# JML Reference Manual

DRAFT, $Revision: 1.235 $
$Date: 2008/07/17 20:40:09 $

**Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, Werner Dietl**

Version Information:
@(#) $Id: jmlrefman.texinfo,v 1.235 2008/07/17 20:40:09 wdietl Exp $

# Table of Contents

# 1 Introduction

JML is a notation for formally specifying the behavior and interfaces of Java [Arnold-Gosling-Holmes00] [Gosling-etal00] classes and methods.

The goal of this reference manual is to precisely record the design of JML. We include both informal semantics (intentions) and where possible [[[we will eventually include]]] formal semantics (describing when an implementation satisfies a specification). We also discuss the implications for various tools (such as the run-time assertion checker, static checkers such as ESC/Java2, and documentation generators such as jmldoc [Burdy-etal03]).

In this manual we also try to give examples and explanations, and we hope that these will be helpful to readers trying to learn about formal specification using JML. However, this manual is not designed to give all the background needed to write JML specifications, nor to give the prospective user an overview of a useful subset of the language. For this background, we recommend starting with the papers "Design by Contract with JML" [Leavens-Cheon06] and "JML: A notation for detailed design" [Leavens-Baker-Ruby99], and continuing with the paper "Preliminary Design of JML" [Leavens-Baker-Ruby06]. These are all available from the JML web site '`http://www.jmlspecs.org/`', where further readings and examples may also be found.

Readers with the necessary background, and users wanting more details may, we hope, profit from reading this manual. We suggest reading this manual starting with chapters 1-3, skimming chapter 4 quickly, skimming chapter 5 to get the idea of what declarations mean in JML, and then reading the chapters on class specifications (chapter 6) and method specifications (chapter 7), paying particular attention to the examples. After that, one can use the rest of this manual as a reference.

The rest of this chapter describes some of the fundamental ideas and background behind JML.

## 1.1 Behavioral Interface Specifications

JML is a *behavioral interface specification language* (BISL) that builds on the Larch approach [Guttag-Horning93] [Guttag-Horning-Wing85b] and that found in APP [Rosenblum95] and Eiffel [Meyer92b] [Meyer97]. In this style of specification, which might be called model-oriented [Wing90a], one specifies both the interface of a method or abstract data type and its behavior [Lamport89]. In particular JML builds on the work done by Leavens and others in Larch/C++ [Leavens-Baker99] [Leavens96b] [Leavens97c]. (Indeed, large parts of this manual are adapted wholesale from the Larch/C++ reference manual [Leavens97c].) Much of JML's design was heavily influenced by the work of Leino and his collaborators [Leino95] [Leino95b] [Leino98] [Leino-etal00] [Leino-Nelson-Saxe00]. 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`'.

The *interface* of the method or type is the information needed to use it from other programs. 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 exceptions it may throw, and so on. For a field the interface includes its name and type, and its 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 change), and the relations between the calling state and the state in which it either returns normally, throws an exception, or for which it might not return to the caller. The states for which 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 axiom* [Borgida-etal95]. In JML one can also specify other aspects of behavior, such as the time a method can use to execute and the space it may need.

The behavior of an abstract data type (ADT), which 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 [Cheon-etal05], which are specification-only fields, or by specifying some of the fields used in the implementation as `spec_public` or `spec_protected`. 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 [Hayes93] [Spivey92] or Fresco [Wills92b].

## 1.2 A First Example

For example, consider the following JML specification of a simple Java abstract class `IntHeap`. (An explanation of the notation follows the specification. This specification, like the others in this manual, ships with the JML release in the '`JML/org/jmlspecs/samples/jmlrefman`' directory.)

```
package org.jmlspecs.samples.jmlrefman;                  // line 1
                                                         // line 2
public abstract class IntHeap {                          // line 3
                                                         // line 4
    //@ public model non_null int [] elements;           // line 5
                                                         // line 6
    /*@ public normal_behavior                           // line 7
      @   requires elements.length >= 1;                 // line 8
      @   assignable \nothing;                           // line 9
      @   ensures \result                                // line 10
      @        == (\max int j;                           // line 11
      @               0 <= j && j < elements.length;     // line 12
      @               elements[j]);                      // line 13
      @*/                                                // line 14
    public abstract /*@ pure @*/ int largest();          // line 15
                                                         // line 16
    //@ ensures \result == elements.length;              // line 17
    public abstract /*@ pure @*/ int size();             // line 18
};                                                       // line 19
```

The interface of this class consists of lines 1, 3, 15, and 18. Line 3 specifies the class name, and the fact that the class is both public and abstract. Lines 15 and 18, apart from their comments, give the interface information for the methods of this class.

The behavior of this class is specified in the JML annotations found in the special comments that have an at-sign (`@`) as their first character following the usual comment beginning. Such lines look like comments to Java, but are interpreted by JML and its tools. For example, line 5 starts with an annotation comment marker of the form `//@`, and this annotation continues until the `//` towards the end of the line, which starts a comment within the annotation which even JML ignores. The other form of such annotations can be seen on lines 7 through 14, line 17, and on lines 15 and 18. These annotations start with the characters `/*@` and end with either `@*/` or `*/`; within such annotations, at-signs (`@`) at the beginnings of lines are ignored by JML. 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 Chapter 4 [Lexical Conventions], page 26, for more details about annotations.)

The first annotation, on line 5 of the figure above, gives the specification of a field, named `elements`, which is part of this class's behavioral specification. Ignoring, for the moment the extra JML modifiers, one should think of this field, in essence, as being declared like:

```
public int[] elements;
```

That is, it is a public field with an integer array type; within specifications it is treated as such. However, because it is declared in an annotation, this field cannot be manipulated by Java code. Therefore, for example, the fact that the field is declared public is not a problem, because it cannot be directly changed by Java code.

Such declarations of fields in annotations should be marked as specification-only fields, using the JML modifier `model`.[1] A model field should be thought of as an abstraction of a set of concrete fields used in the implementation of this type and its subtypes. (See Section 8.4 [Represents Clauses], page 58, for a discussion of how to specify the connection between the concrete fields and such model fields. See also the paper by Cheon et al. [Cheon-etal05].) That is, we imagine that objects that are instances of the type `IntHeap` have such a field, whose value is determined by the concrete fields that are known to Java in the actual object. Of course at runtime, objects of type `IntHeap` have no such field, the model fields are purely imaginary. Model fields are thus a convenient fiction that is useful for describing the behavior of an ADT. One does not have to worry about their cost (in space or time), and should only be concerned with how they clarify the behavior of an ADT.

The other annotation used on line 5 is `non_null`. This just says that in any publicly-visible state, the value of `elements` must not be `null`. It is thus a simple kind of invariant (see Section 8.2 [Invariants], page 50).

In the above specification of `IntHeap`, the specification of each method 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. In JML, it is also possible to put the specification just before the semicolon (`;`) following the method's interface information (see Chapter 9 [Method Specifications], page 61), but we will usually not to do that in this document.

The specification of the method `largest` is given on lines 7 through 15. Line 7 says that this is a public, normal behavior specification. JML permits several different specifications for a given method, which can be of different privacy levels [Ruby-Leavens00] [Leavens-Mueller07]. The modifier `public` says that the specification is intended for use by clients. (If the privacy modifier had been `protected`, for example, then the specification would have been intended for subclasses.)

The keyword `normal_behavior` tells JML several things. First, it says that the specification is a heavyweight method specification, as opposed to a lightweight method specification like that given on line 17. A *heavyweight* specification uses one of JML's behavior keywords, like `normal_behavior`, which tells JML that the method specification is intended to be complete. By contrast, a *lightweight* specification does not use one of JML's behavior keywords, and tells JML that the specification is incomplete in the sense that it contains only some of what the specifier had in mind.[2] Second, the keyword `normal_behavior` tells JML that when the precondition of this method is met, then the method must return normally, without throwing an exception. In other words, it says that the exceptional postcondition is `false`, which prohibits the method from throwing an exception when the precondition holds. (Third, it says that the divergence condition defaults to `false`. See Chapter 9 [Method Specifications], page 61, for more details.)

The heart of the method specification of `largest` is found on lines 7 through 13. This part of the specification gives the method's precondition, on line 8, frame axiom, on line 9, and normal postcondition, on lines 10 through 13. The precondition is contained in the

[1] This is the usual way to declare a specification-only field; it is also possible to use the `ghost` modifier (see Section 2.2 [Model and Ghost], page 11).

[2] Lightweight specifications come from ESC/Java.

`requires` clause on line 8. The frame axiom is contained in the `assignable` clause on line 9. The normal postcondition is contained in the `ensures` clause on lines 10-13.[3]

The precondition in the requires clause on line 8 says that the length of `elements` must be at least 1 before this method can be called. If that is not true, then the method is under no obligation to fulfill the rest of the specified behavior.

The frame axiom in the assignable clause on line 9 says that the method may not assign to any locations (i.e. fields of objects) that are visible outside the method and which existed before the method started execution. (The method may still modify its local variables.) This form of the frame axiom is quite common.[4] Note that in assignable clauses and in assertions, JML uses keywords that start with a backslash (\), to avoid interfering with identifiers in the user's program. Examples of this are `\nothing` on line 9 and `\result` on line 10.

The postcondition in the ensures clause, on lines 10 through 13, says that the result of the method (`\result`) must be equal to the maximum integer found in the array `elements`. This postcondition uses JML's `\max` quantifier (lines 11 through 13). Such a quantifier is always parenthesized, and can consist of three parts. The first part of a quantifier is a declaration of some quantified variables, in this case the integer `j` on line 11. The second part is a *range predicate*, on line 12, which constrains the quantified variables. The third part is the *body* of the quantifier, on line 13, which in this case describes the elements of the array from which the maximum value is taken.

The methods `largest` and `size` are both specified using the JML modifier `pure`. This modifier says that the method has no side effects, and allows the method to be used in assertions if desired.

The method `size` is specified using a lightweight specification, which is given on line 17. The ensures clause on line 17 says nothing about the precondition, frame axiom, exceptional postcondition, or divergence condition of `size`, although the use of `pure` on line 18 gives an implicit frame axiom. Such a form of specification is useful when one only cares to state (the important) part of a method's specification. It is also useful when first learning JML, and when one is using tools, such as ESC/Java2, that do not need heavyweight specifications.

The specifications of the method `largest` above is very precise: it gives a complete specification of what the method does. Even the specification of `size` has a fairly complete normal postcondition. We can also give JML specifications that are far less detailed. For example, we could just specify that the result of `size` is non-negative, with a normal postcondition such as

```
//@ ensures \result >= 0;
```

instead of the postcondition given earlier. Such incomplete specifications give considerably more freedom to implementations, and can often be useful for hiding implementation details. However, one should try to write specifications that capture the important properties expected of callers (preconditions) and implementations (postconditions) [Meyer92a] [Liskov-Guttag86].

---

[3] JML also has various synonyms for these keywords; one can use `pre` for `requires`, `modifies` or `modifiable` for `assignable`, and `post` for `ensures` if desired. See Chapter 9 [Method Specifications], page 61, for more details.

[4] However, unlike Larch BISLs and earlier versions of JML, this is not the default for an omitted `assignable` clause (see Section 9.9.9 [Assignable Clauses], page 80). Thus line 9 cannot be omitted without changing the meaning of the specification.

## 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 specification of the behavior of Java program modules (i.e., classes and interfaces), and documentation of Java code,
- the possibility of tool support [Burdy-etal03].

Although we would like tools that would help with reasoning about concurrent aspects of Java programs, the current version of JML focuses on the sequential behavior of Java code. While there is work in progress on extending JML to support concurrency, 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. A goal of JML is to make the notation indifferent to the precise 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 [Hall90] and Chapter 7 of [Guttag-Horning93]). 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 [Huisman01] [Jacobs-Poll01] [Ruby06].
- JML specifications can be used by several tools that can help debug and improve the code [Burdy-etal03].

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 [Ruby-Leavens00] [Ruby06].

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 (and friend classes). (See Chapter 16 [Refinement], page 124, for how to record each level in JML.)

The reader may also wish to consult the "Preliminary Design of JML" [Leavens-Baker-Ruby06] 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 possibility 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 typechecking – 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 of course. 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 typechecking. In particular JML is designed to support static analysis (as in ESC/Java [Leino-etal00]), formal verification (as in the LOOP tool [Huisman01] [Jacobs-etal98]), recording of dynamically obtained invariants (as in Daikon [Ernst-etal01]), runtime assertion checking (as in JML's runtime assertion checker, jmlc [Cheon-Leavens02b] [Cheon03]), unit testing [Cheon-Leavens02], and documentation (as in JML's jmldoc tool). The paper by Burdy et al. [Burdy-etal03] is a recent 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 Status and Plans for JML

JML is still in development. As you can see, this reference manual is still a draft, and there are some holes in it. [[[And some notes for the authors by the authors that look like this.]]]

Influences on JML that may lead to changes in its design include our desire to specify programs written using the unique features of MultiJava [Clifton-etal00], an eventual integration with Bandera [Corbett-etal00] or other tools for specification of concurrency, aspect-oriented programming, and the evolution of Java itself. Another influence is the ongoing effort to use JML on examples, in designing the JML tools, and efforts to give a formal semantics to JML.

## 1.5 Historical Precedents

JML combines ideas from Eiffel [Meyer92a] [Meyer92b] [Meyer97] with ideas from model-based specification languages such as VDM [Jones90] and the Larch family [Guttag-Horning93] [LeavensLarchFAQ] [Wing87] [Wing90a]. It also adds some ideas from the refinement calculus [Back88] [Back-vonWright89a] [Back-vonWright98] [Morgan-Vickers94] [Morgan94] (see Chapter 16 [Refinement], page 124). In this section we describe the advantages and disadvantages of these approaches. Readers unfamiliar with these historical precedents may want to 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 [Hoare69]. Later Hoare adapted these axiomatic techniques to the specification and correctness proofs of abstract data types [Hoare72a]. 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 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 in the implementation, which ensures that the rest of the program is also independent of the particular data structures used [Liskov-Guttag86] [Meyer97] [Meyer92a] [Parnas72]. Second, it allows the specification to be written even when there are no implementation data structures, as is the case for `IntHeap`.

This idea of model-oriented specification has been followed in VDM [Jones90], VDM-SL [Fitzgerald-Larsen98] [ISO96], Z [Hayes93] [Spivey92], and the Larch family [Guttag-Horning93]. 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 (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 [Guttag-Horning93]), 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 [Meyer92a] [Meyer92b] [Meyer97]. 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.

## 1.6 Acknowledgments

The work of Leavens and 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 Cheon, Clifton, Leavens, Ruby, and others has been supported in part by NSF grants CCR-0097907, CCR-0113181, CCF-0428078, and CCF-0429567. Support from the NSF continues under a Computing Research Infrastructure (CRI) grant jointly to several institutions: CNS 08-08913 (Leavens at U. of Central Florida, and a subcontact to Rajan and Basu at Iowa State Unviversity), CNS 07-07874 (Cheon at UTEP), CNS 07-07701 (Clifton at Rose Hulman), CNS 07-07885 (Flanagan at U. Cal. Santa Cruz), CNS 07-08330 (Naumann at Stevens), and CNS 07-09169 (Robby at Kansas State). The work of Poll is partly supported by the Information Society Technologies (IST) Programme of the European Union, as part of the VerifiCard project, IST-2000-26328.

Thanks to Bart Jacobs, Rustan Leino, Arnd Poetzsch-Heffter, and Joachim van den Berg, for many discussions about the semantics of JML specifications. Thanks for 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, and Jesus Ravelo 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" [Leavens-Baker-Ruby06] for more acknowledgments relating to the earlier history, design, and implementation of JML.

# 2 Fundamental Concepts

This chapter discusses fundamental concepts that are used in explaining the semantics of JML.

## 2.1 Types can be Classes and Interfaces

In this manual we use *type* to mean either a class, interface, or primitive value type in Java. (Primitive value types include `boolean`, `int`, etc.)

A *reference type* is a type that is not a primitive value type, that is either a class or interface. When it is not necessary to emphasize that primitive value types are not included, we often shorten "reference type" to just "type".

## 2.2 Model and Ghost

In JML one can declare various names with the modifier `model`; for example one can declare model fields, methods, and even types. One can also declare some fields as `ghost` fields. JML also has a `model import` directive (see Chapter 5 [Compilation Units], page 35).

The meaning of a feature declared with `model` is that it is only present for specification purposes. For example a model field is an imaginary field that is only used for specifications and is not available for use in Java code outside of annotations. Similarly, a model method is a method that can be used in annotations, but cannot be used in ordinary Java code. A model import directive imports names that can be used only within annotations.

The most common and useful model declarations are model fields. A model field should be thought of as the abstraction of one or more non-model (i.e., Java or *concrete*) fields [Cheon-etal05]. (By contrast, some authors refer to what JML calls model fields as "abstract fields" [Leino98].) The value of a model field is determined by the concrete fields it abstracts from; in JML this relationship is specified by a `represents` clause (see Section 8.4 [Represents Clauses], page 58). (Thus the values of the model fields in an object determines its "abstract value" [Hoare72a].) A model field also defines a data group [Leino98], which collects model and concrete fields and is used to tell JML what concrete fields may be assigned by various methods (see Chapter 10 [Data Groups], page 85).

Unlike model fields, model methods and model types are not abstractions of non-model methods or types. They are simply methods or types that we imagine that the program has, to help in a specification.

A `ghost` field is similar to a model field, in that it is also only present for purposes of specification and thus cannot be used outside of annotations. However, unlike a model field, a ghost field does not have a value determined by a represents clause; instead its value is directly determined by its initialization or by a *set-statement* (see Chapter 12 [Statements and Annotation Statements], page 104).

Although these model and ghost names are used only for specifications, JML uses the same namespace for such names as for normal Java names. Thus, one cannot declare a field to be both a model (or ghost) field and a normal Java field in the same class (or in a refinement, see Chapter 16 [Refinement], page 124). Similarly, a method is either a model method or not. In part, this is done because JML has no syntactic distinction between Java and JML field access or method calls. This decision makes it an error for someone to use the same name as a model or ghost feature in an implementation. In such a case if the Java code is considered to be the goal, one can either change the name of the JML feature or have one declaration in which the Java feature is modified with the JML modifier `spec_public`. See Section 2.4 [Privacy Modifiers and Visibility], page 12, for more about `spec_public`.

## 2.3 Lightweight and Heavyweight Specifications

In JML one is not required to specify behavior completely. Indeed, JML has a style of method specification case, called *lightweight*, in which the user only says what interests them. On the other hand, in a *heavyweight* specification case, JML expects that the user is fully aware of the defaults involved. In a heavyweight specification case, JML expects that a user only omits parts of the specification case when the user believes that the default is appropriate.

Users distinguish these between such cases of method specifications by using different syntaxes. See Section 9.2 [Organization of Method Specifications], page 61, for details, but in essence in a method specification case that uses one of the behavior keywords (such as `normal_behavior`, `exceptional_behavior`, or `behavior`) is heavyweight, while one that does not use such a keyword is lightweight.

## 2.4 Privacy Modifiers and Visibility

Java code that is not within an annotation uses the usual access control rules for determining visibility (or accessibility) of Java [Arnold-Gosling-Holmes00] [Gosling-etal00]. That is, a name declared in package $P$ and type $P.T$ may be referenced from outside $P$ only if it is declared as `public`, or if it is declared as `protected` and the reference occurs within a subclass of $P.T$. This name may be referenced from within $P$ but outside of $P.T$ only if it is declared as `public`, default access, or `protected`. Such a name may always be referenced from within $P.T$, even if it is declared as `private`. See the Java language specification [Gosling-etal00] for details on visibility rules applied to nested and inner classes.

Within annotations, JML imposes some extra rules in addition to the usual Java visibility rules [Leavens-Baker-Ruby06] [Leavens-Mueller07]. These rules depend not just on the declaration of the name but also on the visibility level of the context that is referring to the name in question. For purposes of this section, the *annotation context* of a reference to a name is the smallest grammatical unit with an attached (or implicit) visibility. For example, this annotation context can be a method specification case, an invariant, a history constraint, or a field declaration. The visibility level of such an annotation context can be `public`, `protected`, `private`, or default (package) visibility.

The JML rule, in essence, is that an annotation context cannot refer to names that are more hidden than the context's own visibility. That is, for a reference to a name $x$ to be legal, the visibility of the annotation context that contains the reference to $x$ must be at least as permissive as the declaration of $x$ itself. The reason for this restriction is that the people who are allowed to see the annotation should be able to see each of the names used in that annotation [Meyer97], otherwise they might not understand it. For example, public clients should be able to see all the declarations of names in publicly visible annotations, hence public annotations should not contain protected, default access, or private names.

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

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

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

```
public class PrivacyDemoLegalAndIllegal {
   public int pub;
   protected int prot;
   int def;
   private int priv;

   //@ public invariant pub > 0;      // legal
   //@ public invariant prot > 0;     // illegal!
   //@ public invariant def > 0;      // illegal!
   //@ public invariant priv < 0;     // illegal!

   //@ protected invariant pub > 1;   // legal
   //@ protected invariant prot > 1;  // legal
   //@ protected invariant def > 1;   // illegal!
   //@ protected invariant priv < 1;  // illegal!

   //@ invariant pub > 1;              // legal
   //@ invariant prot > 1;             // legal
   //@ invariant def > 1;              // legal
   //@ invariant priv < 1;             // illegal!

   //@ private invariant pub > 1;      // legal
   //@ private invariant prot > 1;     // legal
   //@ private invariant def > 1;      // legal
   //@ private invariant priv < 1;     // legal
}
```

Note that in a lightweight method specification, the privacy level is assumed to be the same privacy level as the method itself. That is, for example, a protected method with a lightweight method specification is considered to be a protected annotation context for purposes of checking proper visibility usage [Leavens-Baker-Ruby06] [Mueller02]. See Section 2.3 [Lightweight and Heavyweight Specifications], page 12, for more about the differences between lightweight and heavyweight specification cases.

The ESC/Java2 system has the same visibility rules as described above. (However, this was not true of the old version of ESC/Java [Leino-Nelson-Saxe00].)

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

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

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

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

```
//@ public model int length;
private int _length; //@ in length;
//@ private represents length <- _length;
```

The above desugaring allows one to change the underlying field without affecting the readers of the specification.

The desugaring of `spec_protected` is the same as for `spec_public`, except that one uses `protected` instead of `public` in the desugared form.

## 2.5 Instance vs. Static

In Java, a feature of a class or interface may declared to be `static`. This means that the feature is not part of instances of that type, and it means that references to that feature (from outside the type and its subtypes) must use a qualified name of the form *T.f*, which refers to the static feature *f* in type *T*.

A feature, such as a field or method, of a type that is not static is an *instance* feature. For example, in a Java interface, all the methods declared are instance methods, although fields are static by default. In a Java class the default is that all features are instance features, unless the modifier `static` is used.

In JML declarations follow the normal Java rules for determining whether they are instance or static features of a type. However, within annotations it is possible to explicitly label features as `instance` (see Chapter 6 [Type Definitions], page 37 for the syntax). The use of the `instance` modifier is necessary to declare model and ghost instance fields in interfaces, since otherwise the Java default modifier for fields in interfaces (static) would apply.

It is also useful, in JML, to label invariants as either static or instance invariants. See Section 8.2.1 [Static vs. instance invariants], page 54, for more on this topic.

## 2.6 Locations and Aliasing

A *location* is a field of an object or a local variable. A *local variable* is either a variable declared inside a method or a formal parameter of a method.

An *access path* is an expression either of the form $x$, where $x$ is an identifier, or $p\,.\,x$, where $p$ is an access path and $x$ is an identifier.[1] (In forming an access path, we ignore visibility.)

In a given program state, $s$, a location $l$ is *aliased* if there are two or more access paths that, in $s$, both denote $l$. The access paths in question are said to be *aliases* for $l$. Similarly, we say that an object $o$ is aliased in a state $s$ if there are two access paths that, in $s$, both have $o$ as their value. In Java, it is impossible to alias local variables, so the only aliasing possible involves objects and their fields.

## 2.7 Expression Evaluation and Undefinedness

Within JML annotations, Java expressions have the values that are defined in the Java Language Specification [Gosling-etal00]. This has consequences on the interpretation of assertion expressions [Chalin07] [Rioux-Chalin07]: an assertion is taken to be valid if and only if its interpretation

- does not cause an exception to be raised, and
- yields the value true.

Note that this interpretation of assertions, said to be based on "strong validity" [Chalin07], was made the default assertion semantics for JML in 2007. Prior to that, assertions were interpreted using a classical defintion of validity [Leavens-etal05] [Leavens-Baker-Ruby06] [Gries-Schneider95] [Jones95e].

The strong validity semantics for assertion evaluation means that exceptions may arise during evaluation of subexpressions within assertions. These exceptions should be avoided by the specifier and tools are encouraged to warn users when they detect that an exception may arise during assertion evalution.

To avoid exceptions during assertion evaluation, specifiers should practice good Java coding habits, and write specifications that prevent such exceptions. To do this, one can use left-to-right ordering of evaluation of subexpressions and the short-curcuit nature of the Java operators `&&` and `||`. JML also evaluates the its two implication operators, `==>` and `<==` in short-curcuit fashion. Within a specification case, the precondition can protect the rest of the specification from exceptions [Leavens-Wing98]. That is, one can assume that the precondition holds in the remainder of the clauses in a specification case. JML also evaluates multiple occurrences of clauses of the same kind (such as `requires` or `ensures`) within a spec case in top to bottom order, so earlier clauses can protect later ones, just as if they were combined with `&&`.

## 2.8 Null is Not the Default

One common problem that occurs in Java and JML specifications is the possibility of `null` dereferences. For example, if `x` is `null` then `x.f` and `x.m()` both result in a

[1] By an identifier, we technically mean an *ident* in the Java grammar. See Section 4.6 [Tokens], page 29, for details.

`NullPointerException`. Such null pointer exceptions cause undefinedness in expression evaluation, as described above (see Section 2.7 [Expression Evaluation and Undefinedness], page 15).

To avoid having to constantly specify that declarations (other than local variables) are non-null, JML makes them implicitly `non_null` by default. That is, unless a

- member field (see Section 7.1.2 [Field and Variable Declarations], page 47),
- formal parameter, (see Section 7.1.1.1 [Formal Parameters], page 44),
- method return type (see Section 7.1.1 [Method and Constructor Declarations], page 43), or
- bound variable (see Section 11.4.24.5 [Modifiers for Bound Variables], page 101)

is explicitly annotated with the modifier `nullable`, that declaration is assumed to be `non_null`.

For a field whose type is an array of reference types, such as a field of type `Object[]`, both the field that refers to the array and the elements of the array are `non_null` by default. If a field whose type is an array of reference types is declared as `nullable`, then both the reference to the array and all of its elements may potentially be null. To specify that the field is not null but the elements may be null, use an invariant to state that the field cannot contain null, as follows.

```
private /*@ spec_public nullable @*/ Object[] a;
//@ public invariant a != null;
```

While these defaults differ from Java, research has found that in most cases a declaration is expected to be non-null [Chalin-Rioux05]. More importantly, since one of the most common mistakes in JML specifications (and in Java programs) is forgetting to specify that a declaration is non-null, making the default be that they cannot hold null helps eliminate a source of common errors in specifications.

See Section 6.2.12 [Nullity Modifiers], page 42, for more details on the nullity modifiers.

## 2.9 Language Levels

One of JML's goals is to provide a single language that can be used with a variety of different tools. However, JML is also an evolving language that is used as a research vehicle by many groups. The evolution of JML means that some features are not completely documented or implemented. Use of JML in research means that some tools will have features that are not supported by other tools. All of this has the potential to threaten portability and to make JML more difficult to learn and use.

The research groups working on JML are committed to making these problems as invisible to non-researchers as possible, and for this reason have defined several *language levels*. The goal of defining these language levels is to make it easier to learn and use JML and its various tools.

We define the following language levels.[2]

- Level 0 should be supported by all JML tools and constitutes the heart of JML. All users should be familiar with these level 0 features. They are fundamental to all uses

[2] Thanks to Patrice Chalin for pushing to define these. Patrice, Joe Kiniry, Peter Müller, Adam Darvas, and David Naumann participated in the initial discussions about what should be in each level.

of JML, including its use as a design by contract language, as documentation, and as formal specification for formal verification efforts. Thus the level 0 features should be the ones that tutorial materials concentrate on. Users should be able to count on these features being understood and checked by all tools.

- Level 1 should be supported by most JML tools and should be a first priority for developers after implementing the Level 0 features. There are three categories of features that level 1 adds to level 0. The first is the redundancy features of JML, which are useful in documentation, but not absolutely vital. The second is features that are sugars for features present in level 0. The third is various features for which modular static verification is still problematic, although a runtime assertion checking semantics has been implemented. This includes the use of methods and constructor calls in assertions.
- Level 2 contains features that are more specialized to particular uses of JML, but are still useful for several different tools. It also contains some features that are mainly needed to explain JML's semantics, and have not been heavily used (so far).
- Level 3 features are even less commonly used and more exotic features. The semantics of some of these features are not yet well understood, and the features are not implemented by many tools.
- Level C contains features related to specification and verification of concurrent Java programs. Some of these are from ESC/Java [Leino-Nelson-Saxe00], and others are from [Rodriguez-etal05].
- Level X contains experimental features, which may eventually be moved to other levels. Many tools will have other experimental features not documented here.

When learning JML, one should focus on levels 0 features first, as these form the heart of the language which should be understood by all JML tools. Features at level 1 are next in importance and should be supported by most tools that are interested in having a large user base. Features at higher levels are less important and may not be present in all tools. Users should feel free to ignore them unless they meet some specific need.

The language levels also provide guidance for tool designers. JML tools should parse all of the syntax in this reference manual that is not marked as experimental. This is the most important way to guarantee portability for users, and the easiest way for tools to get feedback. In addition, tools should check at least level 0, and preferably level 1 features. Features at levels 2 and 3 are candidates for the tool to just parse and ignore, if they are not features of interest for that tool. Experimental features may ignored (or added) by any tool.

Many tool developers may want to start off supporting only a subset of JML defined by level 0 and then move on to higher levels.

It is also suggested that tools give users optional feedback, perhaps in a verbose mode, as to which features are fully and partially supported. Clearly stating which JML levels are supported in a tool release is also very important.

More details are provided in the subsections below.

### 2.9.1 Level 0 Features

The features in this level form the core of JML and should be understood and checked by all JML tools. Beginning users should pay the most attention to these features. These features include all of Java and the syntax described in the rest of this section.

Many, but not all, of the JML additions to Java's *modifiers* (see Section 6.2 [Modifiers], page 39) are level 0 features. The following modifiers are included in level 0.

- The *modifier* `spec_public` (see Section 6.2.2 [Spec Public], page 40).
- The *modifier* `spec_protected` (see Section 6.2.3 [Spec Protected], page 40).
- The *modifier* `instance` (see Section 6.2.7 [Instance], page 41).
- The *modifier* `model` (see Section 6.2.5 [Model], page 41), as applied to field declarations (see Section 7.1.2.1 [JML Modifiers for Fields], page 47). Note that this modifier as applied to other declarations is not a level 0 feature.
- The *modifier* `ghost` (see Section 6.2.6 [Ghost], page 41), as applied to both field and variable declarations (see Section 7.1.2 [Field and Variable Declarations], page 47).
- The *modifier* `helper` (see Section 6.2.8 [Helper], page 41).

Type specifications (see Chapter 8 [Type Specifications], page 50) are a level 0 feature, although not all clauses and features of type specifications are level 0. The following type-level clauses are included in level 0.

- Object invariants, that is an *invariant* (see Section 8.2 [Invariants], page 50) that is either written in an interface using the *modifier* `instance` (see Section 6.2.7 [Instance], page 41) or one that is written in a class and that does not use the *modifier* `static` (see Section 8.2.1 [Static vs. instance invariants], page 54).
- The functional form of a *represents-clause* (see Section 8.4 [Represents Clauses], page 58). That is, a represents clause that uses *l-arrow-or-eq* and (not `\such_that`).
- The *initially-clause* (see Section 8.5 [Initially Clauses], page 59).
- The *type-spec* `\TYPE` (optionally, as a type of array element). See Section 7.1.2.2 [Type-Specs], page 48, for more details.

Method specifications (see Chapter 9 [Method Specifications], page 61) are a level 0 feature. This includes the ability to combine specification cases using `also` (see Section 9.6.5 [Semantics of nested behavior specification cases], page 69) and specification inheritance [Dhara-Leavens96] [Leavens-Naumann06] [Leavens06b]. It also includes the use of `\not_specified` for all specification clauses that are at level 0. However, not all clauses and features of method specifications are level 0. The following parts of method specifications are included in level 0. Redundancy features of method specifications are only present at level 1, not at level 0. The details are described below.

- Lightweight specification cases (see Section 9.4 [Lightweight Specification Cases], page 63), although not all clauses that are allowed in the syntax are in level 0.
- Heavyweight specification cases (see Section 9.5 [Heavyweight Specification Cases], page 65) that do not use the keyword `code`. This includes *behavior-spec-case* (see Section 9.6 [Behavior Specification Cases], page 65), *normal-behavior-spec-case* (see Section 9.7 [Normal Behavior Specification Cases], page 70), and *exceptional-behavior-spec-case* (see Section 9.8 [Exceptional Behavior Specification Cases], page 71). However, note that not all clauses that are allowed in the syntax are in level 0.
- The *requires-clause* (see Section 9.9.2 [Requires Clauses], page 74). The redundant form of this clause (`requires_redundantly`, `pre_redundantly`) is a level 1 feature.
- The *ensures-clause* (see Section 9.9.3 [Ensures Clauses], page 74). The redundant form of this clause (`ensures_redundantly`, `post_redundantly`) is a level 1 feature.

- The *signals-clause* (see Section 9.9.4 [Signals Clauses], page 75). The redundant form of this clause (`signals_redundantly`, `exsures_redundantly`) is a level 1 feature.
- The *signals_only-clause* (see Section 9.9.5 [Signals-Only Clauses], page 77). The redundant form of this clause (`signals_only_redundantly`) is a level 1 feature.
- The *assignable-clause* (see Section 9.9.9 [Assignable Clauses], page 80). The redundant form of this clause (`assignable_redundantly`, `modifiable_redundantly`, `modifies_redundantly`) is a level 1 feature.

Only static data groups (see Chapter 10 [Data Groups], page 85) are part of level 0.

- The *in-group-clause* (see Section 10.1 [Static Data Group Inclusions], page 85) kind of *jml-data-group-clause* that attaches to field declarations (see Section 7.1.2 [Field and Variable Declarations], page 47).

Some of JML's extensions to Java's *expression* syntax (see Chapter 11 [Predicates and Specification Expressions], page 87), but not all of them, can be used at level 0. Note that calls to pure methods and constructors in *spec-expression*s are *not* part of level 0, but are only found at level 1. We describe the level 0 specification expressions below.

- The *result-expression* (see Section 11.4.1 [Backslash result], page 90).
- The *old-expression* (see Section 11.4.2 [Backslash old and Backslash pre], page 90).
- The *fresh-expression* (see Section 11.4.9 [Backslash fresh], page 94).
- The *nonnullelements-expression* (see Section 11.4.14 [Backslash nonnullelements], page 96).
- The *informal-description* (see Section 11.4.15 [Informal Predicates], page 96).
- The *typeof-expression* (see Section 11.4.16 [Backslash typeof], page 96).
- The *elemtype-expression* (see Section 11.4.17 [Backslash elemtype], page 97).
- The *type-expression* (see Section 11.4.18 [Backslash type], page 97).
- The *spec-quantified-expr* (see Section 11.4.24 [Quantified Expressions], page 98) forms that use the *quantifier* keywords `\forall` and `\exists` (see Section 11.4.24.1 [Universal and Existential Quantifiers], page 99).

