Java Modeling Language (JML) Reference Manual

2nd edition

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

DRAFT December 6, 2021

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

Preface

This document gives the definition of the Java Modeling Language (JML), a language in which to write behavioral specifications for Java programs. Since about 2000 or so, JML has been the most widely known specification language for writing behavioral interface specifications for Java programs. It was first a vehicle for discussing theoretical and soundness issues in specification and verification of object-oriented software. It then also became the language to use 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 [40] should be rewritten and expanded to represent the current state of JML. In the process, many outstanding semantic and syntactic issues have been either resolved or clarified. This document is the result of that collaborative effort. Consequently this document is a completely revised, rewritten and expanded reference manual for JML, while borrowing (and copying) much 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` — and 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`.

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

- interaction with JSR 308

- various @NonNull annotations in different packages

- 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

Chapter 1

Introduction

JML is a *behavioral interface specification language* (BISL) that builds on the Larch approach [24] [25] and Eiffel [47] [48] (and other languages such as APP [57]). In this style of specification, which might be called model-oriented [63], one specifies both the interface of a method or abstract data type and its behavior [34]. In particular JML builds on the work done by Leavens and others in Larch/C++ [38] [35] [36]. (Indeed, large parts of this manual are adapted wholesale from the Larch/C++ reference manual [36].) Much of JML's design was heavily influenced by the work of Leino and his collaborators [41] [42] [44], then subsequently by Cok's work on ESC/Java2 [20] and OpenJML [17], the work on the KeY tool [9], and by work on other specification languages such as Spec# [6], ACSL [8], SPARK [5], and Dafny [43]. 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 or type describes a set of state transformations that it can perform. A behavior of a method is specified by describing

- a set of states in which calling the method is defined,
- a set of locations that the method is allowed to assign to (and hence may change), and
- the relations between the calling state and the states in which it either: (a) returns normally, (b) throws an exception, or (c) does not return to the caller.

The states for which calling the method is defined are formally described by a logical assertion, called the method's *precondition*. The allowed relationships between these states and the states that may result from normal return are formally described by another logical assertion called the method's *normal postcondition*. Similarly the relationships between these pre-states and the states that may result from throwing an exception are described by the method's *exceptional postcondition*. The states for which the method need not return to the caller are described by the method's *divergence condition*; however, explicit specification of divergence is rarely used in JML. The set of locations the method is allowed to assign to is described by the method's *frame condition* [10].

The behavior of an abstract data type (ADT) that is implemented by a class in Java is specified by describing a set of abstract fields for its objects and by specifying the behavior of its methods (as described above). The abstract fields for an object can be specified either by using JML's model and ghost fields [16], 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 [27] [60] or Fresco [61].

1.2 A First Example

As a first example, consider the JML specification of the 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 comments that have an at-sign (`@`) as their first character following the usual comment

```
 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.

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 `@*/`. In such JML annotations, as in lines 12–14, at-signs at the beginnings of lines are ignored by JML. The other form of such annotations can be seen on lines 9, which is a JML annotatation 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 an annotation. (See section 4. Lexical Conventions, for more details about 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`. (See section ? Spec Public, for more details.) That is, as far as the JML specifications are concerned (but not for Java code), `count` can be used an if it were declared to be **`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 section ? Invariants, for more about 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 section 9. Method Specifications, for more details about method specifications).

The specification of the constructor `Counter` is given on lines 12–13. The constructor's precondition is the assertion 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 axiom 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.

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 assertions, 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 modules. As it is a behavioral interface specification language, JML specifies how to use such Java program modules from *within* a Java program; hence JML is *not* 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 modules?

The two main benefits in using JML are:

- the precise, unambiguous description of the behavior of Java program modules (i.e., classes and interfaces), and documentation of Java code,
- the possibility of tool support [11].

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 [56], the current version of JML does not have features that help 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 program modules.

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 the parameter mechanisms, whether the method is **`final`**, **`protected`**, etc.; if one used a specification language such as 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 doesn't have 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 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 [26] and Chapter 7 of [24]). 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 [30, 31, 59].
- JML specifications can be used by several tools that can help debug and improve the code [11].

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 "subclassing contract" allows programmers to implement derived classes of such a base class without looking at its code [58, 59].

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" [39] 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 all 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) [19, 21, 17, 18], or the KeY tool [9].[1] Other tools for JML [11] include Daikon [22], which can infer some JML specifications from execution traces during testing, and the runtime assertion checker (RAC) of OpenJML [18] the RAC found in AspectJML [53][2] and recording of dynamically obtained invariants (as in Daikon), runtime assertion checking (as in JML's runtime assertion checker, jmlc [13, 12]), unit testing [14], and documentation (as in JML's jmldoc tool). The paper by Burdy et al. [11] 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 [40], which has been evolving over many years and has many contributors. This current edition of the reference manual is largely a rewrite of the previous draft. Some sections, particularly introductory and overview material, are taken nearly verbatim from the draft

[1]There have been other formal verification tools for JML, including the LOOP tool [30, 32].

[2]AspectJML is a further evolution of a previous RAC called ajmlc [55, 54]. There was also a RAC tool from Iowa State, called jmlc [12, 13, 15], that is no longer maintained.

manual. However, the current version also incorporates the experience of building tools for JML by the OpenJML and KeY developers, many decisions about new features or deprecated features made at JML workshops, and discussions about JML on the JML mailing lists. 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 17.

1.6 Historical Precedents and Antecedents

JML combines ideas from Eiffel [46] [47] [48] with ideas from model-based specification languages such as VDM [33] and the Larch family [24] [37] [62] [63]. It also adds some ideas from the refinement calculus [2] [3] [4] [50] [49]. 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 [28]. Later Hoare adapted these axiomatic techniques to the specification and correctness proofs of abstract data types [29]. 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 [45] [48] [46] [52]. 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 [33], VDM-SL [23] [51], Z [27] [60], and the Larch family [24]. 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 [24]), but these are expected to be extended as needed. (The advantage of being able to extend the mathematical vocabulary is similar to one advantage of object-oriented programming: one can use a vocabulary that is close to the way one thinks about a problem.)

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 [46] [47] [48]. Eiffel is a programming language with built-in specification constructs. It features pre- and postconditions, although it has no direct support for frame axioms. Programmers like Eiffel because they can easily read the assertions, which 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 Larch/C++.

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.

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 [40] and discussions by the JML community at various JML workshops [**?**].

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.

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.

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.

Thanks to Leo Freitas, Robin Greene, Jesus Ravelo, and Faraz Hussain for comments and questions on earlier versions of this document. Thanks to the many who have worked on the JML checker used to check the specifications in this document. Leavens thanks Iowa State University and its computer science department for helping foster and support the initial work on JML.

See the “Preliminary Design of JML” [39] for more acknowledgments relating to the earlier history, design, and implementation of JML.

Chapter 2

Structure of this Manual

Describe overall organization

2.1 Typographical conventions

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

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

