significant: earlier *<ensures-clause>* expressions may state conditions that enable later ones to be well-defined. In addition, all of the preconditions in the same specification case can be assumed to be true.

The default ensures clause is `ensures true;`, which puts no requirements on the body of the method.

8.4.3 `assignable` clause

Grammar:
TODO

8.4.4 `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 predicate. The identifier may be omitted if it is not needed in the predicate.

A `signals` clause states a condition that must be true if a method exits via an exception of the given type (or derived from it).

The semantics is that *if the method terminates by throwing an exception, then the predicate must be true* (if the program is verified). The converse, namely, *if the predicate is true then the method terminates with the given exception*, is not necessarily true.

There may be more than one `signals` clause in a specification case. There is no meaning to their order.

The default `signals` clause is `signals (Exception) true;`, which puts no requirements on the body of the method.

8.4.5 `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, under the preconditions of the specification case, only the listed Java Exceptions may be thrown. That is, if the method terminates with an exception it must be or be derived from one of the listed exceptions. The token `\nothing` denotes an empty list (no exceptions may be thrown). In contrast to the Java throws list, if any kind of `RuntimeException` is to be permitted by the JML specification, it must be explicitly listed.

There is no point to listing exceptions in a *<signals-only-clause>* that are not (implicitly) 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. `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 `RuntimeException`.

Note that the exceptions listed in a *<signals-only-clause>* have an effect on the use of `allow` and `forbid` annotations (§??).

8.5 Advanced JML method specification clauses

These clauses are less commonly used and may be less-well-supported by tools.

8.5.1 `accessible` clause

Grammar:

TODO

8.5.2 `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>*(§8.5.5) 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 [24]. Explicitly writing a diverges clause allows one to obtain a partial correctness specification [31].

As an example of the use of `diverges`, consider the abort method in the following class. (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), and 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.

```
1 package org.jmlspecs.samples.jmlrefman;
2 public abstract class Diverges {
3
```

```
4       /*@ public behavior
5         @     diverges true;
6         @     assignable \nothing;
7         @     ensures false;
8         @     signals (Exception) false;
9         @*/
10      public static void abort();
11  }
```

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.

8.5.3 `measured_by` clause

Grammar:

TODO

8.5.4 `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>*(§8.5.5) 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 [51, 65]. 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 predicate is `true`.

See [65] for more about the `when` clause.

8.5.5 `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.

An `old` clause declares and initializes a single variable of a Java or JML type. The initializer for the declared variable is evaluated in the pre-state. The scope of the newly declared variable is the remainder of the *<behavior>* or *<nested-clause-seq>* in which it occurs. Like Java declarations, the new variable name hides other variables of the same name, including 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 *<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 nay textually preceding `requires` clauses are true.

The purpose of the clause is to be able to capture and name the value of a subexpression that is used more than once, or just to break up long expressions for the sake of clarity.

8.5.6 `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 [39].

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` and other JML operators appropriate for postconditions.

In any specification case, an omitted `duration` clause means the same as a duration clause giving an unreasonably large amount of time.

See §12.34 for information about the `\duration` expression that can be used in the duration clause to specify the duration of other methods.

8.5.7 `working_space` clause

Grammar:

```
<working-space-clause> ::=
   <working-space-keyword> <opt-name> <expression>
       [ if <expression> ] ;
<working-space-keyword> ::= working_space |working_space_redundantly
```

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, of course. This is adapted from the work of Krone, Ogden, and Sitaraman on RESOLVE [39].

The expression in a working space clause must have type `\bigint`. It is to be understood in units of bytes. It provides a guarantee of the maximum amount of additional space used by the call.

The expression is evaluated in the post-state and thus may use `\old` and other JML operators appropriate for postconditions. This is because it is considered to be evaluated in the post-state. 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 unreasonably large number of bytes.

See §12.35, for information about the `\working_space` expression that can be used to describe the working space needed by a method call. See §12.36, for information about the `\space` expression that can be used to describe the heap space occupied by an object.

8.5.8 `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. If no *<type-name>* is given, the *<identifier>* must name a method in the enclosing class; otherwise it must

name a method in the named type. 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.

What about generic methods?

The callable clause is a postcondition clause. It states that all methods called within its body (including any called by method calls in its body, recursively) are contained in the the list of methods in the clause. The term `\nothing` denotes an empty list of method signatures.

8.5.9 `captures` clause

Grammar:

```
<captures-clause> ::=
    <captures-keyword> <opt-name> <expression> ...  ;
<captures-keyword> ::= captures | captures_redundantly
```

8.6 Model Programs (`model_program` clause)

8.6.1 Structure and purpose of model programs

8.6.2 `extract` clause

TODO

8.6.3 `choose` clause

TODO

8.6.4 `choose_if` clause

TODO

8.6.5 `or` clause

TODO

8.6.6 `returns` clause

TODO

8.6.7 `continues` clause

TODO

8.6.8 `breaks` clause

TODO

8.7 Modifiers for method specifications

8.7.1 `pure` and @Pure

A `pure` modifier on a method declaration states two things:

-
- (a) the method has no side-effects, that is, every *<behavior>* for the method declared in the same class as the method (that is, excluding inherited behaviors) is implicitly `assignable \nothing;` (b) the method may be used in specifications

Note that (a) alone does not allow a method to be used in specifications.

Need to distinguish pure and strictly_pure

8.7.2 `non_null`, `nullable`, @NonNull, and @Nullable

These modifiers are converted into type annotations. They are discussed in §3.5 and §3.4.2.

8.7.3 `model` and @Model

This modifier identifies a method declared within a JML annotation as a *model method*. The method may not also have a `ghost` modifier. Model methods are discussed in §7.8.

8.7.4 `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.

8.7.5 `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 §1.

Typically `helper` methods are internal utility methods and are often (though not necessarily) private.

May helper methods be overridden, if they remain helper

8.7.6 `heap_free` and @HeapFree

A method marked as `heap_free` is one whose result is independent of the state of the heap (a claim that tools should check). A `heap_free` method is implicitly also `pure`. A `heap_free` 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.

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.

It seems that values of immutable types, such as Java Strings or enums or records, could also be allowed. One would need to check that any final (constant) references are not just a container for mutable references, and also that there values are only used once initialization is complete. Also are there any situations in which non-static but final fields could be used?