  (The *quantifier* keywords `\max`, `\min`, `\product`, and `\sum` (see Section 11.4.24.2 [Generalized Quantifiers], page 99), as well as `\num_of` (see Section 11.4.24.3 [Numerical Quantifier], page 100, are all level 1 features.)
- The `<:` operator (see Section 11.6.1 [Subtype operator], page 102).
- The `<==>` and `<=!=>` operators (see Section 11.6.2 [Equivalence and Inequivalence Operators], page 102).
- The `==>` and `<==` operators (see Section 11.6.3 [Forward and Reverse Implication Operators], page 102).
- The syntax for *store-ref*s (see Section 11.7 [Store Refs], page 103).

All of the Java statements and most of the JML extensions for adding assertions to statements and annotation statements (see Chapter 12 [Statements and Annotation Statements], page 104) are at level 0. But redundancy features of the JML extensions are only present at level 1, not at level 0. We describe the level 0 extension to statements below.

- Using the *modifier* `ghost` in *local-declaration*s (see Section 12.1.1 [Modifiers for Local Declarations], page 105).

- The *possibly-annotated-loop* statement (see Section 12.2 [Loop Statements], page 105), with a *loop-invariant* (see Section 12.2.1 [Loop Invariants], page 107). The redundant forms of *loop-invariant*s, namely those that use the keywords `maintaining_redundantly` and `loop_invariant_redundantly` are level 1 features. Furthermore, the *variant-function* is a level 1 feature.
- The *assert-statement* (see Section 12.3 [Assert Statements], page 109). Note that the *assert-redundantly-statement*, which uses the keyword `assert_redundantly`, is in level 1.
- The non-redundant form of the *assume-statement* (see Section 12.4.1 [Assume Statements], page 110). Use of the keyword `assume_redundantly` is a level 1 feature.
- The *set-statement* (see Section 12.4.2 [Set Statements], page 110).

The ability to use a `.spec` file (see Section 16.1 [File Name Suffixes], page 124) to give a separate specification for a compilation unit that only appears in binary form (e.g., in a `.class` file) is a level 0 feature. Use of the *refine-prefix* (see Chapter 16 [Refinement], page 124) is a level 1 feature.

Some syntax from the Universe type system (see Chapter 18 [Universe Type System], page 132) is included in level 0. However, `readonly` is considered to be in level X, as is the semantics of the Universe type system. The `rep` and `peer` modifiers are included in level 0 because, in some form, they are important to the semantics of several level 0 features [Mueller-Poetzsch-Heffter-Leavens03] [Mueller-Poetzsch-Heffter-Leavens06].

- The `\rep` and `rep` *ownership-modifier*s (see Section 18.2 [Rep and Peer], page 133).
- The `\peer` and `peer` *ownership-modifier*s (see Section 18.2 [Rep and Peer], page 133).

### 2.9.2 Level 1 Features

The features in this level will be understood and checked by many JML tools. They are quite important in practice, especially the use of methods and constructors in writing the specifications of other methods and constructors. Also useful are all of JML's redundancy features (see Chapter 13 [Redundancy], page 113), which are included here for level 0 features and for other features at level 1.

The following additions to Java's *modifier*s (see Section 6.2 [Modifiers], page 39) are level 1 features.

- Method or constructor declarations that use the *modifier* `model` (see Section 7.1.1.2 [Model Methods and Constructors], page 44). However, note that using `model` on a field declarations is a level 0 feature and that using `model` on a type declaration is a level 2 feature.
- *import-definition*s that use the modifier `model` (see Section 5.2 [Import Definitions], page 36).
- The *modifier* `pure` (see Section 6.2.4 [Pure], page 41).
- The *modifier* `uninitialized` (see Section 6.2.10 [Uninitialized], page 42).

The following type-level clauses (see Chapter 8 [Type Specifications], page 50) are included in level 1.

- Attaching a *method-specification* to a *class-initializer-decl* (see Section 7.2 [Class Initializer Declarations], page 48).

- Static invariants, that is an *invariant* (see Section 8.2 [Invariants], page 50) that is either written in an interface without using the *modifier* `instance` (see Section 6.2.7 [Instance], page 41), or one that is written in a class and that uses the *modifier* `static` (see Section 8.2.1 [Static vs. instance invariants], page 54).
- Both object and static *history-constraints* (see Section 8.3 [Constraints], page 55).
- The *axiom-clause* (see Section 8.6 [Axioms], page 59).
- The *maps-into-clause* (see Section 10.2 [Dynamic Data Group Mappings], page 86) kind of *jml-data-group-clause* that attaches to field declarations (see Section 7.1.2 [Field and Variable Declarations], page 47).

The following features of method specifications (see Chapter 9 [Method Specifications], page 61) are included in level 1.

- The *spec-var-decls* that may occur in a specification case (see Section 9.9.1 [Specification Variable Declarations], page 73).
- The *redundant-spec* parts of a method specification (see Chapter 13 [Redundancy], page 113) are also included in level 1. The following describes these parts.
  - The *implications* (`implies_that`) part of a *redundant-spec* (see Section 13.1 [Redundant Implications and Redundantly Clauses], page 113).
  - The *examples* (`for_example`) part of a *redundant-spec*.

The following extensions to Java's *expression* syntax (see Chapter 11 [Predicates and Specification Expressions], page 87) are included in level 1.

- The *spec-quantified-expr* (see Section 11.4.24 [Quantified Expressions], page 98) forms that use the *quantifier* keywords `\max`, `\min`, `\product`, and `\sum` (see Section 11.4.24.2 [Generalized Quantifiers], page 99), as well as `\num_of` (see Section 11.4.24.3 [Numerical Quantifier], page 100).

  (Note that the `\max` quantifier is distinct from the *max-expression* (see Section 11.4.20 [Backslash max], page 97), which is a level C feature. Also, note that the *quantifier* keywords `\forall` and `\exists` are level 0 features.)
- Calls to pure methods and constructors (see Section 7.1.1.3 [Pure Methods and Constructors], page 44) in *spec-expressions* (see Chapter 11 [Predicates and Specification Expressions], page 87).
- The *set-comprehension* expression (see Section 11.5 [Set Comprehensions], page 101).

The following additions to Java's statement syntax (see Chapter 12 [Statements and Annotation Statements], page 104) are included in level 1.

- The use of redundant forms of *loop-invariants* (see Section 12.2.1 [Loop Invariants], page 107) namely those that use the keywords `maintaining_redundantly` and `loop_invariant_redundantly`. Non-redundant *loop-invariants* are in level 0.
- The *possibly-annotated-loop* statement (see Section 12.2 [Loop Statements], page 105), with a *variant-function* (see Section 12.2.2 [Loop Variant Functions], page 108).
- The *assert-redundantly-statement* (see Section 12.3 [Assert Statements], page 109); that is, assert statements that use the keyword `assert_redundantly`. The non-redundant *assert-statements* are a level 0 feature.

- The redundant form of the *assume-statement* (see Section 12.4.1 [Assume Statements], page 110); that is, assume statements that use the keyword `assume_redundantly`. The non-redundant *assume-statements* are a level 0 feature.

The *refine-prefix* (see Chapter 16 [Refinement], page 124). However, the ability to use a `.spec` file to give a separate specification for a compilation unit that only appears in binary form (e.g., in a `.class` file) is a level 0 feature.

The `\bigint` type (see Section 19.1 [Backslash bigint], page 139) from the safe math extensions (see Chapter 19 [Safe Math Extensions], page 139) is a level 1 feature.

### 2.9.3 Level 2 Features

Level 2 contains features that are more specialized to particular uses of JML, but are still useful for several different tools. It also contains some features that are mainly needed to explain JML's semantics, and have not been heavily used (so far).

The *nowarn-pragma* (see Section 4.2 [Lexical Pragmas], page 26).

The following type-level clauses (see Chapter 8 [Type Specifications], page 50) are included in level 2.

- The relational form of a *represents-clause* (see Section 8.4 [Represents Clauses], page 58). That is, a represents clause that uses `\such_that`. Note that the functional form of such represents clauses is a level 0 feature.
- The *readable-if-clause* clause (see Section 8.7 [Readable If Clauses], page 59).
- The *writable-if-clause* clause (see Section 8.8 [Writable If Clauses], page 59).

The following features of method specifications (see Chapter 9 [Method Specifications], page 61) are included in level 2.

- The *diverges-clause* (see Section 9.9.7 [Diverges Clauses], page 79).
- The *accessible-clause* (see Section 9.9.10 [Accessible Clauses], page 81).
- The *callable-clause* (see Section 9.9.11 [Callable Clauses], page 82).
- The *measured-by-clause* (see Section 9.9.12 [Measured By Clauses], page 82).
- The *captures-clause* (see Section 9.9.13 [Captures Clauses], page 82).
- The *working-space-clause* (see Section 9.9.14 [Working Space Clauses], page 83).
- The *duration-clause* (see Section 9.9.15 [Duration Clauses], page 83).
- The *model-program* style of method specification (see Chapter 14 [Model Programs], page 117).
- The *refining-statement* (see Section 12.4.3 [Refining Statements], page 110).
- The `extract` modifier (see Section 14.2 [Extracting Model Program Specifications], page 119).

The following extensions to Java's *expression* syntax (see Chapter 11 [Predicates and Specification Expressions], page 87) are included in level 2.

- The *not-assigned-expression* (see Section 11.4.3 [Backslash not_assigned], page 92).
- The *not-modified-expression* (see Section 11.4.4 [Backslash not_modified], page 92).
- The *only-accessed-expression* (see Section 11.4.5 [Backslash only_accessed], page 93).
- The *only-assigned-expression* (see Section 11.4.6 [Backslash only_assigned], page 93).

- The *only-called-expression* (see Section 11.4.7 [Backslash only_called], page 94).
- The *only-captured-expression* (see Section 11.4.8 [Backslash only_captured], page 94).
- The *reach-expression* (see Section 11.4.10 [Backslash reach], page 95).
- The *is-initialized-expression* (see Section 11.4.21 [Backslash is_initialized], page 98).
- The *invariant-for-expression* (see Section 11.4.22 [Backslash invariant_for], page 98).
- The *lblneg-expression* and the *lblpos-expression* (see Section 11.4.23 [Backslash lblneg and lblpos], page 98).

The following additions to Java's statement syntax (see Chapter 12 [Statements and Annotation Statements], page 104) are included in level 2.

- The *unreachable-statement* (see Section 12.4.4 [Unreachable Statements], page 111).
- The *debug-statement* (see Section 12.4.5 [Debug Statements], page 112)
- The *hence-by-statement* (see Section 12.4.6 [Hence By Statements], page 112).

Note that all the *model-prog-statements* (see Chapter 14 [Model Programs], page 117) are at level 2, because the model program style of method specification is at this level.

Aside from the `\bigint` type (see Section 19.1 [Backslash bigint], page 139), which is a level 1 feature, the rest of the safe math extensions (see Chapter 19 [Safe Math Extensions], page 139) are level 2 features. This includes the following particulars.

- The `\real` type (see Section 19.2 [Backslash real], page 139).
- The *modifiers* `code_bigint_math`, `code_java_math`, `code_safe_math`, `spec_bigint_math`, `spec_java_math`, and `spec_safe_math` (see Section 6.2.11 [Math Modifiers], page 42).

### 2.9.4 Level 3 Features

Level 3 features are more exotic and even less commonly used. The semantics of some of these features are not yet well understood, and the features are not implemented by many tools.

- *type-definitions* that use the modifier `model` (see Section 6.1.2 [Modifiers for Type Definitions], page 38).
- The *duration-expression* (see Section 11.4.11 [Backslash duration], page 95).
- The *space-expression* (see Section 11.4.12 [Backslash space], page 95).
- The *working-space-expression* (see Section 11.4.13 [Backslash working space], page 96).

### 2.9.5 Level C Features

The features in this level are related to the specification of concurrency. This includes features inherited from ESC/Java having to do with concurrency. The features of this level are as follows.

- The *monitors-for-clause* clause (see Section 8.9 [Monitors For Clause], page 60).
- The *when-clause* (see Section 9.9.8 [When Clauses], page 80).
- The *lockset-expression* (see Section 11.4.19 [Backslash lockset], page 97).
- The *max-expression* (see Section 11.4.20 [Backslash max], page 97). Note that this is *not* the quantifier `\max` (see Section 11.4.24.2 [Generalized Quantifiers], page 99), which is a level 1 feature.

- The `<` and `<=` operators applied to test ordering of locks (see Section 11.6.4 [Lockset Ordering], page 103).

### 2.9.6 Level X Features

The features in this level are experimental. They are as follows.

- The MultiJava extensions to JML (see Chapter 17 [MultiJava Extensions to JML], page 131), including the syntax for *multijava-top-level-declaration* (see Section 17.1 [Augmenting Method Declarations], page 131) and *multijava-param-declaration* (see Section 17.2 [MultiMethods], page 131).
- The `\readonly` and `readonly` *ownership-modifiers* from the Universe type system (see Chapter 18 [Universe Type System], page 132). Note that the `\peer` and `\rep` modifiers are level 0 features.

# 3 Syntax Notation

We use an extended Backus-Naur Form (BNF) grammar to describe the syntax of JML. The extensions are as follows [Ledgard80].

- Nonterminal symbols are written as follows: *nonterminal*. That is, nonterminal symbols appear in an *italic* font (in the printed manual).
- Terminal symbols are written as follows: `terminal`. In a few cases it is also necessary to quote terminal symbols, such as when using '`|`' as a terminal symbol instead of a meta-symbol.
- Square brackets ([ and ]) surround optional text. Note that `[` and `]` are terminals.
- The notation ... means that the preceding nonterminal or group of optional text can be repeated zero (0) or more times.

For example, the following gives a production for a non-empty list of *init-declarator*s, separated by commas.

*init-declarator-list* ::= *init-declarator* [ `,` *init-declarator* ] ...

To remind the reader that the notation '...' means zero or more repetitions, we try to use '...' only following optional text, although, in cases such as the following, the brackets could have been omitted.

*modifiers* ::= [ *modifier* ] ...

As in the above examples, we follow the C++ standard's conventions [ANSI95] in using nonterminal names of the form *X-list* to mean a comma-separated list, and nonterminal names of the form *X-seq* to mean a sequence not separated by commas. An example of a sequence is the following

*spec-case-seq* ::= *spec-case* [ `also` *spec-case* ] ...

We use "//" to start a comment (to you, the reader) in the grammar.

A complete summary of the JML grammar appears in an appendix (see Appendix A [Grammar Summary], page 141). When reading the HTML version of this appendix, one can click on the names of nonterminals to bring that nonterminal's definition to the top of the browser's window. This is helpful when dealing with such a large grammar.

Another help in dealing with the grammar is to use the index (see [Index], page 179). Every nonterminal and terminal symbol in the grammar is found in the index, and each definition and use is noted.

# 4 Lexical Conventions

This chapter presents the lexical conventions of JML, that is, the microsyntax of JML.

Throughout this chapter, grammatical productions are to be understood lexically. That is, no *white-space* (see Section 4.1 [White Space], page 26) may intervene between the characters of a token. (However, outside this chapter, the opposite of this convention is in force.)

The microsyntax of JML is described by the production *microsyntax* below; it describes what a program looks like from the point of view of a lexical analyzer [Watt91].

*microsyntax* ::= *lexeme* [ *lexeme* ] . . .
*lexeme* ::= *white-space* | *lexical-pragma* | *comment*
        | *annotation-marker* | *doc-comment* | *token*
*token* ::= *ident* | *keyword* | *special-symbol*
        | *java-literal* | *informal-description*

In the rest of this section we provide more details on each of the major nonterminals used in the above grammar.

## 4.1 White Space

Blanks, horizontal and vertical tabs, carriage returns, formfeeds, and newlines, collectively called *white space*, are ignored except as they serve to separate tokens. Newlines and carriage returns are special in that they cannot appear in some contexts where other whitespace can appear, and are also used to end Java-style comments (see Section 4.3 [Comments], page 27).

*white-space* ::= *non-nl-white-space* | *end-of-line*
*non-nl-white-space* ::= a blank, tab, or formfeed character
*end-of-line* ::= *newline* | *carriage-return*
        | *carriage-return newline*
*newline* ::= a newline character
*carriage-return* ::= a carriage return character

## 4.2 Lexical Pragmas

ESC/Java [Leino-etal00] has a single kind of "lexical pragma", `nowarn`, whose syntax is described below in general terms. The JML checker currently ignores these lexical pragmas, but `nowarn` is only recognized within an annotation. Note that, unlike ESC/Java, the semicolon is mandatory. This restriction seems to be necessary to prevent lexical ambiguity.

*lexical-pragma* ::= *nowarn-pragma*
*nowarn-pragma* ::= `nowarn` [ *spaces* ] [ *nowarn-label-list* ] `;`
*spaces* ::= *non-nl-white-space* [ *non-nl-white-space* ] . . .
*nowarn-label-list* ::= *nowarn-label* [ *spaces* ]
        [ `,` [ *spaces* ] *nowarn-label* [ *spaces* ] ] . . .
*nowarn-label* ::= *letter* [ *letter* ] . . .

See Section 4.6 [Tokens], page 29, for the syntax of *letter*.

## 4.3 Comments

Both kinds of Java comments are allowed in JML: multiline C-style comments and single line C++-style comments. However, if what looks like a comment starts with the at-sign (`@`) character, or with a plus sign and an at-sign (`+@`), then it is considered to be the start of an annotation by JML, and not a comment. Furthermore, if what looks like a comment starts with an asterisk (`*`), then it is a documentation comment, which is parsed by JML.

*comment* ::= *C-style-comment* `|` *C++-style-comment*
*C-style-comment* ::= `/*` [ *C-style-body* ] *C-style-end*
*C-style-body* ::= *non-at-plus-star* [ *non-stars-slash* ] . . .
  `|` `+` *non-at* [ *non-stars-slash* ] . . .
  `|` *stars-non-slash* [ *non-stars-slash* ] . . .
*non-stars-slash* ::= *non-star*
  `|` *stars-non-slash*
*stars-non-slash* ::= `*` [ `*` ] . . . *non-star-slash*
*non-at-plus-star* ::= any character except `@`, `+`, or `*`
*non-at* ::= any character except `@`
*non-star* ::= any character except `*`
*non-slash* ::= any character except `/`
*non-star-slash* ::= any character except `*` or `/`
*C-style-end* ::= [ `*` ] . . . `*/`
*C++-style-comment* ::= `//` [ `+` ] *end-of-line*
  `|` `//` *non-at-plus-end-of-line* [ *non-end-of-line* ] . . . *end-of-line*
  `|` `//+` *non-at-end-of-line* [ *non-end-of-line* ] . . . *end-of-line*
*non-end-of-line* ::= any character except a newline or carriage return
*non-at-plus-end-of-line* ::= any character except `@`, `+`, newline, or carriage return
*non-at-end-of-line* ::= any character except `@`, newline, or carriage return

## 4.4 Annotation Markers

If what looks to Java like a comment starts with an at-sign (`@`) as its first character, then it is not considered a comment by JML. We refer to the tokens between `//@` and the following *end-of-line*, and between pairs of annotation start ( `/*@` or `/*+@` ) and end ( `*/` or `@*/` or `@+*/` ) markers as *annotations*.

Annotations must hold entire grammatical units of JML specifications, in the sense that the text of some nonterminals may not be split across two separate annotations. For example the following is illegal, because the *postcondition* of the ensures clause is split over two annotations, and thus each contains a fragment instead of a complete grammatical unit.

```
//@ ensures 0 <= x              // illegal!
//@      && x < a.length;
```

Implementations are not required to check for such errors. However, note that ESC/Java [Leino-Nelson-Saxe00] and ESC/Java2 assume that nonterminals that define clauses are not split into separate annotations, and so effectively do check for them.

Annotations look like comments to Java, and are thus ignored by it, but they are significant to JML. One way that this can be achieved is by having JML drop (ie., ignore) the character sequences that are *annotation-markers*: `//@`, `//+@`, `/*@`, `/*+@`, and `@+*/`, `@*/`.

The at-sign (`@`) in `@*/` is optional, and more than one at-sign may appear in it and the other annotation markers. However, JML will recognize *jml-keywords* only within annotations.

Within annotations, on each line, initial white-space and any immediately following at-signs (`@`) are ignored. The definition of an annotation marker is given below.

> *annotation-marker* ::= `//@` [ `@` ] . . . `|` `//+@` [ `@` ] . . .
> `|` `/*@` [ `@` ] . . . `|` `/*+@` [ `@` ] . . . `|` [ `@` ] . . . `@+*/` `|` [ `@` ] . . . `*/`
> *ignored-at-in-annotation* ::= `@`

## 4.5 Documentation Comments

If what looks like a C-style comment starts with an asterisk (`*`) then it is a *documentation comment*. The syntax is given below. The syntax *doc-comment-ignored* is used for documentation comments that are ignored by JML.

> *doc-comment* ::= `/**` [ `*` ] . . . *doc-comment-body* `*/`
> *doc-comment-ignored* ::= *doc-comment*

At the level of the rest of the JML grammar, a documentation comment that does not contain an embedded JML method specification is essentially described by the above, and the fact that a *doc-comment-body* cannot contain the two-character sequence `*/`.

However, JML and `javadoc` both pay attention to the syntax inside of these documentation comments. This syntax is really best described by a context-free syntax that builds on a lexical syntax. However, because much of the documentation is free-form, the context-free syntax has a lexical flavor to it, and is quite line-oriented. Thus it should come as no surprise that the first non-whitespace, non-asterisk (ie., not `*`) character on a line determines its interpretation.

> *doc-comment-body* ::= [ *description* ] . . .
> [ *tagged-paragraph* ] . . .
> [ *jml-specs* ] [ *description* ]
> *description* ::= *doc-non-empty-textline*
> *tagged-paragraph* ::= *paragraph-tag* [ *doc-non-nl-ws* ] . . .
> [ *doc-atsign* ] . . . [ *description* ] . . .
> *jml-specs* ::= *jml-tag* [ *method-specification* ] *end-jml-tag*
> [ *jml-tag* [ *method-specification* ] *end-jml-tag* ] . . .

The microsyntax or lexical grammar used within documentation comments is as follows. Note that the token *doc-nl-ws* can only occur at the end of a line, and is always ignored within documentation comments. Ignoring *doc-nl-ws* means that any asterisks at the beginning of the next line, even in the part that would be a JML *method-specification*, are also ignored. Otherwise the lexical syntax within a *method-specification* is as in the rest of JML. This method specification is attached to the following method or constructor declaration. (Currently there is no useful way to use such specifications in the documentation comments for other declarations.) Note the exception to the grammar of *doc-non-empty-textline*.

*paragraph-tag* ::= `@author` | `@deprecated` | `@exception`
  | `@param` | `@return` | `@see`
  | `@serial` | `@serialdata` | `@serialfield`
  | `@since` | `@throws` | `@version`
  | `@` *letter* [ *letter* ] ...
*doc-atsign* ::= `@`
*doc-nl-ws* ::= *end-of-line*
  [ *doc-non-nl-ws* ] ... [ `*` [ `*` ] ... [ *doc-non-nl-ws* ] ... ]
*doc-non-nl-ws* ::= *non-nl-white-space*
*doc-non-empty-textline* ::= *non-at-end-of-line* [ *non-end-of-line* ] ...
*jml-tag* ::= `<jml>` | `<JML>` | `<esc>` | `<ESC>`
*end-jml-tag* ::= `</jml>` | `</JML>` | `</esc>` | `</ESC>`

A *jml-tag* marks the (temporary) end of a documentation comment and the beginning of text contributing to a method specification. The corresponding *end-jml-tag* marks the reverse transition. The *end-jml-tag* must match the corresponding *jml-tag*.

## 4.6 Tokens

Character strings that are Java reserved words are made into the token for that reserved word, instead of being made into an *ident* token. Within an *annotation* this also applies to *jml-keywords*. The details are given below.

*ident* ::= *letter* [ *letter-or-digit* ] ...
*letter* ::= `_`, `$`, `a` through `z`, or `A` through `Z`
*digit* ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
*letter-or-digit* ::= *letter* | *digit*

Several strings of characters are recognized as keywords or reserved words in JML. These fall into three separate categories: Java keywords, JML predicate keywords (which start with a backslash), and JML keywords. Java keywords are truly reserved words, and are recognized in all contexts. The nonterminal *java-reserved-word* represents the reserved words in Java (as in the JDK version 1.4).

The *jml-keywords* are only recognized as keywords when they occur within an annotation, but outside of a *spec-expression store-ref-list* or *constrained-list*. JML predicate keywords are also only recognized within annotations, but they are recognized only inside *spec-expressions*, *store-ref-lists*, and *constrained-lists*.

There are options to the JML tools that extend the language in various ways. When an option to parse the syntax for MultiJava [Clifton-etal00] is in turned on, the word `resend`, which is the only word in the nonterminal *multijava-reserved*, is recognized as a reserved word. It is thus recognized in all contexts. When this option is on, the *multijava-separators* (see below) are also recognized.

Similarly, when an option to parse the syntax for the Universe type system [Dietl-Mueller05] is used, the words listed in the nonterminal `java-universe-reserved` also act like reserved words in Java (and are thus recognized in all contexts). When an option to recognize the Universe system syntax in annotations is used, these words instead act as *jml-keywords* and are only recognized in annotations. However, even when no Universe options are used, `pure` is recognized as a keyword in annotations, since it is also a *jml-*

*keyword*. (The Universe type system support in JML is experimental. Most likely the list of `java-universe-reserved` will be added to the list of *jml-keywords* eventually.)

However, even without the Universe option being on, the `jml-universe-pkeyword` syntax is recognized within JML annotations in the same way as JML predicate keywords are recognized.

The details are given below.

```
keyword ::= java-reserved-word
        | jml-predicate-keyword | jml-keyword
java-reserved-word ::= abstract | assert
        | boolean | break | byte
        | case | catch | char
        | class | const | continue
        | default | do | double
        | else | extends | false
        | final | finally | float
        | for | goto | if
        | implements | import | instanceof
        | int | interface | long
        | native | new | null
        | package | private | protected
        | public | return | short
        | static | strictfp | super
        | switch | synchronized | this
        | throw | throws | transient
        | true | try | void
        | volatile | while
        | multijava-reserved      // When the MultiJava option is on
        | java-universe-reserved  // When the Universe option is on
multijava-reserved ::= resend
java-universe-reserved ::= peer | pure
        | readonly | rep
jml-predicate-keyword ::= \TYPE
        | \bigint | \bigint_math | \duration
        | \elemtype | \everything | \exists
        | \forall | \fresh
        | \into | \invariant_for | \is_initialized
        | \java_math | \lblneg | \lblpos
        | \lockset | \max | \min
        | \nonnullelements | \not_assigned
        | \not_modified | \not_specified
        | \nothing | \nowarn | \nowarn_op
        | \num_of | \old | \only_accessed
        | \only_assigned | \only_called
        | \only_captured | \pre
        | \product | \reach | \real
        | \result | \same | \safe_math
```

```
      | \space | \such_that | \sum
      | \typeof | \type | \warn_op
      | \warn | \working_space
      | jml-universe-pkeyword
jml-universe-pkeyword ::= \peer | \readonly | \rep
jml-keyword ::= abrupt_behavior | abrupt_behaviour
      | accessible | accessible_redundantly
      | also | assert_redundantly
      | assignable | assignable_redundantly
      | assume | assume_redundantly | axiom
      | behavior | behaviour
      | breaks | breaks_redundantly
      | callable | callable_redundantly
      | captures | captures_redundantly
      | choose | choose_if
      | code | code_bigint_math |
      | code_java_math | code_safe_math
      | constraint | constraint_redundantly
      | constructor | continues | continues_redundantly
      | decreases | decreases_redundantly
      | decreasing | decreasing_redundantly
      | diverges | diverges_redundantly
      | duration | duration_redundantly
      | ensures | ensures_redundantly | example
      | exceptional_behavior | exceptional_behaviour
      | exceptional_example
      | exsures | exsures_redundantly | extract
      | field | forall
      | for_example | ghost
      | helper | hence_by | hence_by_redundantly
      | implies_that | in | in_redundantly
      | initializer | initially | instance
      | invariant | invariant_redundantly
      | loop_invariant | loop_invariant_redundantly
      | maintaining | maintaining_redundantly
      | maps | maps_redundantly
      | measured_by | measured_by_redundantly
      | method | model | model_program
      | modifiable | modifiable_redundantly
      | modifies | modifies_redundantly
      | monitored | monitors_for | non_null
      | normal_behavior | normal_behaviour
      | normal_example | nowarn
      | nullable | nullable_by_default
      | old | or
      | post | post_redundantly
      | pre | pre_redundantly
```

```
        | pure | readable
        | refine | refines | refining
        | represents | represents_redundantly
        | requires | requires_redundantly
        | returns | returns_redundantly
        | set | signals | signals_only
        | signals_only_redundantly | signals_redundantly
        | spec_bigint_math | spec_java_math
        | spec_protected | spec_public | spec_safe_math
        | static_initializer | uninitialized
        | unreachable | weakly
        | when | when_redundantly
        | working_space | working_space_redundantly
        | writable
        | jml-universe-keyword
jml-universe-keyword ::= peer | readonly | rep
```

The following describes the special symbols used in JML. The nonterminal *java-special-symbol* is the special symbols of Java, taken without change from Java [Gosling-Joy-Steele96].

```
special-symbol ::= java-special-symbol | jml-special-symbol
java-special-symbol ::= java-separator | java-operator
java-separator ::= ( | ) | { | } | ‘[’ | ‘]’ | ; | , | .
        | multijava-separator   // When the MultiJava option is on
multijava-separator ::= @ | @@
java-operator ::= = | < | > | ! | ~ | ? | :
        | == | <= | >= | != | && | ‘||’ | ++ | --
        | + | - | * | / | & | ‘|’ | ^ | % | << | >> | >>>
        | += | -= | *= | /= | &= | ‘|=’ | ^= | %=
        | <<= | >>= | >>>=
jml-special-symbol ::= ==> | <== | <==> | <=!=>
        | -> | <- | <: | .. | ‘{|’ | ‘|}’
```

The nonterminal *java-literal* represents Java literals which are taken without change from Java [Gosling-Joy-Steele96].

```
java-literal ::= integer-literal
        | floating-point-literal | boolean-literal
        | character-literal | string-literal | null-literal

integer-literal ::= decimal-integer-literal
        | hex-integer-literal | octal-integer-literal
decimal-integer-literal ::= decimal-numeral [ integer-type-suffix ]
decimal-numeral ::= 0 | non-zero-digit [ digits ]
digits ::= digit [ digit ] . . .
digit ::= 0 | non-zero-digit
non-zero-digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
integer-type-suffix ::= l | L
hex-integer-literal ::= hex-numeral  [ integer-type-suffix ]
```

```
hex-numeral ::= 0x hex-digit [ hex-digit ] . . .
      | 0X hex-digit  [ hex-digit ] . . .
hex-digit ::= digit | a | b | c | d | e | f
      | A | B | C | D | E | F
octal-integer-literal ::= octal-numeral  [ integer-type-suffix ]
octal-numeral ::= 0 octal-digit [ octal-digit ] . . .
octal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

floating-point-literal ::= digits . [ digits ]
        [ exponent-part ] [ float-type-suffix ]
      | . digits [ exponent-part ] [ float-type-suffix ]
      | digits exponent-part [ float-type-suffix ]
      | digits [ exponent-part ] float-type-suffix
exponent-part ::= exponent-indicator signed-integer
exponent-indicator ::= e | E
signed-integer ::= [ sign ] digits
sign ::= + | -
float-type-suffix ::= f | F | d | D

boolean-literal ::= true | false

character-literal ::= ' single-character ' | ' escape-sequence '
single-character ::= any character except ', \, carriage return, or newline
escape-sequence ::= \b    // backspace
       | \t               // tab
       | \n               // newline
       | \r               // carriage return
       | \'               // single quote
       | \"               // double quote
       | \\               // backslash
       | octal-escape
       | unicode-escape
octal-escape ::= \ octal-digit [ octal-digit ]
       | \ zero-to-three octal-digit octal-digit
zero-to-three ::= 0 | 1 | 2 | 3
unicode-escape ::= \u hex-digit hex-digit hex-digit hex-digit

string-literal ::= " [ string-character ] . . . "
string-character ::= escape-sequence
       | any character except ", \, carriage return, or newline

null-literal ::= null
```

An *informal-description* looks like `(* some text *)`. It is used in predicates (see Section 11.1 [Predicates], page 87) and store-ref expressions (see Section 11.7 [Store Refs], page 103) as an escape from formality.

The exact syntax is given below.

*informal-description* ::= `(*` *non-stars-close* [ *non-stars-close* ] . . . `*)`
*non-stars-close* ::= *non-star*
    `|` *stars-non-close*
*stars-non-close* ::= `*` [ `*` ] . . . *non-star-close*
*non-star-close* ::= any character except `)` or `*`

# 5 Compilation Units

A compilation unit in JML is similar to that in Java, with some additions. It has the following syntax.

> *compilation-unit* ::= [ *package-definition* ]
> [ *refine-prefix* ]
> [ *import-definition* ] . . .
> [ *top-level-definition* ] . . .
> *top-level-definition* ::= *type-definition*
> | *multijava-top-level-declaration* // *When parsing MultiJava*

The *compilation-unit* rule is the start rule for the JML grammar. (In this syntactic rule and in all other rules in the rest of the body of this manual, *white-space* may appear between any two tokens. See Chapter 4 [Lexical Conventions], page 26, for details.)

See Chapter 6 [Type Definitions], page 37, for the syntax and semantics of *type-definition*s. See Section 17.1 [Augmenting Method Declarations], page 131, for the syntax and semantics of *multijava-top-level-declaration*. See Chapter 16 [Refinement], page 124, for a discussion of the *refine-prefix* and its uses.

Some JML tools may support various optional extensions to JML. This manual partially describes two such extensions: MultiJava [Clifton-etal00] and the Universe type system [Dietl-Mueller05]. Comments in the grammar indicate optional productions; these are only used by tools that select an option to parse the syntax in question. Tools for JML do not have to support these extensions to JML, and may themselves support other JML extensions. In general, JML tools will support a (hopefully well-documented) variant of the language described in this manual.

The Java code in a compilation unit must be legal Java code (or legal code in the Java extension, such as MultiJava, selected by any options); in particular it must obey all of Java's static restrictions. For example, at most one of the type definitions in a compilation unit may be declared `public`. See the *Java Language Specification* [Gosling-etal00] for details.

As in Java, JML can be implemented using files to store compilation units. When this is done there must also be a correspondence between the name of any public type defined in a compilation unit and the file name. This is done exactly as in Java, although JML allows additional file name suffixes. See Section 16.1 [File Name Suffixes], page 124, for details on the file name suffixes allowed in JML.

The specification of the compilation unit consists of the specifications of the *top-level-definition*s it contains, placed in the declared package (if any). The interface part of this specification is determined as in Java [Gosling-etal00] (or as in the Java extension used). The specifications of each *type-definition* are computed by starting from an environment that contains the declared package (if any), each top-level definition in the compilation unit (to allow for mutual recursion), and the imports [Gosling-etal00]. In JML, not only is the package `java.lang` implicitly imported, but also there is an implicit model import of `org.jmlspecs.lang`. (See Section 5.2 [Import Definitions], page 36, for the meaning of a model import.)

Ignoring refinement, a Java compilation unit satisfies such a JML specification if it satisfies the specified *package-definition* (if any), and if for each specified *type-definition*, there is a corresponding Java *type-definition* that satisfies that type's JML specification. Furthermore, if the JML specification does not contain a public type, then the Java compilation unit may not contain a public type.