```
Boxed examples
```

comments on how grammars are written; reference to Java grammar

2.2 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 typewriter font: **forall ( ) .**
- 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: `*` `+` `?`
- a post-fixed ... indicates a comma-separated list of 0 or more elements: <*expression*> `...`
- 1-or-more comma-separated elements is written as <*expression*> `(` `,` <*expression*> `)` `*`
- a production has the form: <*non-terminal*> `::=` ...

Chapter 3

JML concepts

This chapter describes some general design principles and concepts of the Java Modeling Language. We also define several important concepts that are used throughout this manual and discuss the overall way that specifications are processed and used.

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. *Reference for this claim?*

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; 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 §16.

3.2 Specification inheritance

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

For example, suppose method `m` in derived class `C` overrides method `m` in parent class

`P`. In a context where we call method `m` on an object `o` with static type `P`, we will expect the specifications for `P.m` to be obeyed. However, `o` may have dynamic type `C`. Thus `C.m`, the method actually executed by the call `o.m()`, must obey all the specifications of `P.m`. `C.m` may have additional specifications, that is, additional behaviors, constraining its behavior further, but it may not relax any of the specifications given for `P.m`.

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

I think the specification marked code applies also to derived classes that do not override the parent class implementation

3.3 JML modifiers and Java annotations

The Java Modeling Language was defined prior to the introduction of annotations in Java. Some, 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.

3.3.1 Modifiers

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

Each such modifier has an equivalent Java annotation. For example

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

can be written equivalently as

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

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

```
import org.jmlspecs.annotation.Pure;
@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, and in runtime checking, a compiled library of he annotation must be available at runtime. 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 §G.4.5.

3.3.2 Type modifiers

Some modifiers are actually type modifiers. In particular `non_null` and `nullable` are in this category. Thus the description of the previous subsection (§G.2) apply to these as well.

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

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

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

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

3.4 Model and Ghost

To be written

3.5 Visibility

To be written - note material written in Method Specifications section

3.6 Evaluation and well-formedness of JML expressions

To be written - note material written in Expressions chapter

3.7 Null and non-null references

To be written

Discuss defaults for binary classes; also default specification

Nonnullbydfault is an extension?

3.8 Static and Instance

To be written

3.9 Observable purity

To be written - perhaps this is a separate chapter It might be early days to put this into the standard.

3.10 Location sets and Dynamic Frames

To be written - see section in DRM on Data Groups

3.11 Arithmetic modes

To be written - see later chapter - where shall we put this discussion

3.12 Immutable types and functions

To be written These are the abstract data types I take it.

3.13 Race condition detection

To be written - see later chapter - where shall we put this discussion Should that be part of the standard?

3.14 Redundant specifications

To be written

3.15 Controlling warnings

To be written - nowarn specifications - perhaps in the Syntax chapter; comment on possibility of unsoundness

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[1]) these elements:

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

More to write here

3.17 Interaction with other tools

3.17.1 Interaction with Type Annotations in Java 1.8

To be written

3.17.2 Interaction with the Checker framework

To be written

3.17.3 Interaction with FindBugs

To be written

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.

This standardization document describes all of JML. However, some portions of the language are considered *Core*, whereas others might be conveniences, are rarely used or applicable only in less common situations. This leaves it possible that some tools might only implement the Core.

[1]Tools implementing JML may add additional methods.

The following table identifies the Core functionality and indicates which tools implement which items.

Also some syntactical redundancy features are not part of the core.

The entries in the table have these meanings:

- **Core**– an JML construct in the Core
- 1 – part of level 1 above Core
- **Ext**- an extension to JML (not defined as standard)
- **Dep**- deprecated features of JML
- **—** not supported by tool
- **– —** not supported by tool
- **+**– supported by tool
- **ESC**– supported by tool for static checking only (not runtime)
- **RAC**– supported by tool for runtime checking only (not static)
- **++**- supported by tool

This table is being edited and is not correct

keyword	**Core**	**KeY**	**OpenJML**	**Comments**
`<==`	1	+	+	
`<==>`	**Core**	++	+	
`==>`	**Core**	++	+	
`//@`	**Core**	++	+	
`/*@ @*/`	**Core**	++	+	
`(* *)`	1	– –	+	MU: If we find a good semantics
`accessible`	+	+	–	
`also`	+	++	+	
`assert`	+	++	+	
`assignable`	+	++	+	KeY: also for loops!
`assume`	+	+	+	
`axiom`	+	+	+	
`behavior`	+	++	+	MU: BrE is syntactic sugar
`\\bigint`	+	++	+	
`code`	+	–	+	
`constraint`	+	+	+	
`\count`	+	–	+	
`decreases`	+	++	+	
`diverges`	+	++	+	

keyword	Core	KeY	OpenJML	Comments
`\dl_`	–	–		MU: or some other means of tool-spec exts.
`\elemtype`	+	+		
`ensures`	+	++		
`\everything`	+	+		
`exceptional_behavior`	+	++		
`\exists`	+	++		
`\forall`	+	++		
`\fresh`	+	++		
`ghost`	+	++		
`helper`	+	++		
`\\index`	+	++		the new addition discussed in Bad Her.
`initially`	+	?		
`instance`	+	– –		
`\invariant_for`	+	++		
`\locset`	+	++		builtin datatype
`loop_invariant`	+	++		`maintains` might be more systematic
`\max`	+	++		semantics interesting
`measured_by`	++		generalised version with list of items	
`\min`	+	++		
`model`	+	++		fields, methods
`model`	–	–		classes
`model`	–	– –		import statements
`non_null`	+	++		
`\nonnullelements`	+	+		
`normal_behavior`	+	++		
`no_state`	–	+		
`\nothing`	+	++		
`nullable`	+	++		
`nullable_by_default`	+	++		
`\num_of`	+	+		
`\old`	+	++		w/o label
`\old`	–	+		w/ label
`private`	+	++		for invariants and contracts
`\product`	+	++		
`protected`	+	–		for invariants and contracts. Meaning must be clarified.
`pure`	+	++		@Pure is syntactic sugar
`\real`	+	++		

keyword	Core	KeY	OpenJML	Comments
`represents`	+	++		
`requires`	+	++		
`\result`	+	++		
`signals`	+	++		
`signals_only`	+	++		
`spec_bigint_math`	+	+		
`spec_java_math`	+	+		
`spec_protected`	+	++		
`spec_public`	+	++		
`spec_safe_math`	+	–		
`\static_invariant_for`	–	++		
`\strictly_nothing`	**Ext**	–	+	
`strictly_pure`	**Ext**	–	+	
`\sum`	1	++		
`two_state`	**Ext**	+	–	
`\type`	1	–	+	
`\TYPE`	1	–	+	
`\typeof`	**Core**	–	+	
`\values`	**Core**	+	+	
block contracts	1	++	+	
annotations instead of modifiers	1	–	+	I would not add them to the core, but explain them only in an appendix on syntactical variations. It should be added to the core when annotations are an alternative for all specifications
`//` comments in specs	1	++	+	
`//` JML in Javadoc	**Dep**	–	–	
`{| ... |}`	–		not widely used, is it?	