8.7.7 `query`, `secret`, @Query, and @Secret

These modifiers relate to observational purity and are still experimental.

8.7.8 `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 §13. 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 *TBD*

8.7.9 `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.

8.7.10 `@Options`

This is perhaps too tool-specific

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, that 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; those that do not apply are silently ignored.

This is useful when there is not a built-in modifier for a particular option. For example, one could write

```
public void TestOption {
  @org.jmlspecs.annotation.Options({"-progress","-timeout=1"})
  public void m() {
    //@ assert false;
  }
}
```

Here method `m` is processed with the given timeout and verboseness level, wdespite 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.

8.7.11 `extract` and @Extract

This modifier applies only to methods with bodies.

TBD

8.8 TODO Somewhere

<:: token

lots more backslash tokens

Chapter 9

Field 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.

9.1 Field and Variable Modifiers

The modifiers permitted on a field, variable, or formal parameter declaration are shown in Table 9.1.

9.1.1 non_null and nullable (@NonNull, @Nullable)

Discuss @NonNull TestJava a, b;
Need to discuss relationship with JSR308

The non_null and nullable 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.

9.1.2 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. Section 1 contains a detailed discussion of the effect of information hiding using Java visibility on JML specifications.

Listing 9.1: Use of spec_public

Table 9.1: Modifiers allowed on field, variable and parameter declarations

Modifier	Where	Purpose
non_null	field, var, param	the variable may not be null (§9.1.1)
nullable	field, var, param	the variable may be null
spec_public	field	visibility is public in specs
spec_protected	field	visibility is protected in specs
model	field	representation field
ghost	field, var	specification only field
uninitialized	var	TBD
instance	field	not static
monitored	field	guarded by a lock
secret	field, var, param	hidden field
peer	field, param	TBD
rep	field, param	TBD
readonly	field, param	TBD

```
1 private /*@ spec_public */ int value;
2
3 //@ ensures value == i;
4 public setValue(int i) {
5   value = i;
6 }
```

For example, Listing 9.1 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. The solution is to declare, in JML, that `value` is `spec_public`, as shown in the Listing.

9.1.3 `ghost` and `@Ghost`

TODO – see later section

9.1.4 `model` and `@Model`

TODO – see later section

9.1.5 `uninitialized` and `@Uninitialized`

TODO

9.1.6 `instance` and `@Instance`

The JML `instance` modifier is the opposite of the Java `static` modifier; that is, an `instance` entity is a member of 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 }
```

9.1.7 `monitored` and `@Monitored`

9.1.8 `query`, `secret` and `@Query`, `@Secret`

TODO

9.1.9 `peer`, `rep`, `readonly` (`@Peer`, `@Rep`, `@Readonly`)

TODO

Check readonly vs. read_only, Readonly vs. ReadOnly

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 (§1).

- a JML field must be one of either ghost or model, and not both
- a ghost field in an interface must be static

TODO - a small example?

9.3 Model fields

TODO

- a JML field must be one of either ghost or model, and not both

- a non-final model field may not have an initializer (the value of a model field is constrained by specifications, including `represents` clauses — §1.

9.4 Datagroups: `in` and `maps` clauses

TODO

9.5 `maps` clause

<maps-clause> `::=` **`maps`** *<storeref>* `\into` *<identifier>* `...` `;`

Type information:

- Each *<identifier>* must be a datagroup (including model fields).

The `maps` clause states that given *<storeref>* is a member of each of the given datagroups.

Allow a list of storerefs?

Allow null dereferences in the storerefs

Chapter 10

Default specifications and specification inference

A default specification for a Java method is assumed wherever (a) there is a library method with no source code or specification file, (b) a method with Java source code but not explicit specifications, or (c) an implicit (compiler constructed) method. This chapter defines the specifications that JML assumes in these cases.

10.1 Class specifications

A class without specifications does not add any defaults – no modifiers or specification clauses. However, the specifications of static and instance initialization are essentially inferred by inlining the class as described in the following subsections.

Default visibility of clauses

10.1.1 Static initialization

Write this

10.1.2 Instance initialization

Write this

10.2 Field specifications

A field without modifiers or specification clauses does not by default have any.

10.3 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:

```
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 + contents of method's 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, the method is by default `volatile`

If the method has a `pure` modifier then the default is

```
1  requires true;
2  accessible \everything;
3  assignable \nothing;
4  captures \everything;
5  callable \everything;
6  ensures true;
7  signals (Exception e) true;
8  signals_only RuntimeException + contents of method's 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, the method is by default not `volatile`

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 `Errors`) 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.

10.4 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 the `pure` modifier if any method it overrides is marked `pure`. It may also declare itself `pure` even if its parents do not.

A method may add its own modifiers (e.g., `spec_public`, `spec_protected`, `helper`) independently of its inheriting behaviors.

10.5 Library methods

To ensures soundness, the defaults for library methods without either source code or explicit specifications are these:

```
requires true;
accessible \everything;
assignable \everything;
captures \everything;
callable \everything;
ensures true;
signals (Exception e) true;
signals_only RuntimeException + contents of method's throws declaration;
diverges true;
when true;
measured_by <very large number>;
duration <very large number>;
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`.

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.

10.6 Object()

As the `java.lang.Object` class has no superclass, its default constructor has a simple default specification:

```
requires true;
accessible \nothing;
assignable \nothing;
captures \nothing;
callable \nothing;
ensures true;
signals (Exception e) false;
signals_only \nothing;
diverges false;
when true;
measured_by 0;
duration <very large number>;
working_space <very large number>
```

10.7 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
- 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.

```
class P {
  public P() {
    //@ assert (this instanceof Con) ==> ((Con)this).f == 0;
  }
}

public class Con extends P {
  public int f = 1;
  {
    f = 2;
  }
}
```

10.8 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.

10.8.1 Specification in .jml file

If there is a .jml file containing the specification of the parent 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.

```
// (portion of) .jml file
class A {
  //@ pure
  public A() {}
}
```

10.8.2 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() {}
}
```

10.8.3 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

```
1     /*@ public normal_behavior
2       @   assignable \nothing;
3       @   reads \nothing;
4       @*/
5     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.

10.9 Enums

Write this

10.10 Records

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 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 modification `immutable`
- 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.