The syntax and semantics of *package-definition*s and *import-definition*s are discussed in the subsections below.

## 5.1 Package Definitions

The syntax of a *package-definition* is as in Java [Gosling-etal00].

*package-definition* ::= `package` *name* `;`
*name* ::= *ident* [ `.` *ident* ] . . .

A Java package definition satisfies the JML specification only if it is the same as that specified. That is, the Java code has to be the same (modulo *white-space*) as the JML specification.

## 5.2 Import Definitions

The syntax of a *import-definition* is as follows. The only difference from the Java syntax [Gosling-etal00] is the optional `model` modifier.

*import-definition* ::= [ `model` ] `import` *name-star* `;`
*name-star* ::= *ident* [ `.` *ident* ] . . . [ `. *` ]

An *import-definition* may use the `model` modifier if and only if the whole *import-definition* is entirely contained within a single annotation. For example, the following is illegal.

```
/*@ model @*/ import com.foo.*; // illegal!
```

To write an import that affects both the JML annotations and Java code, just use a normal java import, without using the `model` modifier.

The effect on the interface computed for a compilation unit of an *import-definition* without the `model` keyword is the same as in Java [Gosling-etal00]. Such import directives affect the computation of the interface of the Java code as well as the JML specification (that is, they apply to both equally).

When the `model` keyword is used, the import only has an effect on the JML annotations (and not on the Java code). The abbreviation permitted by the use of such an import, however, is the same as would be effected by a normal Java import. Such model imports can affect the computation of the interface of the JML specification by being used in the declarations of model and ghost features.

Both normal Java and model imports do not themselves contribute to the interface of a JML specification. As such, they do not have to be present in a correct implementation of the specification. An implementation could, for example, use different forms of import, or it could use fully qualified names instead of imports, and achieve the same effect as using the imports in the specification.

# 6 Type Definitions

The following is the syntax of type definitions.

> *type-definition* ::= *class-definition*
> | *interface-definition*
> | `;`

The specification of a *type-definition* is determined as follows. If the *type-definition* consists only of a semicolon (`;`), then the specification is empty. Otherwise the specification is that of the class or interface definition. Such a specification must be satisfied by the corresponding class or interface definition.

The rest of this chapter discusses class and interface definitions, as well as the syntax of modifiers.

## 6.1 Class and Interface Definitions

Class and interface definitions are quite similar, as interfaces may be seen as a special kind of class definition that only allows the declaration of abstract instance methods and final static fields (in the Java code [Gosling-etal00]). Their syntax is also similar.

> *class-definition* ::= [ *doc-comment* ] *modifiers* `class` *ident*
> [ *class-extends-clause* ] [ *implements-clause* ]
> *class-block*
> *class-block* ::= `{` [ *field* ] . . . `}`
>
> *interface-definition* ::= [ *doc-comment* ] *modifiers* `interface` *ident*
> [ *interface-extends* ]
> *class-block*

Documentation comments for classes and interfaces may not contain JML specification information. See Section 4.5 [Documentation Comments], page 28, for the syntax of documentation comments.

See Chapter 7 [Class and Interface Member Declarations], page 43, for the syntax and semantics of *fields*, which form the essence of classes and interfaces.

The rest of this section discusses subtyping for classes and interfaces and also the particular modifiers used in classes and interfaces.

### 6.1.1 Subtyping for Type Definitions

Classes in Java can use single inheritance and may also implement any number of interfaces. Interfaces may extend any number of other interfaces.

> *class-extends-clause* ::= [ `extends` *name* [ `weakly` ] ]
> *implements-clause* ::= `implements` *name-weakly-list*
> *name-weakly-list* ::= *name* [ `weakly` ] [ `,` *name* [ `weakly` ] ] . . .
> *interface-extends* ::= `extends` *name-weakly-list*

The meaning of inheritance in JML is similar to that in Java. In Java, a when class $S$ names a class $T$ in $S$'s *class-extends-clause*, then $S$ is a *subclass* of $T$ and $T$ is a *superclass* of $S$; we also say that $S$ *inherits* from $T$. This relationship also makes $S$ a *subtype* of $T$, meaning that variables of type $T$ can refer to objects of type $S$. In Java, when $S$ is a subclass of $T$, then $S$ inherits all the instance fields and methods from $T$.

A class may also implement several interfaces, declared in its *implements-clause*; the class thus becomes a subtype of each of the interfaces that it implements.

Similarly, an interface may extend several other interfaces. In Java, such an interface inherits all of the abstract methods and static final fields from the interfaces it extends. When interface $U$ extends another interface $V$, then $U$ is a subtype of $V$.

In JML, model and ghost features, as well as specifications are inherited. A subtype inherits from its supertypes:

- all instance fields, including model and ghost fields,
- instance methods are also inherited and their specifications,
- instance invariants and instance history constraints.

It is an error for a type to inherit a field $x$ from two different supertypes if that field is declared with different types.

It is an error for a type to inherit a method with the same formal parameter types but with either different return types or with conflicting throws clauses [Gosling-etal00]. (There are other restrictions on method inheritance that apply when MultiJava is used [Clifton-etal00].)

In Java one cannot inherit method implementations from interfaces, but this is possible in JML, where one can implement a model method in an interface. It is illegal for a class or interface to inherit two different implementations of a model method.

In JML, instance methods have to obey the specifications of all methods they override. This, together with the inheritance of invariants and history constraints, forces subtypes to be behavioral subtypes [Dhara-Leavens96] [Leavens-Naumann06] [Leavens06b]. However, history constraints are not inherited from supertypes whose names are marked with `weakly` in the relevant clause. Such subtypes are *weak behavioral subtypes*, and should only be used in ways that do not permit cross-type aliasing [Dhara-Leavens94b] [Dhara97].

See the report, "Desugaring JML Method Specifications" [Raghavan-Leavens05] for more about the details of specification inheritance in JML.

### 6.1.2 Modifiers for Type Definitions

In addition to the Java modifiers that can be legally attached to a class or interface definition [Gosling-etal00], in JML one can use the following modifiers.

```
pure model
spec_java_math spec_safe_math spec_bigint_math
code_java_math code_safe_math code_bigint_math
nullable_by_default
```

See Section 6.2 [Modifiers], page 39, for the syntax and semantics of modifiers in general.

A type definition may be modified with the JML modifier keyword `pure`. The effect of declaring a type `pure` is that all constructor and instance method declarations within the type are automatically declared to be pure (see Section 7.1.1.3 [Pure Methods and Constructors], page 44, for more about pure methods). Hence, once an object of a class is created, it will be immutable, and furthermore, none of its instance methods will have any side effects. However, its static methods may still have side effects, as the `pure` does not apply to the static methods declared in a type. Furthermore, although an override of a pure method must be pure, instance methods declared in subtypes that do not override this supertype's methods need not be pure. Hence, such a subtype does not necessarily have immutable objects. So, in essence, declaring a class pure is merely a shorthand for declaring all of the constructors and instance methods pure.

[[[ Pure does not make a class immutable either, since a method might return a reference to an internal representation which is then modified by some non-pure method in its class. Is it sufficient if all fields are also fields of pure types (recursively)? Then there are arrays. And also all fields would need to be private to have immutability. - DRC ]]]

A type declaration that is declared with the modifier `model` is a specification-only type. Hence, such a type may not be used in Java code, and may only be used in annotations. It follows that the entire type definition must be contained within an annotation comment, and consequently annotations within the type definition do not need to be separately enclosed in annotation comments, as is demonstrated in the example below. The scope rules for a model type definition are the same as for Java type definitions, except that a model type definition is not in scope for any Java code, only for annotations.

[[[Model types are seldom used in JML. Since the runtime assertion checker doesn't work with them, I wonder if it would be best to get rid of them completely. You could always just define a Java type, which would be useful for runtime assertion checking.]]]

[[[ May a model type definition appear in more than one specification file of a refinement sequence, with any member declarations being combined together? I'd prefer that it only be allowed to appear once and be required to be completely defined in one spec file - easier for tools. – DRCok ]]]

[[[Need to explain the math modifiers.]]]

## 6.2 Modifiers

The following is the syntax of modifiers.

*modifiers* ::= [ *modifier* ] . . .
*modifier* ::= `public` | `protected` | `private`
    | `abstract` | `static` |
    | `final` | `synchronized`
    | `transient` | `volatile`
    | `native` | `strictfp`
    | `const`          *// reserved but not used in Java*
    | *jml-modifier*
*jml-modifier* ::= `spec_public` | `spec_protected`
    | `model` | `ghost` | `pure`
    | `instance` | `helper`
    | `uninitialized`
    | `spec_java_math` | `spec_safe_math` | `spec_bigint_math`
    | `code_java_math` | `code_safe_math` | `code_bigint_math`
    | `non_null` | `nullable` | `nullable_by_default`
    | `extract`

The *jml-modifiers* are only recognized as keywords in annotation comments. See Chapter 4 [Lexical Conventions], page 26, for more details.

The Java modifiers have the same meaning as in Java [Gosling-etal00].

Note that although the *modifiers* grammar non-terminal is used in many places throughout the grammar, not all modifiers can be used with every grammar construct. See the discussion regarding each grammar construct, which is summarized in Appendix B [Modifier Summary], page 159.

In the following we first discuss the suggested ordering of modifiers The rest of this section discusses the JML-specific modifiers in general terms. Their use and meaning for each kind of grammatical construct should be consulted directly for more details.

### 6.2.1 Suggested Modifier Ordering

There are various guidelines for ordering modifiers in Java [[[citations?]]]. As JML has several extra modifiers, we also suggest an ordering; although this ordering is not enforced, various tools may give warnings if the suggestions are not followed, as following a standard ordering tends to make reading declarations easier. For use in JML, we suggest the following ordering groups, where the ones at the top should appear first (leftmost), and the ones at the bottom should appear last (rightmost). In each line, the modifiers are either mutually exclusive, or their order does not matter (or both).

```
public private protected spec_public spec_protected
abstract static
model ghost pure
final synchronized
instance helper
transient volatile
native strictfp
monitored uninitialized
spec_java_math spec_safe_math spec_bigint_math
code_java_math code_safe_math code_bigint_math
non_null nullable nullable_by_default
code extract
peer rep readonly
```

### 6.2.2 Spec Public

The `spec_public` modifier allows one to declare a feature as public for specification purposes. It can only be used when the feature has a more restrictive visibility in Java. A `spec_public` field is also implicitly a data group.

### 6.2.3 Spec Protected

The `spec_protected` modifier allows one to declare a feature as protected for specification purposes. It can only be used when the feature has a more restrictive visibility in Java. That is, it can only be used to change the visibility of a field or method that is, for Java, either declared `private` or default access (package visible). A `spec_protected` field is also implicitly a data group.

### 6.2.4 Pure

In general terms, a *pure* feature is one that has no side effects when executed. In essence `pure` only applies to methods and constructors. The use of `pure` for a type definition is shorthand for applying that modifier to all constructors and instance methods in the type (see Section 6.1.2 [Modifiers for Type Definitions], page 38).

See Section 7.1.1.3 [Pure Methods and Constructors], page 44, for the exact semantics of pure methods and constructors.

### 6.2.5 Model

The `model` modifier introduces a specification-only feature. For fields it also has a special meaning, which is that the field can be represented by concrete fields. See Section 2.2 [Model and Ghost], page 11.

The modifiers `model` and `ghost` are mutually exclusive.

A `model` field may not be declared to be `final`. This is because model fields are abstractions of concrete fields, and thus it would complicate JML to allow final model fields. If you feel that you want a final model field, what you should use instead is a final ghost field. See Section 6.2.6 [Ghost], page 41.

Note that in an interface, a model field is implicitly declared to be `static`. Thus if you want an instance field, you should use the modifier `instance`, so that the field will act as if it were a member of all objects whose type is a subtype of that interface. Conversely, in a class, a model field is implicitly declared to be `instance`. Thus, if you want a static field, you should use the modifier `static`, so that the value of the model field is shared by all instances of the class and its subclasses.

### 6.2.6 Ghost

The `ghost` modifier introduces a specification-only field that is maintained by special set statements. See Section 2.2 [Model and Ghost], page 11.

The modifiers `ghost` and `model` are mutually exclusive.

A ghost field declared in an interface is not `final` by default. If you want a final `ghost` field in an interface, you must declare it to be `final` explicitly. Ghost fields in classes are also not final by default.

In an interface, a ghost field is implicitly declared to be `static`. Thus if you want an instance field, you should use the modifier `instance`, so that the field will act as if it were a member of all objects whose type is a subtype of that interface. Conversely, in a class, a ghost field is implicitly declared to be `instance`. Thus, if you want a static field, you should use the modifier `static`, so that the value of the ghost field is shared by all instances of the class and its subclasses.

### 6.2.7 Instance

The `instance` modifier says that a field is not static. See Section 2.5 [Instance vs. Static], page 14.

### 6.2.8 Helper

The `helper` modifier may be used on a private method or constructor to say that its specification is not augmented by invariants and history constraints that would otherwise be relevant. Normally, an invariant applies to all methods in a class or interface. However, an exception is made for methods and constructors declared with the `helper` modifier. See Section 8.2 [Invariants], page 50. [[[ Just on private? or just on non-overridable methods? or just on non-overridden methods? - DRC]]]

### 6.2.9 Monitored

The `monitored` modifier may be used on a non-model field declaration to say that a thread must hold the lock on the object that contains the field (i.e., the `this` object containing the field) before it may read or write the field [Leino-Nelson-Saxe00].

### 6.2.10 Uninitialized

The `uninitialized` modifier may be used on a field declaration to say that despite the initializer, the location declared is to be considered uninitialized. Thus, the field should be assigned in each path before it is read. [Leino-Nelson-Saxe00].

### 6.2.11 Math Modifiers

[[[Need explanation of these.]]]

### 6.2.12 Nullity Modifiers

Any declaration (other than that of a local variable) whose type is a reference type is implicitly declared `non_null` unless (explicitly or implicitly) declared `nullable`. Hence reference type declarations are assumed to be non-null by default (see Section 2.8 [Null is Not the Default], page 15).

A declaration can be *explicitly* declared `nullable` by annotating it with the `nullable` modifier. A declaration is *implicitly* declared `nullable` when the (outer most) class or interface containing the declaration is adorned by the class-level modifier `nullable_by_default`.

Attempting to use both the `non_null` and `nullable` modifiers is a compile time error.

# 7 Class and Interface Member Declarations

The nonterminal *field* describes all the members of classes and interfaces (see Section 6.1 [Class and Interface Definitions], page 37).

*field* ::= *member-decl*
  | *jml-declaration*
  | *class-initializer-decl*
  | ;

Also see Section E.2.1 [Non-null by Default], page 165. In the rest of this chapter we describe mostly the syntax and Java details of member declarations and class initializers. See Chapter 8 [Type Specifications], page 50, for the syntax and semantics of *jml-declaration*, and, more generally, how to use JML to specify the behavior of types.

## 7.1 Java Member Declarations

The following gives the syntax of Java member declarations.

*member-decl* ::= *method-decl*
  | *variable-definition*
  | *class-definition*
  | *interface-definition*

See Section 6.1 [Class and Interface Definitions], page 37, for details of *class-definition* and *interface-definition*. We discuss method and variable declarations below.

### 7.1.1 Method and Constructor Declarations

The following is the syntax of a method declaration.

*method-decl* ::= [ *doc-comment* ] . . .
    *method-specification*
    *modifiers* [ *method-or-constructor-keyword* ]
    [ *type-spec* ] *method-head*
    *method-body*
  | [ *doc-comment* ] . . .
   *modifiers* *method-or-constructor-keyword*
   [ *type-spec* ] *method-head*
   [ *method-specification* ]
   *method-body*
*method-or-constructor-keyword* ::= `method` | `constructor`
*method-head* ::= *ident* *formals* [ *dims* ] [ *throws-clause* ]
*method-body* ::= *compound-statement* | ;
*throws-clause* ::= `throws` *name* [ , *name* ] . . .

Notice that the specification of a method (see Chapter 9 [Method Specifications], page 61) may appear either before or after the *method-head*.

The use of `non_null` as a *modifier* in a *method-decl* really is shorthand for a postcondition describing the normal result of a method, indicating that it must not be null. It can also be seen as a modifier on the method's result type, saying that the type returned does not contain null.

The use of `extract` as a *modifier* in a *method-decl* is shorthand for writing a model program specification. See Section 14.2 [Extracting Model Program Specifications], page 119, for an explanation of this modifier.

#### 7.1.1.1 Formal Parameters

*formals* ::= `(` [ *param-declaration-list* ] `)`
*param-declaration-list* ::= *param-declaration*
    [ `,` *param-declaration* ] . . .
*param-declaration* ::= [ *param-modifier* ] . . . *type-spec ident* [ *dims* ]
    `|` *multijava-param-declaration* // *When MultiJava parsing is on*
*param-modifier* ::= `final` `|` `non_null` `|` `nullable`

See Section 7.1.2.2 [Type-Specs], page 48, for more about the nonterminals *type-spec* and *dims*. See Section 17.2 [MultiMethods], page 131, for details of *multijava-param-declaration*.

The modifier `non_null` when attached to a formal parameter is shorthand for a precondition that says that the corresponding actual parameter may not be null. The type of a parameter that has the `non_null` modifier must be a reference type [Raghavan-Leavens05].

The `non_null` modifier on a parameter is inherited in the same way as the equivalent precondition would be, so it need not be declared on every declaration of the same method in a subtype or refinement. The `non_null` modifier may be added to a method in a refinement file (see Chapter 16 [Refinement], page 124), and thus does not have to appear in any particular file in a refinement sequence. It can be added to a method override in a subtype, but that will generally make the method non-implementable, as the method must also satisfy an inherited specification without the corresponding precondition.

#### 7.1.1.2 Model Methods and Constructors

A method or constructor that uses the modifier `model` is called a *model method or constructor*. Since a model method is not visible to Java code, the entire method, including its body, should be written in an annotation.

As usual in JML (see Section 2.2 [Model and Ghost], page 11), a model method or constructor is a specification-only feature. A model method or constructor may have either a body or a specification, or both. The specification may be used in various verification tools, while the body allows it to be executed during runtime assertion checking. Model methods may also be abstract, and both model methods and constructors may be final.

[[[ Can constructors be final? Why? - DRC ]]]

It is usual in JML to declare model methods and constructors as `pure`. However, it is possible to have a model method or constructor that is not pure; such methods are useful in model programs (see Chapter 14 [Model Programs], page 117). On the other hand, aside from their use in model programs, most model methods only exist to be called in assertions, and since only pure methods can be called in assertions, they should usually be declared as `pure`.

#### 7.1.1.3 Pure Methods and Constructors

This subsubsection, which describes the effect of the `pure` modifier on methods and constructor declarations, is quoted from the preliminary design document [Leavens-Baker-Ruby06].

We say a method is *pure* if it is either specified with the modifier `pure` or is a method that appears in the specification of a `pure` interface or class. Similarly, a constructor is pure if it is either specified with the modifier `pure` or appears in the specification of a `pure` class.

A *pure method* that is not a constructor implicitly has a specification that does not allow any side-effects. That is, its specification has the clauses

```
diverges false;
assignable \nothing;
```

added to each specification case; if the method has no specification given explicitly, then these clauses are added as a lightweight specification. For this reason, if one is writing a pure method, it is not necessary to otherwise specify an assignable clause (see Section 9.9.9 [Assignable Clauses], page 80), although doing so may improve the specification's clarity.

A *pure constructor* has the clauses

```
diverges false;
assignable this.*;
```

added to each specification case; if the constructor has no specification given explicitly, then these clauses are added as a lightweight specification. This specification allows the constructor to assign to the non-static fields of the class in which it appears (including those inherited from its superclasses and ghost model instance fields from the interfaces that it implements).

Implementations of pure methods and constructors will be checked to see that they meet these conditions on what locations they can assign to. To make such checking modular, some JML tools prohibit a pure method or constructor implementation from calling methods or constructors that are not pure. However, more sophisticated tools could more directly check the intended semantics [Salcianu-Rinard05].

A pure method or constructor must also be provably terminating. Although JML does not force users to make such proofs of termination, users writing pure methods and constructors are supposed to make pure methods total in the sense that whenever, a pure method is called it either returns normally or throws some exception. This is supposed to lessen the possibility that assertion evaluation could loop forever, aids the runtime assertion checker, which turns exceptions into arbitrary values of the appropriate result type, and helps make pure methods more like mathematical functions for verification purposes. [[[ I think this has changed - exceptions in a pure method make the result undefined, not arbitrary - DRC]]]

Furthermore, a pure method is supposed to always either terminate normally or throw an exception, even for calls that do not satisfy its precondition. Static verification tools for JML should enforce this condition, by requiring a proof that a pure method implementation satisfies the following specification

```
private behavior
  requires true;
  diverges false;
  assignable \nothing;
```

(and similarly for constructors, except that the assignable clause becomes `assignable this.*;` for constructors).

However, this implicit verification condition is a specification, and thus cannot be used in reasoning about calls to the method, even calls from within the class itself and recursive calls from within the implementation. For this reason we recommend writing the method or constructor specification in such a way that the effective precondition of the method is "true," making the proof of the above implicit verification condition trivial, and allowing the termination behavior of the implementation to be relied upon by all clients.

Recursion is permitted, both in the implementation of pure methods and the data structures they manipulate, and in the specifications of pure methods. When recursion is used in a specification, the proof of well-formedness for the specification involves the use of JML's `measured_by` clause.

Since a pure method may not go into an infinite loop, if it has a non-trivial precondition, it should throw an exception when its normal precondition is not met. This exceptional behavior does not have to be specified or programmed explicitly, but technically there is an obligation to meet the specification that the method never loops forever.

Furthermore, a pure method must be deterministic, in the sense that when called in a given state, it must always return the same value. Similarly a pure constructor should be deterministic in the sense that when called in a given state, it always initializes the object in the same way.

A pure method can be declared in any class or interface, and a pure constructor can be declared in any class. JML will specify the pure methods and constructors in the standard Java libraries as pure.

As a convenience, instead of writing `pure` on each method declared in a class and interface, one can use the modifier `pure` on classes and interfaces and classes. This simply means that each non-static method and each constructor declared in such a class or interface is `pure`. Note that this does not mean that all methods inherited (but not declared in and hence not overridden in) the class or interface are pure. For example, every class inherits ultimately from `java.lang.Object`, which has some methods, such as `notify` and `notifyAll` that are manifestly not pure. Thus each class will have some methods that are not pure. Despite this, it is convenient to refer to classes and interfaces declared with the `pure` modifier as *pure*.

In JML the modifiers `model` and `pure` are orthogonal. (Recall something declared with the modifier `model` does not have to be implemented, and is used purely for specification purposes.) Therefore, one can have a model method that is not pure (these might be useful in JML's model programs) and a pure method that is not a model method. Nevertheless, usually a model method (or constructor) should be pure, since there is no way to use non-pure methods in an assertion, and model methods cannot be used in normal Java code.

By the same reasoning, model classes should, in general, also be pure. Model classes cannot be used in normal Java code, and hence their methods are only useful in assertions (and JML's model programs). Hence it is typical, although not required, that a model class also be a pure class.

As can be seen from the semantics, if a pure method has a return type of `void`, then it can essentially only do nothing. So, while pure methods with `void` as their return type are not illegal, they are useless.

#### 7.1.1.4 Helper Methods and Constructors

The `helper` modifier may only be used on a private method or constructor. [[[ This restriction needs to be clarified - ESC/Java limits helper to non-overridable methods. ]]] Such a helper method or constructor has a specification that is not augmented by invariants and history constraints that would otherwise apply to it. It can thus be thought of as not really a method or constructor, but merely an abbreviation device. However, whatever specifications are given explicitly for such a method or constructor still apply. See Section 8.2 [Invariants], page 50, for more details.

### 7.1.2 Field and Variable Declarations

The following is the syntax of field and variable declarations.

*variable-definition* ::= [ *doc-comment* ] . . . *modifiers variable-decls*
*variable-decls* ::= [ `field` ] *type-spec variable-declarators* `;`
      [ *jml-data-group-clause* ] . . .
*variable-declarators* ::= *variable-declarator*
      [ `,` *variable-declarator* ] . . .
*variable-declarator* ::= *ident* [ *dims* ] [ `=` *initializer* ]
*initializer* ::= *expression* `|` *array-initializer*
*array-initializer* ::= `{` [ *initializer-list* ] `}`
*initializer-list* ::= *initializer* [ `,` *initializer* ] . . . [ `,` ]

The `field` keyword is not normally needed, but can be used to change JML's parsing mode. Within an annotation, such as within a declaration of a model method, it is sometimes necessary to switch from JML annotation mode to JML spec-expression mode, in order to parse words that are JML keywords but should be recognized as Java identifiers. This can be accomplished in a field declaration by using the keyword `field`, which changes parsing to spec-expression mode. [[[ When does the mode revert back? e.g. in a method declaration - DRC]]]

[[[Needs example, move elsewhere?]]]

In a non-Java file, such as a file with suffix '`.refines-java`' (see Chapter 16 [Refinement], page 124), one may omit the initializer of a *variable-declarator*, even one declared to be `final`. In such a file, one may also omit the body of a *method-decl*. Of course, in a '`.java`' file, one must obey all the rules of Java for declarations that are not in annotations.

See Chapter 10 [Data Groups], page 85, for more about *jml-data-group-clause*s. See Section 11.2 [Specification Expressions], page 87, for the syntax of *expression*. In the following we discuss the modifiers for field and variable declarations and *type-spec*s.

#### 7.1.2.1 JML Modifiers for Fields

The `ghost` and `model` modifiers for fields both say that the field is a specification-only field; it thus cannot be accessed by the Java code. The difference is that a ghost field is explicitly manipulated by initializations and set statements (see Chapter 12 [Statements and Annotation Statements], page 104), whereas a model field cannot be explicitly manipulated. Instead a model field is indirectly given a value by a represents clause (see Section 8.4 [Represents Clauses], page 58). See Section 2.2 [Model and Ghost], page 11, for a general discussion of this distinction in JML.

While fields can be declared as either model or ghost fields, a field cannot be both. Furthermore, local variables cannot be declared with the `model` modifier.

The `non_null` modifier in a variable declaration is shorthand for an invariant saying that each variable declared in the *variable-decls* may not be null. This invariant has the same visibility as the visibility declaration of the *variable-definition* itself. See Section 8.2 [Invariants], page 50, for more about invariants.

The `monitored` modifier says that each variable declared in the *variable-decls* can only be accessed by a thread that holds the lock on the object that contains the field [Leino-Nelson-Saxe00]. It may not be used with model fields.

The `instance` modifier says that the field is to be found in instances instead of in class objects; it is the opposite of `static`. It is typically only needed for model or ghost fields declared in interfaces. When used in an interface, it makes the field both non-static and non-final (unless the `final` modifier is used explicitly). See Section 2.5 [Instance vs. Static], page 14. [[[ So how does one declare a static non-final field in an interface? - DRC ]]]

#### 7.1.2.2 Type-Specs

The syntax of a *type-spec* is as in Java [Gosling-etal00], except for the addition of the type `\TYPE` and the possibility of using *ownership-modifiers*. The *ownership-modifiers* are only available when the Universe type system is turned on. See Chapter 18 [Universe Type System], page 132, for how to do that, and for the syntax and semantics of *ownership-modifiers*.

```
type-spec ::= [ ownership-modifiers ] type [ dims ]
        | \TYPE [ dims ]
type ::= reference-type | built-in-type
reference-type ::= name
dims ::= '[' ']' [ '[' ']' ] . . .
```

The type `\TYPE` represents the kind of all Java types. It can only be used in annotations. It is equivalent to `java.lang.Class`.

## 7.2 Class Initializer Declarations

The following is the syntax of class initializers.

```
class-initializer-decl ::= [ method-specification ]
                           [ static ] compound-statement
          | method-specification static_initializer
          | method-specification initializer
```

The first form above is the form of Java class instance and static initializers. The initializer is static, and thus run when the class is loaded, if it is labeled `static`. The effect of the initializer can be specified by a JML method specification (see Chapter 9 [Method Specifications], page 61), which treats the initializer as a private helper method with return type `void`, whose body is given by the *compound-statement* (see Chapter 12 [Statements and Annotation Statements], page 104).

The last two forms are used in JML to specify static and instance initializers without giving the body of the initializer. They would be used in annotations in non-Java files (see Chapter 16 [Refinement], page 124). At most one of each of these may appear in a type specification file. Such a specification is satisfied if there is at least one corresponding initializer in the implementation, and if the sequential composition of the bodies of the corresponding initializer(s), when considered as the body of a private helper method with return type `void`, satisfy the specification given (see Chapter 9 [Method Specifications], page 61).

Note that, due to this semantics, the *method-specification*s for an initializer can only have private specification cases.

[[[ But initializers can be interspersed between field initializations, which will affect their meaning. Thus I think the composition has to include the field initializations. The effect is that the post-condition of the JML initializer refers to the state before a constructor begins executing; a static_initializer refers to the state after class loading, I think. – DRCok ]]] [[[ Is the restriction to private true for static initialization as well - don't think it should be. - DRCOk ]]]

# 8 Type Specifications

This chapter describes the way JML can be used to specify abstract data types (ADTs).

Overall the mechanisms used in JML to specify ADTs can be described as follows. First, the interface of a type is described using the Java syntax for such a type's declaration (see Chapter 7 [Class and Interface Member Declarations], page 43); this includes any required fields and methods, along with their types and visibilities, etc. Second, the behavior of a type is described by declaring model and ghost fields to be the client (or subtype) visible abstractions of the concrete state of the objects of that type, by writing method specifications using those fields, and by writing various *jml-declaration*s to further refine the logical model defined by these fields. These *jml-declaration*s can also be used to record various design and implementation decisions.

The syntax of these *jml-declaration*s is as follows.

*jml-declaration* ::= *modifiers invariant*
  | *modifiers history-constraint*
  | *modifiers represents-clause*
  | *modifiers initially-clause*
  | *modifiers monitors-for-clause*
  | *modifiers readable-if-clause*
  | *modifiers writable-if-clause*
  | *axiom-clause*

The semantics of each of kind of *jml-declaration* is discussed in the sections below. However, before getting to the details, we start with some introductory examples.

## 8.1 Introductory ADT Specification Examples

[[[Need examples here, which should be first written into the org.jmlspecs.samples.jmlrefman package and then included and discussed here.]]]

## 8.2 Invariants

The syntax of an invariant declaration is as follows.

*invariant* ::= *invariant-keyword predicate* `;`
*invariant-keyword* ::= `invariant` | `invariant_redundantly`

An example of an invariant is given below. The invariant in the example has default (package) visibility, and says that in every state that is a visible state for an object of type `Invariant`, the object's field `b` is not null and the array it refers to has exactly 6 elements. In this example, no postcondition is necessary for the constructor since the invariant is an implicit postcondition for it.

```
package org.jmlspecs.samples.jmlrefman;

public abstract class Invariant {

    boolean[] b;

    //@ invariant b != null && b.length == 6;
```

```
        //@ assignable b;
        Invariant() {
            b = new boolean[6];
        }
    }
```

Invariants are properties that have to hold in all visible states. The notion of visible state is of crucial importance in the explanation of the semantics of both invariants and constraints. A state is a *visible state* for an object $o$ if it is the state that occurs at one of these moments in a program's execution:

- at end of a non-helper constructor invocation that is initializing $o$,
- at the beginning of a non-helper finalizer invocation that is finalizing $o$,
- at the beginning or end of a non-helper non-static non-finalizer method invocation with $o$ as the receiver,
- at the beginning or end of a non-helper static method invocation for a method in $o$'s class or some superclass of $o$'s class, or
- when no constructor, destructor, non-static method invocation with $o$ as receiver, or static method invocation for a method in $o$'s class or some superclass of $o$'s class is in progress.

Note that visible states for an object $o$ do not include states at the beginning and end of invocations of *helpers*: constructors or methods declared with the `helper` modifier (see Section 9.6.4 [Helper methods and constructors], page 69). Thus the post-state of a helper constructor and the pre- and post-states of helper methods are not visible states.

A state is a *visible state* for a type $T$ if it occurs after static initialization for $T$ is complete and it is a visible state for some object that has type $T$.

JML distinguishes *static* and *instance* invariants. These are mutually exclusive and any invariant is either a static or instance invariant. An invariant may be explicitly declared to be static or instance by using one of the modifiers `static` or `instance` in the declaration of the invariant. An invariant declared in a class declaration is, by default, an instance invariant. An invariant declared in an interface declaration is, by default, a static invariant.

For example, the invariant declared in the class `Invariant` above is an instance invariant, because it occurs inside a class declaration. If `Invariant` had been an interface instead of a class, then this invariant would have been a static invariant.

A static invariant may only refer to static fields of an object. An instance invariant, on the other hand, may refer to both static and non-static fields.

The distinction between static and instance invariants also affects when the invariants are supposed to hold. A static invariant declared in a type $T$ must hold in every state that is a visible state for type $T$. An instance invariant declared in a type $T$ must hold for every object $o$ of type $T$, for every state that is a visible state for $o$.

For reasoning about invariants we make a distinction between assuming, establishing, and preserving an invariant. A method or constructor *assumes* an invariant if the invariant must hold in its pre-state. A method or constructor *establishes* an invariant if the invariant must hold in its post-state. A method or constructor *preserves* an invariant if the invariant is both assumed and established.

JML's verification logic enforces invariants by making sure that each non-helper method, constructor, or finalizer:

- assumes the static invariants of all types, $T$, for which its pre-state is a visible state for $T$,
- establishes the static invariants of all types, $T$, for which its post-state is a visible state for $T$,
- assumes the instance invariants of all objects, $o$, for which its pre-state is a visible state for $o$, and
- establishes the instance invariants of all objects, $o$, for which its post-state is a visible state for $o$.

This means that each non-helper constructor found in a class $C$ preserves the static invariants of all types, including $C$, that have finished their static initialization, establishes the instance invariant of the object under construction, and, modulo creation and deletion of objects, preserves the instance invariants of all other objects. (Objects that are created by a constructor must have their instance invariant established; and objects that are deleted by the action of the constructor can be assumed to satisfy their instance invariant in the constructor's pre-state.) Note in particular that, at the beginning of a constructor invocation, the instance invariant of the object being initialized does not have to hold yet.

Furthermore, each non-helper non-static method found in a type $T$ preserves the static invariants of all types that have finished their static initialization, including $T$, and, modulo creation and deletion of objects, preserves the instance invariants of all objects, in particular the receiver object. However, finalizers do only assume the instance invariant of the receiver object, and do not have to establish it on exit.

The semantics given above is highly non-modular, but is in general necessary for the enforcement of invariance when no mechanisms are available to prevent aliasing problems, or when constructs like (concrete) public fields are used [Poetzsch-Heffter97]. Of course, one would like to enforce invariants in a more modular way. By a modular enforcement of invariants, we mean that one could verify each type independently of the types that it does not use, and that a well-formed program put together from such verified types would still satisfy the semantics for invariants given above. That is, each type would be responsible for the enforcement of the invariants it declares and would be able to assume, without checking, the invariants of other types it uses.

To accomplish this ideal, it seems that some mechanism for object ownership and alias control [Noble-Vitek-Potter98] [Mueller-Poetzsch-Heffter00] [Mueller-Poetzsch-Heffter00a] [Mueller-Poetzsch-Heffter01a] [Mueller02] [Mueller-Poetzsch-Heffter-Leavens03] is necessary. However, this mechanism is still not a part of JML, although some design work in this direction has taken place [Mueller-Poetzsch-Heffter-Leavens06].

On the other hand, people generally assume that there are no object ownership alias problems; this is perhaps a reasonable strategy for some tools, like run-time assertion checkers, to take. The alternative, tracking which types and objects are in visible states, and checking every applicable invariant for every type and object in a visible state, is obviously impractical.