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`$(\cdot,\cdot)$
- `\subset`$(\cdot,\cdot)$
- `\disjoint`$(\ldots)$ pairwise disjointness
- `(\collect ...; ...; ...)` a variable binder in the sense of

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

  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`$(\cdot)$ with the meaning

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

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

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 §16. The latter are quite similar to the former, 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, and ill-formed floating-point literals.

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 text of JML annotations 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

```
public
//@ invariant a != null;
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

  `/[/|*]([+|-]<identifier>)*@+`

  is a JML annotation comment if

  - (a) there are no <*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

```
//@ requires a == b;

/*@@@ requires true;
      ensures a == b;
  @@@*/
}
```

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:

```
//+RAC@ requires true;
//+RAC-DEBUG@ requires true;
/*+OPENJML@@@ requires true; @@@*/
//-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:

```
//-RAC@ requires true;
//+OPENJML-RAC@ requires true;
//+DEBUG@ requires true;
//+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
- `KEY` — reserved for use by the KeY tool and presumed to be defined when that tool is used

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, the identifiers are all alphabetic after the backslash).

- JML defines additional operators: `..`, `==>`, `<==>`, `<=!=>`, `<:`, `<:=`, `<#`, and `<#=` .
- 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.

In addition, 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`).

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 `*/` that 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

```
/*@
  @ requires true; // precondition
  @ writes a; // frame condition
  @ ensures a > 0; // postcondition
  @*/
```

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

```
/*@
  @ requires true; // precondition
  @ //-RAC@ writes a; // frame condition,
  @                   // ignored during RAC
  @ ensures a > 0; // postcondition
  @*/
```

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

```
//@ requires a
//@         && b
//@         && 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 two issues that can arise with the syntactical design of JML. First, 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.

Second, 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 (§**??**) — 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 (§**??**) – these are placed with Java top-level declarations, such as `import` statements or model class declarations
- type specification clauses (§**??**) — these are placed where Java places members of types, such as field and method declarations
- method specifications (§**??**) — these are placed in conjunction with the declaration of a method's signature. They in turn consist of
  - keywords
  - punctuation
  - clauses
- field specifications (§**??**) — these are placed in conjunction with a field declaration
- statement specifications (§**??**) — 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. They all occur at the beginning of a grammatical unit (as defined in the previous section). 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 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 variable 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.

[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

- There is a situation that is unavoidable. A Java class `Parent` may contain a declaration of a JML name `n` that is appropriately distinct from any potentially conflicting name in `Parent`. However, unknown to the specifier of `Parent`, a class `Child` can later be derived from `Parent` and the (Java) author of `Child`, not knowing about the JML specifications of `Parent`, may declare a name `n` in `Child`. In such a case. with a local Java entity and an inherited JML entity having the same name, what does the name refer to? In Java code, the name refers of course to the (one) Java declaration. In JML code the ambiguity is resolved in favor of the Java name. In this case the JML entity could be referred to in JML code within `Child` as `super.n`.

Need a more general solution to the name resolution problem of the last bullet above. Do we need a pseudo operator something like `\jml(n)` to force resolution as a JML name? Or do we need a syntax to write JML identifiers that cannot be Java identifiers, such as `‘n` or a convention to write JML identifiers that are unlikely to be Java identifiers, such as `$$n`? Or perhaps syntax like `super.Parent.n` to limit the name resolution to the `Parent` class, where it will find the one JML declaration of `n`? The last is perhaps the most future-proof and Java-like.