- 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.
- 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*
- The generated `hashCode` method has a `public behavior` with an `ensures true;` postcondition. *other clauses*
- The generated `toString` method has a `public behavior` specification case in which the ensures postcondition is `ensures true;`.The record's `.hashCode()` method is `pure` if all of the component types have `pure .hashCode()` methods. The record's `.toString()` method is `pure` if all of the component types have `pure .toString()` methods.

other clauses

Thus for an example declaration
`record Count(int number, /*@ nullable */ T value) XXX`
we have the specification

```
1  final class Count {
2      //@ spec_public nullable
3      final private int number;
4
5      //@ spec_public
6      final private 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.

10.10.1 Lambda functions

Write this

10.10.2 Loops

Move this to where loop specs are discussed

A loop typically has four specification clauses:

- a loop invariant that constrains the value of the loop index (or `\count` value §1)
- a loop invariant that gives the inductive predicate stating what the loop is accomplishing
- an `assigns` clause that states what the loop body modifies
- a `decreases` clause need to demonstrate loop termination

JML does not specify a default for any of these, though for simple loops all but the second are quite easy to infer.

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.

Many JML specification statements end with a semicolon. That 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.

Grammar:

```
<jml-statement> ::=
        <jml-assert-statement>          §11.1
      | <jml-assume-statement>          §11.2
      | <jml-local-variable>            §11.3
      | <jml-local-class>               §11.4
      | <jml-ghost-label>               §11.5
      | <jml-unreachable-statement>     §11.7
      | <jml-set-statement>             §11.8
      | <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 warning if the expression is not provable. A run-time 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. *Mattias - does KeY behave this way?*

Clarify the recommended behavior of Java assert statements

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 the assume statements.

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 the following code 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.
- Java annotations
- `non_null`, `nullable` - these may modify the *<decl-type>* in the declaration, if it is a Java reference type

Variables declared in such a ghost declaration may be used in subsequent JML expressions and they may be assigned values in `set` statements (§11.8).

Any other JML modifiers?

Grammar needs to permit array initializers

11.4 Local model class declarations

<jml-local-class> ::=

Need a grammar entry – requires 'model'; permits absence of method implementation

Should have active agreement to support this features

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 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 }
```

11.5 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 labelled statement. A label name may shadow the name of a previously labeled statement. 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` (§12.15) and `\fresh` (§12.16) expressions can refer to the program state at a particular statement label. JML also allows inserting statement labels into the source code (§11.5).

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

Implicit 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 extension 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. 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.

Type checking requirements:

- the *<java-statement>* may be any single executable Java statement, including a block statement

The DRM requires a set statement to take an assignment expression.

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; 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**
    | **decreases_redundantly**
    | **decreasing_redundantly**
<loop-frame> ::=
    <loop-frame-keyword> (**nothing** | *<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. *Write this*

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 just after the loop update ststement has been executed and 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 the very common pattern for loop specifications.

```
1 // a is a non-null array to be initialized
2 //@ maintaining 0 <= k <= a.length;
3 //@ maintaining \forall int j; 0<=j<k; a[j] == j*j;
4 //@ assigning a[*];
5 //@ decreasing 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 following `assert` statement is true.

Fix the example for the final agreed upon keywords

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 is independent of the loop index. 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.

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*[1] are very similar to those of a method specification (cf. §1). A method specification states preconditions on the legal states in which a method may be called and gives conditions on what the effects of a method's execution may be, including comparisons between the pre-state (before the method call) and the post-state (after the method completion). Similarly, a block specification makes assertions about the execution of the statement (possibly a block statement) that follows the block specification, or, if a **begin** JML statement (§11.11) immediately follows the statement specification, then the specification applies to the sequence of statements within the **begin-end** block:

- 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, each other clause must be satisfied in the post-state, that is after execution of the following statement or **begin-end** block.
- The `\result` expression may not be used in any clause

There are a few conceptual differences between the method and statement specifications.

- In the pre- and post-states any local (including ghost) variables that are in scope may be used in the clause expressions.
- As there is no return statement, the `\result` expression may not be used.
- There is no inheritance of specification cases as there might be for method specifications.
- The `returns`, `breaks`, `continues`, and `throws` specification clauses are permitted in a

The motivation for a block specification is that it summarizes the behavior of the subsequent Java statement or 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 statement specification without needing to use the implementation.

[1] 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 §11

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 specification.

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 the begin and end do not introduce a local scope and can be inserted in the code without modifying the Java code per se.

Begin and 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 statement specifications as described in the next subsection

inline_loop statement

Chapter 12

JML Expressions

Grammar:

```
<jml-expression> ::=
        <result-expression>                          §12.12
      | <exception-expression>                        §12.13
      | <informal-expression>                         §12.20
      | <old-expression>                              §12.15
      | <quantified-expression>                       §12.17
      | <nonnullelements-expression>                  §12.18
      | <fresh-expression>                            §12.16
      | <type-expression>                             §12.21
      | <typeof-expression>                           §12.22
      | <elemtype-expression>                         §12.23
      | <invariant-for-expression>                    §12.25
      | <static-invariant-for-expression>             §12.26
      | <is-initialized-expression>                   §12.24
 Missing some - check the list
      | <java-math-expression>                        §??
      | <safe-math-expression>                        §??
      | <bigint-math-expression>                      §??
      | <duration-expression>                         §12.34
      | <working-space-expression>                    §12.35
      | <space-expression>                            §12.36
      | <expression> ( <# | <#= ) <expression>
```

shift bit logical dot cast new methodcall ops

Need sections on \count and \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 four types:

- infix operations that use non-alphanumeric symbols (e.g., `<==>`)
- identifiers that begin with a backslash (e.g., `\result`)
- identifiers that begin with a backslash but have a functional form (e.g., `\old`)
- methods defined in JML whose syntax is Java-like (e.g., `JML.informal(...)`)

The Java-like forms replicate some of the backslash forms. The backslash forms are traditional JML and more concise. However, the preference for new JML syntax is to use the Java-like form since supporting such syntax requires less modification of JML tools.

12.2 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.

12.3 Java operations used in JML

Because of the pure expression rule (cf. §12.2), some Java operators are not permitted in JML expressions:

- allowed: `+ - * / % == != <= >= < > .^ & | && || << >> >>> ?:`
- prohibited: `++ -- = += -= *= /= %= &= |= ^= <<= >>= >>>=`

12.4 Precedence of infix operations

JML infix operators may be mixed with Java operators. The new JML operators have precedences that fit within the usual Java operator precedence order, as shown in Table 12.1.