Therefore, assuming or ignoring the problems with object ownership and alias control, one obtains a simple and more modular way to check invariants. This is as follows.

- Each non-helper constructor declared in a class $C$, must preserve the static invariant of $C$, if $C$ is finished with its static initialization, and must establish the instance invariant of the object being constructed.
- Each non-helper non-static non-finalizer method declared in a type $T$, must preserve the static invariant of $T$, if $T$ is finished with its static initialization, and must preserve the instance invariant of the receiver object.
- Each non-helper static method declared in a type $T$, must preserve the static invariant of $T$, if $T$ is finished with its static initialization.

When doing such proofs, one may assume the static invariant of any type (that is finished with its static initialization), and one may also assume the instance invariant of any other object.

In this, more modular, style of checking invariants, one can think of all the static invariants in a class as being implicitly conjoined to the pre- and postconditions of all non-helper constructors and methods, and the instance invariants in a class as being implicitly conjoined to the postcondition of all non-helper constructors, and to the pre- and postconditions of all non-helper methods.

As noted above, `helper` methods and constructors are exempt from the normal rules for checking invariants. That is because the beginning and end of invocations of these `helper` methods and constructors are not visible states, and therefore they do not have to preserve or establish invariants. Note that only `private` methods and constructors can be declared as `helper`. See Section 7.1.1.4 [Helper Methods and Constructors], page 46.

The following subsections discuss other points about the semantics of invariants:

- Invariants can be declared `static`; see Section 8.2.1 [Static vs. instance invariants], page 54.
- Invariants can be declared with the access modifiers `public`, `protected`, and `private`, or be left with default access; see Section 8.2.3 [Access Modifiers for Invariants], page 55.
- Invariants should also hold in case a constructor or method terminates abruptly, by throwing an exception; see Section 8.2.2 [Invariants and Exceptions], page 54.
- A class inherits all visible invariants specified in its superclasses and superinterfaces; see Section 8.2.4 [Invariants and Inheritance], page 55.
- Although some aspects of invariants are discussed in isolation here, the full explanation of their semantics can only be given considered together with that of method specifications. After all, a method only has to preserve invariants when one of the preconditions (i.e., `requires` clauses) specified for that method holds. So invariants are an integral part of the explanation of method specifications in Chapter 9 [Method Specifications], page 61.
- When considering an individual method body, remember that invariants should not just hold in the beginning and the end of it, but also at any program point halfway where another (non-`helper`) method or constructor is invoked. After all, these program points are also visible states, and, as stated above, invariants should hold at all visible states.
- A method invocation on an object should not just preserve the instance invariants of that object and the static invariants of the class, but it should preserve the invariants of all other (reachable) objects as well [Poetzsch-Heffter97].

It should be noted that the last two points above are not specific to Java or JML, but these are tricky issues that have to be considered for any notion of invariant in an object-oriented languages. Indeed, these two issues make the familiar notion of invariant a lot more complicated than one might guess at first sight!

### 8.2.1 Static vs. instance invariants

As discussed above (see Section 8.2 [Invariants], page 50), invariants can be declared `static` or `instance`. Just like a static method, a static invariant cannot refer to the current object `this` and thus cannot refer to instance fields of `this` or non-static methods of the type.

Instance invariants must be established by the constructors of an object, and must be preserved by all non-helper instance methods. If an object has fields that can be changed without calling methods (usually a bad idea), then any such changes must also preserve the invariants. For example, if an object has a public field, each assignment to that field must establish all invariants that might be affected.

Static methods do not have a receiver object for which they need to assume or establish an instance invariant, since they have no receiver object. However, a static method may assume instance invariants of other objects, such as argument objects passed to the method.[1]

Static invariants must be established by the static initialization of a class, and must be preserved by all non-helper constructors and methods, i.e., by both static and instance methods.

The table below summarizes this:

```
           | static          non-helper     non-helper    non-helper
           | initialization  static method  constructor   instance method
-------------------------------------------------------------------
static     | establish       preserve       preserve      preserve
invariant  |
           |
instance   | (irrelevant)    (irrelevant)   establish     preserve,
invariant  |                                              if not a
                                                          finalizer
```

A word of warning about terminology. As stated above, we call an invariant about static properties "static invariants" and we call an invariant about the dynamic properties of objects an "instance invariant" or, equivalently, an "object invariant." This terminology is contrary to the literature but it is more accurate with respect to the nomenclature of Java.

### 8.2.2 Invariants and Exceptions

Methods and constructors should preserve and establish invariants both in the case of normal termination and in the case of abrupt termination (i.e., when an exception is thrown). In other words, invariants are implicitly included in both normal postconditions, i.e., `ensures` clauses, and in exceptional postconditions, i.e., `signals` clauses, of methods and constructors.

The requirement that invariants hold after abrupt termination of a method or constructor may seen excessively strong. However, it is the only sound option in the long run. After

[1] Thanks to Peter Müller for clarifying this paragraph.

all, once an object's invariant is broken, no guarantees whatsoever can be made about subsequent method invocations on that object. When faced with a method or constructor that may violate an invariant in case it throws an exception, one will typically try to strengthen the precondition of the method to rule out this exceptional behavior or try to weaken the invariant. Note that a method that does not have any side effects when it throws an exception automatically preserves all invariants.

### 8.2.3 Access Modifiers for Invariants

Invariants can be declared with any one of the Java access modifiers `private`, `protected`, and `public`. Like class members, invariants declared in a class have `package` visibility if they do not have one of these keywords as modifier. Similarly, invariants declared in an interface implicitly have `public` visibility if they do not have one of these keywords as modifier.

The access modifier of an invariant affects which members, i.e. which fields and which (pure) methods, may be used in it, according to JML's usual visibility rules. See Section 2.4 [Privacy Modifiers and Visibility], page 12, for the details and an example using invariants.

The access modifiers of invariants do *not* affect the obligations of methods and constructors to maintain and establish them. That is, *all* non-`helper` methods are expected to preserve invariants irrespective of the access modifiers of the invariants and the methods. For example, a public method must preserve private invariants as well as public ones.

[[[ JML's visibility restrictions still allow some highly dubious invariants. E.g., a private invariant can refer to a public field, which, if this public field is not final, means the invariant is not really enforceable. Tools should warn about (or forbid??) invariants which refer to non-final non-model fields that have a looser access control than the invariant itself has. ]]]

### 8.2.4 Invariants and Inheritance

Each type inherits all the instance invariants specified in its superclasses and superinterfaces. [[[Erik wrote: "Static invariants are not inherited", but there seems to be some kind of static field inheritance in Java...]]] [[[ DRCok- but all the static invariants of a superclass have to be maintained by the subclass methods - isn't this equivalent to inheritance?]]]

The fact that (instance) invariants are inherited is one of the reasons why the use of the keyword `super` is not allowed in invariants. [[[ Is this true? - I don't understand this. DRCok ]]]

## 8.3 Constraints

History constraints [Liskov-Wing93b] [Liskov-Wing94], which we call *constraints* for short, are related to invariants. But whereas invariants are predicates that should hold in all visible states, history constraints are relationships that should hold for the combination of each visible state and any visible state that occurs later in the program's execution. Constraints can therefore be used to constrain the way that values change over time.

The syntax of history constraints in JML is as follows.

*history-constraint* ::= *constraint-keyword* *predicate*
  [ `for` *constrained-list* ] `;`
*constraint-keyword* ::= `constraint` `|` `constraint_redundantly`
*constrained-list* ::= *method-name-list* `|` `\everything`

*method-name-list* ::= *method-name* [ `,` *method-name* ] ...
*method-name* ::= *method-ref* [ `(` [ *param-disambig-list* ] `)` ] | *method-ref-start* `. *`
*method-ref* ::= *method-ref-start* [ `.` *method-ref-rest* ] ...
  | `new` *reference-type*
*method-ref-start* ::= `super` | `this` | *ident*
*method-ref-rest* ::= `this` | *ident*
*param-disambig-list* ::= *param-disambig* [ `,` *param-disambig* ] ...
*param-disambig* ::= *type-spec* [ *ident* [ *dims* ] ]

Because methods will not necessarily change the values referred to in a constraint, a constraint will generally describe reflexive and transitive relations.

For example, the constraints in the example below say that the value of field `a` and the length of the array `b` will never change, and that the length of the array `c` will only ever increase.

```
package org.jmlspecs.samples.jmlrefman;

public abstract class Constraint {

    int a;
    //@ constraint a == \old(a);

    boolean[] b;

    //@ invariant b != null;
    //@ constraint b.length == \old(b.length) ;

    boolean[] c;

    //@ invariant c != null;
    //@ constraint c.length >= \old(c.length) ;

    //@ requires bLength >= 0 && cLength >= 0;
    Constraint(int bLength, int cLength) {
      b = new boolean[bLength];
      c = new boolean[cLength];
    }
}
```

Note that, unlike invariants, constraints can – and typically do – use the JML keyword `\old`.

A constraint declaration may optionally explicitly list one or more methods. It is the listed methods that must *respect* the constraint. If no methods are listed, then all non-helper methods of the class (and any subclasses) must respect the constraint. A method respects a history constraint iff the pre-state and the post-state of a non-static method invocation are in the relation specified by the history constraint. So one can think of history constraints as being implicitly included in the postcondition of relevant methods. However, history constraints do not apply to constructors and destructors, since constructors do not have a pre-state and destructors do not have a post-state.

Private methods declared as `helper` methods do not have to respect history constraints, just like these do not have to preserve invariants.

A few points to note about history constraints:

- Constraints can be declared `static`; see Section 8.3.1 [Static vs. instance constraints], page 57.
- Constraints can be declared with the access modifiers `public`, `protected`, and `private`; see Section 8.3.2 [Access Modifiers for Constraints], page 58.
- Constraints should also hold if a method terminates abruptly by throwing an exception.
- A class inherits all constraints specified in its superclasses and superinterfaces; see Section 8.3.3 [Constraints and Inheritance], page 58.
- Although some aspects of constraints are discussed in isolation here, the full explanation of their semantics can only be given considered together with that of method specifications. After all, a method only has to respect constraints when one of the preconditions (ie. `requires` clauses) specified for that method holds. So constraints are an integral part of the explanation of method specifications in Chapter 9 [Method Specifications], page 61.
- When considering an individual method body, remember that constraints not only have to hold between the pre-state and the post-state, but between all visible state that arise during execution of the method. So, given that any program points in the method where (non-`helper`) methods or constructors are invoked are also visible states, constraints should also hold between the pre-state and any such program points, between these program points themselves, and between any such program points and the post-state.
- A method invocation on an object `o` should not just respect the constraints of `o`, but should respect the constraints of all other (reachable) objects as well.

These aspects of constraints are discussed in more detail below.

### 8.3.1 Static vs. instance constraints

History constraints can be declared `static`. Non-`static` constraints are also called *instance* constraints. Like a static invariant, a static history constraint cannot refer to the current object `this` or to its fields.

Static constraints should be respected by all constructors and all methods, i.e., both static and instance methods.

Instance constraints must be respected by all instance methods.

The table below summarizes this:

```
           | static          non-helper     non-helper    non-helper
           | initialization  static method  constructor   instance method
-------------------------------------------------------------------
static     | (irrelevant)    respect        respect       respect
constraint|
           |
instance   | (irrelevant)    (irrelevant)   (irrelevant)  respect
```

`constraint|`

Instance constraints are irrelevant for constructors, in that here there is no pre-state for a constructor that can be related (or not) to the post-state. However, if a visible state arises during the execution of a constructor, then any instance constraints have to be respected.

In the same way, and for the same reason, static constraints are irrelevant for static initialization.

### 8.3.2 Access Modifiers for Constraints

The access modifiers `public`, `private`, and `protected` pose exactly the same restrictions on constraints as they do on invariants, see Section 8.2.3 [Access Modifiers for Invariants], page 55.

### 8.3.3 Constraints and Inheritance

Any class inherits all the instance constraints specified in its superclasses and superinterfaces. [[[Static constraints are not inherited.]]] [[[ But they still apply to subclasses, no ? and it says they are above - David]]]

The fact that (instance) constraints are inherited is one of the reasons why the use of the keyword `super` is not allowed in constraints. [[[ Needs explanation - David ]]]

## 8.4 Represents Clauses

The following is the syntax for `represents` clauses.

*represents-clause* ::= *represents-keyword store-ref-expression*
        *l-arrow-or-eq spec-expression* `;`
    `|` *represents-keyword store-ref-expression* `\such_that`
     *predicate* `;`
*represents-keyword* ::= `represents` `|` `represents_redundantly`
*l-arrow-or-eq* ::= `<-` `|` `=`

The first form of `represents` clauses (with `<-` or `=`) is called a *functional abstraction*. This form defines the value of the *store-ref-expression* in a visible state as the value of the *spec-expression* that follows the *l-arrow-or-eq*.

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

- The left-hand side of a `represents` clause must be a reference to a model field (See Chapter 7 [Class and Interface Member Declarations], page 43, for details of model fields). Although it is a *store-ref-expression*, wild cards and array ranges are not permitted.
- In the functional abstraction form, the type of right-hand side of a `represents clause` must be assignment-compatible to the type of left-hand side.
- In the relational abstraction form, the type of right-hand side of a `represents clause` must be `boolean`.

A `represents` clause can be declared as `static` (See Chapter 6 [Type Definitions], page 37, for `static` declarations). In a `static represents` clause, only static elements can be referenced both in the left-hand side and the right-hand side. In addition, the following restriction is enforced:

- A `static represents` clause must be declared in the type where the model field on the left-hand side is declared.

Unless explicitly declared as `static`, a `represents` clause is non-`static` (for exceptions see see Chapter 6 [Type Definitions], page 37). A non-`static represents` clause can refer to both `static` and non-`static` elements on the right-hand side.

- A non-`static represents` clause must not have a static model field in its left-hand side.
- A non-`static represents` clause must be declared in a type descended from (or nested within) the type where the model field on the left-hand side is declared.

Note that represents clauses can be recursive. That is, a represents clause may name a field on its right hand side that is the same as the field being represented (named on the left hand side). It is the specifier's responsibility to make sure such definitions are well-defined. But such recursive represents clauses can be useful when dealing with recursive datatypes [Mueller-Poetzsch-Heffter-Leavens03].

## 8.5 Initially Clauses

The *initially-clause* has the following syntax.

*initially-clause* ::= `initially` *predicate* `;`

The meaning is that each non-helper (see Section 6.2.8 [Helper], page 41) constructor for each concrete subtype of the enclosing type (including that type itself, if it is concrete) must establish the *predicate*. Thus, the predicate can be thought of as implicitly conjoined to the postconditions of all non-helper constructors.

## 8.6 Axioms

An *axiom-clause* has the following syntax.

*axiom-clause* ::= `axiom` *predicate* `;`

Such a clause specifies that a theorem prover should assume that the given predicate is true (whenever such an assumption is needed).

[[[ example needed ]]]

## 8.7 Readable If Clauses

The syntax of the *readable-if-clause* is as follows.

*readable-if-clause* ::= `readable` *ident* `if` *predicate* `;`

Such a clause gives a condition that must be true before the field named by *ident* can be read. This field must be one declared in the type in which the declaration appears, or in a supertype of the class.

## 8.8 Writable If Clauses

The syntax of the *writeable-if-clause* is as follows.

*writable-if-clause* ::= `writable` *ident* `if` *predicate* `;`

Such a clause gives a condition that must be true before the field named by *ident* can be written. This field must be one declared in the type in which the declaration appears, or in a supertype of the class.

## 8.9 Monitors For Clause

The *monitors-for-clause* is adapted from ESC/Java [Leino-Nelson-Saxe00] [Rodriguez-etal05]. It has the following syntax.

*monitors-for-clause* ::= `monitors_for` *ident*
*l-arrow-or-eq* *spec-expression-list* `;`

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

Note that the righthand-side of the *monitors-for-clause* is not just a *store-ref-list*, but is in fact a *spec-expression-list*, where each *spec-expression* evaluates to a reference to an object.

# 9 Method Specifications

Although the use of pre- and postconditions for specification of the behavior of methods is standard, JML offers some features that are not so standard. A good example of such a feature is the distinction between normal and exceptional postconditions (in `ensures` and `signals` clauses, respectively), and the specification of frame conditions using `assignable` clauses. Another example of such a feature is that JML uses privacy modifiers to allow one to write different specification that are intended for different readers; for example, one can write a public specification for clients, a protected specification for subclasses, and a private specification to record implementation design decisions. Yet another such feature is the use of redundancy to allow one to point out important consequences of a specification for readers [Tan95] [Leavens-Baker99].

JML provides two constructs for specifying methods and constructors:

- pre- and postconditions, and
- model programs.

This chapter only discusses the first of these, which is by far the most common. Model programs are discussed in Chapter 14 [Model Programs], page 117.

## 9.1 Basic Concepts in Method Specification

[[[Discuss the “client viewpoint” here and give some basic examples here.]]]

[[[Perhaps discuss other common things to avoid repeating ourselves below...]]]

## 9.2 Organization of Method Specifications

The following gives the syntax of behavioral specifications for methods. We start with the top-level syntax that organizes these specifications.

> *method-specification* ::= *specification* `|` *extending-specification*
> *extending-specification* ::= `also` *specification*
> *specification* ::= *spec-case-seq* [ *redundant-spec* ]
>         `|` *redundant-spec*
> *spec-case-seq* ::= *spec-case* [ `also` *spec-case* ] . . .

Redundant specifications (*redundant-spec*) are discussed in Chapter 13 [Redundancy], page 113.

A *method-specification* of a method in a class or interface *must* start with the keyword `also` if (and only if) this method is already declared in the parent type that the current type extends, in one of the interfaces the class implements, or in a previous file of the refinement sequence for this type. Starting a *method-specification* with the keyword `also` is intended to tell the reader that this specification is in addition to some specifications of the method that are given in the superclass of the class, one of the interfaces it implements, or in another file in the refinement sequence.

A *method-specification* can include any number of *spec-cases*, joined by the keyword `also`, as well as a *redundant-spec*. Aside from the *redundant-spec*, each of the *spec-cases* specifies a behavior that must be satisfied by a correct implementation of the method or constructor. That is, whenever a call to the specified method or constructor satisfies the precondition of one of its *spec-cases*, the rest of the clauses in that *spec-case* must also be satisfied by the implementation [Dhara-Leavens96] [Leavens-Naumann06] [Leavens06b] [Raghavan-Leavens05] [Wills92b] [Wing83]. Model program specification cases, which have no explicit preconditions, must be satisified by all implementations.

The *spec-cases* in a *method-specification* can have several forms:

*spec-case* ::= *lightweight-spec-case* `|` *heavyweight-spec-case*
`|` *model-program*

Model programs are discussed in Chapter 14 [Model Programs], page 117. The remainder of this chapter concentrates on lightweight and heavyweight behavior specification cases. JML distinguishes between

- *heavyweight specification cases*, which start with one of the keywords `behavior`, `normal_behavior` or `exceptional_behavior`, or one of their British variant spellings keywords `behaviour`, `normal_behaviour` or `exceptional_behaviour` (these are also called behavior, normal behavior, and exceptional behavior specification cases, respectively), and
- *lightweight specification cases*, which do not contain one of these behavior keywords.

A lightweight specification case is similar to a behavior specification case, but with different defaults [Leavens-Baker-Ruby06]. It also is possible to desugar all such specification cases into behavior specification cases [Raghavan-Leavens05].

## 9.3 Access Control in Specification Cases

Heavyweight specification cases may be declared with an explicit access modifier, according to the following syntax.

*privacy* ::= `public` `|` `protected` `|` `private`

The access modifier of a heavyweight specification case cannot allow more access than the method being specified. So a `public` method may have a `private` behavior specification, but a `private` method may not have a `public` public specification. A heavyweight specification case without an explicit access modifier is considered to have default (package) access.

Lightweight specification cases have no way to explicitly specify an access modifier, so their access modifier is implicitly the same as the method being specified. For example, a lightweight specification of a `public` method has `public` access, implicitly, but a lightweight specification of a `private` method has `private` access, implicitly. Note that this is a different default than that for heavyweight specifications, where an omitted access modifier always means package access.

The access modifier of a specification case affects only which annotations are visible in the specification and does *not* affect the semantics of a specification case in any other way.

JML's usual visibility rules apply to specification cases. So, for example, a public specification case may only refer to public members, a protected specification case may refer to both public and protected members, as long as the protected members are otherwise accessible according to Java's rules, etc. See Section 2.4 [Privacy Modifiers and Visibility], page 12, for more details and examples.

## 9.4 Lightweight Specification Cases

### Syntax

The following is the syntax of lightweight specification cases. These are the most concise specification cases.

*lightweight-spec-case* ::= *generic-spec-case*
*generic-spec-case* ::= [ *spec-var-decls* ]
*spec-header*
[ *generic-spec-body* ]
| [ *spec-var-decls* ]
*generic-spec-body*
*generic-spec-body* ::= *simple-spec-body*
| {| *generic-spec-case-seq* |}
*generic-spec-case-seq* ::= *generic-spec-case*
[ `also` *generic-spec-case* ] . . .
*spec-header* ::= *requires-clause* [ *requires-clause* ] . . .
*simple-spec-body* ::= *simple-spec-body-clause*
[ *simple-spec-body-clause* ] . . .
*simple-spec-body-clause* ::= *diverges-clause*
| *assignable-clause* | *captures-clause*
| *when-clause* | *working-space-clause*
| *duration-clause* | *ensures-clause*
| *signals-only-clause* | *signals-clause*

[[[ Is this list missing measured_by, accessible, callable? – DRC ]]]

As far as the syntax is concerned, the only difference between a lightweight specification case and a *behavior-specification-case* (see Section 9.6 [Behavior Specification Cases], page 65) is that the latter has the keyword `behavior` and possibly an access control modifier.

A lightweight specification case always has the same access modifier as the method being specified, see Section 9.3 [Access Control in Specification Cases], page 62. To specify a different access control modifier, one must use a heavyweight specification.

### Semantics

A lightweight specification case can be understood as syntactic sugar for a behavior specification case, except that the defaults for omitted specification clauses are different for lightweight specification cases than for behavior specification cases. So, for example, apart from the class names, method `m` in class `Lightweight` below

```
package org.jmlspecs.samples.jmlrefman;

public abstract class Lightweight {

    protected boolean P, Q, R;
    protected int X;

    /*@ requires P;
      @  assignable X;
```

```
      @  ensures Q;
      @  signals (Exception) R;
      @*/
    protected abstract int m() throws Exception;
}
```

has a specification that is equivalent to that of method `m` in class `Heavyweight` below.

```
package org.jmlspecs.samples.jmlrefman;

public abstract class Heavyweight {

    protected boolean P, Q, R;
    protected int X;

    /*@ protected behavior
      @   requires P;
      @   diverges false;
      @   assignable X;
      @   when \not_specified;
      @   working_space \not_specified;
      @   duration \not_specified;
      @   ensures Q;
      @   signals_only Exception;
      @   signals (Exception) R;
      @*/
    protected abstract int m() throws Exception;
}
```

As this example illustrates, the default for an omitted clause in a lightweight specification is `\not_specified` for all clauses, except `diverges`, which has a default of `false`, and `signals` [Leavens-Baker-Ruby06]. The default for an omitted `signals` clause is to only permit the exceptions declared in the method's header to be thrown. Thus, if the method declares that exceptions `DE1` and `DE2` may be thrown, then the default for an omitted `signals` clause is

signals (Exception e) e instanceof DE1 `||` e instanceof DE2;

It is intended that the meaning of `\not_specified` may vary between different uses of a JML specification. For example, a static checker might treat a `requires` clause that is `\not_specified` as if it were `true`, while a verification logic may decide to treat it as if it were `false`.

A completely omitted specification is taken to be a lightweight specification. If the default (zero-argument) constructor of a class is omitted because its code is omitted, then its specification defaults to an assignable clause that allows all the locations that the default (zero-argument) constructor of its superclass assigns — in essence a copy of the superclass's default constructor's assignable clause. If some other frame is desired, then one has to write the specification, or at least the code, explicitly.

A method or constructor with code present has a *completely omitted* specification if it has no *specification-cases* and does not use annotations like `non_null` or `pure` that add implicit specifications.

If a method or constructor has code, has a completely omitted specification, and does not override another method, then its meaning is taken as the lightweight specification `diverges \not_specified;`. Thus, its meaning can be read from the lightweight column of table above, except that the diverges clause is not given its usual default. This is done so that the default specification when no specification is given truly says nothing about the method's behavior. However, if a method with code and a completely omitted specification overrides some other method, then its meaning is taken to be the lightweight specification `also requires false;`. This somewhat counter-intuitive specification is the unit under specification conjunction with `also`; it is used so as not to change the meaning of the inherited specification.

If the code is annotated with keywords like `non_null` or `pure` that add implicit specifications, then these implicit specifications are used instead of the default. Code with such annotations is considered to have an implicit specification.

## 9.5 Heavyweight Specification Cases

There are three kinds of heavyweight specification cases, called behavior, normal behavior, and exceptional behavior specification cases, beginning (after an optional privacy modifier) with the one of the keywords `behavior`, `normal_behavior`, or `exceptional_behavior`, respectively.

*heavyweight-spec-case* ::= *behavior-spec-case*
    `|` *exceptional-behavior-spec-case*
    `|` *normal-behavior-spec-case*

Like lightweight specification cases, normal behavior and exceptional behavior specification cases can be understood as syntactic sugar for special kinds of `behavior` specification cases [Raghavan-Leavens05].

## 9.6 Behavior Specification Cases

The behavior specification case is the most general form of specification case. All other forms of specification cases simply provide some syntactic sugar for special kinds of `behavior` specification cases.

### Syntax

As far as the syntax is concerned, the only difference between a `behavior` specification case and a lightweight one is the optional access control modifier, *privacy*, and the keyword `behavior` (or the British variant, `behaviour`). One can use either the British or the American spelling of this keyword, although for historical reasons most examples will use the American spelling.

*behavior-spec-case* ::= [ *privacy* ] [ `code` ] *behavior-keyword*
    *generic-spec-case*
*behavior-keyword* ::= `behavior` `|` `behaviour`

See Section 15.2 [Code Contracts], page 122, for details of the semantics of *behavior-spec-cases* that use the `code` keyword.

## Semantics

To explain the semantics of a behavior specification case we make a distinction between flat and nested specification cases:

- *Flat* specification cases are of the form

  `behavior` [ *spec-var-decls* ] [ *spec-header* ] *simple-spec-body*

  A flat specification case is just made up of a sequence of method specification clauses, ie. `require`, `ensures`, etc. clauses, and its semantics is explained directly in Section 9.6.1 [Semantics of flat behavior specification cases], page 66.

- *Nested* specification cases are all other specification cases. They use the special brackets `{|` and `|}` to nest specification clauses and possibly also `also` inside these brackets to join several specification cases.

  A nested specification case can be syntactically desugared into a list of one or more simple specification cases, joined by the `also` keyword [Raghavan-Leavens05]. This is explained in Section 9.6.5 [Semantics of nested behavior specification cases], page 69.

## Invariants and constraints

The semantics of a behavior specification case for a method or constructor in a class depends on the invariants and constraints that have been specified. This is discussed in Section 8.2 [Invariants], page 50 and Section 8.3 [Constraints], page 55. In a nutshell, methods must preserve invariants and respect constraints, and constructors must establish invariants.

### 9.6.1 Semantics of flat behavior specification cases

Below we explain the semantics of a simple *behavior-spec-case* case with precisely one `requires` clause, one `diverges` clause, one `measured_by` clause, one `assignable` clause, one `accessible` clause, one `callable` clause, one `when` clause, one `ensures` clause, one `duration` clause, one `working_space` clause, one `signals_only` clause, and one `signals` clause.

A `behavior` specification case can contain any number of these clauses, and there are defaults that allow any of them to be omitted. However, as explained in Section 9.9 [Method Specification Clauses], page 73, any `behavior` specification case is equivalent with a `behavior` specification case of this form.

### 9.6.2 Non-helper methods

Consider a non-`helper` instance method `m`, and a specification case of the following form.

```
behavior
  forall T1 x1; ... forall Tn xn;
  old U1 y1 = F1; ... old Uk yk = Fk;
  requires P;
  measured_by Mbe if Mbp;
  diverges D;
  when W;
  accessible R;
  assignable A;
  callable p1(...), ..., pl(...);
  captures Z;
```

```
ensures Q;
signals_only E1, ..., Eo;
signals (E e) S;
working_space Wse if Wsp;
duration De if Dp;
```

The meaning of this specification case is as follows.

Consider a particular call of the method *m*.

The state of the program after passing parameters to *m*, but before running any of the code of *m* is called the *pre-state* of the method call.

Suppose all applicable invariants hold in the pre-state of this call.

For every possible value of the variables declared in the `forall` clauses, *x1*, ..., *xn*, the following must be true. (If there are no `forall` clauses, then the following just has to hold all by itself.)

Suppose that the variable *y1* is bound to the pre-state value of *F1* in the pre-state (i.e., the beginning of the method, after parameter passing), and in turn each of the `old` variable declarations are bound to the values of the corresponding expressions, also evaluated in the pre-state, and finally *yk* is bound to the value of *Fk* in the pre-state. These bindings can depend on previously defined `old` variable declarations in the specification case. (If there are no `old` clauses, then no such variables are bound.) We call the state with such bindings in place the *augmented pre-state*.

Suppose also that with these binding (i.e., in the augmented pre-state), that the precondition, *P*, from the `requires` clause, holds.

If the method has a `measured_by` clause, and if the predicate in the `measured_by` clause, *Mbp*, is true in the augmented pre-state, and if this call is in the control flow of another instance of this method, *Caller*, then the value of the expression *Mbe* in this call's augmented pre-state must be non-negative and strictly less than the value of *Mbe* in the pre-state of *Caller*. (If the `measured_by` clause is omitted, there is no such requirement.) For example, consider a method `fib` that calls itself directly and has an integer parameter `n` and for which the `measured_by` clause has `n` as its expression (*Mbe*), and the default predicate (*Mbp*) is true; then recursive calls of `fib` that appear in the body of `fib` must have actual argument exprssions whose value is (non-negative and) strictly less than `n`, such as `n-1` and `n-2`.[1]

Then one of the following must also hold:

- the `diverges` predicate, *D*, holds in the augmented pre-state and the execution of the method does not terminate (i.e., it loops forever or the Java virtual machine exits in such a way that the method call does not return or throw an exception). (If the `diverges` clause is omitted, then the default for *D* is `false`, and hence these outcomes are effectively prohibited.) or
- the Java virtual machine throws an error (i.e., an instance of `java.lang.Throwable` whose type does not inherit from `java.lang.Exception`, usually an instance of `java.lang.Error`), or
- the method terminates by returning or throwing an exception, reaching a state called its *post-state*, in which all of the following hold.

[1] Thanks to Jesus Ravelo for correcting the semantics of measured-by clauses.

- The method's execution only reaches its commit point (a label in the method body with the name "`commit`" [Rogriguez-etal05]) in a state such that the `when` clause's condition, *W*, holds. (If the condition does not hold, then the method's execution waits for a concurrent thread to make it true, and then proceeds. There is no guarantee that the method will proceed the first time this condition holds, so the condition may have to hold many times before the thread may proceed to its commit point.) (If the `when` clause is omitted, there is no need to have a commit point in the method, and the method need not wait for the execution of concurrent threads.)
- During execution of the method (which includes all directly and indirectly called methods and constructors), only locations that either did not exist in the pre-state, that are local to the method (including the method's formal parameters), or that are either named in the lists *R* and *A* found in the `accessible` and `assignable` clauses or that are dependees (see Chapter 10 [Data Groups], page 85) of such locations, are read from. The set of locations named by the accessible and assignable clauses (and hence the elements of their data groups) are computed in the pre-state. (If the `accessible` clause is omitted, it defaults to `accessible \everything;`, which allows all locations to be accessed.)
- During execution of the method, only locations that either did not exist in the pre-state, that are local to the method, or that are either named by the `assignable` clause's list, *A*, or are dependees (see Chapter 10 [Data Groups], page 85) of such locations, are assigned to. The set of locations named by the assignable clause (and hence the elements of their data groups) are computed in the pre-state. (If the `assignable` clause is omitted, it defaults to `assignable \everything;`, which allows all locations to be assigned.)
- During execution of the method, the only methods and constructors called are those listed in the `callable` clause's list *p1*, ..., *pl*. (If the `callable` clause is omitted, it defaults to `callable \everything;`, which allows all methods and constructors to be called.)

  The form *p*`.*` refers to all methods of the object denoted by *p*.
- During execution of the method, of the formal parameters whose type is a reference type, only those listed in the `captures` clause's list, *Z*, may be assigned to fields of some object or to array elements. (References in formals may freely be assigned to local variables, however, as these are "borrowed" but not captured [Boyland00]. If the `captures` clause is omitted, then all such formals may be assigned freely.)
- If the execution of the method terminates by returning normally, then the normal postcondition, *Q*, given in the `ensures` clause, holds in the post-state.
- If the execution of the method terminates by throwing an exception of some type *Ea* that is a subtype of `java.lang.Exception`, then:
  - the type *Ea* must be a subtype of some type in the list *E1*, ..., *Eo*, listed in the `signals_only` clause (this list of types has as its default the list in the method's `throws` clause), and
  - if *Ea* is a subtype of the type *E* given in the `signals` clause, then the exceptional postcondition *R* must hold in the post-state, augmented by a binding from the variable *e* to the exception object thrown.

- All applicable invariants and history constraints hold in the post-state.
- If the predicate in the `working_space` clause, *Wsp*, was true in the augmented pre-state, then the method execution had available to it the amount of heap space, in bytes, *Wse* [Krone-Ogden-Sitaraman03]. (Note that the expression *Wse* may depend on post-state values so this expression is conceptually evaluated in the post-state, although it may use `\old()` to refer to pre-state values. If the `working_space` clause is omitted, there is no restriction placed on the maximum space that the method call may during its execution.)
- If the predicate in the `duration` clause, *Dp*, was true in the augmented pre-state, then the method execution used no more than the number of virtual machine cycles given by the expression *De* [Krone-Ogden-Sitaraman03]. (Note that the expression *De* may depend on post-state values so this expression is conceptually evaluated in the post-state, although it may use `\old()` to refer to pre-state values. If the `duration` clause is omitted, there is no restriction placed on the maximum number of virtual machine cycles that the call may use during its execution.)

In all of these clauses, the value of a formal parameter is always considered to be the value they had in the pre-state. That is the actual post-state value they take in an execution is not considered, as explained in See Section 9.9.6 [Parameters in Postconditions], page 78.

### 9.6.3 Non-helper constructors

The semantics of a flat specification case for a (non-`helper`) constructor is the same as that for a (non-`helper`) method given above, except that:

- any instance invariants of the object being initialized by the constructor are not assumed to hold in the precondition,
- any instance constraints do not have to be established as implicit part of the postcondition of the constructor.

These two differences are also discussed in Section 8.2 [Invariants], page 50 and Section 8.3 [Constraints], page 55.

### 9.6.4 Helper methods and constructors

The semantics of a flat specification case for a helper method (or constructor) is the same as that for a non-helper method (or constructor) given above, except that:

- the instance invariants for the current object and the static invariants for the current class are not assumed to hold in the pre-state, and do not have to be established in the post-state.
- the instance constraints for current object and the static constraints for the current class do not have to be established in the post-state

These differences are also discussed in Section 8.2 [Invariants], page 50 and Section 8.3 [Constraints], page 55.

### 9.6.5 Semantics of nested behavior specification cases

We now explain how all behavior specification cases can be desugared into a list of one or more flat specification cases joined by the `also` keyword [Raghavan-Leavens05]. The semantics of a behavior specification case is then simply the semantics of this desugared version.