4.4 JML Lexical Grammar

In the following grammar, the lexical syntax is defined using regular expressions, using the standard symbols: parentheses for grouping, brackets for a choice of one character, `?`, `*`, `+` for 0 or 1, 0 or more and 1 or more repetitions. An identifier within 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 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>
      <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> ::=
       <identifier>
       |<reserved-word>
       |<literal>
       |<operator>
       |<separator>
       |<white-space>
       |<simple-jml-comment>
       |<plain-java-comment>
       |<line-terminator>
```

<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

Chapter 5

JML Types

In order 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 (§12) 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 `.equals` and `==`.

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 [7]).

Location set type? Object set type? Built-in collection types

5.1 `\bigint`

The `\bigint` type is the set of mathematical integers (i.e., $\mathbb{Z}$). Just as Java primitive integral types are implicitly converted (see *numeric promotion* in the JLS, Ch. 5) to `int` or `long`, all Java primitive integral types implicitly convert to `\bigint` where needed. When `\bigint` values need to be auto-boxed into an Object, they are boxed as `java.math.BigInteger` values; similarly when JML specifications are compiled for runtime checking, `\bigint` values are represented as `java.math.BigInteger` values. Within JML specifications, however, 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 and binary + and −, `*`, `/`, `%`, comparison operators, ==, and `!=`. Also, these types can be used in quantified expressions and variables of these types can be declared as ghost or model variables.

5.2 `\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 implicitly convert to `\real` where needed. When `\real` values need to be auto-boxed into an Object, they are boxed as `???TODO` values; similarly when JML specifications are compiled for runtime checking, `\real` values are represented as `???TODO` values. Within JML specifications, however, the `\real` type is treated as a primitive type. Integral values, including `\bigint`, are implicitly converted to `\real` where necessary.

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

5.3 `\TYPE`

TODO

OLD STUFF:

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

TBD - what about other constructors or acccessors of TYPE values

5.4 `\locset`

The `\locset` type is the type of *MU,DISCUSS: finite* sets of heap locations, that is, of left-hand-side-values that can occur on the left hand side of an assignment statement. There are three kinds of heap locations:

1. references to static fields (ClassName.staticFieldName)
2. references to non-static fields in objects which are pairs (o,f) consisting of an object reference o and a reference to 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 as the target of `accessible` and `assignable` clauses. *(Weigl) This is rather misleading: a type is clause? It turnaround the first-citizen character as it only focus on these two clauses.* In earlier versions of JML these clauses only took static lists of locations (cf. §**??**), but in order to reason about linked data structures, first-class expressions representing sets of locations are needed. *MU: Actually, with datagroups these were already dynamic, and also "o.f" could mean something different depending on the value of o. What is new is that are first-class cizizens and that they can be stored in entities.*

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

- `\locset()` - constructs an empty set
- `\locset(<storeref> ...)` - constructs a set containing the designated locations
- `obj.*` - describes the a location set contain all fields of the given object *obj*
- `ary[*]` and `ary[n..m]` - describes a location set where all either all index position of *ary* are included, or in the second case only the index position from *n* (inclusively) to $m-1$ (exclusively).

- `(\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>`*` ...)` - union of `\locsets`
- `\intersection(`*`<expr>`*` ...)` - intersection of `\locsets`
- `\disjoint(`*`<expr>`*` ...)` - true iff the arguments are pair-wise disjoint
- `\subset(`*`<expr>`*`,` *`<expr>`*`)` - true iff the first expression evaluates to a subset of the evaluation of the second
- `\setminus(`*`<expr>`*`,`*`<expr>`*`)` - 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 (says `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>)

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 `package-info.java` file. 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 §**??**.

6.1 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.

Is this correct?

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

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.

Anything to say about model declarations within a model class? Can we give a good use case/motivation for using this feature? Should we distinguish model and ghost class declarations?

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.

Need to work out implicit specs for enum and record types

Specifications at the type level serve three different 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 §**??**, 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 §**??**. 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.

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 a String argument, which is a string of command-line options and corresponding arguments. 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.

Or does Option take an array of String

7.2 invariant clause

TODO

7.3 constraint clause

TODO

7.4 initially clause

Grammar:

An `initially` clause for a type is equivalent to an additional postcondition for each constructor of the type, as if an additional `ensures` clause (with the predicate stated by the `initially` clause) is added to every behavior of each constructor in the type.

Say more about initially clauses in interfaces and enums

7.5 ghost fields

TODO

7.6 model fields

TODO

7.7 represents clause

TODO

7.8 model methods and model classes

TODO

7.9 initializer and static_initializer

TODO

7.10 axiom

TODO

7.11 readable if clause and writable if clause

TODO

7.12 monitors_for clause

TODO

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 also contains JML 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>
      |<old-clause>
      |<forall-clause>
      |<assignable-clause>
      |<accessible-clause>
      |<callable-clause>
      |<ensures-clause>
      |<signals-clause>
      |<signals-only-clause>
      |<diverges-clause>
      |<measured-by-clause>
      |<duration-clause>
      |<when-clause>
      |<working-space-clause>
      |<captures-clause>
      |<method-program-block>

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

Meta-parser rules:

- Each of the behavior keywords spelled **behaviour** is equivalent to the corresponding keyword spelled **behavior**. The latter is more common.
- A behavior beginning with **normal_behavior** may not contain a <*signals-clause*> or a <*signals-only-clause*>. It implicitly contains the clause **signals (Exception e) false;**.
- A behavior beginning with **exceptional_behavior** may not contain a <*ensures-clause*>. It implicitly contains the clause **ensures false;**.

  Is this a requirement or a style recommendation?
- *May any clauses appear after a nested-clause?*

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

OK to relax the rules on which clauses can be present where in parsing, and then enforce or advise in later checking?

FIXME - the method-spec production is not correct - the first also might be omitted whichever one it is

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 <*nested-clause*>$_A$ is a combination of n <*clause-seq*> as in
{| <*clause-seq*>$_{S1}$ (**also** <*clause-seq*>$_{Si}$) $\star$ **|}**
then
(<*java-visibility*>$_V$ (**code**)?$_W$<*behavior-id*>$_X$)?<*clause*>$*_D$<*nested-clause*>$_A$<*clause-seq*>$_E$
is equivalent to a sequence of n <*behavior*> constructions
(<*java-visibility*>$_V$ (**code**)?$_W$<*behavior-id*>$_X$)?<*clause*>$*_D$<*clause-seq*>$_{S1}$<*clause-seq*>$_E$
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, forall 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 §3.2.

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.
- `private` behaviors are 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. 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.

Note that 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 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 §**??**). Any of the kinds of behavior keywords (`behavior`, `normal_behavior`, `exceptional_behavior`) may be prefixed by a Java visibility keyword (`public`, `protected`, `private`); the absence of a visibility keyword indicate 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 name (of a type, method or field) in a method specification that is visible to a client must also be 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.

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

A protected behavior is visible to any client in the same or 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 the 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.

8.1.6 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

Comment on comparison with ACSL

8.2 Method specifications as Annotations

8.3 Modifiers for methods

TODO

8.4 Common JML method specification clauses

TODO

8.4.1 `requires` clause

TODO

The order of <*requires-clause*> clauses 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.

8.4.2 `ensures` clause

TODO

The order of <*ensures-clause*>s is also significant in that earlier expressions can assert conditions that are required for later expressions to be well-defined.

8.4.3 `assignable` clause

TODO

8.4.4 `signals` clause

TODO

8.4.5 `signals_only` clause

TODO

8.5 Advanced JML method specification clauses

TODO

8.5.1 `accessible` clause

TODO

8.5.2 `diverges` clause

TODO

8.5.3 `measured_by` clause

TODO

8.5.4 `when` clause

TODO

8.5.5 `old` clause

TODO

Any declarations in <*old-clause*> and <*forall-clause*> clauses must precede any uses of the declared variables.

8.5.6 `forall` clause

TODO

Any declarations in <*old-clause*> and <*forall-clause*> clauses must precede any uses of the declared variables.

8.5.7 `duration` clause

TODO

8.5.8 `working_space` clause

TODO

8.5.9 `callable` clause

TODO

8.5.10 `captures` clause

TODO

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

TBD

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

TBD

8.7.3 `model` and @Model

TBD

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.

TBD

8.7.5 `helper` and @Helper

TBD

8.7.6 `function` and @Function

TBD

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

TBD

8.7.8 `code_java_math`, `code_bigint_math`, `code_safe_math`, `spec_java_math`, `spec_bigint_math`, `spec_safe_math`

TBD - add annotations

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 Java annotation applies to class or method declarations. It is available only as a Java annotation.

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;
  }

  @org.jmlspecs.annotation.Options("-specspath=")
  public void n() {
    //@ assert false;
  }
}
```

Here method `m` is processed with the given timeout and verboseness level, while method `n` has the specspath set to an empty list. 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

constructor field method nowarn

`<::` 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)

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.

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

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

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

if the Java visibility is less than protected. Section **??** contains a detailed discussion of the effect of information hiding using Java visibility on JML specifications.