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)`.

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

12.5 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

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	quantified	
postfix: . [] method calls		left
prefix: unary + - ! ~ cast new		right
* / %		left
binary + -		left
<< >> >>>		left
<= < >= > instanceof	<: <# <#=	chainable
== !=		left
&		left
^		left
|		left
&&		left
||		left
	==> <==	right
	<==> <=!=>	left
?:		right
..		none
assignment, assign-op		right
lowest precedence		

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 JML operators are given in the section describing that operator. The rules for Java operators *used in JML expressions* are given here. They presume that the expressions are type correct. The [[ ]] notation denotes that the enclosed expression is well-defined. In the following e, e_1, etc. are *<expression>*s, & is short-circuiting conjunction, and $\Rightarrow$ is short-circuiting implication.

(literals and names)			**true**
(parenthesis)	$[\![(e)]\!]$	$\equiv$	$[\![e]\!]$
(dot access) f is a field	$[\![e.f]\!]$	$\equiv$	$[\![e]\!]$ & $e \neq null$, where 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 *What about overflow?*
(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 ||)	$[\![e_1 \;\|\|\; e_2]\!]$	$\equiv$	$[\![e_1]\!]$ & $(\neg e_1 \Rightarrow [\![e_2]\!])$
(arithmetic operations)	$[\![e_1 \; op \; e_2]\!]$	$\equiv$	$[\![e_1]\!]$ & $[\![e_2]\!]$, for operators + − * *What about 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 used
- 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 argument, 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 appropriate 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. *Is the result undefined if the RHS is out of range?*

floating point operations?

12.6 Chaining of comparison operators

Grammar:

```
<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[1] 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.

Do <: <# <#= chain?

[1] They are chainable in Dafny, by comparison.

12.7 org.jmlspecs.lang.JML

Say more

12.8 Implies operator: ==>

Grammar:
<expression> ::= *<expression>* **==>** *<expression>*

Well-defined:
$$[\![e_1 \texttt{ ==> } e_2]\!] \equiv [\![e_1]\!] \mathbin{\&} (e_1 \implies [\![e_2]\!])$$

Type information:

- two arguments, each an expression of boolean type
- result is boolean

The ==> operator denotes implication and is a short-circuit operator. It 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.

Do we agree that <== is deprecated

12.9 Equivalence and inequivalence: <==> <=!=>

Grammar:
<expression> ::=
<expression> **<==>** *<expression>*
| *<expression>* **<=!=>** *<expression>*

Well-defined:
$$[\![e_1 \texttt{ <==> } e_2]\!] \equiv [\![e_1]\!] \mathbin{\&} [\![e_2]\!]$$
$$[\![e_1 \texttt{ <=!=> } e_2]\!] \equiv [\![e_1]\!] \mathbin{\&} [\![e_2]\!]$$

Type information:

- two arguments, each an expression of boolean type
- the expression is well-defined if both operands are well-defined
- result is boolean

The <==> operator denotes equivalence: it 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: it 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.

12.10 JML subtype: <:

Is this the time to have both <:= and <:, with the latter being a proper subtype?

Type information:

- two arguments, each of type `\TYPE`
- well-defined iff both operands are well-defined
- result is boolean

The <: operator denotes JML subtyping: the result is true if the left operand is a subtype of the right operand. Note that the argument types are `\TYPE`, that is JML types (cf. §1). *Say more about relationship to Java subtyping*

Note that the operator would be better named <:=, since it is true if the two operands are the same type.

12.11 Lock ordering: <# <#=

Type information:

- two arguments, each of reference type
- well-defined iff both operands are well-defined and both are not null
- 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.

12.12 \result

Grammar:
<result-expression> ::= **\result**

Well-defined:
[[**\result**]] ≡ **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 `ensures`, `duration`, and `working_space` clauses

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`.

12.13 \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 the `signals`, `duration`, and `working_space` clauses

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 ...  ;
```

Must we allow for exception to be null in duration and workingspace clauses; what is the type in duration or workingspace clauses?

12.14 \count (\index)

Grammar:
<count-expression> ::= **\count** | **\index**

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 preferred spelling of this term is `\count`. `\index` will be eventually deprecated. The rationale is that the expression connotes the count of the number of times the loop body has been executed, not the value of a loop index variable (though often those are the same).

12.15 \old, \pre, and \past

Grammar:
<old-expression> ::=

```
        ( \old( <expression> ( , <label> )? )
        ( \pre( <expression> )
        ( \past( <expression> )
<label> ::= <id>
```

Type information: The type of the expression is the type of the first argument. Note though that the expression may be evaluated in different state than the current state and different variable names may be in scope.

Well-definedness: The expression is well-defined if the first argument is well-defined and any label argument names either a built-in label (§11.6) or an in-scope Java or JML ghost label (S11.5).

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 show in the following example.

```
1 public void m(int i) {
2   a: {
3      a: {} // forbidden nested use
4      b: {}
5   }
6   a: {} // permitted subsequent use
7   //@ assert \old(i,a) ==m ... // refers to the most recent use of a
8   //@ assert \old(i,b) ==m ... // Error - b is out of scope
9 }
```

12.15.1 \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. §11.6). 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 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.

```
1 public class Old {
2
3   public boolean k;
4
5   //@ requires k;
6   public void m() {
7
8     //@ assert k; // k is this.k, a boolean
9
```

```
10      int k = 0;
11      //@ assert k >= 0; // k is the local k, an int
12      //@ assert \old(k); // k is this.k, a boolean
13    }
14  }
```

12.15.2 \pre

The \pre expression is simply an abbreviation for \old with the built-in label Pre (cf. §11.6).

12.15.3 \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

12.16 \fresh

Grammar:

<fresh-expression> ::=
    \fresh(*<expression>* [, *<java-identifier>*]**)**

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 (§11.6) or a Java statement label or a JML ghost label (§11.5). If omitted, the built-in label Old is implicit.
- expression type is boolean
- \fresh may be used only in postcondition clauses or statement specifications

Well-definedness: The argument must be well-defined and non-null. The second argument, if present, must be the identifier corresponding to an in-scope label or a built-in label.

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.