The desugaring is as follows. Consider a specification of the form.

```
spec-var-decls
spec-header
{|
    GenSpecCase1
  also
    ...
  also
    GenSpecCasen
|}
```

The above desugars to the following.

```
    spec-var-decls
    spec-header
    GenSpecCase1
  also
    ...
  also
    spec-var-decls
    spec-header
    GenSpecCasen
```

In the above desugaring either the *spec-var-decls* or the *spec-header* (or both) may be omitted.

The meaning of the desugared list of specification cases is explained in Section 9.2 [Organization of Method Specifications], page 61. The meaning of a single simple specification case is explained in Section 9.6.1 [Semantics of flat behavior specification cases], page 66.

## 9.7 Normal Behavior Specification Cases

A `normal_behavior` specification case is just syntactic sugar for a `behavior` specification case with an implicit `signals` clause

```
signals (java.lang.Exception) false;
```

ruling out abrupt termination, i.e., the throwing of any exception. Note that this includes unchecked exceptions, since in Java, `RuntimeException` is a subclass of `Exception`.

The following gives the syntax of the body of a normal behavior specification case.

*normal-behavior-spec-case* ::= [ *privacy* ] [ `code` ] *normal-behavior-keyword*
*normal-spec-case*
*normal-behavior-keyword* ::= `normal_behavior` | `normal_behaviour`
*normal-spec-case* ::= *generic-spec-case*

As far as syntax is concerned, the only difference between a *normal-spec-case* and a *generic-spec-case* is that normal behavior specification cases cannot include *signals-clauses* or *signals-only-clauses*.

The semantics of a normal behavior specification case is the same as the corresponding `behavior` specification case (see Section 9.6 [Behavior Specification Cases], page 65) with the addition of the following *signals-clause*

```
signals (java.lang.Exception) false;
```

So a normal behavior specification case specifies a precondition which guarantees normal termination; i.e., it prohibits the method from throwing an exception.

## 9.8 Exceptional Behavior Specification Cases

The following gives the syntax of the body of an exceptional behavior specification case.

*exceptional-behavior-spec-case* ::= [ *privacy* ] [ `code` ] `exceptional-behavior-keyword`
*exceptional-spec-case*
*exceptional-behavior-keyword* ::= `exceptional_behavior` | `exceptional_behaviour`
*exceptional-spec-case* ::= *generic-spec-case*

As far as syntax is concerned, the only difference between an *exceptional-spec-case* and a *generic-spec-case* is that exceptional behavior specification cases cannot include *ensures-clauses*.

The semantics of an exceptional behavior specification case is the same as the corresponding behavior specification case (see Section 9.6 [Behavior Specification Cases], page 65) with the addition of the following `ensures` clause.

```
ensures false;
```

So an exceptional behavior specification case specifies a precondition which guarantees that the method throws an exception, if it terminates, i.e., a precondition which prohibits the method from terminating normally.

### 9.8.1 Pragmatics of Exceptional Behavior Specifications Cases

Note that an exceptional behavior specification case says that some exception *must* be thrown if its precondition is met (assuming the diverges clause predicate is `false`, as is the default.) Beware of the difference between specifying that an exception *must* be thrown and specifying that an exception *may* be thrown. To specify that an exception *may* be thrown you should *not* use an exceptional behavior, but should instead use a behavior specification case [Leavens-Baker-Ruby06].

For example, the following method specification

```
package org.jmlspecs.samples.jmlrefman;

public abstract class InconsistentMethodSpec {

    /** A specification that can't be satisfied. */
    /*@  public normal_behavior
     @    requires z <= 99;
     @    assignable \nothing;
     @    ensures \result > z;
     @ also
     @  public exceptional_behavior
     @    requires z < 0;
```

```
     @    assignable \nothing;
     @    signals (IllegalArgumentException) true;
     @*/
    public abstract int cantBeSatisfied(int z)
        throws IllegalArgumentException;
}
```

is *inconsistent* because the preconditions `z <= 99` and `z < 0` overlap, for example when `z` is `-1`. When both preconditions hold then the exceptional behavior case specifies that an exception *must* be thrown and the normal behavior case specifies that an exception *must not* be thrown, but the implementation cannot both throw and not throw an exception.

Similarly, multiple exceptional specification cases with overlapping preconditions may give rise to an inconsistent specification. For example, the following method specification

```
package org.jmlspecs.samples.jmlrefman;

public abstract class InconsistentMethodSpec2 {

    /** A specification that can't be satisfied. */
    /*@  public exceptional_behavior
     @     requires z < 99;
     @     assignable \nothing;
     @     signals_only IllegalArgumentException;
     @ also
     @   public exceptional_behavior
     @     requires z > 0;
     @     assignable \nothing;
     @     signals_only NullPointerException;
     @*/
    public abstract int cantBeSatisfied(int z)
        throws IllegalArgumentException, NullPointerException;
}
```

is inconsistent because, again, the two preconditions overlap, and the `signals_only` clauses do not permit the same exception to be thrown in both cases.

There is an important distinction to be made between the `signals` and the `signals_only` clauses in JML. The `signals_only` clause says what exceptions may be thrown (when the specification case's precondition is met); this clause does not say anything about the state of the exception object or other locations in the system. On the other hand, the `signals` clause only describes what must be true of the system state when an exception is thrown, and does not say anything about what exceptions may be thrown. For example, consider the following specification.

```
package org.jmlspecs.samples.jmlrefman;

public abstract class SignalsClause {

    /*@ signals (IllegalArgumentException) x < 0;
      @ signals (NullPointerException) x < 0;
```

```
      @*/
    public abstract int notPrecise(int x) throws RuntimeException;
}
```

The above allows a method to throw either an `IllegalArgumentException` or a `NullPointerException` when `x` is less than 0, but in that condition the method might also throw a different exception altogether, as long as that exception was permitted by the method's declaration header. The only thing ruled out by this specification is throwing either a `IllegalArgumentException` or a `NullPointerException` when `x` is not less than 0. Thus from such a specification one may draw the conclusion that `x < 0` only when one of these two exceptions is thrown.

Therefore, if one just wants to specify the exceptions that are permitted to be thrown in a specific situation, one should use the `signals_only` clause.

## 9.9 Method Specification Clauses

The different kinds of clauses that can be used in method specifications are discussed in this section. See Section 9.4 [Lightweight Specification Cases], page 63, for the overall syntax that ties these clauses together.

### 9.9.1 Specification Variable Declarations

The syntax of *spec-var-decls* is as follows.

*spec-var-decls* ::= *forall-var-decls* [ *old-var-decls* ]
    `|` *old-var-decls*

The scope of the variables declared in the *spec-var-decls* is the entire specification case in which they appear. The two types of such declarations are described below.

#### 9.9.1.1 Forall Variable Declarations

The syntax of the *forall-var-decls* is as follows.

*forall-var-decls* ::= *forall-var-declarator* [ *forall-var-declarator* ] . . .
*forall-var-declarator* ::= `forall` [ *bound-var-modifiers* ] *quantified-var-declarator* `;`

When a *forall-var-declarator* is used, it specifies that the specification case that follows must hold for every possible value of the declared variables. In other words, it is a universal quantification over the specification case.

Note that if such variables are used in preconditions, then they can be thought to range over all values that satisfy the preconditions. The bound variable may not rename earlier bound variables in the specification, nor the formal parameters of the method declaration.

#### 9.9.1.2 Old Variable Declarations

The syntax of the *old-var-decls* is as follows. See Section 7.1.2.2 [Type-Specs], page 48, for the syntax of *type-spec*. [[[Give cross ref for *spec-variable-declarators* when ready.]]]

*old-var-decls* ::= *old-var-declarator* [ *old-var-declarator* ] . . .
*old-var-declarator* ::= `old` [ *bound-var-modifiers* ] *type-spec* *spec-variable-declarators* `;`

An *old-var-declarator* allows abbreviation within a specification case. The names defined in the *spec-variable-declarators* can be used throughout the specification case for the values of their initializers. As the name suggests, the expressions are evaluated in the method's pre-state. The bound variable may not rename earlier bound variables in the specification, nor the formal parameters of the method declaration.

[[[Example]]]

### 9.9.2 Requires Clauses

A requires clause specifies a precondition of method or constructor. Its syntax is as follows.

> *requires-clause* ::= *requires-keyword* *pred-or-not* `;`
> | *requires-keyword* `\same ;`
> *requires-keyword* ::= `requires` | `pre`
> | `requires_redundantly` | `pre_redundantly`
> *pred-or-not* ::= *predicate* | `\not_specified`

The *predicate* in a `requires` clause can refer to any visible fields and to the parameters of the method. See Section 2.4 [Privacy Modifiers and Visibility], page 12, for more details on visibility in JML.

Any number of requires clauses can be included a single specification case. Multiple requires clauses in a specification case mean the same as a single requires clause whose precondition predicate is the *conjunction* of these precondition predicates in the given requires clauses. For example,

```
requires P;
requires Q;
```

means the same thing as:

```
requires P && Q;
```

When a requires clause is omitted in a specification case, a default requires clause is used. For a lightweight specification case, the default precondition is `\not_specified`. The default precondition for a heavyweight specification case is `true`.

At most one precondition in a specification case can use `\same`, and `\same` cannot be used in the only specification case for a method unless the method is an override. Similarly, `\same` cannot be used in the only specification case for a constructor or a static method. Another restriction is that `\same` cannot be used in a requires clause of a nested specification case (see Section 9.6.5 [Semantics of nested behavior specification cases], page 69).

When the precondition is `\same` in a specification case, it means that the specification case being written has, effectively, the same precondition as that specified in the other (non-`\same`) specification cases. That is, `\same` stands for the disjunction of the preconditions in all non-`\same` specification cases of the method's specification from the current class together with the inherited specification cases defined in its supertypes (i.e., in its superclasses and implemented interfaces).

### 9.9.3 Ensures Clauses

An ensures clause specifies a normal postcondition, i.e., a property that is guaranteed to hold at the end of the method (or constructor) invocation in the case that this method (or constructor) invocation returns without throwing an exception. The syntax is as follows See Section 9.9.2 [Requires Clauses], page 74, for the syntax of *pred-or-not*.

> *ensures-clause* ::= *ensures-keyword* *pred-or-not* `;`
> *ensures-keyword* ::= `ensures` | `post`

```
        | ensures_redundantly | post_redundantly
```

A *predicate* in an `ensures` clause can refer to any visible fields, the parameters of the method, `\result` if the method is non-void, and may contain expressions of the form `\old(`*E*`)`. See Section 2.4 [Privacy Modifiers and Visibility], page 12, for more details on visibility in JML.

Informally,

```
ensures Q;
```

means

> if the method invocation terminates normally (ie. without throwing an exception), then predicate $Q$ holds in the post-state.

In an ensures clause, `\result` stands for the result that is returned by the method. The postcondition $Q$ may contain expressions of the form `\old(e)`. Such expressions are evaluated in the pre-state, and not in the post-state, and allow $Q$ to express a relation between the pre- and the post-state. If parameters of the method occur in the postcondition $Q$, these are always evaluated in the pre-state, not the post-state. In other words, if a method parameter $x$ occurs in $Q$, it is treated as `\old(`$x$`)`. For a detailed explanation of this see Section 9.9.6 [Parameters in Postconditions], page 78.

Any number of ensures clauses can be given in a single specification case. Multiple ensures clauses in a specification case mean the same as a single ensures clause whose postcondition predicate is the *conjunction* of the postcondition predicates in the given ensures clauses. So

```
ensures P;
ensures Q;
```

means the same as

```
ensures P && Q;
```

Note that, in JML's semantics for expressions within assertions, the order of evaluation of $P$ and $Q$ does not matter. See Section 2.7 [Expression Evaluation and Undefinedness], page 15, for more details on this topic.

When an ensures clause is omitted in a specification case, a default ensures clause is used. For a lightweight specification case, the default precondition is `\not_specified`. The default precondition for a heavyweight specification case is `true`.

### 9.9.4 Signals Clauses

In a specification case a `signals` clause specifies the exceptional or abnormal postcondition, i.e., the property that is guaranteed to hold at the end of a method (or constructor) invocation when this method (or constructor) invocation terminates abruptly by throwing a given exception.

The syntax is as follows. See Section 9.9.2 [Requires Clauses], page 74, for the syntax of *pred-or-not*.

*signals-clause* ::= *signals-keyword* `(` *reference-type* [ *ident* ] `)`
          [ *pred-or-not* ] `;`
*signals-keyword* ::= `signals` | `signals_redundantly`
      | `exsures` | `exsures_redundantly`

In a *signals-clause* of the form

```
signals (E e) P;
```

*E* has to be a subclass of `java.lang.Exception`, and the variable *e* is bound in *P*. If *E* is a checked exception (i.e., if it does not inherit from `java.lang.RuntimeException` [Arnold-Gosling-Holmes00] [Gosling-etal00]), it must either be one of the exceptions listed in the method or constructor's `throws` clause, or a subclass or a superclass of such a declared exception.

Informally,

```
signals (E e) P;
```

means

> If the method (or constructor) invocation terminates abruptly by throwing an exception of type *E*, then predicate *P* holds in the final state for this exception object *E*.

A signals clause of the form

```
signals (E e) R;
```

is equivalent to the signals clause

```
signals (java.lang.Exception e) (e instanceof E) ==> R;
```

Several signals clauses can be given in a single lightweight, behavior or exceptional behavior specification case. Multiple signals clauses in a specification case mean the same as a single signals clause whose exceptional postcondition predicate is the *conjunction* of the exceptional postcondition predicates in the given signals clauses. This should be understood to take place after the desugaring given above, which makes all the signals clauses refer to exceptions of type `java.lang.Exception`. Also, the names in the given signals clauses have to be standardized [Raghavan-Leavens05]. So for example,

```
signals (E1 e) R1;
signals (E2 e) R2;
```

means the same as

```
signals (Exception e)   ((e instanceof E1) ==> R1)
                     && ((e instanceof E2) ==> R2);
```

Note that this means that if an exception is thrown that is both of type *E1* and of type *E2*, then both *R1* and *R2* must hold.

[[[EXAMPLE]]]

Beware that a `signals` clause specifies when a certain exception *may* be thrown, not when a certain exception *must* be thrown. To say that an exception must be thrown in some situation, one has to exclude that situation from other signals clauses and from ensures clause (and any diverges clauses). It may also be useful to use the `signals_only` clause in such specifications (see Section 9.9.5 [Signals-Only Clauses], page 77).

[[[EXAMPLE?]]]

When a behavior or exceptional specification case has no *signals-clause*, a default signals clause is used. For a heavyweight specification case, the default signals clause is `signals (Exception) true;`. Since normal behavior specification cases do not have signals clauses, no default applies for such specification cases. For a lightweight specification case, the default is `signals \not_specified;`.

### 9.9.5 Signals-Only Clauses

A `signals_only` clause is an abbreviation for a *signals-clause* (see Section 9.9.4 [Signals Clauses], page 75) that specifies what exceptions may be thrown by a method, and thus, implicitly, what exceptions may *not* be thrown.

The syntax is as follows.

*signals-only-clause* ::= *signals-only-keyword reference-type* [ `,` *reference-type* ] . . . `;`
    `|` *signals-only-keyword* `\nothing ;`
*signals-only-keyword* ::= `signals_only` `|` `signals_only_redundantly`

All of the *reference-type*s named in a *signals-only-clause* must be subtypes of `java.lang.Exception`. Each *reference-type* that is a checked exception type (i.e., that does not inherit from `java.lang.RuntimeException` [Arnold-Gosling-Holmes00] [Gosling-etal00]), must either be one of the exceptions listed in the method or constructor's `throws` clause, or a subclass or a superclass of such a declared exception.

A *signals-only-clause* of the form

```
signals_only E1, E2, ..., En;
```

is considered to be an abbreviation (syntactic sugar) for the following *signals* clause (see Section 9.9.4 [Signals Clauses], page 75).

```
signals (java.lang.Exception e)
            e instanceof E1
         || e instanceof E2
         || ...
         || e instanceof En;
```

That is, such a clause specifies that if the method or constructor throws an exception, it must be an instance of one of the types named.

Several *signals-only-clause*s can be given in a single lightweight, behavior or exceptional behavior specification case. Multiple such clauses in a specification case mean the same as a single clause whose list contains only the names *Ej* that are subtypes of some type named in all of the given *signals-only-clause*s. Thus, the meaning is a kind of intersection of the `signals_only` clauses. Since this may be confusing, only one `signals_only` clause should ever be used in a given specification case.

The `signals_only` clause is useful for specifying when a certain exception, or one of a small set of exceptions, *must* be thrown. To say that an exception must be thrown in some situation, one has to exclude the method from returning normally in that situation (using an ensures clause or the precondition of some other specification case) and from not terminating (by using the diverges clause).

[[[Example]]]

If the `signals_only` is omitted from a specification case, a default `signals_only` clause is provided. The same default is used for both lightweight and heavyweight behavior and exceptional behavior specification cases. (Since normal behavior specification cases cannot throw exceptions at all, there is no default `signals_only` clause for such specification cases.) This default prohibits any exception not declared by the method in the method's header from being thrown. Thus the exact default depends on the method header. If the method header does not list any exceptions that can be thrown, then the default is `signals_only`

`\nothing;` (which means that the method cannot throw any exceptions). However, if the method header declares that the method may throw exceptions *DE_1*, ..., *DE_n*, *Err_1*, ..., *Err_m*, where each *DE_i* is a subtype of `java.lang.Exception`, and each *Err_j* is not a subtype of `java.lang.Exception`, then the default `signals_only` clause is as follows.

```
signals_only DE_1, ..., DE_n
```

For example, if the method has the header

public void foo() throws E1, E2

then the default `signals_only` clause would be

signals_only E1, E2;

It is important to note that the set of exceptions included in the default `signals` clause described above never includes `java.lang.Throwable`, and does not include `java.lang.Error` or any of its subtypes. Furthermore, this default would not normally include `java.lang.RuntimeException` or any of its subtypes, because Java explicitly allows RuntimeExceptions to be thrown even if they are not declared in the method header's `throws` clause. Since such unchecked, runtime exceptions are not usually listed in the method header, they would not find their way into the default `signals_only` clause. In JML, however, if you wish to allow such runtime exceptions, you can either explicitly list them in the method header or, more usually, you would list them in an explicit `signals_only` clause.

### 9.9.6 Parameters in Postconditions

Parameters of methods are passed by value in Java, meaning that parameters are local variables in a method body, which are initialized when the method is called with the values of the parameters for the invocation.

This leads us to the following two rules:

- The parameters of a method or constructor can never be listed in its assignable clause.
- If parameters of a method (or constructor) are used in a normal or exceptional postcondition for that method (or constructor), i.e., in an ensures or signals clause, then these always have their value in the pre-state of the method (or constructor), not the post-state. In other words, there is an implicit `\old()` placed around any occurrence of a formal parameter in a postcondition.

The justification for the first convention is that clients cannot observe assignments to the parameters anyway, as these are local variables that can only be used by the implementation of the method. Given that clients can never observe these assignments, there is no point in making them part of the contract between a class and its clients.

The justification for the second convention is that clients only know the initial values of the parameter that they supply, and do not have any knowledge of the final values that these variables may have in the post-state.

The reason for this is best illustrated by an example. Consider the following class and its method specifications. Without the convention described above the implementations given for methods `notCorrect1` and `notCorrect2` would satisfy their specifications. However, clearly neither of these satisfies the specification when read from the caller's point of view.

```
package org.jmlspecs.samples.jmlrefman;
```

```
public abstract class ImplicitOld {

    /*@ ensures 0 <= \result && \result <= x;
      @ signals (Exception) x < 0;
      @*/
    public static int notCorrect1(int x) throws Exception {
        x = 5;
        return 4;
    }

    /*@ ensures 0 <= \result && \result <= x;
      @ signals (Exception) x < 0;
      @*/
    public static int notCorrect2(int x) throws Exception {
        x = -1;
        throw new Exception();
    }

    /*@ ensures 0 <= \result && \result <= x;
      @ signals (Exception) x < 0;
      @*/
    public static int correct(int x) throws Exception {
        if (x < 0) {
            throw new Exception();
        } else {
            return 0;
        }
    }
}
```

The convention above rules out such pathological implementations as `notCorrect1` above; because mention of a formal parameter name, such as `x` above, in postconditions always means the pre-state value of that name, e.g., `\old(x)` in the example above.

### 9.9.7 Diverges Clauses

The diverges clause is a seldom-used feature of JML. It says when a method may loop forever or otherwise not return to its caller, by either throwing an exception or returning normally. The syntax is as follows See Section 9.9.2 [Requires Clauses], page 74, for the syntax of *pred-or-not*.

*diverges-clause* ::= *diverges-keyword* *pred-or-not* `;`
*diverges-keyword* ::= `diverges` | `diverges_redundantly`

When a diverges clause is omitted in a specification case, a default diverges clause is used. For both lightweight and heavyweight specification cases, the default diverges condition is `false`. Thus by default, specification cases give total correctness specifications [Dijkstra76]. Explicitly writing a diverges clause allows one to obtain a partial correctness specification [Hoare69]. Being able to specify both total and partial correctness specification cases for a method leads to additional power [Hesselink92] [Nelson89].

As an example of the use of `diverges`, consider the `exit` method in the following class. (This example is simplified from the specification of Java's `System.exit` method. This specification says that the method can always be called (the implicit precondition is `true`), may always not return to the caller (i.e., diverge), and may never return normally, and may never throw an exception. Thus the only thing the method can legally do, aside from causing a JVM error, is to not return to its caller.

```
package org.jmlspecs.samples.jmlrefman;

public abstract class Diverges {

    /*@ public behavior
      @    diverges true;
      @    assignable \nothing;
      @    ensures false;
      @    signals (Exception) false;
      @*/
    public static void abort();

}
```

The diverges clause is also useful to specify things like methods that are supposed to abort the program when certain conditions occur, although that isn't really good practice in Java. In general, it is most useful for examples like the one given above, when you want to say when a method cannot return to its caller.

### 9.9.8 When Clauses

The `when` clause allows concurrency aspects of a method or constructor to be specified [Lerner91] [Rodriguez-etal05]. A caller of a method will be delayed until the condition given in the `when` clause holds. What is checked is that the method does not proceed to its commit point, which is the start of execution of statement with the label `commit`, until the given predicate is true.

The syntax is as follows. See Section 9.9.2 [Requires Clauses], page 74, for the syntax of *pred-or-not*.

*when-clause* ::= *when-keyword* *pred-or-not* `;`
*when-keyword* ::= `when` | `when_redundantly`

When a when clause is omitted in a specification case, a default when clause is used. For a lightweight specification case, the default when condition is `\not_specified`. The default when condition for a heavyweight specification case is `true`.

See [Rodriguez-etal05] for more about the `when` clause and JML's plans for support of multithreading.

### 9.9.9 Assignable Clauses

An assignable clause gives a frame axiom for a specification. It says that, from the client's point of view, only the locations named, and locations in the data groups associated with these locations, can be assigned to during the execution of the method. The values of all subexpressions used in assignable clauses, such as `i-1` in `a[i-1]`, are computed in the pre-state of the method, because the assignable clause only talks about locations that exist in the pre-state.

See Chapter 10 [Data Groups], page 85, for more about specification of data groups. However, locations that are local to the method (or methods it calls) and locations that are created during the method's execution are not subject to this restriction.

The syntax is as follows. See Section 11.7 [Store Refs], page 103, for the syntax of *store-ref-list*.

> *assignable-clause* ::= *assignable-keyword store-ref-list* `;`
> *assignable-keyword* ::= `assignable` | `assignable_redundantly`
>     | `modifiable` | `modifiable_redundantly`
>     | `modifies` | `modifies_redundantly`

When an assignable clause is omitted in a specification case, a default assignable clause is used. This default has a default *store-ref-list*. For a lightweight specification case, the default *store-ref-list* is `\not_specified`. The default *store-ref-list* for a heavyweight specification case is `\everything`.

If one wants the opposite of the default (for a heavyweight specification case), then one can specify that a method cannot assign to any locations by writing:

```
assignable \nothing;
```

Using the modifier `pure` on a method achieves the same effect as specifying `assignable \nothing`, but does so for the method's entire specification as opposed to a single *specification-case*.

Assignable clauses are subject to several restrictive rules in JML. The first rule has to do with fields of model objects. Because model objects are abstract and do not have a concrete state or concrete fields, the JML typechecker does not allow fields of model objects to be listed in the assignable clause; that is, such expressions do not specify a set of locations (concrete fields) that can be assigned to. Thus expressions like `f.x` are not allowed in the assignable clause when `f` is a model field.

[[[Flesh out other restrictions. Refer to [Mueller-Poetzsch-Heffter-Leavens03] for details.]]]

### 9.9.10 Accessible Clauses

The accessible clause is a seldom-used feature of JML. Together with the `assignable` clause (see Section 9.9.9 [Assignable Clauses], page 80), it says what (pre-existing) locations a method may read during its execution. It has the following syntax.

> *accessible-clause* ::= *accessible-keyword store-ref-list* `;`
> *accessible-keyword* ::= `accessible` | `accessible_redundantly`

During execution of the method (which includes all directly and indirectly called methods and constructors), only locations that either did not exist in the pre-state, that are local to the method (including the method's formal parameters), or that are either named in the lists found in the `accessible` and `assignable` clauses or that are dependees (see Chapter 10 [Data Groups], page 85) of such locations, are read from. Note that locations that are local to the method (or methods it calls) and locations that are created during the method's execution are not subject to this restriction and may be read from freely.

When an accessible clause is omitted in a code contract specification case, a default accessible clause is used. This default has a default *store-ref-list* which is `\everything`.

See Chapter 15 [Specification for Subtypes], page 122, for more discussion and examples.

### 9.9.11 Callable Clauses

The callable clause says what methods may be called, either directly or indirectly, by the method being specified. It has the following syntax.

> *callable-clause* ::= *callable-keyword callable-methods-list* `;`
> *callable-keyword* ::= `callable | callable_redundantly`
> *callable-methods-list* ::= *method-name-list* `|` *store-ref-keyword*

During execution of a method, the only methods and constructors that may be called are those listed in the `callable` clause's list.

When a callable clause is omitted in a code contract specification case, a default callable clause is used. This default has a default *callable-methods-list* which is `\everything`.

See Chapter 15 [Specification for Subtypes], page 122, for more discussion and examples.

### 9.9.12 Measured By Clauses

A measured by clause can be used in a termination argument for a recursive specification. It has the following syntax.

> *measured-clause* ::= *measured-by-keyword* `\not_specified ;`
>     `|` *measured-by-keyword spec-expression* [ `if` *predicate* ] `;`
> *measured-by-keyword* ::= `measured_by | measured_by_redundantly`

The *spec-expression* in a measured by clause must have type `int`.

In both lightweight and heavyweight specification cases, an omitted measured by clause means the same as a measured by clause of the following form.

```
measured_by \not_specified;
```

### 9.9.13 Captures Clauses

The captures clause has the following syntax.

> *captures-clause* ::= *captures-keyword store-ref-list* `;`
> *captures-keyword* ::= `captures | captures_redundantly`

The captures clause says that references to the *store-ref*s listed can be retained after the method returns, for example in a field of the receiver object or in a static field. Therefore, the captures clause specifies when an object, passed as an actual parameter in a method call, may be captured during the call.

An actual parameter object (including the receiver this) is captured if it appears on the right-hand side of an assignment statement during the call. This can also happen indirectly through another method or constructor call or by returning the parameter object as the method result (we assume the result will be assigned to a field or local variable after the call).

The captures clause is used to prevent certain kinds of representation exposure as part of an alias control technique. For example, if an object should not be aliased, then that object must not be passed to a method that may capture it, i.e., may create an alias to it (this includes the receiver). Furthermore, objects used as part of the abstract representation of a type should not be aliased, and thus should not be passed to methods that capture it. JML tools will eventually prevent such aliasing.

When a captures clause is omitted in a method specification case, then a default captures clause is used. This default has a default *store-ref-list* which is `\everything`. Thus when omitted, a method is allowed to capture any of the actual parameter objects or the receiver.

### 9.9.14 Working Space Clauses

A *working-space-clause* can be used to specify the maximum amount of heap space used by a method, over and above that used by its callers. The clause applies only to the particular specification case it is in, of course This is adapted from the work of Krone, Ogden, and Sitaraman on RESOLVE [Krone-Ogden-Sitaraman03].

*working-space-clause* ::= *working-space-keyword* `\not_specified ;`
  `|` *working-space-keyword spec-expression* [ `if` *predicate* ] `;`
*working-space-keyword* ::= `working_space | working_space_redundantly`

The *spec-expression* in a working space clause must have type `long`. It is to be understood in units of bytes.

The *spec-expression* in a working space clause may use `\old` and other JML operators appropriate for postconditions. This is because it is considered to be evaluated in the post-state, and provides a guarantee of the maximum amount of additional space used by the call. In some cases this space may depend on the `\result`, exceptions thrown, or other post-state values. [[[ There is however no way to identify the exception thrown - DRCok]]]

In both lightweight and heavyweight specification cases, an omitted working space clause means the same as a working space clause of the following form.

```
working_space \not_specified;
```

See Section 11.4.13 [Backslash working space], page 96, for information about the `\working_space` expression that can be used to describe the working space needed by a method call. See Section 11.4.12 [Backslash space], page 95, for information about the `\space` expression that can be used to describe the heap space occupied by an object.

### 9.9.15 Duration Clauses

A duration clause can be used to specify the maximum (i.e., worst case) time needed to process a method call in a particular specification case. [[[ Tools are simpler if the argument can simply be an arbitrary expression rather than a method call. – DRCok ]]] This is adapted from the work of Krone, Ogden, and Sitaraman on RESOLVE [Krone-Ogden-Sitaraman03].

*duration-clause* ::= *duration-keyword* `\not_specified ;`
  `|` *duration-keyword spec-expression* [ `if` *predicate* ] `;`
*duration-keyword* ::= `duration | duration_redundantly`

The *spec-expression* in a duration clause must have type `long`. It is to be understood in units of [[[the JVM instruction that takes the least time to execute, which may be thought of as the JVM's cycle time.]]] The time it takes the JVM to execute such an instruction can be multiplied by the number of such cycles to arrive at the clock time needed to execute the method in the given specification case. [[[This time should also be understood as not counting garbage collection time.]]]

The *spec-expression* in a duration clause may use `\old` and other JML operators appropriate for postconditions. This is because it is considered to be evaluated in the post-state, and provides a guarantee of the maximum amount of additional space used by the call. In some cases this space may depend on the `\result`, exceptions thrown, or other post-state values. [[[ There is no way to identify the exception thrown - DRCok]]]

In both lightweight and heavyweight specification cases, an omitted duration clause means the same as a duration clause of the following form.

```
duration \not_specified;
```

See Section 11.4.11 [Backslash duration], page 95, for information about the `\duration` expression that can be used in the duration clause to specify the duration of other methods.

# 10 Data Groups

A *data group* is a set of locations; data groups are used in JML's frame axioms (see Section 9.9.9 [Assignable Clauses], page 80) to name such sets of locations in a way that does not expose representation details [Leino98].

Each field in a program defines a data group, whose name is the same as that of the field.

The main purpose for putting locations into data groups is so that these locations may be assigned during the executions of methods that have permission to assign to the data group. For example, if locations `x.f` and `x.y` are in data group `x.d`, then an assignable clause of the form

```
assignable x.d;
```

allows `x.d`, `x.f`, `x.y`, and any other locations in the data group of `x.d` to be assigned during the execution of a method.

One should always put private or protected fields that are used to compute the value of a public model field (see Section 8.4 [Represents Clauses], page 58) into the data group of that model field. However, one can also put other fields into a model field's data group, just to allow them to be assigned when the model field is assignable.

It is sometimes convenient to declare a data group without any other information about the model of data. This can be done using the type `org.jmlspecs.models.JMLDataGroup`. This type has exactly one non-null object, named `JMLDataGroup.IT`. For example, the class `java.lang.Object` has the following data group declaration.

```
//  public non_null model JMLDataGroup objectState;
```

The `objectState` data group provides a convenient way to talk about "the state" of an object without committing to any modeling or representation details.

[[[ needs discussion - default data groups ]]]

To place a field or array element in a data group, one uses the following syntax.

*jml-data-group-clause* ::= *in-group-clause* `|` *maps-into-clause*

The details of the two kinds of data group clauses are discussed below.

## 10.1 Static Data Group Inclusions

*in-group-clause* ::= *in-keyword group-list* `;`
*in-keyword* ::= `in` `|` `in_redundantly`
*group-list* ::= *group-name* [ `,` *group-name* ] . . .
*group-name* ::= [ *group-name-prefix* ] *ident*
*group-name-prefix* ::= `super .` `|` `this .`

The *in-group-clause* puts the field being declared in all the data groups named in the *group-list*.

[[[needs discussion]]]

## 10.2 Dynamic Data Group Mappings

See Section 11.7 [Store Refs], page 103, for the definition of *spec-array-ref-expr*.

*maps-into-clause* ::= *maps-keyword member-field-ref* `\into` *group-list* `;`
*maps-keyword* ::= `maps` | `maps_redundantly`
*member-field-ref* ::= *ident* `.` *maps-member-ref-expr*
    | *maps-array-ref-expr* [ `.` *maps-member-ref-expr* ]
*maps-member-ref-expr* ::= *ident* | `*`
*maps-array-ref-expr* ::= *ident maps-spec-array-dim*
    [ *maps-spec-array-dim* ] . . .
*maps-spec-array-dim* ::= '`[`' *spec-array-ref-expr* '`]`'

The *maps-into-clause* describes elements of a data group that are determined dynamically, through a field reference or an array index, or a field of an array index. The pattern `*` may be used to specify all fields of an object or all elements of an array.

The fields of a model object do not denote locations because model objects are abstract and do not have concrete fields. Therefore, in JML, the maps clause is not allowed in the declaration of a model field because such maps clauses do not denote a specific set of locations to be added to a data group, and this is the primary purpose of the maps clause (see also the discussion of model fields in the assignable clause).

[[[ needs discussion ]]]

# 11 Predicates and Specification Expressions

This chapter describes predicates in JML and JML's extensions to Java's expressions. It also describes store references, which are similar to specification expressions, but are used to describe locations instead of values. Details are found in the sections below.

## 11.1 Predicates

A *predicate* The following gives the syntax of predicates, which are simply *spec-expression*s that must have a boolean value. See Section 11.2 [Specification Expressions], page 87, for the syntax of specification expressions.

*predicate* ::= *spec-expression*

## 11.2 Specification Expressions

The following gives the syntax of specification expressions in JML. See Section 11.3 [Expressions], page 87, for the syntax of *expression*.

*spec-expression-list* ::= *spec-expression*
    [ , *spec-expression* ] . . .
*spec-expression* ::= *expression*

Within a *spec-expression*, one cannot use any of the operators (such as `++`, `--`, and the assignment operators) that would necessarily cause side effects. In addition, one can use extensions that are specific to JML, in particular the JML primary expressions.

## 11.3 Expressions

The JML syntax for expressions extends the Java syntax with several operators and primitives.