Listing 9.1: Use of spec_public

```

private /*@ spec_public */ int value;

//@ ensures value == i;
public setValue(int i) {
  value = i;
}
```

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`.

```
public interface MyCollection {
  //@ model instance int size; // a public instance JML field
  final int MAX = 100; // a public static Java field
}
```

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 (§**??**).

- 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 — §**??**.

9.4 Datagroups: `in` and `maps` clauses

TODO

Chapter 10

JML Statement Specifications

JML Statement specifications 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 that follow.

Many 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). The semicolon is required if the statement is succeeded by another statement within the same JML comment.

Grammar:

```
<jml-statement> ::=
      <jml-assert-statement>          §10.1
     |<jml-check-statement>           §10.2
     |<jml-assume-statement>          §10.3
     |<jml-local-declaration>         §10.4
     |<jml-unreachable-statement>     §10.6
     |<jml-reachable-statement>       §10.7
     |<jml-set-statement>             §10.8
     |<jml-debug-statement>           §10.8
     |<jml-havoc-statement>           §??
     |<jml-show-statement>            §10.9
     |<jml-hence-by-statement>        §??
     |<jml-loop-specification>        §??
     |<jml-refining-specification>    §??
```

10.1 `assert` statement and Java assert statement

Grammar:
<jml-assert-statement> `::=` **`assert`** *<jml-expression>* `;`

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 runtime assertion checking tool is expected to check whether the asserted expression is true and to issue a warning message if it is not true in the given execution of the program.

In static-checking, after an assert statement, the asserted predicate is assumed to be true (see the `check` statement in §**??** for an alternative). For example, in

```
// c possibly null
//@ assert c != null;
//@ 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 enaabled (via the Java `-ea` option).

In executing a Java program, a Java assert statement. 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.

10.2 `check` statement

Grammar:
<jml-check-statement> `::=` **`check`** *<jml-expression>* `;`

Type checking requirements:

- the <*jml-expression*> must be boolean

A `check` statement behaves just like a JML `assert` statement except for this: after a `check` statement, the predicate is *not* assumed to be true, as it is for an `assert`. Thus, to repeat the example above, in this code

```
// c possibly null
//@ check c != null;
//@ int i = c.value;
```

a tool should give two errors: one that the `check` statement is not provable and a second that there might be a null-dereference in the `c.value` expression.

A `check` statement is useful for inquiring about the truth of a given predicate without otherwise disturbing the logic of a program.

10.3 `assume` statement

Grammar:
<jml-assume-statement> `::=` **`assume`** *<jml-expression>* `;`

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 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.

10.4 local ghost variable declarations

Grammar:
<jml-local-declaration> `::=`
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 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

Any JML modifiers?

variable type may be a JML type

Grammar needs to permit array initializers

10.5 local model class declarations

model declarations?

10.6 `unreachable` statement

Grammar:

```
<jml-unreachable-statement> ::=
      unreachable <jml-expression>? ;
```

Type checking requirements:

- the optional <*jml-expression*> must be boolean; if not present its default value is **true**.

The `unreachable` statement asserts that no feasible execution path will ever reach this statement when the given expression is true. Thus

`unreachable` <*expr*>;

is equivalent to

`assert !`<*expr*>; .

As an `unreachable` statement rarely has an expression and is typically the only statement within its JML comment, the terminating semicolon is routinely omitted.

Runtime-checking can only check that no `unreachable` statement is executed in the current execution of a program.

The unreachable statement with an expression is an extension to standard JML.

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.

10.7 `reachable` statement

Grammar:

```
<jml-reachable-statement> ::=
      reachable <jml-expression>?  ;
```

Type checking requirements:

- the optional <*jml-expression*> must be boolean; if not present its default value is **true**.

The `reachable` statement asserts that there exists a feasible execution path that reaches this statement with the given predicate true. It does *not* mean that whenever execution reaches this point that the predicate is always true; for that you should use a combination of `reachable` and `assert`.

The examples that follow are explained by the comments:

```
void m1(int i) {
  //@ assert i == 0; // ERROR: i can be any integer, not just 0
}
void m2(int i) {
  //@ reachable i == 0; // OK. It is possible to reach the
                        // statement with i == 0
}
void m3(int i) {
  if (i > 0) {
    //@ reachable // OK - reachable in some scenario
  }
}
void m4(int i) {
  if (i > 0) {
    //@ reachable i == 0;//ERROR: reachable but never with i==0
  }
}
```

As an `unreachable` statement rarely has an expression and is typically the only statement within its JML comment, the terminating semicolon is routinely omitted.

The `reachable` statement is especially useful for checking the feasibility of a program, answering questions such as can execution ever go down a certain execution path; it is also used to check whether the specifications for a method are accidentally contradictory, in which case the method body is not feasible. For example, verification of the following code will fail at the `reachable` statement because the precondition contradicts the else branch of the if-statement; if the precondition holds, the else branch will never be executed.

```
//@ requires i > 0;
void m(int i) {
  if (i > 0) { ...  }
  else {
    //@ reachable
    throw new RuntimeException("Argument not positive");
  }
}
```

The usual execution of an underlying solver checks whether there are any feasible paths for which any assertions are false. The reachability test is different and typically requires separate executions of underlying solvers, as the test is now to find at least one path that reaches the given statement. If there are multiple `reachable` statements in a method, the check is for each one of them individually; they are not required to all be reachable for the same initial state.

Runtime-checking can only check that if it happens to execute a `reachable` statement in the current execution of a program, then the predicate in the statement is true. This is not a particularly useful statement. In fact the predicate might be false on legitimate executions. Accordingly, RAC tools should ignore `reachable` statements.

The reachable statement is an extension to standard JML.

It has been common practice to insert `assert false;` statements to check whether a given program point is feasible; if it is feasible, the `assert false;` statement will cause a static checking warning. The `reachable` statement accomplishes the same purpose with clearer syntax and without the need for temporary edits to the program.

10.8 `set` and `debug` statements

Grammar:
<jml-set-statement> ::= **set** *<java-statement>*
<jml-debug-statement> ::= **debug** *<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; the DRM is inconsistent in how it describes debug statements.

A `set` statement marks a statement that is executed during runtime assertion checking or symbolically executed during static checking. 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.

The semantics of a `debug` statement is the same as the `set` statement except that by default a `debug` statement is ignored. A `debug` statement is only executed when enabled by enabling the optional annotation key `DEBUG` (cf. §**??**). That is, a (single-line, standalone) `debug` statement

`//@ debug` *statement*

is equivalent to

`//+DEBUG@ set` *statement*

This connection between debug and the DEBUG optional key is an extension.

DRC: I propose deprecating the debug statement in favor of //+DEBUG@ show ...

10.9 `show` statement

Grammar:
<*jml-show-statement*> ::= **show** <*jml-expression*> ... **;**

Type information:
The expressions in the `show` statement may have any type.

The show statement is a debugging statement and may be ignored by tools. If implemented, the expected behavior is this:

- When executed during runtime-assertion-checking, it prints out (as with `System.out.println`) the values of the given expressions. As the expressions may be JML expressions, they are not accessible to debugging of the Java program itself.

- In static checking, if a proof of a method fails with a counterexample, then the counterexample contains the values of the expressions for inspection, associated with some identifying identifier.

The `show` statement provides functionality similar to the `\lbl` expression, but more conveniently. As is the case for all JML expressions, the `show` statement has no side-effects.

10.10 loop specifications

Grammar:
<loop-specification> `::=` (*<loop-clause>*) `*`
<loop-clause> `::=` *<loop-invariant>* | *<loop-variant>* | *<loop-assignable>*
<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-assignable> `::=`
<loop-assignable-keyword> *<location-set>* (**,** *<location-set>* |) `*` **;**
<loop-assignable-keyword> `::=` **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 *(or just a long?)*
- 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

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*

10.11 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

10.12 statement (block) specification

Grammar:

> <*statement-specification*> ::= **refining** <*behavior-seq*>

The semantics of a statement specification are very similar to those of a method specification (cf. §**??**). 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 statement specification makes assertions about the execution of the statement (possibly a block statement) that follows the statement specification, or, if a **begin** JML statement 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

The primary conceptual differences between the method and statement specifications are that

- 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

The motivation for a statement 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.

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.

Write this more formally? as a desugaring?

Do we really need the refining keyword? DRC - Actually I think we want to require it for easier parsing.

Need comment from Gary and Mattias

10.13 `hence_by` statement

Write this *Should this be removed?*

inline_loop statement

Chapter 11

JML Expressions

Grammar:

```
<jml-expression> ::=
        <result-expression>                                    §11.12
       |<exception-expression>                                 §11.13
       |<informal-expression>                                  §11.19
       |<old-expression>                                       §??
       |<key-expression>                                       §11.15
       |<lbl-expression>                                       §11.16
       |<nonnullelements-expression>                           §11.17
       |<fresh-expression>                                     §11.18
       |<type-expression>                                      §11.20
       |<typeof-expression>                                    §11.21
       |<elemtype-expression>                                  §11.22
       |<invariant-for-expression>                             §??
       |<is-initialized-expression>                            §11.23
 Missing some - check the list
       |<duration-expression>                                  §??
       |<working-space-expression>                             §??
       |<space-expression>                                     §??
       |(  <jml-expression>  )
       |<jml-expression> ( <# |<#= ) <jml-expression>
```

shift bit logical dot cast new methodcall ops

Need sections on \count and \values

11.1 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.

11.2 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.

11.3 Java operations used in JML

Because of the pure expression rule (cf. §11.2), some Java operators are not permitted in JML expressions:

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

11.4 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 11.1.

11.5 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 11.1: Java and JML precedence (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 <*jml-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?

11.6 Chaining of comparison operators

Grammar:

```
<jml-expr> ::=
        <jml-expr> ( [<|<=] <jml-expr> )+
       |<jml-expr> ( [>|>=] <jml-expr> )+
```

Well-defined:

$$[[\, e_1 \; op \; e_2 \; op \; ... \; op \; e_n]] \equiv \forall i \; [[e_i]]$$

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.

11.7 org.jmlspecs.lang.JML

Say more

11.8 Implies operator: `==>`

Grammar:
`<jml-expr> ::= <jml-expr> ==> <jml-expr>`

Well-defined:

$$[[\, e_1 \textbf{ ==> } e_2 \,]] \equiv [[\, e_1 \,]] \;\&\; (e_1 \implies [[\, e_2 \,]])$$

Type information:

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

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

11.9 Equivalence and inequivalence: `<==>` `<=!=>`

Grammar:
```
<jml-expr> ::=
        <jml-expr> <==> <jml-expr>
      | <jml-expr> <=!=> <jml-expr>
```

Well-defined:

$$[[\, e_1 \textbf{ <==> } e_2 \,]] \equiv [[\, e_1 \,]] \;\&\; [[\, e_2 \,]]$$
$$[[\, e_1 \textbf{ <=!=> } e_2 \,]] \equiv [[\, e_1 \,]] \;\&\; [[\, e_2 \,]]$$

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.

11.10 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. §**??**). *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.

11.11 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 lock ordering operator enables the specifier to write appropriate statements about the desired ordering.

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.

11.12 `\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`.

11.13 `\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?

11.14 `\old`, `\pre`, and `\past`

Grammar:

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

Any pre-defined labels?

\past is an extension

Text needed

11.15 `\key`

The \key expression is an OpenJML extension

Grammar:

```
<key-expression> ::= \key( <string-literal> )
```

Type information:

- The argument must be a compile-time string literal
- The expression has type boolean

This expression enables a JML expression to vary depending on the settings of optional, tool-specified keys. These are the same keys as are used for optional annotation comments (cf. §**??**). The `\key` expression is translated during parsing to a true or false boolean literal depending on whether the key is defined or not (such keys do not have values). Some tools may subsequently use constant folding to avoid processing or executing unreachable parts of expressions. However, the remainder of the expression must still be parseable and type-correct. To avoid parsing some optional text entirely use optional JML annotations.

Example: The expression

!\key("RAC") ==> state == 0

might be used in a situation where `state` is a model field that is not compiled in RAC mode. This expression is equivalent to

- `true` if `RAC` is defined (so that `\key("RAC")` is true)
- `state == 0` if `RAC` is not defined (so that `\key("RAC")` is false)

Mechanisms for defining keys are provided by tools and are not defined in JML itself.

11.16 `\lblpos`, `\lblneg`, `\lbl`, and `JML.lblpos()`, `JML.lblneg()`, `JML.lbl()`

\lbl and the JML forms are extensions

Grammar:
```
<lbl-expression> ::=
          ( ( \lbl <id> <jml-expression> ) )
        | ( ( \lblpos <id> <jml-expression> ) )
        | ( ( \lblneg <id> <jml-expression> ) )
```

Type information: `\lblpos`, `\lblneg`

- the first argument is an id, that is a legal Java identifier (not within quotes)
- the second argument is a boolean expression
- well-defined iff the boolean expression is well-defined
- the expression has type boolean; its value is the value of the expression in the argument

TODO: I don't think the grammar above is correct

Type information: `JML.lblpos()`, `JML.lblneg()`

- first argument is a String literal, the second has type boolean

- well-defined iff the arguments are well-defined
- expression type is boolean; its value is the value of the second argument

Type information: `\lbl`

- the first argument is an id, that is a legal Java identifier (not within quotes)
- the second argument has any (non-void) type
- well-defined iff the second argument is well-defined
- expression type and value are the same as the type and value of the second argument expression

Type information: `JML.lbl()`

- first argument is a String literal, the second has any (non-void) type
- well-defined iff the arguments are well-defined
- expression type and value are the same as the type and value of the second argument expression

These expressions are used for debugging. When static checking finds that some assertion is invalid and generates a counterexample for that invalid assertion, then the value of the expression contained in each of these $< lbl - expression >$s is reported as part of the counterexample, in association with the id or String literal. When runtime assertion checking is used, these expressions print the String or id and the value of the expression. In each case, the construct simply passes on the type and value of its expression, possibly generating some debug output in the process.

The `\lblpos`, `\lblneg`, and `\lbl` expressions have a non-standard syntax:

```
                    ( \lbl <id> <expression>)
```

with no comma between what would be the arguments. Here the <*id*> is a Java identifier. The `JML.lbl` forms are standard functional forms: the first argument is a String literal; the second is an expression. A String literal is required instead of a String expression because the value is used to identify the output as coming from this expression and it is not evaluated during static checking.

In the positive and negative forms, the argument is a boolean expression. Output is generated by `\lblpos` only if the expression is true in the counterexample; output is generated by `\lblneg` only if the expression is false in the counterexample. In the neutral forms (`\lbl` and `JML.lbl`), output is generated whatever the value. For any type, the value is converted to a String value as is customary in Java (e.g., using toString()).

Examples: *Examples needed*

11.17 `\nonnullelements`

Grammar:

`<nonnullelements-expression>` `::=`

```
        \nonnullelements ( <jml-expression> ... )
```

Type information:

- strict JML: one argument, an expression of array type, may be null
- OpenJML extension: one or more arguments, each an expression of array type
- well-defined iff all arguments are well-defined
- expression type is boolean

What about an array or sequence of arrays

The arguments of the `\nonnullelements` expressions must be expressions that each evaluate to an array. The `\nonnullelements` expression is true iff for each argument, the argument is non-null and each element of the argument's array value is not null.

Perhaps allow the argument to be reference type - result is true if the argument is non-null and, if an array, all elements are non-null. Are elements non-null recursively?

Allow any number of arguments?

11.18 `\fresh`

Grammar:

```
<fresh-expression> ::=
        \fresh( <jml-expression> ... )
```

Type information:

- strict JML: one argument, an expression of reference type
- OpenJML extensions: one or more arguments, each an expression of reference type
- well-defined iff all arguments are non-null and well-defined
- expression type is boolean
- use: `\fresh` may be used only in `ensures` and `signals` clauses

The arguments of the `\fresh` expression must be expressions that evaluate to non-null references. The `\fresh` expression is true iff each argument is a reference that was not allocated in the pre-state.

Standardize the JML extension?

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

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

Well-defined:
  [[(.∗)]] ≡ **true**
  [[**JML.informal(***e***)**]] ≡ [[*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()  ...
```

11.20 `\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 §**??**.

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

```
//@ ...  \type(int) ...
//@ ...  \type(Integer) ...
//@ ...  \type(java.lang.Integer) ...
//@ ...  \type(java.util.LinkedList<String>) ...
//@ ...  \type(java.util.LinkedList<String>[]) ...
//@ ...  \type(T) ...
//@ ...  \type(java.util.LinkedList<T>) ...
```

11.21 `\typeof`

Grammar:
<typeof-expression> **::=**
    \typeof (*<jml-expression>* **)**

Well-defined:

[[**typeof(***e***)**]] ≡ [[*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:

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

11.22 `\elemtype`

Grammar:
```
<elemtype-expression> ::=
       \elemtype ( <jml-expression> )
```

Well-defined:

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

Type information:

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

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

Examples:

```
//@ assert \elemtype(\type(int[])) == \type(int);
//@ 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?

11.23 `\is_initialized`

Grammar:
```
<is-initialized-expression> ::=
       \is_initialized ( <jml-expression> ...   )
      |\is_initialized ( )
```

Text needed

11.24 `\invariant_for`

Grammar:

```
<invariant-for-expression> ::=
      \invariant_for ( <jml-expression> ...   )
     |\invariant_for ( )
```

Allow expr or type in grammar

Type information:

- Strict JML: one argument, of type `Object` or a typename, but not an expression of primitive type
- Extension: any number of arguments
- well-defined iff the argument is well-defined and the value is either a type name or has reference type that is not an array type
- expression has type `boolean`

This expression with one expression argument is equivalent to the conjunction (with `&&`) of the static and non-static JML-visible 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.

OpenJML allows as an extension a typename argument. The expression with a typename argument is equivalent to the conjunction (with `&&`) of the *static* JML-visible invariants in the named type and all its super classes and interfaces (recursively).

In each case 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.

Strict JML requires that there be just one argument. As an extension, OpenJML allows any number of arguments. The expression is then equivalent to the conjunction of the values for each argument, in order, conjoined by the short-circuit operator `&&`. When there are no arguments, the value of `\invariant_for()` is `true`.

Are the invariants of superclasses and interfaces included?

Are both static and non-static invariants included?

The DRAFT JML reference manual does not mention JML-visibility, but I presume that must be the case.

multiple arguments, type-name argument

The DJMLRM does not mention a static-only version of \invariant_for - so this is an extension?

11.25 `\not_modified`

Grammar:

```
<not-modified-expression> ::=
        \not_modified ( <jml-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`
- use: only in `ensures` or `signals` clauses

A `\not_modified` expression is a two-state expression that may occur only in `ensures` or `signals` 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.*

11.26 `\lockset` and `\max`

Text needed

11.27 `\reach`

Text needed

11.28 `\duration`

Grammar:

```
<duration-expression> ::=
       \duration ( <jml-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

11.29 `\working_space`

Grammar:

```
<working-space-expression> ::=
       \working_space ( <jml-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.

11.30 `\space`

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

Type information:

- one argument, of any reference type
- well-defined iff the argument is well-defined *Must the argument be well-defined?*
- expression has type `long`

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

not-assigned, not-modified, only-accessed, only-called, only-assigned, only-captured, spec-quantified-expr *Text needed*

Chapter 12

Arithmetic modes

12.1 Description of 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. Chalin [**?**] surveyed programmer expectations and desires and identified three useful 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 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 (in the description below, this mode is called 'bigint' mode, but is the mathematical mode for both integers and reals).

The relationship between real vs. floats and doubles is very much under discussion

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

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

In JML, the type of mathematical integers is expressed as `\bigint` and the type of mathematical reals is expressed as `\real`. Static checking can reason about these types using usual logics with arithmetic; runtime checking uses `java.math.BigInteger` to represent `\bigint` and `org.jmlspecs.lang.Real` to represent `\real`.