12.17 Quantified expressions

Grammar:

```
<quantified-expression> ::=
       <quantifier> <type-name> <java-identifier> ;
       [ [ <expression> ]; ] <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*, $V(x)$.

Well-definedness: The quantified-expression is well-defined iff the two sub-expressions are well-defined. For a quantifier `Q`

$$[[\ \texttt{Q T x; R(x); V(x)}\ ]] \equiv$$
$$(\forall\, T x;\ [[\ \text{R(x)}\ ]]\ )$$
$$\wedge(\forall\, T x;\ R(x)) \implies [[\ \text{V(x)}\ ]]\ )$$

The `\choose` expression has an additional well-definedness condition given below.

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 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.

12.17.1 \forall, \exists

[I reformulated this] 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.

[A brief discussion that this without dispute on primitive data types, but that there several ways of reading this for T a reftype. Although this is relevant for all generalised quantifiers I would mention this here.]

12.17.2 \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 (if one exists). Thus the `\choose` expression is well-defined only if such a value exists. That is,

[[`(\choose T x; R(x); V(x)(`]] $\equiv$
$(\forall\, T\, x;$ [[R(x)]] $)$
$\wedge\, (\forall\, T\, x;\ R(x) \implies$ [[V(x)]] $)$
$\wedge\, (\exists\, T\, x;\ R(x) \wedge V(x)\ )$

The value of a *choose-expression* is any value that satisfies its range and value predicates; its result is non-deterministic 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 = (\choice int k; 1 <= k <= 2);
2 //@ assert x == 1 || x == 2;  // true
3 //@ assert x == 1; // false
```

the first assert is valid, but the second is not. In runtime-checking the second will be reported true or false non-deterministically . [Well ... I would say the first is valid (ie. true for all possible non-deterministic choices, whereas the second one is invalid, ie. not true for all possible nondeterministic choices, but (in this case) true for some of the choicees. x==42 would indeed always be false.]

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))` .

The choose operator implements Hilbert's choice operator ε introduced in XXX. The property last explained is called extensionality. All generalised quantifiers in JML are extensional in the sense that whenever R(x) and V(x) is replaced by a semantically equivalent R' and V', the expression yields the same value.

12.17.3 \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 $V(x)$. Note that `\max` is overloaded with the max-locset expression (§??). *Needs default*
- `\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 $V(x)$. *Needs default*

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) comparison function is supplied

We should leave room to generalize these operations to any commutative-associative binary function, even if we do not add them to JML now.

12.18 \nonnullelements

Grammar:

```
<nonnullelements-expression> ::=
        \nonnullelements ( <expression> )
```

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
- the expression is well-defined iff the argument is well-defined (it may be null)
- 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?

12.19 Arithmetic mode scope

Grammar:

```
<arithmetic-mode-expression> ::=
        <arithmetic-mode-name> ( <expression> )
<arithmetic-mode-name> ::= \java_math | \safe_math | \bigint_math
```

The function-like expression `\java_math`, `safe_math`, and `bigint_math` may be used in Java code or in specifications to change the arithmetic mode (§13) for the enclosed expression. Such arithmetic mode contexts override the default set for the program or the specific method and can be nested.

12.20 informal expression: `(*...*)` and `JML.informal()`

Grammar:

```
<informal-expression> ::=
        (*  .*  *)
      | JML.informal ( <expression> )
```

Well-defined:

$[\![\text{(*} .* \text{*)}]\!] \equiv \textbf{true}$

$[\![\textbf{JML.informal(}e\textbf{)}]\!] \equiv [\![e]\!]$

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()  ...
```

12.21 \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 type, see §1.

Examples: (T is an in-scope type variable)

```
1 //@ ... \bs type(int) ...
2 //@ ... \bs type(Integer) ...
3 //@ ... \bs type(java.lang.Integer) ...
4 //@ ... \bs type(java.util.LinkedList<String>) ...
5 //@ ... \bs type(java.util.LinkedList<String>[]) ...
6 //@ ... \bs type(T) ...
7 //@ ... \bs type(java.util.LinkedList<T>) ...
```

12.22 \typeof

Grammar:

<typeof-expression> ::=
    \typeof (<expression>)

Well-defined:

$[\![\backslash\textbf{typeof}(e)]\!] \equiv [\![e]\!] \ \&\ e \neq null$

Type information:

- one expression argument, of any type
- well-defined iff the argument is well-defined and not null
- 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`).

Verify that primitive types are allowed

Examples:

```
1 Object o = new Integer(5); \\
2 // o has static type Object, but dynamic type Integer
3 //@ assert \bs typeof(o) == \bs type(Integer); // - true
4 //@ assert \bs typeof(o) == \bs type(Object); // - false
5 //@ assert \bs typeof(5) == \bs type(int); // - true
```

12.23 \elemtype

Grammar:

<elemtype-expression> ::=
    \elemtype (<expression>)

Well-defined:

$[\![\backslash\textbf{elemtype}(e)]\!] \equiv [\![e]\!] \ \&\ e \neq null$ & (e has array type)

Type information:

- one argument, of type `\TYPE`
- expression has type `\TYPE`

This operator returns the static element type of an array type.

Examples:

```
1 //@ assert \elemtype(\type(int[])) == \type(int);
2 //@ assert \elemtype(\type(int)) == \type(int); // -- undefined
```

Fix this text. Should we allow array values or only type expressions? Should a non-array value be undefined or yield null?

12.24 \is_initialized

Grammar:

```
<is-initialized-expression> ::=
        \is_initialized ( <type-name> ... )
      | \is_initialized ( )
```

Type information: The argument must be the name of a reference type.

The value of this expression is true iff the class named as the argument has completed its static initialization.

12.25 \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.

Extensions: multiple arguments as sugar for the conjunction of multiple instances of `invariant_for`

12.26 \static_invariant_for

Grammar:

```
<static-invariant-for-expression> ::=
```

```
\static_invariant_for ( <type-name> )
```

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. If the type is a generic type, it must be fully parameterized. 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 being 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?

Extensions: - multiple arguments as sugar for the conjunction of multiple instances of `static_invariant_for`

12.27 \not_modified

Grammar:

```
<not-modified-expression> ::=
       \not_modified ( <expression> ... )
      | \not_modified ( )
```

Type information:

- Strict JML: one argument, an expression of any type other than void
- Extension: any number of arguments, each expression of any type other than void
- well-defined iff the arguments are well-defined
- result type is `boolean`

A `\not_modified` expression is a two-state expression that may occur only in post-condition 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.

The RM says the argument is a store-ref list, rather than an expression. Which do we want? A store-ref-list allows constructions such as o. or a[*] or a[1..6] but not a+b.*

12.28 \not_assigned

Grammar:

```
<not-assigned-expression> ::=
      \not_assigned ( <store-ref-expression> ... )
```

Type information: Each argument must be properly typed. The expression has boolean type.

Well-definedness: The expression is well-defined iff each argument is well-defined.

This expression may be used only in postconditions or in statement specifications. 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 statement specification (e.g., an `assert` statement), the expression is true if none of the arguments have been assigned to since the beginning of the innermost block contract, or of the method body if there is no enclosing block contract, and up to the position of the containing specification statement.

12.29 \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> ... )
```

Type information: Each argument must be properly typed. The expression has boolean type.

Well-definedness: The expression is well-defined iff each argument is well-defined.

The argument list of this expression denotes a locset, as described in §1.

This expression may be used only in postconditions or in statement specifications. 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 statement specification (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 expression.

12.30 \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 §8.5.8).

The argument list of this expression denotes a set of methods, as described in §8.5.8. If there are no arguments, the set of methods is empty.

This expression may be used only in postconditions or in statement specifications. 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 statement specification (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.

12.31 \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 of the `\max` expression is `Object`.

Well-definedness:

[[`\lockset`]] ≡ *true*
[[`\max`(*<expression>*)]] ≡
    [[*<expression>*]] ∧ *<expression>* ≠ `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.

Issue: We don't currently have a built-in set type

12.32 \reach

Text needed

12.33 Set comprehension

Text needed

12.34 \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 `long`

Here we say that the argument is an expression, whereas the DRM says it must be an explicit method or constructor call.

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 specification.

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.

Say more about what a virtual machine cycle is.

What about runtime assertion checking

12.35 \working_space

Grammar:
<working-space-expression> ::=
\working_space (<expression>)

Type information:

- one argument, of any type, including void
- well-defined iff the argument is well-defined
- expression has type `long`

Here we allow the argument to be any expression. The DRM requires the argument to be a method or constructor call.

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.

12.36 \space

Grammar:
<space-expression> ::=
\space (<expression>)

Type information:

- one argument, of any reference type
-
- expression has type `\bigint`

Well-definedness:
[[`\space(`<*expression*>`)`]] ≡ [[<*expression*>]]

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.

What about padding for alignment

12.37 Store-ref expressions

Grammar:
<store-ref-expression> : : =
<non-wild-store-ref-expression>
| *<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 array-range store-ref)
| *<non-wild-store-ref-expression>* [*]
(a wild-array store-ref)

Type information: For store-ref expression o and a type-name T,

- A *<store-ref-expression>* does not have a type, but a *<non-wild-store-ref-expression>* does have a type
- In $o.f$, f must name a field of the type of o, which must have reference type; the type of the expression is the type of the field f.
- In $o.*$, o must be a reference type
- In $T.f$, T must name a reference type, f must name a static field of T; the type of the expression is the type of the field f.
- 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`, and the type of the expression is the element type of the array
- In $o[i..j]$, o must be an array type, i and j must be (convertible to) `\bigint`, and the type of the expression is the element type of the array
- In $o[*]$, o must be an array type, and the type of the expression is the element type of the array

Well-definedness:
$[[o.f]] \equiv [[o]] \wedge o \neq \texttt{null}$
$[[o.*]] \equiv [[o]] \wedge o \neq \texttt{null}$
$[[T.f]] \equiv true$
$[[T.*]] \equiv true$
$[[a[i]]] \equiv [[a]] \wedge a \neq \texttt{null} \wedge 0 \leq i < a.length$
$[[a[i..j]]] \equiv [[a]] \wedge a \neq \texttt{null} \wedge 0 \leq i \wedge j < a.length$
$[[a[*]]] \equiv [[a]] \wedge a \neq \texttt{null}$

A store-ref expression denotes a set of memory locations, that is a `\locset` (§1).

- 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
- A static wild-field store-ref (`T.*` denotes the set of locations of all the static fields of the given type
- 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
- A wild-array store-ref ($a[*]$) denotes the locations of all of the array elements of the given array

Note that the grammar does not allow constructions like $o.*.*$, but it does allow $a[*].f$ and $a[*][*]$.

Questions: Does T. include supertype fields? Does o.* include static fields? fields of the dynamic type? fields of super types?*

Include `\till` as a range

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 [14] 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.

Chalin proposed that most of the time, programmers would like Safe mode semantics for programming language operations and Math mode for specification expressions.

Static checking can reason about these types using usual logics with arithmetic; runtime checking uses `java.math.BigInteger` to represent `\bigint`.

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 mode 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.

Change the annotations to be simply @CodeMath and @SpecMath with a value?

The arithmetic mode affects the semantics of these operators:

- arithmetic: unary plus, unary minus, and binary `+ - * / %`
- shift operations: `<< >> >>>`
- cast operation
- *Math functions ???*

The semantics of these operations in each mode are described in the following sections.

Say more about the explicit semantics

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
- `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.

[1] https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.17.3

- 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 produces a divide-by-zero error

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 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 or square roots of negative numbers) 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) is false and the equals operation on `Double` and `Float` values is different than that == 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`.

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

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

Describe the intent, syntax and semantics of model programs

Chapter 17

Specification `.jml` files

Agreed that jml files completely supplant Java files if both are present for a given .top-level class. In that case any jml annotation comments in the java file are completely ignored – they need not even be parseable.

Resolved?: MU: I think this section should go into a later chapter, it is too technical here. We mention all JML artifacts before actually having introduced them. Resolved?: GL: Yes, maybe it should be in a chapter on lexical analysis. Resolved?: DRC: Moved it here - is this OK

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]

[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 no obsolete and no longer supported; it was complex and not used.

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-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 (up to renaming), 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 and annotations.
- A Java method of a class or interface need not have a JML declaration (in which case various default specifications might apply).

Field declarations

- Fields declared in a non-model JML 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.
- 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 JML initializer clauses.

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. §10).

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, modifiers and 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 Observable 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 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.