The precedence of operators in JML expressions is similar to that in Java The precedence levels are given in the following table, where the parentheses, quantified expressions, `[]`, `.`, and method calls on the first three lines all have the highest precedence, and for the rest, only the operators on the same line have the same precedence.

highest `new () \forall \exists \max \min`
    `\num_of \product \sum` *informal-description*
    `[] .` and method calls
  unary `+` and `-` `~` `!` (typecast)
  `* / %`
  `+` (binary) `-` (binary)
  `<< >> >>>`
  `< <= > >= <: instanceof`
  `== !=`
  `&`
  `^`
  `|`
  `&&`
  `||`
  `==> <==`

```
           <==> <=!=>
           ?:
lowest     = *= /= %= += -= <<= >>= >>>= &= ^= |=
```

The following is the syntax of Java expressions, with JML additions. The additions are the operators `==>`, `<==`, `<==>`, `<=!=>`, and `<:`, and the syntax found under the nonterminals *jml-primary* (see Section 11.4 [JML Primary Expressions], page 89) and *set-comprehension* (see Section 11.5 [Set Comprehensions], page 101). The JML additions to the Java syntax can only be used in assertions and other annotations. Furthermore, within assertions, one cannot use any of the operators (such as `++`, `--`, and the assignment operators) that would necessarily cause side effects.

```
expression-list ::= expression [ , expression ] . . .
expression ::= assignment-expr
assignment-expr ::= conditional-expr
                [ assignment-op assignment-expr ]
assignment-op ::= = | += | -= | *= | /= | %= | >>=
        | >>>= | <<= | &= | ‘|=’ | ^=
conditional-expr ::= equivalence-expr
                [ ? conditional-expr : conditional-expr ]
equivalence-expr ::= implies-expr
                [ equivalence-op implies-expr ] . . .
equivalence-op ::= <==> | <=!=>
implies-expr ::= logical-or-expr
            [ ==> implies-non-backward-expr ]
        | logical-or-expr <== logical-or-expr
            [ <== logical-or-expr ] . . .
implies-non-backward-expr ::= logical-or-expr
            [ ==> implies-non-backward-expr ]
logical-or-expr ::= logical-and-expr [ ‘||’ logical-and-expr ] . . .
logical-and-expr ::= inclusive-or-expr [ && inclusive-or-expr ] . . .
inclusive-or-expr ::= exclusive-or-expr [ ‘|’ exclusive-or-expr ] . . .
exclusive-or-expr ::= and-expr [ ^ and-expr ] . . .
and-expr ::= equality-expr [ & equality-expr ] . . .
equality-expr ::= relational-expr [ == relational-expr] . . .
        | relational-expr [ != relational-expr] . . .
relational-expr ::= shift-expr < shift-expr
        | shift-expr > shift-expr
        | shift-expr <= shift-expr
        | shift-expr >= shift-expr
        | shift-expr <: shift-expr
        | shift-expr [ instanceof type-spec ]
shift-expr ::= additive-expr [ shift-op additive-expr ] . . .
shift-op ::= << | >> | >>>
additive-expr ::= mult-expr [ additive-op mult-expr ] . . .
additive-op ::= + | -
mult-expr ::= unary-expr [ mult-op unary-expr ] . . .
mult-op ::= * | / | %
```

*unary-expr* ::= `(` *type-spec* `)` *unary-expr*
  | `++` *unary-expr*
  | `--` *unary-expr*
  | `+` *unary-expr*
  | `-` *unary-expr*
  | *unary-expr-not-plus-minus*
*unary-expr-not-plus-minus* ::= `~` *unary-expr*
  | `!` *unary-expr*
  | `(` *built-in-type* `)` *unary-expr*
  | `(` *reference-type* `)` *unary-expr-not-plus-minus*
  | *postfix-expr*
*postfix-expr* ::= *primary-expr* [ *primary-suffix* ] . . . [ `++` ]
  | *primary-expr* [ *primary-suffix* ] . . . [ `--` ]
  | *built-in-type* [ '`[`' '`]`' ] . . . `. class`
*primary-suffix* ::= `.` *ident*
  | `. this`
  | `. class`
  | `.` *new-expr*
  | `. super (` [ *expression-list* ] `)`
  | `(` [ *expression-list* ] `)`
  | '`[`' *expression* '`]`'
  | [ '`[`' '`]`' ] . . . `. class`
*primary-expr* ::= *ident* | *new-expr*
  | *constant* | `super` | `true`
  | `false` | `this` | `null`
  | `(` *expression* `)`
  | *jml-primary*
*built-in-type* ::= `void` | `boolean` | `byte`
  | `char` | `short` | `int`
  | `long` | `float` | `double`
*constant* ::= *java-literal*
*new-expr* ::= `new` *type* *new-suffix*
*new-suffix* ::= `(` [ *expression-list* ] `)` [ *class-block* ]
  | *array-decl* [ *array-initializer* ]
  | *set-comprehension*
*array-decl* ::= *dim-exprs* [ *dims* ]
*dim-exprs* ::= '`[`' *expression* '`]`' [ '`[`' *expression* '`]`' ] . . .
*array-initializer* ::= `{` [ *initializer* [ `,` *initializer* ] . . . [ `,` ] ] `}`
*initializer* ::= *expression*
  | *array-initializer*

[[[Need to have semantics of the new things explained here.]]]

## 11.4 JML Primary Expressions

The following is the syntax of *jml-primary*.

*jml-primary* ::= *result-expression*
  | *old-expression*

| *not-assigned-expression*
| *not-modified-expression*
| *only-accessed-expression*
| *only-assigned-expression*
| *only-called-expression*
| *only-captured-expression*
| *fresh-expression*
| *reach-expression*
| *duration-expression*
| *space-expression*
| *working-space-expression*
| *nonnullelements-expression*
| *informal-description*
| *typeof-expression*
| *elemtype-expression*
| *type-expression*
| *lockset-expression*
| *max-expression*
| *is-initialized-expression*
| *invariant-for-expression*
| *lblneg-expression*
| *lblpos-expression*
| *spec-quantified-expr*

All of the JML keywords that can be used in expressions which would otherwise start with an alphabetic character start with a backslash (\), so that they cannot clash with the program's variable names.

The new expressions that JML introduces are described below. Several of the descriptions below quote, without attribution, descriptions from [Leavens-Baker-Ruby06].

### 11.4.1 `\result`

The syntax of a *result-expression* is as follows.

*result-expression* ::= `\result`

The primary `\result` can only be used in `ensures`, `duration`, and `workingspace` clauses of a non-void method. Its value is the value returned by the method. Its type is the return type of the method; hence it is a type error to use `\result` in a void method or in a constructor.

### 11.4.2 `\old` and `\pre`

An *old-expression* has the following syntax. See Section 11.2 [Specification Expressions], page 87, for the syntax of *spec-expression*.

*old-expression* ::= `\old` `(` *spec-expression* [ `,` *ident* ] `)`
  | `\pre` `(` *spec-expression* `)`

An expression of the form `\old(`*Expr*`)` refers to the value that the expression *Expr* had in the pre-state of a method.

JML uses Java's reference semantics, hence the pre-state value of an expression whose type is a reference type is simply the reference; it is *not* a clone of the object the reference points to. For example, suppose in the pre-state that `v` is field that holds a reference to a `HashMap`; concretely, suppose that the location stored in `v` is 0x952ab340. Then the expression `\old(v)` denotes the pre-state value of `v`, which is the same reference, i.e., it is the address 0x952ab340. Note that `\old(v)` is not a reference to a copy of the `HashMap` stored at that location, but simply a copy of the location's address (the reference), which is the value of `v`. If the fields of the object at that location have changed in the post-state, then changes to those fields will be visible through `\old(v)`; for example, `\old(v).size()` will be the same as `v.size()`. To write a post-condition that refers to `v`'s size in the pre-state, one should instead write `\old(v.size())`. Indeed as a general rule, it is always safest to use `\old()` only around expressions whose type is a value type or a type with immutable values, such as `String`.

Expressions of this form may be used in both normal and exceptional postconditions (see Chapter 9 [Method Specifications], page 61, for more about such `ensures` and `signals` clauses), in history constraints, in duration and working space clauses, and also in assertions that appear in the bodies of methods (see Chapter 12 [Statements and Annotation Statements], page 104, for more about `assert` and `assume` statements, loop invariants, and variant functions).

However, we recommend that inside the bodies of methods, one of the two other forms of *old-expression* (see below) be used instead. The reason for this is that the reader may wonder whether `\old(`*Expr*`)` in the body of a method means the pre-state value of *Expr* (which it does) or the value of *Expr* before some previous statement (which it does not).

An expression of the form `\pre(`*Expr*`)` also refers to the value that the expression *Expr* had in the pre-state of a method. Expressions of this form may only be used in assertions that appear in the bodies of methods (i.e., in `assert` and `assume` statements, and in loop invariants and variant functions). That is, such expressions may not be used in specification cases, and hence may not appear in normal or exceptional postconditions, in history constraints, or in duration and working space clauses.

An expression of the form `\old(`*Expr*`,` *Label*`)` refers to the value that the expression *Expr* had when control last reached the statement label *Label*. That is, it refers to the value of the expression just before control reached the statement the label is attached to. Expressions of this form may only be used in assertions that appear in the bodies of methods (i.e., in `assert` and `assume` statements, and in loop invariants and variant functions). That is, such expressions may not be used in specification cases, and hence may not appear in normal or exceptional postconditions, in history constraints, or in duration and working space clauses.

In an expression of the form `\old(`*Expr*`,` *Label*`)`, *Label* must be a label defined in the current method. The type of `\old(`*Expr*`)`, `\old(`*Expr*`,` *Label*`)`, or `\pre(`*Expr*`)`, is simply the type of *Expr*.

It is a type error if `\old()` or `\pre()` encloses a free occurrence of a quantified variable. For example, in the following, `\old()` encloses a free occurrence of the quantified variable `i`, which is declared in the surrounding quantifier, and thus the example is illegal.

```
(\forall int i; 0 <= i && i < 7; \old(i < y))  // illegal
```

The problem with the above example is that there is no easy way to evaluate `\old(i < y)` in the pre-state.

However, constructions like the following are legal, as in the first the use of `\old()` does not enclose the quantified variable, `i`, and in the second use of `\old()` does not enclose a free occurrence of the quantified variable (the variable is bound by the declaration which is inside of `\old()`.

```
(\forall int i; 0 <= i && i < 7; i < \old(y))  // ok
\old((\forall int i; 0 <= i && i < 7; i < y))  // ok
```

### 11.4.3 `\not_assigned`

The syntax of a *not-assigned-expression* is as follows. See Section 11.7 [Store Refs], page 103, for the syntax of *store-ref-list*.

*not-assigned-expression* ::= `\not_assigned` `(` *store-ref-list* `)`

The JML operator `\not_assigned` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. It asserts that the locations in the data group (see Chapter 10 [Data Groups], page 85) named by the argument were not assigned to during the execution of the method being specified (or all methods to which a history constraint applies). For example, `\not_assigned(xval,yval)` says that the locations in the data groups named by `xval` and `yval` were not assigned during the method's execution.

A predicate such as `\not_assigned(x.f)` refers to the entire data group named by `x.f` not just to the location `x.f` itself. This allows one to specify absence of even temporary side-effects in various cases of a method. See Section 11.4.4 [Backslash not_modified], page 92, for ways to specify that just the value of a given field was not changed, which allows temporary side effects.

The `\not_assigned` operator can be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that all (concrete) locations in that model field's data group were not assigned. [[[A real example would help here.]]]

The type of a `\not_assigned` expression is `boolean`.

### 11.4.4 `\not_modified`

The syntax of a *not-modified-expression* is as follows. See Section 11.7 [Store Refs], page 103, for the syntax of *store-ref-list*.

*not-modified-expression* ::= `\not_modified` `(` *store-ref-list* `)`

The JML operator `\not_modified` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. It asserts that the values of the named fields are the same in the post-state as in the pre-state; for example, `\not_modified(xval,yval)` says that the fields `xval` and `yval` have the same value in the pre- and post-states (in the sense of the `equals` method for their types).

A predicate such as `\not_modified(x.f)` refers to the location named by `x.f`, not to the entire data group of `x.f`. This allows one to specify benevolent side-effects, as one can name `x.f` (or a data group in which it participates) in an assignable clause, but use `\not_modified(x.f)` in the postcondition. See Section 11.4.3 [Backslash not_assigned], page 92, for ways to specify that no assignments were made to any location in a data group, disallowing temporary side effects.

The `\not_modified` operator can be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that only the value of the model field is unchanged (in the sense of its type's equals operation); concrete fields involved in its representation may have changed. [[[A real example would help here.]]]

The type of a `\not_modified` expression is `boolean`.

### 11.4.5 `\only_accessed`

The syntax of an *only-accessed-expression* is as follows. See Section 11.7 [Store Refs], page 103, for the syntax of *store-ref-list*.

> *only-accessed-expression* ::= `\only_accessed ( ` *store-ref-list* ` )`

The JML operator `\only_accessed` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. Used in a method's postcondition (perhaps implicitly in a history constraint), it asserts that the method's execution only reads from a subset of the data groups named by the given fields. For example, `\only_accessed(xval,yval)` says that no fields, outside of the data groups of `xval` and `yval` were read by the method. This includes both direct reads in the body of the method, and reads during calls that were made by the method (and methods those methods called, etc.).

A predicate such as `\only_accessed(x.f)` refers to the entire data group named by `x.f` not just to the location `x.f` itself.

The `\only_accessed` operator can be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that the (concrete) locations in that model field's data group are permitted to be accessed during the method's execution.

The type of an `\only_accessed` expression is `boolean`.

### 11.4.6 `\only_assigned`

The syntax of an *only-assigned-expression* is as follows. See Section 11.7 [Store Refs], page 103, for the syntax of *store-ref-list*.

> *only-assigned-expression* ::= `\only_assigned ( ` *store-ref-list* ` )`

The JML operator `\only_assigned` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. Used in a method's postcondition (perhaps implicitly in a history constraint), it asserts that the method's execution only assigned to a subset of the data groups named by the given fields. For example, `\only_assigned(xval,yval)` says that no fields, outside of the data groups of `xval` and `yval` were assigned by the method. This includes both direct assignments in the body of the method, and assignments during calls that were made by the method (and methods those methods called, etc.).

A predicate such as `\only_assigned(x.f)` refers to the entire data group named by `x.f` not just to the location `x.f` itself.

The `\only_assigned` operator can be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that the (concrete) locations in that model field's data group are permitted to be assigned during the method's execution.

The type of an `\only_assigned` expression is `boolean`.

### 11.4.7 \only_called

The syntax of an *only-called-expression* is as follows. See Section 8.3 [Constraints], page 55, for the syntax of *method-name-list*.

*only-called-expression* ::= `\only_called ( `*method-name-list*` )`

The JML operator `\only_called` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. Used in a method's postcondition (perhaps implicitly in a history constraint), it asserts that the method's execution only called from a subset of methods given in the *method-name-list*. For example, `\only_called(p,q)` says that methods, apart from `p` and `q`, were called during this method's execution.

The type of an `\only_called` expression is `boolean`.

### 11.4.8 \only_captured

The syntax of an *only-captured-expression* is as follows. See Section 11.7 [Store Refs], page 103, for the syntax of *store-ref-list*.

*only-captured-expression* ::= `\only_captured ( `*store-ref-list*` )`

The JML operator `\only_captured` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. Used in a method's postcondition (perhaps implicitly in a history constraint), it asserts that the method's execution only captured references from a subset of the data groups named by the given fields. For example, `\only_captured(xv,yv)` says that no references, outside of the data groups of `xv` and `yv` were captured by the method.

A reference is *captured* when it is stored into a field (as opposed to a local variable). Typically a method captures a formal parameter (or a reference stored in a static field) by assigning it to a field in the method's receiver (the `this` object), a field in some object (or to an array element), or to a static field.

A predicate such as `\only_captured(x.f)` refers to the references stored in the entire data group named by `x.f` in the pre-state, not just to those stored in the location `x.f` itself. However, since the references being captured are usually found in formal parameters, the complications of data groups can usually be ignored.

The `\only_captured` operator can be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that the (concrete) locations in that model field's data group are permitted to be captured during the method's execution.

The type of an `\only_captured` expression is `boolean`.

### 11.4.9 \fresh

The syntax of a *fresh-expression* is as follows. See Section 11.2 [Specification Expressions], page 87, for the syntax of *spec-expression-list*.

*fresh-expression* ::= `\fresh ( `*spec-expression-list*` )`

The operator `\fresh` asserts that objects were freshly allocated; for example, `\fresh(x,y)` asserts that `x` and `y` are not null and that the objects bound to these identifiers were not allocated in the pre-state. The arguments to `\fresh` can have any reference type, and the type of the overall expression is `boolean`.

Note that it is wrong to use `\fresh(this)` in the specification of a constructor, because Java's `new` operator allocates storage for the object; the constructor's job is just to initialize that storage.

### 11.4.10 `\reach`

The syntax of a *reach-expression* is as follows. See Section 11.2 [Specification Expressions], page 87, for the syntax of *spec-expression*.

*reach-expression* ::= `\reach (` *spec-expression* `)`

The `\reach` expression allows one to refer to the set of objects reachable from some particular object. The syntax `\reach(`$x$`)` denotes the smallest `JMLObjectSet` containing the object denoted by $x$, if any, and all objects accessible through all fields of objects in this set. That is, if $x$ is `null`, then this set is empty otherwise it contains $x$, all objects accessible through all fields of $x$, all objects accessible through all fields of these objects, and so on, recursively. If $x$ denotes a model field (or data group), then `\reach(`$x$`)` denotes the smallest `JMLObjectSet` containing the objects reachable from $x$ or reachable from the objects referenced by fields in that data group.

### 11.4.11 `\duration`

The syntax of a *duration-expression* is as follows. See Section 11.3 [Expressions], page 87, for the syntax of *expression*.

*duration-expression* ::= `\duration (` *expression* `)`

`\duration`, which describes the specified maximum number of virtual machine cycle times needed to execute the method call or explicit constructor invocation expression that is its argument; e.g., `\duration(myStack.push(o))` is the maximum number of virtual machine cycles needed to execute the call `myStack.push(o)`, according to the contract of the static type of `myStack`'s type's `push` method, when passed argument `o`. Note that the expression used as an argument to `\duration` should be thought of as quoted, in the sense that it is not to be executed; thus the method or constructor called need not be free of side effects. Note that the argument to `\duration` is an *expression* instead of just the name of a method, because different method calls, i.e., those with different parameters, can take different amounts of time [Krone-Ogden-Sitaraman03].

The argument expression passed to `\duration` must be a method call or explicit constructor invocation expression; the type of a `\duration` expression is `long`.

For a given Java Virtual Machine, a *virtual machine cycle* is defined to be the minimum of the maximum over all Java Virtual Machine instructions, $i$, of the length of time needed to execute instruction $i$.

### 11.4.12 `\space`

The syntax of a *space-expression* is as follows. See Section 11.2 [Specification Expressions], page 87, for the syntax of *spec-expression*. [[[ Shouldn't this take an expression instead of a spec-expression? - DRC]]]

*space-expression* ::= `\space (` *spec-expression* `)`

`\space`, which describes the amount of heap space, in bytes, allocated to the object referred to by its argument [Krone-Ogden-Sitaraman03]; e.g., `\space(myStack)` is number of bytes in the heap used by `myStack`, not including the objects it contains. The type of the *spec-expression* that is the argument must be a reference type, and the result type of a `\space` expression is `long`.

### 11.4.13 `\working_space`

*working-space-expression* ::= `\working_space` `(` *expression* `)`

`\working_space`, which describes the maximum specified amount of heap space, in bytes, used by the method call or explicit constructor invocation expression that is its argument; e.g., `\working_space(myStack.push(o))` is the maximum number of bytes needed on the heap to execute the call `myStack.push(o)`, according to the contract of the static type of `myStack`'s type's `push` method, when passed argument `o`. Note that the expression used as an argument to `\working_space` should be thought of as quoted, in the sense that it is not to be executed; thus the method or constructor called need not be free of side effects. The detailed arguments are needed in the specification of the call because different method calls, i.e., those with different parameters, can use take different amounts of space [Krone-Ogden-Sitaraman03]. The argument expression must be a method call or explicit constructor invocation expression; the result type of a `\working_space` expression is `long`.

### 11.4.14 `\nonnullelements`

The syntax of a *nonnullelements-expression* is as follows. See Section 11.2 [Specification Expressions], page 87, for the syntax of *spec-expression*.

*nonnullelements-expression* ::= `\nonnullelements` `(` *spec-expression* `)`

The operator `\nonnullelements` can be used to assert that an array and its elements are all non-null. For example, `\nonnullelements(myArray)`, is equivalent to [Leino-Nelson-Saxe00]

```
myArray != null &&
(\forall int i; 0 <= i && i < myArray.length;
                myArray[i] != null)
```

### 11.4.15 Informal Predicates

An *informal-description* is some text enclosed in `(*` and `*)`. See Section 4.6 [Tokens], page 29, for details of its syntax. It is used as an escape form formality.

An informal description used as a predicate has type boolean. Hence the text in an informal description should describe a condition, for example `(* the value of x is displayed *)`.

The value of an informal description is only known to the user, not to any JML tools, so it is never executable. Informal descriptions should thus be avoided when possible, but can be used to avoid formalizing everything when doing so would be too expensive.

### 11.4.16 `\typeof`

The syntax of a *typeof-expression* is as follows. See Section 11.2 [Specification Expressions], page 87, for the syntax of *spec-expression*.

*typeof-expression* ::= `\typeof` `(` *spec-expression* `)`

The operator `\typeof` returns the most-specific dynamic type of an expression's value [Leino-Nelson-Saxe00]. The meaning of `\typeof(`$E$`)` is unspecified if $E$ is null. If $E$

has a static type that is a reference type, then `\typeof(`*E*`)` means the same thing as *E*`.getClass()`. For example, if `c` is a variable of static type `Collection` that holds an object of class `HashSet`, then `\typeof(c)` is `HashSet.class`, which is the same thing as `\type(HashSet)`. If *E* has a static type that is not a reference type, then `\typeof(`*E*`)` means the instance of `java.lang.Class` that represents its static type. For example, `\typeof(true)` is `Boolean.TYPE`, which is the same as `\type(boolean)`. Thus an expression of the form `\typeof(`*E*`)` has type `\TYPE`, which JML considers to be the same as `java.lang.Class`.

### 11.4.17 \elemtype

The syntax of a *elemtype-expression* is as follows.

*elemtype-expression* ::= `\elemtype (` *spec-expression* `)`

The `\elemtype` operator returns the most-specific static type shared by all elements of its array argument [Leino-Nelson-Saxe00]. For example, `\elemtype(\type(int[]))` is `\type(int)`. The argument to `\elemtype` must be an expression of type `\TYPE`, which JML considers to be the same as `java.lang.Class`, and its result also has type `\TYPE` (see Section 7.1.2.2 [Type-Specs], page 48). If the argument is not an array type, then the result is `null`. For example, `\elemtype(\type(int))` and `\elemtype(\type(Object))` are both `null`.

### 11.4.18 \type

The syntax of a *type-expression* is as follows. See Section 7.1.2.2 [Type-Specs], page 48, for the syntax of *type*.

*type-expression* ::= `\type (` *type* `)`

The operator `\type` can be used to introduce literals of type `\TYPE` in expressions. An expression of the form `\type(T)`, where `T` is a type name, has the type `\TYPE`. Since in JML `\TYPE` is the same as `java.lang.Class`, an expression of the form `\type(`*T*`)` means the same thing as *T*`.class`, if *T* is a reference type. If *T* is a primitive type, then `\type(T)` is equivalent to the value of the `TYPE` field of the corresponding reference type. Thus `\type(boolean)` equals `Boolean.TYPE`.

For example, in

```
\typeof(myObj) <: \type(PlusAccount)
```

the use of `\type(PlusAccount)` is used to introduce the type `PlusAccount` into this expression context.

### 11.4.19 \lockset

The syntax of a *lockset-expression* is as follows.

*lockset-expression* ::= `\lockset`

The `\lockset` primitive denotes the set of locks held by the current thread. It is of type `JMLObjectSet`. (This is an adaptation from ESC/Java [Leino-etal00] [Leino-Nelson-Saxe00] for dealing with threads.)

### 11.4.20 \max

The syntax of a *max-expression* is as follows. See Section 11.2 [Specification Expressions], page 87, for the syntax of *spec-expression*.

*max-expression* ::= `\max ( ` *spec-expression* `)`

The `\max` operator returns the "largest" (as defined by `<`) of a set of lock objects, given a lock set as an argument. The result is of type Object. (This is an adaptation from ESC/Java [Leino-etal00] [Leino-Nelson-Saxe00] for dealing with threads.)

If you are looking to take the maximum of several integers, use the max quantifier (see Section 11.4.24.2 [Generalized Quantifiers], page 99).

### 11.4.21 `\is_initialized`

The syntax of the *is-initialized-expression* is as follows. See Section 7.1.2.2 [Type-Specs], page 48, for the syntax of *reference-type*

*is-initialized-expression* ::= `\is_initialized (` *reference-type* `)`

The `\is_initialized` operator returns true just when its *reference-type* argument is a class that has finished its static initialization. It is of type `boolean`.

### 11.4.22 `\invariant_for`

*invariant-for-expression* ::= `\invariant_for (` *spec-expression* `)`

The `\invariant_for` operator returns true just when its argument satisfies the invariant of its static type; for example, `\invariant_for((MyClass)o)` is true when `o` satisfies the invariant of `MyClass`. The entire `\invariant_for` expression is of type `boolean`.

### 11.4.23 `\lblneg` and `\lblpos`

The syntax of the two kinds of labeled expressions is as follows. See Section 11.2 [Specification Expressions], page 87, for the syntax of *spec-expression*.

*lblneg-expression* ::= `( \lblneg` *ident spec-expression* `)`
*lblpos-expression* ::= `( \lblpos` *ident spec-expression* `)`

Parenthesized expressions that start with `\lblneg` and `\lblpos` can be used to attach labels to expressions [Leino-Nelson-Saxe00]; these labels might be printed in various messages by support tools, for example, to identify an assertion that failed. Such an expression has a *label* and a *body*; for example, in

```
(\lblneg indexInBounds 0 <= index && index < length)
```

the label is `indexInBounds` and the body is the expression `0 <= index && index < length`. The value of a labeled expression is the value of its body, hence its type is the type of its body. The idea is that if this expression is used in an assertion and its value is `false` (e.g., when doing run-time checking of assertions), then a warning will be printed that includes the label `indexInBounds`. The form using `\lblpos` has a similar syntax, but should be used for warnings when the value of the enclosed expression is `true`.

### 11.4.24 Quantified Expressions

*spec-quantified-expr* ::= `(` *quantifier quantified-var-decls* `;`
  [ [ *predicate* ] `;` ]
  *spec-expression* `)`
*quantifier* ::= `\forall | \exists`
  `| \max | \min`
  `| \num_of | \product | \sum`

*quantified-var-decls* ::= [ *bound-var-modifiers* ] *type-spec quantified-var-declarator*
[ , *quantified-var-declarator* ] . . .
*quantified-var-declarator* ::= *ident* [ *dims* ]
*spec-variable-declarators* ::= *spec-variable-declarator*
[ , *spec-variable-declarator* ] . . .
*spec-variable-declarator* ::= *ident* [ *dims* ]
[ = *spec-initializer* ]
*spec-array-initializer* ::= { [ *spec-initializer*
[ , *spec-initializer* ] . . . [ , ] ] }
*spec-initializer* ::= *spec-expression*
| *spec-array-initializer*

Note that each quantified expression includes a set of parentheses; these parentheses cannot be omitted. The first part of a quantified expression is the *quantifier*, which determines the operation to be performed. Every quantifier starts with a backslash (\). Following the quantifier are *quantified-var-decls*, which declare *bound variables* whose scope is the *spec-quantified-expr*. The bound variables may not conflict with existing local variables, but may hide static and instance fields. The optional predicate between the two semicolons is the *range predicate*; a quantifier ranges over all possible values of its bound variables that satisfy the range predicate (for a discussion of the ranges of values for reference types, see Section 11.4.24.6 [Quantifying over Reference Types], page 101). If the range predicate is omitted, it defaults to `true`. The final *spec-expression* is called the *body* of the quantifier.

We discuss the various kinds of quantified expressions below.

### 11.4.24.1 Universal and Existential Quantifiers

The quantifiers `\forall` and `\exists`, are universal and existential quantifiers (respectively). For example,

```
(\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])
```

says that the values `a[0]` . . . `a[9]` are sorted.

The body of a universal or existential quantifier must be of type `boolean`. The type of a universal or existential quantified expression as a whole is `boolean`. When the range predicate is not satisfiable, the value of a `\forall` expression is `true` and the value of an `\exists` expression is `false`. For example:

```
(\forall int i; 0 < i && i < 0; 0 < i) == true
(\exists int i; 0 < i && i < 0; 0 < i) == false
```

### 11.4.24.2 Generalized Quantifiers

The quantifiers `\max`, `\min`, `\product`, and `\sum`, are generalized quantifiers that return the maximum, minimum, product, or sum of the values of the expressions given, where the variables satisfy the given range. The expression in the body must be of a built-in numeric type, such as `int` or `double`; the type of the quantified expression as a whole is the type of its body. For example, the following equations are all true (see chapter 3 of [Cohen90]):

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
(\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
(\max int i; 0 <= i && i < 5; i) == 4
(\min int i; 0 <= i && i < 5; i-1) == -1
```

For computing the value of a sum or product, Java's arithmetic is used. [[[ This would depend on the arithmetic mode in force - DRC]]]The meaning thus depends on the type of the expression. For example, in Java, floating point numbers use the IEEE 754 standard, and thus when an overflow occurs, the appropriate positive or negative infinity is returned. However, Java integers wrap on overflow. Consider the following examples.

```
(\product float f; 1.0e30f < f && f < 1.0e38f; f)
  == Float.POSITIVE_INFINITY

(\sum int i; i == Integer.MAX_VALUE || i == 1; i)
  == Integer.MAX_VALUE + 1
  == Integer.MIN_VALUE
```

When the range predicate is not satisfiable, the sum is 0 and the product is 1; for example:

```
(\sum int i; false; i) == 0
(\product double d; false; d*d) == 1.0
```

When the range predicate is not satisfiable for `\max` the result is the smallest number with the type of the expression in the body; for floating point numbers, negative infinity is used. Similarly, when the range predicate is not satisfiable for `\min`, the result is the largest number with the type of the expression in the body. [[[ Or should this be undefined - DRC]]]

### 11.4.24.3 Numerical Quantifier

The numerical quantifier, `\num_of`, returns the number of values for its variables for which the range and the expression in its body are true. The body must have type `boolean`, and the entire quantified expression has type `long`. The meaning of this quantifier is defined by the following equation (see p. 57 of [Cohen90]).

```
(\num_of T x; R(x); P(x)) == (\sum T x; R(x) && P(x); 1L)
```

### 11.4.24.4 Executability of Quantified Expressions

When are universal or existential quantifiers executable for purposes of runtime assertion checking? If the type of the quantified variable is `boolean`, then it is always executable. Otherwise a *spec-quantified-expr* is only executable if the form of the expression matches a pattern that the runtime assertion checker understands. This varies by tool implementation, but you can expect that the runtime assertion checker understands patterns where the range predicate gives a finite range for an ordinal primitive value type (such as `int`) or where the range predicate requires the quantified variable to be drawn from some set. Examples include the following. [[[Make these examples be real examples in the samples directory]]]

```
(\forall int x; 0 <= x && x < somelimit; ...)
(\forall Object x; someSet.has(x); ...)
```

You should get warnings from the `jmlc` tool when assertions are not executable, but you have to use the `-w2` flag to see them.

If a *spec-quantified-expr*, *QE*, is executable, then a tool executing it should only evaluate any range expression in *QE* once per execution of *QE*. Since the value of such a range expression cannot change, this evaluation strategy will not change the value of *QE*, but it will save time to only evaluate the range expression once for each evaluation of *QE*.

#### 11.4.24.5 Modifiers for Bound Variables

*bound-var-modifiers* ::= `non_null` | `nullable`

Logical variables can be bound in

- quantified expressions (see Section 11.4.24 [Quantified Expressions], page 98),
- set comprehension expressions (see Section 11.5 [Set Comprehensions], page 101),
- forall clauses of method contracts (see Section 9.9.1.1 [Forall Variable Declarations], page 73), or
- old clauses of method contracts (see Section 9.9.1.2 [Old Variable Declarations], page 73).

Note that in JML, `non_null` and `nullable` are not reserved words, hence such identifiers can be used as type names. In order to quantify over the elements of a type named `non_null` or `nullable` is necessary to provide an explicit nullity modifier. For example,

```
(\forall non_null non_null nn; ...)
```

where the first `non_null` is one of the *bound-var-modifiers* and the second is the type `non_null`.

#### 11.4.24.6 Quantifying over Reference Types

The range of values for a quantified variable that is declared to be of a reference type:

- Does not include `null` unless the bound variable is declared `nullable` (see Section E.2.1 [Non-null by Default], page 165).
- May include references to objects that are not constructed by the program; one should use a range predicate to eliminate such cases if they are not desired.

## 11.5 Set Comprehensions

The syntax of a *set-comprehension* expression is as follows.

*set-comprehension* ::= `{` [ *bound-var-modifiers* ] *type-spec*
*quantified-var-declarator* '`|`'
*postfix-expr* `&&` *predicate* `}`

The set comprehension notation can be used to succinctly define sets. For example, the following is the `JMLObjectSet` that is the subset of non-`null` `Integer` objects found in the set `myIntSet` whose values are between 0 and 10, inclusive.