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 matah 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

12.2 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`

12.3 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 warning 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).

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. Note though that the default arithmetic mode for specifications is bigint mode, so this situation rarely arises.

12.4 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). Static analysis 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.

12.5 Real arithmetic and non-finite values

TBD -reference what is done in ACSL

Chapter 13

Specification and verification of lambda functions

TODO: to be written

Chapter 14

Universe types

TODO: To be written

Chapter 15

Model Programs

Describe the intent, syntax and semantics of model programs

Chapter 16

Specification `.jml` files

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. GL: Yes, maybe it should be in a chapter on lexical analysis. DRC: Moved it here - is this OK

JML files have a `.jml` extension and have a similar appearance 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 rules below refer to the `.java` file that would have been compiled to produce the `.class` file.

The principle present throughout these rules is that declarations in a JML file either (1) correspond to a declaration in the Java file, having the same name, types, non-JML modifiers and annotations, or (2) do not correspond to a Java declaration, and then must have 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 non-JML modifiers (`public`, `protected`, `private`, `static`, `final`, *What others*). The JML declaration may add additional JML modifiers or 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 non-JML modifiers (`public`, `protected`, `private`, `static`, *What others*). The JML declaration may add additional JML modifiers or annotations.
- *Comment on static and instance; no initializer for JML field declarations*

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. MU: look into JLS and check if that is "same signature"
- Methods that correspond to Java methods must not be declared model and must not have a body. They must have the same set of non-JML 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 non-JML 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 and operators; 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 initializer blocks.

16.1 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 to a `.jml` file, then the JML specification present in the `.jml` file supersedes all of the JML specifications in the `.java` file; those 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. *Where described?*

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. [1] *Be sure we want this superseding deign rather than merging – could use specs in the Java file if there are no JML declaration, just not merge when both have JML declarations.*

[1]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 17

Obsolete and Deprecated Syntax

describe deprecated syntax, obsolete syntax, incompatible changes from past versions

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.

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 §**??** 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 §**??** for details on the use of file names with JML tools.

B.1.5 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 §**??** for details.

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 §**??**.

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

Misc stuff to move, incorporate or delete

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

G.1 Finding specification files and the refine statement

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

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

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

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

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

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

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

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

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

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

G.1.1 Model import statements

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

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

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

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

G.1.2 Modifiers

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

Reuse this table?

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

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

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

G.1.3 JML expressions

Expressions in JML annotations are Java expressions with three adjustments:

- Expressions with side-effects are not allowed. Specifically, JML excludes
  - the ++ and − pre- and post- increment and decrement operations
  - the assignment operator
  - assignment operators that combine an operation with assignment (e.g., +=)
  - method invocations that are not explicitly declared `pure` (cf. §TBD)

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

Table G.1: Summary of JML modifiers. All Java annotations are in the `org.jmlspecs.annotation` package.

`new () [] .` and method calls		
unary + unary − `!` `(typecast)`	-	
`* / %`	L	
+ (binary) − (binary)	L	
« » »>	L	
`< <= > >= <: instanceof <# <#=`	-	`<:` is the JML subtype operation (§**??**); <# and <#= are lock ordering operators (§G.1.3)
`== !=`	L	
`&`	L	
`^`	L	
`|`	L	
`&&`	L	
`||`	L	
`==> <==`	?	JML implies and reverse implies (§**??**)
`<==> <=!=>`	?	JML equivalence and inequivalence (§**??**)
`?:`	-	
`= *= /= %= += -= «= »= »>= &= ≙ |=`	L	Java only

Table G.2: Java and JML operators, in order of precedence, from highest (most tightly binding) to lowest precedence. Operators on the same line have the same precedence. The associativity is given in the central column.

- JML adds additional operators to the Java set of operators, discussed in subsection §**??** below.
- JML adds specific keywords that are used as constants or function-like expressions within JML expressions, discussed in subsection §**??** below

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

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

G.1.4 Code contracts

This section will be added later.

G.2 JML modifiers and Java annotations

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

Modifiers

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

Each such modifier has an equivalent Java annotation. For example

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

can be written equivalently as

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

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

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

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

All of the modifiers, their corresponding Java annotations, and the locations in which they may be used are described in §G.4.5.

Type modifiers

Some modifiers are actually type modifiers. In particular `non_null` and `nullable` are in this category. Thus the description of the previous subsection (§G.2) apply to these as well.

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

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

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

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

Method specification clauses

This section will be added later.

Class specification clauses

This section will be added later.

Statement specifications

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

G.3 org.jmlspecs.lang package

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

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

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

G.4 JML Syntax

G.4.1 Syntax of JML specifications

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

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

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

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

G.4.2 Conditional JML specifications

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

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

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

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

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

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

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

G.4.3 Finding specification files and the refine statement

Discuss obsolete syntax somewhere - including the refines statement

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

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

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

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

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

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

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

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

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

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

G.4.4 Model import statements

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

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

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

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

G.4.5 Modifiers

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

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

Table G.3: Summary of JML modifiers. All Java annotations are in the `org.jmlspecs.annotation` package.

`public`.

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

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

G.4.6 JML expressions

Expressions in JML annotations are Java expressions with three adjustments:

- Expressions with side-effects are not allowed. Specifically, JML excludes
  - the ++ and − pre- and post- increment and decrement operations

  - the assignment operator
  - assignment operators that combine an operation with assignment (e.g., +=)
  - method invocations that are not explicitly declared `pure` (cf. §TBD)

- JML adds additional operators to the Java set of operators, discussed in subsection §**??** below.
- JML adds specific keywords that are used as constants or function-like expressions within JML expressions, discussed in subsection §**??** below

G.4.7 JML types

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

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

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

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

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

TBD - what about other constructors or acccessors of TYPE values

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/se17/html/jls-3.html`.

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

[3] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential non-deterministic 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.

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

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

[6] 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.

[7] 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`.

[8] 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`.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[15] 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.

[16] 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.

[17] David Cok. OpenJML: JML for Java 7 by extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer-Verlag, Berlin, 2011.

[18] David R. Cok, 2018. `http://www.openjml.org`.

[19] David R. Cok. JML and OpenJML for Java 16. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, FT-

fJP 2021, page 65â€“67, New York, NY, USA, 2021. Association for Computing Machinery.

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

[21] 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.

[22] 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.

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

[24] 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.

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

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

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

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

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

[30] 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.

[31] 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.

[32] 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.

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

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

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

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

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

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

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

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

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

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

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

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

[61] 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.

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

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

Index