Fix up page break Review for correctness and completeness.

Missing: non_null_by_default

Add in secret, query

inline; check for others

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.

CHeck the table; add section references; add where allowed; indicate which are type modifiers; turn headings 90 degrees.

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

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 strictfp`	`spec_public spec_protected model ghost pure instance helper non_null nullable nullable_by_default monitored uninitialized final`
Class declaration	`public final abstract strictfp`	`pure model nullable_by_default spec_public spec_protected`
Interface declaration	`public strictfp`	`pure model nullable_by_default spec_public spec_protected`
Nested Class declaration	`public protected private static final abstract strictfp`	`spec_public spec_protected model pure`
Nested interface declaration	`public protected private static strictfp`	`spec_public spec_protected model pure`
Local Class (and local model class) declaration	`final abstract strictfp`	`pure model`

Grammatical construct	Java modifiers	JML modifiers
Type specification (e.g. invariant)	`public protected private static`	`instance`
Field declaration	`public protected private final volatile transient static`	`spec_public spec_protected non_null nullable instance monitored final`
Ghost Field declaration	`public protected private static final`	`non_null nullable instance monitored`
Model Field declaration	`public protected private static`	`non_null nullable instance`
Method declaration in a class	`public protected private abstract final static synchronized native strictfp final`	`spec_public spec_protected pure non_null nullable helper extract`
Method declaration in an interface	`public abstract`	`spec_public spec_protected pure non_null nullable helper`
Constructor declaration	`public protected private`	`spec_public spec_protected helper pure extract`
Model method (in a class or interface)	`public protected private abstract static final synchronized strictfp`	`pure non_null nullable helper extract`
Model constructor	`public protected private`	`pure helper extract`
Java initialization block	`static`	-
JML `initializer` and `static_initializer` annotation	-	-
Formal parameter	`final`	`non_null nullable`
Local variable and local ghost variable declaration	`final`	`ghost non_null nullable uninitialized`

Appendix B

Deprecated and Replaced Syntax

A. Deprecated and Replaced Syntax

The subsections 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. 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.

B.1 Deprecated Syntax

The following syntax is deprecated. Note that it might be supported with a deprecation warning by some tools (e.g., JML2) but not by newer tools.

B.1.1 Deprecated Annotation Markers

The following lexical syntax for annotation markers is deprecated.

<annotation-marker> `::=`

```
    //+@ [ @ ] ...
  | /*+@ [ @ ] ...
  | //-@ [ @ ] ...
  | /*-@ [ @ ] ...
```

B.1.2 Deprecated Represents Clause Syntax

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 section 8.4 Represents Clauses, for the supported syntax.

B.1.3 Deprecated `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 §1 for the supported syntax.

B.1.4 Deprecated File Name Suffixes

The set of file name suffixes supported by JML tools is being simplified. In the future, especially in new tools the suffixes 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 into files with the suffixes '.java' and '.jml'. See §1 for details on the use of file names with JML tools.

B.1.5 Deprecated `weakly` modifier

The `weakly` modifier is not longer supported.

B.1.6 Deprecated `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 §1 for details.

B.1.7 Deprecated reverse-implication (<==) token

The <== token and the reverse-implication expression are deprecated. It was rarely used and a bit confusing.

B.1.8 Deprecated `\not_specified` token

The `\not_specified` token used as an alternative to a predicate in many clauses is deprecated.

B.1.9 Deprecated `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.

B.1.10 Deprecated `hence_by`

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.

B.1.11 Deprecated `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 (§8.5.5) initialized with a `\choose` expression (§12.17.2), as in

```
old T e = (\choose T e; true);
```

B.1.12 Deprecated 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 deprecated.

B.1.13 Deprecated `\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 the language.

B.1.14 Deprecated 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.

B.2 Replaced Syntax

The +-style of JML annotations, that is, JML annotations beginning with `//+@` or `/*+@`, has been replaced by the annotation-key feature described in §1.

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`.

Similarly, the depends clause has been replaced by the mechanism of data groups and the 'in' and 'maps' clauses of variable declarations.

Appendix C

Grammar Summary

Automatic collection of all of the grammar productions listed elsewhere in the document

Appendix D

Type Checking Summary

This was in the DRM outline - is there something to be put in here? If it is to be collected from the rest of the document, we need to place markers to identify the relevant stuff.

Appendix E

Verification Logic Summary

This was in the DRM outline. What was its intent? Is it the same as a section on semantics and translation?

Appendix F

Differences in JML among tools

Some material is in the DRM. Needs to be enhanced. SHould have a detailed comparison with ACSL, for example – see the appendix of the ACSL documentation.

Appendix G

TODO

Be sure to talk about

- 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

Appendix H

Statement translations

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

H.1 While loop

Java and JML statement:

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

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

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

Appendix I

Java expression translations

I.1 Implicit or explicit arithmetic conversions

TODO

I.2 Arithmetic expressions

TODO: need arithmetic range assertions

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

stats(tmp, − **a** *)* ==>
    stats(tmpa, **a** *)*
    T tmp = − *tmpa* **;**

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

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

stats(tmp, **a** ⋆ **b** *)* ==>
    stats(tmpa, **a** *)*

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

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

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

I.3 Bit-shift expressions

TODO

I.4 Relational expressions

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

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

I.5 Logical expressions

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

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

stats(tmp, **a** `&&` **b** *)* ==>

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

stats(tmp, **a** `||` **b** *)* ==>

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

Bibliography

[1] https://docs.oracle.com/javase/specs/jls/se21/html/jls-3.html. 39, 49

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification – The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*. Springer-Verlag, 2016. 2

[3] Krzystof R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981. 14

[4] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, August 1988. 9

[5] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. Technical Report Ser. A, No 92, Abo Akademi University, Department of Computer Science, Lemminkäinengatan 14, 20520 Abo, Finland, 1989. Appears in *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*, Mook, The Netherlands, May/June 1989, Spring-Verlag, LNCS 430, J. W. de Bakker, et al, (eds.), pages 42–66. 9

[6] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 1998. 9

[7] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison Wesley Longman, Inc., Reading, 1997. 2

[8] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, New York, NY, 2005. Springer-Verlag. 2