```
new JMLObjectSet {Integer i | myIntSet.has(i) &&
    i != null && 0 <= i.intValue() && i.intValue() <= 10 }
```

The syntax of JML limits set comprehensions so that the *postfix-expr* following the vertical bar (`|`) is always a method invocation with the bound variable declared in the *quantified-var-declarator* as its parameter; the method may be either the `has` method of an `org.jmlspecs.models.JMLObjectSet` or `org.jmlspecs.models.JMLValueSet`, or the `contains` method of a `java.util.Collection`. This restriction is used to avoid Russell's paradox [Whitehead-Russell25]. The bound variable, whose scope is the *set-comprehension*, may not conflict with existing local variables, but may hide static and instance fields. The bound variable type is used to restrict the objects that become part of the resulting set; if the set called in the *postfix-expr* contains objects that are not assignable to the bound variable, they are not contained in the resulting set comprehension. Thus, the following two set comprehension expressions result in identical sets:

```
new JMLObjectSet {Integer i | s.has(i) && 0 < i.intValue() }
new JMLObjectSet {Object i | s.has(i) && i instanceof Integer &&
    0 < ((Integer) i).intValue() }
```

In practice, one starts either from some relevant set at hand or from the sets found in `JMLObjectSet` and `JMLValueSet` containing the objects of primitive types. The type of a set comprehension is the type named following `new`, which must be `JMLObjectSet` or `JMLValueSet`. The bound variable type must be compatible with the set comprehension type; in particular, the bound variable type must be a subtype of `org.jmlspecs.models.JMLType` if the set comprehension type is `JMLValueSet`.

## 11.6 JML Operators

In this section we describe the various new operators that JML adds to Java expressions. The following can all be used in *spec-expressions*.

### 11.6.1 Subtype operator

The relational operator `<:` compares two reference types and returns true when the type on the left is a subtype of the type on the right [Leino-Nelson-Saxe00]. Although the notation might suggest otherwise, this operator is also reflexive; a type will compare as `<:` with itself. In an expression of the form *E1* `<:` *E2*, both *E1* and *E2* must have type `\TYPE`; since in JML `\TYPE` is the same as `java.lang.Class` the expression *E1* `<:` *E2* means the same thing as the expression *E2*`.isAssignableFrom(`*E1*`)`. As a result, primitive types are not subtypes of `java.lang.Object`, nor of each other, though they are of themselves; so, for example, `Integer.TYPE <: Integer.TYPE` is true.

### 11.6.2 Equivalence and Inequivalence Operators

The operators `<==>` and `<=!=>` work only on boolean-subexpressions and have the same meaning as `==` and `!=`, respectively. However, they have very low precedence, and so are useful at the top-level of a *spec-expression*. Unlike `==` and `!=`, the operators `<==>` and `<=!=>` are also associative and symmetric.

The notation `<==>` can be read "if and only if". It has the same meaning for Boolean values as `==`, but has a lower precedence. Therefore, the expression "`\result <==> size == 0`" means the same thing as "`\result == (size == 0)`".

The notation `<=!=>` can be read "is not equivalent to". It has the same meaning for Boolean values as `!=`, but has a lower precedence. Therefore, the expression "`\result <=!=> size == 0`" means the same thing as "`\result != (size == 0)`".

The expressions on either side of these operators must be of type `boolean`, and the type of the result is also `boolean`.

### 11.6.3 Forward and Reverse Implication Operators

The operators `==>` and `<==` work only on boolean-subexpressions. They compute forward and reverse implications, respectively.

For example, the formula `raining ==> getsWet` is true if either `raining` is false or `getsWet` is true. The formula `getsWet <== raining` means the same thing. The `==>` operator associates to the right, but the `<==` operator associates to the left. The expressions on either side of these operators must be of type `boolean`, and the type of the result is also `boolean`.

These two operators are evaluated in short-circuit fashion, left to right. Thus, in `a ==> b`, if `a` is false, then the expression is true and `b` is not evaluated. Similarly, in `a <== b`, if `a` is true, the expression is true and `b` is not evaluated. In other words, `a ==> b` is equivalent to `!a || b` and `a <== b` is equivalent to `a || !b`.

Because of this short-circuit evaluation, `a ==> b` is not quite equivalent to `b <== a`. For example, `x != null ==> x.a > 0` will be true if `x` is `null`, but `x.a>0 <== x != null` would be undefined (or throw a NullPointerException) if `x` is `null`.

### 11.6.4 Lockset Ordering

JML uses `<` and `<=` to test order of locks. JML extends these two operators, but not `>` and `>=`, as comparisons on Objects. Using `synchronized` statements, Java programs can establish monitor locks to permit only one thread at a time to execute given sections of code. Any object can be used as a lock. In order for ESC/Java [Leino-Nelson-Saxe00] to reason about the possibility of deadlocks among threads, a partial order must be statically declared on lock objects, with `"larger"` objects being objects whose locks should be acquired later. ESC/Java suggests the use of *axiom-clause*s to declare this partial order.

The `<` and `<=` operators test this partial order in assertions. When used in this way, the subexpressions to either side of `<` or `<=` must be reference types, and the result is of type boolean.

## 11.7 Store Refs

The syntax related to the *store-ref* production is used in several places.

*store-ref-list* ::= *store-ref-keyword* `|` *store-ref* [ `,` *store-ref* ] . . .
*store-ref* ::= *store-ref-expression*
        `|` *informal-description*
*store-ref-expression* ::= *store-ref-name* [ *store-ref-name-suffix* ] . . .
*store-ref-name* ::= *ident* `|` `super` `|` `this`
*store-ref-name-suffix* ::= `.` *ident* `|` `. this` `|` '`[`' *spec-array-ref-expr* '`]`' `|` `. *`
*spec-array-ref-expr* ::= *spec-expression*
        `|` *spec-expression* `..` *spec-expression*
        `| *`
*store-ref-keyword* ::= `\nothing` `|` `\everything` `|` `\not_specified`

A *store-ref* denotes a set of locations in general.

The form `\nothing` denotes the empty set of locations. The form `\everything` denotes the set of all locations in the program. The form `\not_specified` denotes a unspecified set of locations, whose usage is determined by the tool.

The form *SR*`.*` refers to all fields of the object denoted by *SR*. Similarly, the form *A*`[*]` refers to all locations of elements in the array *A*. [[[ And their datagroups? - DRC]]]

Otherwise if a *store-ref* refers to a field, it denotes that field's data group (see Chapter 10 [Data Groups], page 85). If a *store-ref* refers to an element or a range of elements, it refers to all of the named locations in that array.

# 12 Statements and Annotation Statements

JML also defines a number of annotation statements that may be interspersed with Java statements in the body of a method, constructor, or initialization block.

The following gives the syntax of statements. These are the standard Java statements, with the addition of annotations, the *hence-by-statement*, *assert-redundantly-statement*, *assume-statement*, *set-statement*, *unreachable-statement*, *debug-statement*, and the various forms of *model-prog-statement*. See Chapter 14 [Model Programs], page 117, for the syntax of *model-prog-statement*, which is only allowed in model programs. [[[ Does this include local class declarations?]]]

*compound-statement* ::= `{` *statement* [ *statement* ] . . . `}`
*statement* ::= *compound-statement*
  | *local-declaration* `;`
  | *ident* `:` *statement*
  | *expression* `;`
  | `if` `(` *expression* `)`
   *statement* [ `else` *statement* ]
  | *possibly-annotated-loop*
  | `break` [ *ident* ] `;`
  | `continue` [ *ident* ] `;`
  | `return` [ *expression* ] `;`
  | *switch-statement*
  | *try-block*
  | `throw` *expression* `;`
  | `synchronized` `(` *expression* `)` *statement*
  | `;`
  | *jml-annotation-statement*
  | *assert-statement*
  | *jml-annotation-statement*
  | *model-prog-statement* // *only allowed in model programs*
*switch-statement* ::= `switch` `(` *expression* `)` `{`
    [ *switch-body* ] . . . `}`
*switch-body* ::= *switch-label-seq* [ *statement* ] . . .
*switch-label-seq* ::= *switch-label* [ *switch-label* ] . . .
*switch-label* ::= `case` *expression* `:` | `default` `:`
*try-block* ::= `try` *compound-statement*
   [ *handler* ] . . .
   [ `finally` *compound-statement* ]
*handler* ::= `catch` `(` *param-declaration* `)` *compound-statement*

The semantics of the Java statements are as in Java [Arnold-Gosling-Holmes00] [Gosling-etal00]. More details on the JML-specific features related to statements are described below.

## 12.1 Local Declaration Statements

The following is the syntax of local declaration statements. See Section 7.1.2 [Field and Variable Declarations], page 47, for the syntax of *variable-decls*.

*local-declaration* ::= *local-modifiers* *variable-decls*

### 12.1.1 Modifiers for Local Declarations

JML allows the modifiers `ghost`, `uninitialized`, `non_null` and `nullable` in addition to Java's `final` modifier on local variable declarations. See Chapter 18 [Universe Type System], page 132, for the grammar of *ownership-modifier*.

*local-modifiers* ::= [ *local-modifier* ] . . .
*local-modifier* ::= `ghost` | `final` `uninitialized` | `non_null` | `nullable`
  | *ownership-modifier* *// when the Universe type system is on*

The JML modifiers are discussed to some extent below. See Section 7.1.2.1 [JML Modifiers for Fields], page 47, for more about these modifiers.

When used as a local variable modifier, `uninitialized` means that the variable should be considered by the tools to be uninitialized, even if it has an initialization. This allows the tools to check for uses before a "real" initialization.

A *local ghost declaration* is a variable declaration with a `ghost` modifier, entirely contained in an annotation. It introduces a new variable that may be used in subsequent annotations within the remainder of the block in which the declaration appears. A ghost variable is not used in program execution as Java variables are, but is used by runtime assertion checkers or a static checker to reason about the execution of the routine body in which the ghost variable is used.

- The variable name may not be already declared as a local variable or local ghost variable or as a formal parameter of the routine in which the declaration appears.
- Each variable declared may have an initializer; the initializer is in the scope of the newly declared variable.
- The modifiers `final`, `uninitialized`, `non_null` and `nullable` may be used on the ghost declaration.

In the following, the body of the method `ghostLocalExample` contains several examples of local ghost declarations.

```
package org.jmlspecs.samples.jmlrefman;

public abstract class GhostLocals {
    void ghostLocalExample() {
        //@ ghost int i = 0;
        //@ ghost int zero = 0, j, k = i+3;
        //@ ghost float[] a = {1, 2, 3};
        //@ ghost Object o;
        //@ final ghost non_null Object nno = new Object();
    }
}
```

## 12.2 Loop Statements

The following is the syntax of loop statements.

*possibly-annotated-loop* ::=
  [ *loop-invariant* ] . . .

[ *variant-function* ] . . .
[ *ident* : ] *loop-stmt*
*loop-stmt* ::= while ( *expression* ) *statement*
| do *statement* while ( *expression* ) ;
| for ( [ *for-init* ] ; [ *expression* ] ; [ *expression-list* ] )
*statement*
*for-init* ::= *local-declaration* | *expression-list*

In JML a loop statement can be annotated with one or more loop invariants, and one or more variant functions. The following class contains an example in the middle of the method `sumArray`. This example has a `while` loop with two loop invariants, which follow the keyword `maintaining`, and a single variant function, which follows the keyword `decreasing`. The invariants and variant function are written above the loop itself. The first loop invariant describes the range that the variable `i` can take, and the second relates `i` and the value in `sum`.

```
package org.jmlspecs.samples.jmlrefman;

/** An example of some simple loops with loop invariants
 *  and variant functions specified.
 */
public abstract class SumArrayLoop {

    /** Return the sum of the argument array. */
    /*@   old \bigint sum =
      @ (\sum int j; 0 <= j && j < a.length; (\bigint)a[j]);
      @   requires Long.MIN_VALUE <= sum && sum <= Long.MAX_VALUE;
      @   assignable \nothing;
      @   ensures \result == sum;
      @*/
    public static long sumArray(int [] a) {
        long sum = 0;
        int i = a.length;

        /*@ maintaining -1 <= i && i <= a.length;
          @ maintaining sum
          @            == (\sum int j;
          @                   i <= j && 0 <= j && j < a.length;
          @                   (\bigint)a[j]);
          @ decreasing i; @*/
        while (--i >= 0) {
            sum += a[i];
        }

        //@ assert i < 0 && -1 <= i && i <= a.length;
        //@ hence_by (i < 0 && -1 <= i) ==> i == -1;
        //@ assert i == -1 && i <= a.length;
        //@ assert sum == (\sum int j; 0 <= j && j < a.length; (\bigint)a[j]);
```

```
            return sum;
        }

    }
```

At the end of the loop, the negation of the loop's test expression and the loop invariants hold. This is shown by the assertions after the loop.

Loop invariants and variant functions are discussed in more detail below. (Thanks to K. Rustan M. Leino, Claude Marche, and Steve M. Shaner for discussions on this topic, including details of the semantics.)

### 12.2.1 Loop Invariants

A loop can specify one or more loop invariants, using the following syntax.

*loop-invariant* ::= *maintaining-keyword* *predicate* `;`
*maintaining-keyword* ::= `maintaining` | `maintaining_redundantly`
    | `loop_invariant` | `loop_invariant_redundantly`

A *loop-invariant* is used to help prove partial correctness of a loop statement.

The meaning of a loop, which does not contain a use of `break` that exits the loop itself (as opposed to some inner loop), such as

```
//@ maintaining J;
while (B) { S }
```

is as follows.

```
while (true) {
  //@ assert J;
  if (!(B)) { break; }
  S
}
```

So that the loop invariant holds at the beginning of each iteration of the loop.

The rule for deducing what is true after the loop can be stated simply if the loop does not contain any `break` statements that exit the loop, and if the loop test, `B`, is both a Java *expression* and a JML *specification-expression* (see Section 11.2 [Specification Expressions], page 87). (This means that `B` is side-effect free.) For such loops, the rule is that, after a loop with condition `B` and invariant `J` the negation of the condition, `(!B)`, conjoined with the invariant, `J`, holds. This is summarized in the following program schema.

```
//@ maintaining J;
while (B) {  // assuming B has no side effects
  S
}
// assert !(B) && J;
```

If the loop contains a `break` statement that exits the loop itself, then more detailed reasoning is necessary to establish what will be true after the loop. The intended condition that should be true after the loop when it is exited via a `break` statement can be recorded in the code using an `assert` statement. For example, if the loop has the form:

```
//@ maintaining J;
while (true) {
```

```
    S1
    if (C) {
       S2
       //@ assert Q;
       break;
    }
    S3
  }
```

then after the loop the asserted condition, `Q`, should hold, assuming there are no other `break` statements that exit the loop.

### 12.2.2 Loop Variant Functions

A loop can also specify one or more variant functions, using the following syntax.

*variant-function* ::= *decreasing-keyword* *spec-expression* `;`
*decreasing-keyword* ::= `decreasing` | `decreasing_redundantly`
      | `decreases` | `decreases_redundantly`

A *variant-function* is used to help prove termination of a loop statement. It specifies an expression of type `long` or `int` that must be no less than 0 when the loop is executing, and must decrease by at least one (1) each time around the loop.

The meaning of a loop such as

```
//@ decreasing E;
while (B) { S }
```

in which $S$ does not use `continue`, is as follows.

```
while (true) {
  long vf = E;      // assuming vf is a fresh variable name
  if (!(B)) { break; }
  S
  //@ assert 0 <= vf;
  //@ assert E < vf;
}
```

If the loop contains a `continue` statement, then the loop variant is checked just before each use of `continue`. For example, if the loop has the form:

```
//@ decreasing E;
while (B) { S1 if (C) { S2 continue; } S3 }
```

then the meaning is as follows.

```
while (true) {
  long vf = E;      // assuming vf is a fresh variable name
  if (!(B)) { break; }
  S1
  if (C) {
     S2
     //@ assert 0 <= vf;
     //@ assert E < vf;
     continue;
```

```
    }
    S3
    //@ assert 0 <= vf;
    //@ assert E < vf;
  }
```

## 12.3 Assert Statements

The syntax of assert and redundant assert statements is as follows.

> *assert-statement* ::= `assert` *expression* [ `:` *expression* ] `;`
>   | `assert` *predicate* [ `:` *expression* ] `;`
> *assert-redundantly-statement* ::= `assert_redundantly` *predicate*
>   [ `:` *expression* ] `;`

Note that Java (as of J2SDK 1.4) also has its own `assert` statement. For this reason JML distinguishes between assert statements that occur inside and outside annotations.

Outside an annotation, an assert statement is a Java assert statement, whose syntax follows the first *assert-statement* production above. Thus in such an assert statement, the first *expression* can have side effects (potentially, although it shouldn't). The second expression is supposed to have type `String`, and will be used in a message should the assertion fail.

Inside an annotation, an assert statement is a JML assert statement, and the second syntax is used for *assert-statement*. Thus instead of an *expression* before the optional colon, there is a JML *predicate*. This predicate cannot have side effects, but can use the various JML extensions to the Java expression syntax (see Section 11.2 [Specification Expressions], page 87, for details.) As in a Java assert statement, the optional expression that follows the colon must be a `String`, which is printed if the assertion fails.

An assert statements tells JML to check that the specified *predicate* is true at the given point in the program. The runtime assertion checker checks such assertions during execution of the program, when control reaches the assert statement. Other tools, such as verification tools, will try to prove that the assertion always holds at that program point, for every possible execution.

The *assert-redundantly-statement* must appear in an annotation. It has the same semantics as the JML form of an assert statement, but is marked as redundant. Thus it would be used to call attention to some property, but need not be checked.

## 12.4 JML Annotation Statements

The following gives the syntax of JML annotation statements. These can appear anywhere in normal Java code, but must be enclosed in annotations. See Section 12.3 [Assert Statements], page 109, for the syntax of the *assert-redundantly-statement*. See Chapter 14 [Model Programs], page 117, for the syntax of additional statements that can only be used in model programs.

> *jml-annotation-statement* ::= *assert-redundantly-statement*
>   | *assume-statement*
>   | *hence-by-statement*
>   | *set-statement*

| *refining-statement*
| *unreachable-statement*
| *debug-statement*

### 12.4.1 Assume Statements

The syntax of an assume statement is as follows. As in a Java assert statement, the optional expression that follows the colon must be a `String`, which is printed if the assumption fails.

*assume-statement* ::= *assume-keyword predicate*
[ `:` *expression* ] `;`
*assume-keyword* ::= `assume` | `assume_redundantly`

In runtime assertion checking, assumptions are checked in the same way that assert statements are checked (see Section 12.3 [Assert Statements], page 109).

However, in static analysis tools, the assume statement is used to tell the tool that the given predicate is assumed to be true, and thus need not be checked.

### 12.4.2 Set Statements

The syntax of a set statement is as follows. See Section 11.3 [Expressions], page 87, for the syntax of *assignment-expr*.

*set-statement* ::= `set` *assignment-expr* `;`

A set statement is the equivalent of an assignment statement but is within an annotation. It is used to assign a value to a ghost variable or to a ghost field. A set statement serves to assist the static checker in reasoning about the execution of the routine body in which it appears.

- the target of the set statement must be a ghost variable or a ghost field
- the right-hand-side of the set statement must be pure (not have side effects)

**Examples:**

```
//@ set i = 0;
//@ set collection.elementType = \type(int);
```

[[[ Questions: must the rhs be pure? Should we allow an arbitrary statement, not just an assignment? such as set ++i; or set i += 5; ]]]

### 12.4.3 Refining Statements

The syntax of a refining statement is as follows. See Section 14.6 [Specification Statements], page 120, for the syntax of *spec-statement* and *generic-spec-statement-case*. See Chapter 12 [Statements and Annotation Statements], page 104, for the syntax of *statement*.

*refining-statement* ::= `refining` *spec-statement statement*
| `refining` *generic-spec-statement-case statement*

A refining statement allows one to annotate a specification with a specification. It has two parts, a *specification* and a *body*. The specification part can be either a *spec-statement* (see Section 14.6 [Specification Statements], page 120), which includes the grammar for a heavyweight specification case, or a *generic-spec-statement-case* (see Section 14.6 [Specification Statements], page 120), which includes the grammar for a lightweight specification case. The body is simply a statement. In particular, the body can be a *compound-statement* or a *jml-annotation-statement*, including a nested *refining-statement*.

Annotating the body with a specification is a way of collecting all the specification information about the statement in one place. Giving such an annotation is especially useful for framing, e.g., writing *assignable-clauses*. For example, by using a refining statement, one can write an assignable clause for a loop statement or for the statement in the body of a loop.

Refining statements are also used in connection with model program specification cases (see Chapter 14 [Model Programs], page 117). Within the implementation of a method with such a model program specification, a refining statement indicates exactly what *spec-statement* is implemented by its body, since its specification part would be exactly that *spec-statement*. This is helpful for "matching" the implementation against the model program specification [Shaner-Leavens-Naumann07].

Note that the scope of any declarations made in the specification part of a refining statement are limited to the specification part, and do not extend into the body. Thus a refining statement is type correct if each of its subparts is type correct, using the surrounding context for separately type checking the specification and body.

The meaning of a refining statement of the form `refining` *S B* is that the body *B* must refine the specification given in *S*. This means that *B* has to obey all the specifications given in *S*. For example, *B* may not assume a stronger precondition than that given by *S*. (Standard defaults are used for omitted clauses in the specification part of a refining statement; thus, if there is no requires clause in a *spec-statement*, then the precondition defaults to true.) Similarly, *B* may not assign to locations that are not permitted to be assigned to by *S*, and, assuming *S*'s precondition held, then when *B* terminates normally it must establish *S*'s normal postcondition. See Chapter 9 [Method Specifications], page 61, for more about what it means to satisfy such a specification.

When `\old()` or `\pre()` are used in the specification part of a refining statement, they have the same meaning as in a specification statement (see Section 14.6 [Specification Statements], page 120).

In execution, a refining statement of the form `refining` *S B* just executes its body *B*. For this reason, typically the `refining` keyword and the specification *S* would be in JML annotations, but the body *B* would be normal Java code (outside of any annotation).

See Chapter 14 [Model Programs], page 117, for more examples.

### 12.4.4 Unreachable Statements

The syntax of the `unreachable` statement is as follows.

*unreachable-statement* ::= `unreachable ;`

The `unreachable` statement is an annotation that asserts that the control flow of a routine will never reach that point in the program. It is equivalent to the annotation `assert false`. If control flow does reach an `unreachable` statement, a tool that checks (by reasoning or at runtime) the behavior of the routine should issue an error of some kind. The following is an example:

```
if (true) {
    ...
```

```
        } else {
                //@ unreachable;
        }
```

### 12.4.5 Debug Statements

The syntax of the `debug` statement is as follows. See Section 11.3 [Expressions], page 87, for the syntax of *expression*.

*debug-statement* ::= `debug` *expression* `;`

A `debug` statement is the equivalent of an expression statement but is within an annotation. Thus, features visible only in the JML scope can also appear in the `debug` statement. Examples of such features include ghost variables, model methods, `spec_public` fields, and JML-specific expression constructs, to name a few.

The main use of the `debug` statement is to help debugging specifications, e.g., by printing the value of a JML expression, as shown below.

```
        //@ debug System.err.println(x);
```

In the above example, the variable `x` may be a ghost variable. Note that using `System.err` automatically flushes output, unlike `System.out`. This flushing of output is helpful for debugging.

As shown in the above example, expressions with side-effects are allowed in the `debug` statement. These include not only methods with side-effects but also increment (`++`) and decrement (`--`) operators and various forms of assignment expressions (e.g., `=`, `+=`, etc.). Thus, the `debug` statement can also be used to assign a value to a variable, or mutate the state of an object.

```
        //@ debug x = x + 1;
        //@ debug aList.add(y);
```

However, a model variable cannot be assigned to, nor can its state be mutated by using the `debug` statement, as its value is given by a `represents` clause (see Section 8.4 [Represents Clauses], page 58).

There is no restriction on the type of expression allowed in the `debug` statement.

Tools should allow debug statements to be turned on or off easily. Thus programmers should not count on debug statements being executed. For example, if one needs to assign to a ghost variable, the proper way to do it is to use a *set-statement* (see Section 12.4.2 [Set Statements], page 110), which would execute even if debug statements are not being executed.

### 12.4.6 Hence By Statements

The syntax of the `hence_by` statement is as follows.

*hence-by-statement* ::= *hence-by-keyword predicate* `;`
*hence-by-keyword* ::= `hence_by | hence_by_redundantly`

The `hence_by` statement is used to record reasoning when writing a proof by intermittent assertions. It would normally be used between two assert statements (see Section 12.3 [Assert Statements], page 109) or between two assume statements (see Section 12.4.1 [Assume Statements], page 110).

[[[Needs example.]]]

# 13 Redundancy

JML has several features that allow the specification of implications [Tan95] and examples [Leavens97c] [Leavens-Baker99]. They are redundant in the sense that they do not constrain an implementation directly. Instead, they are useful for pointing out consequences to the specification's readers, for example to draw attention to some consequences of the specification of a method, or to illustrate it by an example.

In addition to clauses of the form *X*`_redundantly`, such as `requires_redundantly`, `ensures_redundantly`, etc., there are two sections of a method specification that are devoted to such redundant specifications. These sections of a method specification are described by the following grammar.

*redundant-spec* ::= *implications* [ *examples* ] `|` *examples*

The two subsections below explain these features. The description of clauses of the form *X*`_redundantly` is contained in the first section.

## 13.1 Redundant Implications and Redundantly Clauses

A *redudant implication* is a way of stating a claim about a specification. By itself it does not constrain an implication, but can be thought of a stating a theorem to be proven about a specification. Such redundant implications are useful for drawing the reader's attention to some point that might otherwise be overlooked, or that is important for rhetorical purposes [Leavens-Baker99].

Redundant implications can be specified in two ways in JML. The first is by using clauses of the form *X*`_redundantly`. The second is by use of the *implications* section of a method specification, which starts with the keyword `implies_that`. (See Section 9.2 [Organization of Method Specifications], page 61, for the syntax of *spec-case-seq*.)

*implications* ::= `implies_that` *spec-case-seq*

The *implications* section of a method specification says that for each visibility level *V*, and for each *spec-case* of visibility *V* in its *spec-case-seq*, that *spec-case* is refined by the entire non-redundant specification of the method that applies at visibility level *V*. Thus every correct implementation of the non-redundant specification must satisfy each of the *spec-cases* in the *implications* section.

For example, suppose that the (desugared) meaning of the non-redundant part of a method's specification has the form:

```
V behavior        // non-redundant
   requires Pre;
   assignable x1, x2;
   ensures NormPost;
   signals_only Ex1;
   signals (Exception e) ExPost;
```

and suppose that one of the *spec-cases* in its *implications* section has the following (desugared) meaning:

```
V behavior        // redundant
   requires RedPre;
   assignable x1, x2;
```

```
ensures RedNormPost;
signals_only Ex1;
signals (Exception e) RedExPost;
```

Then it must be the case that (by definition of refinement for method specifications [Leavens-Naumann06]) the following implications hold:

- `\old(RedPre) ==> Pre`,
- `(\old(RedPre) && NormPost) ==> RedNormPost`, and
- `(\old(RedPre) && ExPost) ==> RedExPost`.

These implications are only sensible if the specifications have the same visibility (*V*), the same `assignable` clauses, and the same `signals_only` clauses. If the `assignable` clauses differ, one can adjust by adding elements to the non-redundant parts of the assignable clause, to widen it, but preserve its meaning by adding restrictions (e.g., using the `\only_assigned` predicate), to the postconditions. Similar adjustments can be made to the non-redundant `signals_only` clause, by adding exceptions (or supertypes of exceptions) to the non-redundant `signals_only`, preserving its meaning by adding restrictions in the `signals` clause.

Redundant clauses are a syntactic variant of Tan's procedure claims [Tan95]. The meaning of a redundant clause, of the form *X*`_redundantly` is also defined as making a claim about implications, but in this case only one simple implication. The claim is that the predicate in the redundant clause follows from the meaning of the non-redundant *X* clauses.

As an example, consider the following requires clauses.

```
requires Pre;
requires_redundantly RedPre;
```

These state the claim that `Pre ==> RedPre`. That is, in all pre-states, whenever *Pre* is true, then *RedPre* must be true. The same pattern holds for all other clauses and their redundant counterparts, including ensures clauses, signals clauses (which must first be standardized to have the same exception [Raghavan-Leavens05]), invariants, etc.

For example, recall that multiple clauses are conjoined, and thus

```
ensures Q1;
ensures Q2;
ensures_redundantly RedQ1;
ensures_redundantly RedQ2;
```

is equivalent to

```
ensures Q1 && Q2;
ensures_redundantly RedQ1 && RedQ2;
```

In this example, the claim stated is that:

```
(Q1 && Q2) ==> (RedQ1 && RedQ2).
```

If one is using a theorem prover, then these implications can be thought of as theorems to prove (in the context of the overall class or interface specification).

A runtime assertion checker is free to check the specifications in the *implications* section, since they must all hold, as they should be refined by the non-redundant specification. If a redundant specification case in a method's *implications* section is violated, this could indicate that either: (a) the implications described above do not hold, or that (b) there is a violation of the specification by the caller (e.g., if the precondition does not hold) or by the implementation of the method (e.g., if the normal postcondition does not hold).

[[[Needs concrete examples.]]]

## 13.2 Redundant Examples

Examples are, used to point out, to readers or testing tools, particular cases of a method specification [Leavens97c] [Leavens-Baker99] [Leavens-Baker-Ruby06]. The following gives the syntax of the *examples* section of a method specification. This section starts with the `for_example` keyword, and includes one or more *example*s. Each *example* is much like a *spec-case* (see Section 9.2 [Organization of Method Specifications], page 61), but uses various `example` keywords instead of `behavior` keywords, and does not permit *model-program* cases.

```
examples ::= for_example example [ also example ] . . .
example ::= [ [ privacy ] example ]
              [ spec-var-decls ]
              [ spec-header ]
              simple-spec-body
          | [ privacy ] exceptional_example
            [ spec-var-decls ]
            spec-header
            [ exceptional-example-body ]
          | [ privacy ] exceptional_example
            [ spec-var-decls ]
            exceptional-example-body
          | [ privacy ] normal_example
            [ spec-var-decls ]
            spec-header
            [ normal-example-body ]
          | [ privacy ] normal_example
            [ spec-var-decls ]
            normal-example-body
exceptional-example-body ::= exceptional-spec-clause
                             [ exceptional-spec-clause ] . . .
normal-example-body ::= normal-spec-clause
                        [ normal-spec-clause ] . . .
```

As in method *spec-case*s (see Section 9.2 [Organization of Method Specifications], page 61) there are both heavyweight and lightweight examples. A *lightweight* example does not use one of the `example` keywords. A *heavyweight* example uses one of the example keywords. As with *spec-case*s, only heavyweight examples can have a specified visibility; lightweight examples all have the same visibility as the method (or constructor) being specified.

The defaults for omitted clauses in lightweight *example*s are the same as those for omitted clauses in lightweight *spec-case*s. Similarly, heavyweight *example*s have the same defaults as heavyweight *spec-case*s. (See Section 9.6.1 [Semantics of flat behavior specification cases], page 66, for the defaults for a lightweight and heavyweight specification cases.)

As described in the “Preliminary Design of JML” [Leavens-Baker-Ruby06] (section 2.3.2.1) “the specification in each example should be such that:

- the example’s precondition implies the precondition of the expanded meaning of the specified behaviors,
- the example’s assignable clause specifies a subset of the locations that are assignable according to the expanded meaning of the specified behaviors, and
- assuming the example’s assignable clause, the conjunction of:
  - the example’s precondition (wrapped by `\old()`),
  - the precondition of the expanded meaning of the specified behaviors (also wrapped by `\old()`), and
  - the postcondition of the expanded meaning of the specified behaviors

  should be equivalent to the example’s postcondition.

Requiring equivalence to the example’s postcondition means that it can serve as a test oracle for the inputs described by the example’s precondition. If there is only one specified `public normal_behavior`” specification case “and if there are no preconditions and assignable clauses, then the example’s postcondition should the equivalent to the conjunction of the example’s precondition and the postcondition of the `public normal_behavior` specification.”

[[[(Needs concrete examples :-)]]]

# 14 Model Programs

This chapter discusses JML's model programs, which are adapted from the refinement calculus [Back88] [Back-vonWright89a] [Buechi-Weck00] [Morgan94] [Morris87]. Details of JML's design and semantics for model program specifications are described in a recent paper [Shaner-Leavens-Naumann07].

## 14.1 Ideas Behind Model Programs

The basic idea of a model program is that it is a specification that is written as an abstract algorithm. Such an abstract algorithm specifies a method in the sense that the method's execution should be a refinement of the model program.

JML adopts ideas from Büchi and Weck's "grey-box approach" to specification [Buechi-Weck00] [Buechi00]. However, JML structurally restricts the notion of refinement by not permitting all implementations with behavior that refines the model program, but only allowing implementations that syntactically match the model program [Shaner-Leavens-Naumann07]. The current JML notion of matching uses *refining-statements* (see Section 12.4.3 [Refining Statements], page 110), as explained below. This turns out to be a simple and easy to understand technique for specifying and verifying both higher-order features and callbacks.

Consider the following example (from a survey on behavioral subtyping by Leavens and Dhara [Leavens-Dhara00]). In this example, both the methods are specified using model programs, which are explained below.

```
package org.jmlspecs.samples.dirobserver;

//@ model import org.jmlspecs.models.JMLString;
//@ model import org.jmlspecs.models.JMLObjectSetEnumerator;

/** Directories that can be both read and written. */
public interface Directory extends RODirectory {

  /** Add a mapping from the given string
   *  to the given file to this directory.
   */
  /*@ public model_program {
    @   normal_behavior
    @     requires !in_notifier && n != null && n != "" && f != null;
    @     assignable entries;
    @     ensures entries != null
    @        && entries.equals(\old(entries.extend(
    @                                     new JMLString(n), f)));
    @
    @   maintaining !in_notifier && n != null && n != "" && f != null
    @               && e != null;
    @   decreasing e.uniteratedElems.size();
    @   for (JMLObjectSetEnumerator e = listeners.elements();
    @        e.hasMoreElements(); ) {
```

```
          @     set in_notifier = true;
          @     ((DirObserver)e.nextElement()).addNotification(this, n);
          @     set in_notifier = false;
          @   }
          @ }
          @*/
        public void addEntry(String n, File f);

        /** Remove the entry with the given name from this directory. */
        /*@ public model_program {
          @   normal_behavior
          @     requires !in_notifier && n != null && n != "";
          @     assignable entries;
          @     ensures entries != null
          @        && entries.equals
          @               (\old(entries.removeDomainElement(
          @                                             new JMLString(n))));
          @
          @   maintaining !in_notifier && n != null && n != "" && e != null;
          @   decreasing e.uniteratedElems.size();
          @   for (JMLObjectSetEnumerator e = listeners.elements();
          @        e.hasMoreElements(); ) {
          @     set in_notifier = true;
          @     ((DirObserver)e.nextElement()).removeNotification(this, n);
          @     set in_notifier = false;
          @   }
          @ }
          @*/
        public void removeEntry(String n);
    }
```

Both model programs in the above example are formed from a specification statement, which begins with the keyword `normal_behavior` in these examples, and a for-loop. The key event in the for loop bodies is a method call to a method (`addNotification` or `removeNotification`). These calls must occur in a state equivalent to the one reached in the model program for the implementation to be legal.

The specification statements abstract away part of a correct implementation. The `normal_behavior` statements in these examples both have a precondition, a frame axiom, and a postcondition. These mean that the statements that they abstract away from must be able to, in any state satisfying the precondition, finish in a state satisfying the postcondition, while only assigning to the locations (and their dependees) named in the frame axiom. For example, the first specification statement says that whenever `in_notifier` is false, `n` is not null and not empty, and `f` is not null, then this part of the method can assign to `entries` something that isn't null and that is equal to the old value of `entries` extended with a pair consisting of the string `n` and the file `f`.

The model field `entries`, of type `JMLValueToObjectMap`, is declared in the supertype `RODirectory` [Leavens-Dhara00].

Implementations of model programs must match each specification statement in a model program with a corresponding refining statement. In the matching refining statement, the specification part must be textually equal to the specification statement. The body of the refining statement must thus implement the given specification for that statement (see Section 12.4.3 [Refining Statements], page 110).

## 14.2 Extracting Model Program Specifications

Since refining statements contain both specifications and implementations, it is possible to extract a model program specification from an implementation with (zero or more) refining statements. This is done by using the modifier `extract` on the method [Shaner-Leavens-Naumann07]. [[[Give example.]]]

## 14.3 Details of Model Programs

The following gives the syntax of model programs. See Chapter 12 [Statements and Annotation Statements], page 104, for the parts of the syntax of statements that are unchanged from Java. The *jml-compound-statement* and *jml-statement* syntax is the same as the *compound-statement* and *statement* syntax, except that *model-prog-statement*s are not flagged as errors within the *jml-compound-statement* and *jml-statement*s.

> *model-program* ::= [ *privacy* ] [ `code` ] `model_program`
>             *jml-compound-statement*
> *jml-compound-statement* ::= *compound-statement*
> *jml-statement* ::= *statement*
> *model-prog-statement* ::= *nondeterministic-choice*
>       `|` *nondeterministic-if*
>       `|` *spec-statement*
>       `|` *invariant*

## 14.4 Nondeterministic Choice Statement

The syntax of the *nondeterministic-choice* statement is as follows.

> *nondeterministic-choice* ::= `choose` *alternative-statements*
> *alternative-statements* ::= *jml-compound-statement*
>         [ `or` *jml-compound-statement* ] . . .

The meaning is that a correct implementation can dynamically execute (e.g., with an `if` or `switch` statement), one of the alternatives. Code may also make a static choice of one of the alternatives.

## 14.5 Nondeterministic If Statement

> *nondeterministic-if* ::= `choose_if` *guarded-statements*
>         [ `else` *jml-compound-statement* ]
> *guarded-statements* ::= *guarded-statement*
>         [ `or` *guarded-statement* ] . . .
> *guarded-statement* ::= `{`
>         *assume-statement*
>         *jml-statement* [ *jml-statement*] . . . `}`

The meaning of a nondeterministic if statement is that a correct implementation may dynamically choose any of the guarded-statements for which the guard (the first *assume-statement* in the *guarded-statement*) is true. If none of these are true, then it must execute the *jml-compound-statement* given following `else`, but it may not do that if one of the guards in the guarded statements is true.

## 14.6 Specification Statements

The grammar for specification statements appears below. It is unusual, compared to specification statements in refinement calculus, in that it allows one to specify statements that can signal exceptions, or terminate abruptly. The reasons for this are based on verification logics for Java [Huisman01] [Jacobs-Poll01] [Ruby06], which have these possibilities. The meaning of an *abrupt-spec-case* is that the normal termination and signaling an exception are forbidden; that is, the equivalent *spec-statement* using `behavior` would have `ensures false;` and `signals (Exception) false;` clauses. Hence in an *abrupt-spec-case*, JML does not allow use of an *ensures-clause*, *signals-only-clause*, or *signals-clause*.