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`. 53

[10] Patrick Baudin, Pascal Cuoq, Jean-Christophe FilliÃ¢tre, Claude MarchÃ©, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACLS: ANSI/ISO C Specification Language*. CEA LIST and INRIA, Sacly, France, version 1.13 edition, 2018. `https://frama-c.com/download/acsl.pdf`. 2

[11] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2007. 8

[12] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995. 3

[13] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003. 6, 7, 8

[14] Patrice Chalin. JML support for primitive arbitrary precision numeric types: Definition and semantics. *Journal of Object Technology*, 3(6):57–79, June 2004. 147

[15] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April 2003. The author's Ph.D. dissertation. 8

[16] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002. 8

[17] Yoonsik Cheon and Gary T. Leavens. A contextual interpretation of undefinedness for runtime assertion checking. In *AADEBUG 2005, Proceedings of the Sixth International Symposium on Automated and Analysis-Driven Debugging, Monterey, California, September 19–21, 2005*, pages 149–157, New York, NY, September 2005. ACM Press. 8

[18] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6):583–599, May 2005. 3

[19] David Cok. OpenJML: JML for Java 7 by extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA For-*

mal Methods, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer-Verlag, Berlin, 2011. 2, 8

[20] David R. Cok, 2018. http://www.openjml.org. 8

[21] David R. Cok. JML and OpenJML for Java 16. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, FTfJP 2021, page 65â€“67, New York, NY, USA, 2021. Association for Computing Machinery. 8

[22] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-R0413. 2

[23] David R. Cok and Serdar Tasiran. Practical methods for reasoning about java 8's functional programming features. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments*, pages 267–278, Cham, 2018. Springer International Publishing. 8

[24] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976. 92

[25] Michael Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001. 8

[26] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools in Software Development*. Cambridge, Cambridge, UK, 1998. 9

[27] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993. 2, 7, 9, 10

[28] John V. Guttag, James J. Horning, and Jeannette M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985. 2

[29] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990. 7

[30] I. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, Inc., London, second edition, 1993. 3, 9

[31] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969. 3, 9, 92

[32] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972. 9

[33] Marieke Huisman. *Reasoning about Java Programs in higher order logic with PVS and Isabelle*. Ipa dissertation series, 2001-03, University of Nijmegen, Holland, February 2001. 7, 8

[34] IEEE Standards Committee 754. *IEEE Standard for binary floating-point arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in ACM SIGPLAN Notices, 22(2):9-25, 1987. 57

[35] Bart Jacobs and Eric Poll. A logic for the Java modeling language JML. In *Fundamental Approaches to Software Engineering (FASE'2001), Genova, Italy, 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, 2001. 7

[36] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes (preliminary report). In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 329–340. ACM, October 1998. 8

[37] Cliff B. Jones. *Systematic software development using VDM*. International Series in Computer Science. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1986. 2

[38] Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990. 9

[39] Joan Krone, William F. Ogden, and Murali Sitaraman. Modular verification of performance constraints. Technical Report RSRG-03-04, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003. 94, 95

[40] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989. 2

[41] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011. 2

[42] Gary T. Leavens. Larch/C++ Reference Manual. Version 5.14. Available in `ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz` or on the World Wide Web at the URL `http://www.cs.iastate.edu/~leavens/larchc++.html`, October 1997. 2

[43] Gary T. Leavens. Larch frequently asked questions. Version 1.110. Available in `http://www.cs.iastate.edu/~leavens/larch-faq.html`, May 2000. 9

[44] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999. 2

[45] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, March 2006. 7, 11

[46] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual. Available from `http://www.jmlspecs.org`, September 2009. i, 8, 9, 10

[47] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03. 2

[48] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153, New York, NY, October 1998. ACM. 2

[49] K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 112–126, Berlin, 2010. Springer-Verlag. 2

[50] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, October 2000. 2, 82

[51] Richard Allen Lerner. Specifying objects of concurrent systems. Ph.D. Thesis CMU-CS-91-131, School of Computer Science, Carnegie Mellon University, May 1991. 93

[52] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986. 9

[53] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, October 1992. 3, 9, 10

[54] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992. 2, 9, 10

[55] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997. 2, 9, 10

[56] Bertrand Meyer, Alisa Arkadova, and Alexander Kogtenkov. The concept of class invariant in object-oriented programming. *Form. Asp. Comput.*, jan 2024. Just Accepted. 28

[57] Carroll Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994. 9

[58] Carroll Morgan and Trevor Vickers, editors. *On the refinement calculus*. Formal approaches of computing and information technology series. Springer-Verlag, New York, NY, 1994. 9

[59] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, February 2003. 76

[60] International Standards Organization. Information technology – programming languages, their environments and system software interfaces – Vienna Development Method – specification language – part 1: Base language. ISO/IEC 13817-1, December 1996. 9

[61] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. 9

[62] Henrique Rebêlo, Gary T. Leavens, Mehdi Bagherzadeh, Hridesh Rajan, Ricardo Lima, Daniel M. Zimmerman, Márcio Cornélio, and Thomas Thüm. Modularizing crosscutting contracts with aspectjml. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 21–24, New York, NY, USA, 2014. ACM. 8

[63] Henrique Rebêlo, Gary T. Leavens, and Ricardo Massa Lima. Client-aware checking and information hiding in interface specifications with JML/Ajmlc. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, pages 11–12, New York, NY, USA, 2013. ACM. 8

[64] Henrique Rebêlo, Ricardo Lima, Márcio Cornélio, Gary T. Leavens, Alexandre Mota, and César Oliveira. Optimizing jml feature compilation in ajmlc using aspect-oriented refactorings. In *XIII Brazilian Symposium on Programming Languages (SBLP)*, pages 117–130. Brazilian Computer Society, August 2009. 8

[65] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In Andrew P. Black, editor, *ECOOP 2005 — Object-Oriented Programming 19th European Conference, Glasgow, UK*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576. Springer-Verlag, Berlin, July 2005. 6, 82, 93

[66] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995. 2

[67] Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 208–228, New York, NY, October 2000. ACM. 7

[68] Clyde Dwain Ruby. Modular subclass verification: safely creating correct subclasses without superclass code. Technical Report 06-34, Iowa State University, Department of Computer Science, December 2006. 7

[69] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, NY, second edition, 1992. 3, 9

[70] Alan Wills. Specification in Fresco. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992. 3

[71] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987. 9

[72] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, September 1990. 2, 9

Index