```
spec-statement ::= [ privacy ] behavior-keyword
                generic-spec-statement-case
        | [ privacy ] exceptional-behavior-keyword
          exceptional-spec-case
        | [ privacy ] normal-behavior-keyword
          normal-spec-case
        | [ privacy ] abrupt-behavior-keyword
          abrupt-spec-case
generic-spec-statement-case ::= [ spec-var-decls ]
                generic-spec-statement-body
        | [ spec-var-decls ]
          spec-header
          [ generic-spec-statement-body ]
generic-spec-statement-body ::= simple-spec-statement-body
        | {| generic-spec-statement-case-seq |}
generic-spec-statement-body-seq ::= generic-spec-statement-case
          [ also generic-spec-statement-case ] . . .
simple-spec-statement-body ::= simple-spec-statement-clause
                [ simple-spec-statement-clause ] . . .
simple-spec-statement-clause ::= diverges-clause
        | assignable-clause
        | when-clause | working-space-clause | duration-clause
        | ensures-clause | signals-only-clause | signals-clause
        | continues-clause | breaks-clause | returns-clause
abrupt-behavior-keyword ::= abrupt_behavior | abrupt_behaviour
abrupt-spec-case ::= generic-spec-statement-case
```

The meaning of a *spec-statement* is that the code in a correct implementation must refine the given specification. One way to ensure this is to use a *refining-statement* in the implementation that contains the *spec-statement* in its specification part (see Section 12.4.3 [Refining Statements], page 110).

The following subsections describe details of each of the new clauses that may appear in an *abrupt-spec-case* or a *generic-spec-statement-case*.

### 14.6.1 Continues Clause

*continues-clause* ::= *continues-keyword* [ *target-label* ]
        [ *pred-or-not* ] `;`
*continues-keyword* ::= `continues` | `continues_redundantly`
*target-label* ::= `->` `(` *ident* `)`

The meaning of the *continues-clause* is that if the statement that implements the specification statement executes a `continue`, then it must continue to the given *target-label* (if any), and the given predicate (if any) must hold in the state just before the `continue` is executed.

### 14.6.2 Breaks Clause

*breaks-clause* ::= *breaks-keyword* [ *target-label* ]
        [ *pred-or-not* ] `;`
*breaks-keyword* ::= `breaks` | `breaks_redundantly`

The meaning of the *breaks-clause* is that if the statement that implements the specification statement executes a `break`, then it must break to the given *target-label* (if any), and the given predicate (if any) must hold in the state just before the `break` is executed.

### 14.6.3 Returns Clause

*returns-clause* ::= *returns-keyword* [ *pred-or-not* ] `;`
*returns-keyword* ::= `returns` | `returns_redundantly`

The meaning of the *returns-clause* is that if the statement that implements the specification statement executes a `return`, then the given predicate (if any) must hold in the state following evaluation of the return value, but just before the `return` is executed. The predicate (if any) in a returns clause may use `\result` to name the computed return value.

# 15 Specification for Subtypes

This chapter describes how JML specifies a type so that one can program subtypes from the specification, without the need to see the code of the supertypes that have been specified.

The problem of specifying enough about superclasses has been discussed by Kiczales and Lamping [Kiczales-Lamping92] and by Steyaert, et al. [Steyaert-etal96]. This problem is difficult because of the many ways that subclasses can depend on coding details of a superclass. For example, a subclass can depend on the calling pattern among a superclass's method and the fields that a superclass can access [Kiczales-Lamping92] [Steyaert-etal96].

JML builds on the work of Ruby and Leavens to solve this problem [Ruby-Leavens00] [Ruby06], which builds on the earlier works described above. The idea is to write specifications for subclasses in three parts. The first is the usual, public specification, which is primarily for clients but also useful to subclasses, who need to know what public interface they must meet. The second is a protected specification, which specifies fields and methods that are usable by the subclass. The third is the code contract. The code contract has a different syntax in JML than it did in [Ruby-Leavens00]. In the current JML a *code contract* is a heavyweight behavior specification case (see Section 9.5 [Heavyweight Specification Cases], page 65) or as a model program (see Chapter 14 [Model Programs], page 117) that uses the keyword "`code`." The `code` keyword is used just before one of the behavior keywords or just before the keyword `model_program`.

While code contracts can be generated automatically by a tool, as imagined by Ruby and Leavens [Ruby-Leavens00] [Ruby06], they can also be written by users directly. This is sometimes useful for documenting the implementation of a method. The code contract is intended to be created automatically, by a tool (which does not, as of this writing, exist). It has the following syntax.

In code contracts as described in the work of Ruby and Leavens, the main clauses used are the *accessible-clause* and the *callable-clause*. See Section 9.9.10 [Accessible Clauses], page 81, for the syntax and semantics of the *accessible-clause*. See Section 9.9.11 [Callable Clauses], page 82, for the syntax and semantics of the *callable-clause*.

## 15.1 Method of Specifying for Subclasses

[[[This should be a synopsis of Clyde Ruby's dissertation, with an example.]]]

## 15.2 Code Contracts

This section discusses the semantics of "code contracts," which are specification cases that use the "`code`" keyword. (See Section 9.6 [Behavior Specification Cases], page 65, for the detailed syntax of such specification cases.)

This feature was inspired by "does" clause of the Alloy Annotation Language [Khurshid-Marinov-Jackson02].

The modifier `code` may not be used on an abstract method. It follows that the `code` modifier cannot be used to document normal Java methods in interfaces. (In an interface, `code` could only be used in the specification of a model method that has a body.)

Tools for JML should warn the user if `code` is used in a specification case for a constructor, or for a final, static, or private method. It does no harm there, but is not needed.

The meaning of the `code` modifier is just that specification cases or model programs containing them are not inherited. That is, whenever the method is overridden, it does not inherit code contracts from its supertypes.

In verification of a method call, you can use all non-code specification cases, that are visible at a call site, for the statically-determined method being called. Such specifications are inherited by each subtype's method overrides to preserve behavioral subtyping [Dhara-Leavens96] [Leavens-Naumann06] [Leavens06b].

In verification of a method call, you can use a code specification case for a method $m$ given in a class $C$ only if you can prove that the method being called is method $m$ in class $C$. This applies in particular to super calls, which is the main use for such code contracts. (It would also apply to calls to final methods, calls to methods in final classes, and calls to private or static methods.)

# 16 Refinement

This chapter explains JML's notion of refinement files, which uses the following syntax.

> *refine-prefix* ::= *refine-keyword string-literal* `;`
> *refine-keyword* ::= `refine` | `refines`

The *refine-prefix* in a compilation unit says that the declarations in this compilation unit refine the corresponding declarations in the file named by the *string-literal*. The *string-literal* should name a file, complete with a suffix, for example, `"MyType.java-refined"`. The suffix of such a file is used by JML tools to find the file that is the base of a refinement chain, and all other files in the chain are found using the files named in the *refine-prefix* of a previous file in the chain.

One can use either keyword, `refine` or `refines` in a *refine-prefix*, although for historical reasons most examples use `refine`.

The following gives more details about the checks and meaning of this feature of JML.

## 16.1 File Name Suffixes

The JML tools recognize several filename suffixes. The following are considered to be *active* suffixes: '`.refines-java`', '`.refines-spec`', '`.refines-jml`', '`.java`', '`.spec`', and '`.jml`'; There are also three *passive* suffixes: '`.java-refined`', '`.spec-refined`', and '`.jml-refined`'. Files with passive suffixes can be used in refinements but should not normally be passed explicitly to the tools directly. These filename suffixes are ordered from most active to least active, in the order given above. Graphical user interface tools for JML should, by default, only present the active suffixes for selection. Among files in a directory with the same prefix, but with different active suffixes, the one whose suffix appears first in the list of active suffixes above should be considered primary by such a tool.

See Section 16.2 [Using Separate Files], page 124, for guidelines on how to use these suffixes. See Section 16.3 [Refinement Chains], page 125, for details on the semantics of specifications written using separate files.

## 16.2 Using Separate Files

Typically, JML specifications are written into annotation comments in '`.java`' files, and this is certainly the simplest way to use JML and its tools.

However, there are some circumstances in which one may wish to separate the specification from the Java code. An important example of this is when you do not own the sources for the Java code, but wish to specify it. This might happen if you are specifying a class library or framework that you are using. When you do not have control of the code, it is best to put the specification in a different file.

To add specifications to such a library or framework, one would use a filename with an active suffix, such as '`.refines-java`' (or '`.refines-spec`' or '`.refines-jml`'). The file with such a name would hold the specifications of the corresponding Java compilation unit. For example, if one wants to specify the type `LibraryType`, without touching the file '`LibraryType.java`' then one could write specifications in the file '`LibraryType.refines-java`', and include in that file the following *refine-prefix*.

```
refine "LibraryType.java";
```

If you are specifying code for which no sources are available (a class library in binary form), then you should use the '`.spec`' or '`.jml`' suffixes to write the specification. Such specifications act much like those written in '`.refines-spec`' or '`.refines-jml`' files, but would not include a *refine-prefix*. They allow specifications to be written without having to write Java code for the bodies of methods (as do all non-'`.java`') files.

Another reason for writing specifications in different files is to prevent the specifications from "cluttering up" the code (making it hard to see all of the code at once). This is also possible by using separate files for the specification and the code. In such a case one has a choice of suffixes, depending on whether one considers the code to be primary or the specification. If the code is primary, or has been written already, then one can treat the code as if it were written in an extra library, using the '`.refines-java`' (or '`.refines-spec`' or '`.refines-jml`') suffixes to specify the Java files as above.

On the other hand, if the specification is primary, or is to be written first, one could instead use the '`.java-refined`' (or '`.spec-refined`' or '`.jml-refined`') suffixes, and then write a *refine-prefix* in the '`.java`' file. For example, one might specify the class `MyType` in a file named '`MyType.java-refined`'. Then one could write the implementation of `MyType` in a file called '`MyType.java`'. The file '`MyType.java`' would include the following *refine-prefix*:

```
refine "MyType.java-refined";
```

In this case, the specification found in '`MyType.java-refined`' *is a refinement* of the implementation found in '`MyType.java`'.

Combinations of these techniques can also be used, by using several files instead of just a code file and a specification file. See Section 16.3 [Refinement Chains], page 125, for the meaning of JML specifications in this general case.

To summarize, aside from the standard '`.java`' suffix, one would use file name suffixes as follows.

- If you are specifying before coding, but want to keep the specifications in a different file, but you want to have the '`.java`' file refer directly to the specification, then use one of the suffixes: '`.java-refined`', '`.spec-refined`', or '`.jml-refined`'). The '`.java`' file would name the file it refines (as would other files in the chain) in a *refine-prefix*.
- If you have a '`.java`' file, but the *refine-prefix* cannot or should not appear in that '`.java`' file, then use one of the suffixes: '`.refines-java`', '`.refines-spec`', or '`.refines-jml`'.
- If there is no '`.java`' source file that will be available to the tools, the specify the type using a '`.spec`' or '`.jml`' file, without using a *refine-prefix*.

## 16.3 Refinement Chains

Compilation Units that jointly give the specifications of a type form a refinement chain. It begins at a base (or most-refined) compilation unit, proceeding by means of the `refine` annotation links, until a file is found that has no `refine` statement. That file is the end of the refinement chain and is the least-refined compilation unit.

For a given type in a given package, the base of the refinement chain is found as follows. Each entry of the classpath is searched in order for a directory whose name matches the package of the type and that contains a file whose name has a prefix matching the type name and a suffix that is an active suffix as defined above. The first such file found is the base of the refinement chain. If the first classpath entry to contain a candidate file contains more than one candidate file, then the file with the most active suffix is the base of the chain.

The subsequent elements of the refinement chain are given by the filenames provided in the `refine` statements. Each element of the chain is in the same package. Thus the file corresponding to the `refine` statement is the first file found by searching the classpath entries in order and that is in the directory corresponding to the package of the type and has the filename and suffix given in the `refine` statement.

To help ensure that the base is correctly selected, the file with the most active suffix must be the base of a refinement sequence, otherwise the JML typechecker issues an error message. Also, the prefix of the base file must be the same as the public type declared in that compilation unit or an error message is issued. However, it is not necessary that the file being refined have the same prefix as the file at the base of the refinement chain (except that the .java file, if it is in the refinement sequence, must have a name given by the Java rules for naming compilation units). Furthermore, a file with the same prefix as the base file may not be in a different refinement sequence. For example, '`SomeName.java-refined`' can be refined by '`MyType.java`' as long as there is no refinement sequence with '`SomeName`' as the prefix of the base of another refinement.

The JML tools deal with all files in a refinement chain whenever one of them is selected for processing by the tool. This allows all of the specifications that apply to be consistently dealt with at all times. For example, suppose that there are files named '`Foo.refines-java`' and '`Foo.java`', then if a tool selects the '`Foo.java`', e.g., with the command:

```
jmlc *.java
```

then it will see both the '`Foo.refines-java`' and the '`Foo.java`' file (as long as '`Foo.refines-java`' appears in a specification path directory before or with '`Foo.java`').

A given .java file (that is, compilation unit) may have more than one top-level class declaration within it. Only one may be public, and Java requires that the name of that type match the name of the file, so that the definition of the type can be found in the file system. The non-public types within that compilation unit may be referred to only within that compilation unit. Consequently, all specifications of those non-public types must occur along with the specifications of the public type in that compilation unit. For example, suppose a file '`A.java`' contains the Java declaration of types `A` and `B`. Then if the specifications of type `A` are in '`A.refines-java`', the specifications of type `B` must also be in '`A.refines-java`'. For simple one-file programs, the one compilation unit may contain only non-public types. Then the specifications for those types are found in specification files with the same prefix as the filename of the Java file containing the type declarations.

## 16.4 Type Checking Refinements

There are some restrictions on what can appear in the different files involved in a particular refinement. Since the Java compilers only see the '`.java`' files, executable code (that is not just for use in specifications) should only be placed in the '`.java`' files. In particular the following restrictions are enforced by JML.

- When the same method is declared in more than one file in a refinement sequence, most parts of the method declaration must be identical in all the files. (Two method declarations are considered to be declaring the *same method* if they have the same signature, i.e., same name, same generic type parameters, and static formal parameter types.) However, in addition to the signature of such a method, the return type, the names of the formal parameters, the declared exceptions the method may throw, and the non-JML modifiers `public`, `protected`, `private`, `static`, and `final`, must all match exactly in each such declaration in a refinement chain.

- The `model` modifier must appear in all declarations of a given method or it must appear in none of them. It is not permitted to implement a model method with a non-model method or to refine a non-model method with a model method. Use a `spec_public` or `spec_protected` method if you want to use a non-model method in a specification. Also, there may be no nesting of model declarations: model classes and model methods may not contain model or ghost declarations.
- Some of the JML method modifiers do not always have to match in all declarations of the same method in a refinement chain. One may add `pure`, `non_null`, `nullable`, `spec_public`, or `spec_protected` to any of the declarations for a method in any file. However, if `pure` is added to a method specification, then all subsequent declarations of that method in a refinement sequence must also be declared `pure`. Also, it is, of course, not permitted to add `spec_protected` to a method that has been declared `public` or `spec_public` in other declarations. One can add `non_null` or `nullable` to any formal parameter in any file, although good style suggests that all of these annotations appear on one declaration of that method.
- The specification of a refining method declaration must start with the JML keyword `also`; if it does not an error message is issued. A *refining method declaration* is a declaration that overrides a superclass method or refines the specification of the same method in a refinement chain. In JML, method specifications are inherited by subclasses and in refinement chains. The `also` keyword indicates that the current specification is refining the specification inherited either from the superclass or from the previous declaration of the method in a refinement sequence. Therefore, it is an error if the specification of a non-refining method begins with `also` (unless it overrides an inherited method).
- If a non-model method has a body, then the body can only appear in a '`.java`' file; an error message is issued if the body of a non-model method appears in a file with any other suffix. Furthermore, the body of a model method may only appear in one file of a refinement sequence. This means that each method of each class can have at most one method body.
- When the same field is declared in more than one file in a refinement sequence, then the signature of each such declaration must be identical in all the files. (Two field declarations are considered to be declaring the *same field* if they have the same name.) The signature of such a field, including its type, the non-JML modifiers `public`, `protected`, `private`, `static`, and `final`, must all match exactly in each such declaration.
- All declarations of a given field must either use the modifier `model` or not. It is not permitted to implement a model field with a non-model field or vice versa. Use a `spec_public` or `spec_protected` field if you want to use the same name. The same comment holds for `ghost` fields as well.
- Some of the JML field modifiers do not always have to match in all declarations of the

same field in a refinement chain. One may add `non_null`, `nullable`, `spec_public`, or `spec_protected` to any of the declarations for a field in any file. However, it is of course not permitted to add `spec_protected` to a field that has been declared public in other declarations.

- Initializers are not allowed in all field declarations. A non-model field can have an initializer expression but it can only appear in a '`.java`' file because this is where a compiler expects to find it.

  Fields declared using the `ghost` modifier can have an initializer expression in any file, but they may have at most one initializer expression in all the files.

  Model fields cannot have an initializer expression because there is no storage associated with such fields. Use the `initially` clause to specify the initial state of model fields (although the initial state is usually determined from the represents clause).
- Any number of *jml-var-assertion*'s [[[ what is this? the name must have changed - DRC]]] can be declared for any field declaration and these are all conjoined. For example, if a variable `int count` is declared and there are two `initially` clauses, in the same or different files, then these initially clause predicates are conjoined; that is, both must be satisfied initially.
- An initializer block or a static initializer block (with code) may only appear in a '`.java`' file. One can write annotations to specify the effects of such initializers in JML annotations in other files, using the keywords `initializer` and `static_initializer`.

JML uses specification inheritance to impose the specifications of supertypes on their subtypes [Dhara-Leavens96] [Leavens-Naumann06] [Leavens06b] to support the concept of behavioral subtyping [America87] [Leavens90] [Leavens91] [Leavens-Weihl90] [Leavens-Weihl95] [Liskov-Wing94]. JML also supports a notion of weak behavioral subtyping [Dhara-Leavens94b] [Dhara97].

## 16.5 Refinement Viewpoints

In refinements, specification inheritance allows the specifier to separate the public, protected, and private specifications into different files. Public specifications give the public behavior and are meant for clients of the class. Protected specifications are meant for programmers creating subclasses and give the protected behavior of the type; they give the behavior of protected methods and fields that are not visible to clients. Similarly, private specifications are meant for implementors of the class and provide the behavior related to private methods and fields of the class; implementors must satisfy the combined public, protected, and private specifications of a method.

[[[Needs work]]]

### 16.5.1 Default Constructor Refinement

In Java, a default constructor is automatically generated for a class when no constructors are declared in a class. However, in JML, a default constructor is not generated for a class unless the file suffix is '`.java`' (the same constructor is generated as in the Java language). Consider, for example, the refinement sequence defined by the following three files, `RefineDemo.jml-refined`, `RefineDemo.jml`, and `RefineDemo.java`.

```
// ---- file RefineDemo.jml-refined ---
```

```
package org.jmlspecs.samples.jmlrefman;

public class RefineDemo {
   //@ public model int x;

   /*@ public normal_behavior
     @    assignable x;
     @    ensures x == 0; @*/
   public RefineDemo();
}

// ---- file RefineDemo.jml -----------
package org.jmlspecs.samples.jmlrefman;

//@ refine "RefineDemo.jml-refined";

public class RefineDemo {
   protected int x_;
   //@           in x;

   //@ protected represents x <- x_;
}
// ---- file RefineDemo.java ---------
package org.jmlspecs.samples.jmlrefman;

//@ refine "RefineDemo.jml";

public class RefineDemo {
   protected int x_;
   public RefineDemo() { x_ = 0; }
}
```

In the protected specification declared in 'RefineDemo.jml', no constructor is defined. If JML were to generate a default constructor for this class declaration, then the public constructor defined earlier in the refinement chain, in 'RefineDemo.jml-refined', could have a visibility modifier that conflicts with the one automatically generated for the protected specification. (The visibility modifier of an automatically generated default constructor depends on other factors including the visibility of the class. See Section 9.4 [Lightweight Specification Cases], page 63, for more details.) Recall that the signature, including the visibility modifier, must match for every method and constructor declared in a refinement chain. To avoid such conflicts, JML does not generate a default constructor unless the file suffix is '.java' (as part of the standard compilation process).

A similar problem can occur when the only constructor is protected or private as in the refinement sequence defined by the following three files, RefineDemo2.jml-refined, RefineDemo2.jml, and RefineDemo2.java.

```
// ---- file RefineDemo2.jml-refined --
package org.jmlspecs.samples.jmlrefman;
```

```
public class RefineDemo2 {
   //@ public model int x;
   //@ public initially x == 0;
}

// ---- file RefineDemo2.jml ----------
package org.jmlspecs.samples.jmlrefman;

//@ refine "RefineDemo2.jml-refined";
public class RefineDemo2 {
   protected int x_;
   //@           in x;

   //@ protected represents x <- x_;

   /*@ protected normal_behavior
     @    assignable x;
     @    ensures x == 0; @*/
   protected RefineDemo2();
}

// ---- file RefineDemo2.java ---------
package org.jmlspecs.samples.jmlrefman;

//@ refine "RefineDemo2.jml";
public class RefineDemo2 {
   protected int x_;
   protected RefineDemo2() { x_ = 0; }
}
```

In this example, notice that no constructor is defined for the public specification in '`RefineDemo2.jml-refined`'. If a default constructor were generated for this class declaration, then the `protected` constructor defined later in the refinement chain, in '`RefineDemo2.jml`', would have a visibility modifier that conflicts with the one automatically generated and JML would emit an error. Thus JML only generates the default constructor for the executable declaration of a class in the '`.java`' file and only when required by the Java language.

# 17 MultiJava Extensions to JML

This section describes extensions to JML to support the MultiJava [Clifton-etal00] language. All of these extensions are optional and are only used when an option (or special tool) is used to parse this syntax.

The sections below explain the extensions that MultiJava makes to JML.

## 17.1 Augmenting Method Declarations

MultiJava has a feature, called "open classes" [Clifton-etal00] or "augmenting methods" that allows methods to be added to an existing class. It has the following syntax, which, in JML, permits method specifications.

*multijava-top-level-declaration* ::= *multijava-top-level-method*
*multijava-top-level-method* ::= [ *method-specification* ]
        *modifiers* [ `method` ]
        [ *type-spec* ] *extending-method-head method-body*
*extending-method-head* ::= *name* `. ident` *formals* [*dims* ]
                [ *throws-clause* ]

This syntax adds a method to the class named by the *name* in the *extending-method-head*.

The method must satisfy the given *method-specification*, if there is one.

## 17.2 MultiMethods

The other feature in MultiJava is multiple dispatch, which is used to define multimethods. Multiple dispatch is defined using the following syntax.

*multijava-param-declaration* ::= [ *param-modifier* ] . . .
        *type-spec specializer ident* [ *dims* ]
*specializer* ::= `@` *type-spec*
    `| @@` *value-specializer*
*value-specializer* ::= *expression*

See the MultiJava paper [Clifton-etal00] for how the use of a *specializer* affects the meaning of method calls.

# 18 Universe Type System

This section describes how the Universe type system [Dietl-Drossopoulou-Mueller07] [Dietl-Mueller05] [Dietl-Mueller-Schregenberger-08] [Mueller-Poetzsch-Heffter01a] is realized in JML and the impact it has on JML specifications. The Universe type system is a lightweight ownership type system that hierarchically structures the object store and confines the possible effects of expressions.

The syntax for the Universe type system consists of three ownership modifiers.

*ownership-modifiers* ::= *ownership-modifier* [ *ownership-modifier* ]
*ownership-modifier* ::= `\rep` | `\peer` | `\readonly`
  | *reserved-ownership-modifier // with –universesx parse or –universesx full*
*reserved-ownership-modifier* ::= `rep` | `peer` | `readonly`

Depending on the options selected, one can use either form of the modifiers, with or without the backslash, in annotations. The forms without the backslashes are the only ones that can be used in Java code, and when they are enabled, they are treated as new reserved words in both JML annotations and in Java code.

Currently the Universe type checking and the *reserved-ownership-modifier* syntax are not enabled by default in JML, but is only available when various options are used in the tools. It can also be used with different levels of checking. If the `--universesx no` option is used, only the *ownership-modifiers* `\rep`, `\peer`, and `\readonly` are available.

To enable just parsing of the full syntax, one can use the `--universesx parse` option; in this case, all of the syntax is parsed, and `rep`, `peer`, and `readonly` are treated as reserved words. However, with this option, none of the checking described below is done.

To enable checking, but without reserving the keywords `rep`, `peer`, and `readonly`, one uses the `--universesx check` option. With this option, only the *ownership-modifiers* `\rep`, `\peer`, and `\readonly` are available. This allows the use of ownership modifiers in specifications, but not in Java code.

Various other options control the generation of runtime checks and the storage of ownership modifiers in the created class files. See [Dietl-Mueller-Schregenberger08] for a complete list of the different supported compiler options.

One can also enable checking, all of the syntax, and default options by using the `--universesx full` option. An equivalent option is `--universes` (synonym `-e`). This parses and type checks all the *ownership-modifiers*, not only in specifications, but also in Java code.

For a simple reference type, one can use only one *ownership-modifier* where *ownership-modifiers* appears in the grammar. The only case where two *ownership-modifiers* can be used is for array types as described below.

Note that in [Dietl-Drossopoulou-Mueller07] the Universe type system is extended to type genericity as found in Java 5. The JML tools support Generic Universe Types and also recognize the `any` modifier as synonym for `readonly`. As the rest of this report is about non-generic Java, we refer to [Dietl-Drossopoulou-Mueller07] [Dietl-Mueller-Schregenberger08] for details.

In the sections below we just use the forms without the backslashes when discussing the semantics of each form.

## 18.1 Basic Concepts of Universes

The Universe type system organizes objects into ownership contexts [Dietl-Mueller05] [Mueller-Poetzsch-Heffter01a]. Each object has 0 or 1 *owner* objects. The owner of an object (or the absence of an owner) is determined by the `new` expression that creates the object. Once determined, the owner of an object cannot be changed.

An *ownership context* is a set of objects with the same owner. There is also a *root ownership context*, which is the set of all objects that have no owner. Each object thus belongs to exactly one ownership context. The contexts form a hierarchy, with the root ownership context at the top. The owner of an ownership context is not considered to be part of the context it owns, but rather part of that context's parent context.

The Universe type system enforces the "owner-as-modifier" property (see section 1 of [Dietl-Mueller05]). This property says "an object $X$ can be referenced by any other object, but reference chains that do not pass through $X$s owner must not be used to modify $X$" (section 1 of [Dietl-Mueller05]). Thus, if one looks at all the references from outside an ownership context into objects within the context, all of these references must be readonly references, with the exception of any references from the context's owner.

## 18.2 Rep and Peer

The `rep` and `peer` annotations are type modifiers (see Section 7.1.2.2 [Type-Specs], page 48) that specify ownership relative to a receiver object. The *receiver* object is defined as follows:

- For a field access of the form *E.f*, the receiver object is the result of the expression *E*.
- For a call to an instance method of the form *E.m(. . . )*, the receiver object is the result of the expression *E*.
- For all other expressions occurring in the declaration of an instance method or constructor (including the specification), or in an instance invariant or instance history constraint, the receiver object is `this`.
- For all other expressions in the declaration of a static method, there is no receiver object. In this case, the ownership modifier specifies ownership relative to the current ownership context, as explained below.

A `rep` modifier says that the referenced object is owned by the receiver object. Thus if `myList` has a field `head` of type `rep Node`, then `myList.head` is owned by `myList`, because `myList` is the receiver. If `n` is a local variable of type `rep Node` in an instance method, then `n` is owned by `this`. (Formal parameters are treated in exactly the same way as local variables.)

Since the meaning of the `rep` modifier depends on the existence of a receiver object, it cannot be used in static declarations where there is no receiver object. Hence, a `rep` modifier cannot be used in a static field declaration. It also cannot be used in the declaration of a static method or in its specification. Furthermore, it cannot be used in static invariants or static history constraints.

A `peer` modifier says that the referenced object has the same owner as the receiver object. Thus if `myNode` has a field `next` of type `peer Node`, then `myNode.next` is owned by the owner of `myNode`, because `myNode` is the receiver. If `n` is a local variable of type `peer Node` in an instance method, then `n` is owned by the owner of `this`.

The peer modifier can be used in all declarations, even in static declarations. Currently, a peer modifier in a static field declaration leads to type unsafety and should therefore not be used. (The tools give a warning in this situation, and a safe semantics is a subject of current research.) The same remark applies to static invariants and static history constraints.

When used in a static method or its specification, peer refers to the current ownership context. The *current ownership context* for a method execution is defined as follows. For executions of instance methods the current ownership context is the one containing the this object. For executions of static methods, the current ownership context is determined by the current ownership context of the caller and the ownership modifier (rep or peer) used in the call as follows:

- If the call has the form peer $T$.$m$(...), then $m$ executes in the same ownership context as the code making the call (and hence in the current ownership context of the caller).
- If the call has the form rep $T$.$m$(...), then $m$ executes in the ownership context owned by the caller's this object; hence this form of static method call cannot be used in static declarations.

For example, if p is a local variable of type peer Node in a static method, then p is in the current ownership context, because there is no receiver object.

See Section 18.4 [Ownership Modifiers for Array Types], page 134, for the usage of these modifiers with array types.

## 18.3 Readonly

The readonly (or \readonly) modifier does not specify an ownership context. Therefore, following the owner-as-modifier property, references specified with the readonly modifier cannot be used to modify the referenced object. (Note that this does not guarantee that the object referenced cannot change, only that it cannot be changed using this reference.)

A readonly type thus cannot be used as the type of the receiver expression of: a field update, a call to a non-pure instance method (See Section 7.1.1.3 [Pure Methods and Constructors], page 44, for more about pure methods.), or a call to a static method. In more detail, the cases are:

- A field update in general might change the value of the field and always needs to be forbidden on a readonly receiver.
- A (strictly) pure instance method call is guaranteed to preserve the owner-as-modifier property and is therefore allowed on a readonly receiver.
- A non-pure instance method call might change the receiver or objects reachable from it and needs to be forbidden.
- A static method can create new peer objects and therefore a specific current ownership context needs to be provided when a static method is called. Only peer and rep determine a current ownership context and therefore readonly is forbidden as the receiver type of a static method call.

## 18.4 Ownership Modifiers for Array Types

An array of reference types always has two ownership modifiers, the first for the array object itself and the second for the elements. Both modifiers express ownership relative to the receiver object and both modifiers can be any of the *ownership-modifiers*. For example, the type `rep readonly Object[]` says that the array object itself is owned by the receiver object, but the elements are readonly (and hence may belong to an arbitrary ownership context). A `peer rep Object[]` type says that the array object has the same owner as the receiver object and that the array elements are owned by the receiver object.

All array objects in a multidimensional array of a reference type are in the same context, which is determined by the first ownership modifier. For example, if an instance field, `f`, has type `rep peer Object[][]`, then `f` and `f[3]` are both owned by the receiver and `f[3][1]` has the same owner as the receiver object.

For one-dimensional arrays of primitive types, the second ownership modifier is omitted. Primitive types are not owned and do not take an ownership modifier. A one-dimensional array of primitive types is one object that needs to specify ownership information. For example, the type `readonly int[]` says that the array object can belong to any context, but cannot be modified through this reference. A `rep int[]` references an array object that is owned by the receiver object and that manages `int` values.

Multi-dimensional arrays of primitive types have two ownership modifiers, the first for the array object itself and the second for the one-dimensional array at the "lowest" level. All array objects in a multidimensional array are in the same context, which is determined by the first ownership modifier.

For example, if an instance field, `g`, has type `rep peer int[][][]`, then:

- `g` references a `rep peer int[][][]` array object that is owned by the receiver and the array manages `rep peer int[][]` references.
- `g[3]` references a `rep peer int[][]` array object that is owned by the receiver and the array manages `peer int[]` references.
- `g[3][1]` references a `peer int[]` array object that has the same owner as the receiver and the array manages `int` values.
- `g[3][1][0]` is an `int` value.

Note how the first modifier changes when going from a two- or more-dimensional array of a primitive type to a one-dimensional array of a primitive type.

Also note that `java.lang.Object` is a supertype of arrays, in particular also of arrays of primitive type. A `peer int[]` can be assigned to a `peer Object` reference. Then a `rep peer Object[][]` type behaves consistently with the `rep peer int[][][]` type.

Following the convention in Java, array types support covariant subtyping that needs runtime checks on write accesses. For example, a `peer rep Object[]` is a subtype of a `peer readonly Object[]` and when an element is inserted it needs to be checked that it is owned by the receiver object.

## 18.5 Default Ownership Modifiers

If the *ownership-modifiers* are omitted in a *type-spec*, then a default is used. This default is normally `peer`, but there are a few exceptions, described below.

- The ownership modifier of immutable types defaults to `readonly`. Currently, the set of immutable types only includes the Java wrapper types for primitive types (e.g. `java.lang.Integer` and `java.lang.Long`), `java.lang.String`, `java.lang.Class`, and `java.math.BigInteger`.

- The ownership modifiers of local variable declarations are propagated from the initializer expression. If no initializer is present, the other defaults are applied.
- The ownership modifiers of field declarations are propagated from the initializer expression. If no initializer is present, the other defaults are applied. If a field type was already used to determine the ownership modifier of some other field, i.e. it was used in the initializer expression of some other field, then the type cannot be changed any more and the other defaults are used.
- The default modifier for explicit formal parameters to a `pure` method (but not for the receiver, `this`) is `readonly`. (Note that this is not the case for pure constructors, however.)
- The default ownership modifier for a type in the `throws` clause of a method header, and in the declaration of a `catch` clause of a `try` statement is `readonly` [Dietl-Mueller04].
- If, for a type that is an array of references, one of the two ownership modifiers is omitted, then the element type is used to determine the meaning of the ownership modifier. If the element type is a mutable type, then the specified modifier is taken to be the element modifier, and the array's modifier defaults to `peer`. If the element type is an immutable type, then the specified modifier is taken to be the array modifier, and the element modifier defaults to `readonly`.

  For example, the type `readonly Object[]` is the same as `peer readonly Object[]`. A type `rep Integer[]` is the same as `rep readonly Integer[]`. Note that if one wants to specify a `rep` or `readonly` array of mutable references, one is thus forced to use two ownership modifiers; for example, `rep readonly Object[]`.

  One-dimensional arrays of primitive types default to `peer`. For multi-dimensional arrays of primitive types there is no distinction between immutable and mutable types and a single ownership modifier is always taken to be the element modifier.
- In a cast expression of the form `(`$T$`)`$E$, where $T$ is a reference type that is not an array type, the default ownership modifier of $T$ is the ownership modifier of the type of $E$; in this case, if the type of $E$ is an array type, this is the ownership modifier of the array object itself, not the ownership modifier of the elements.

  In a cast expression of the form `(`$T$`)`$E$, where $T$ is an array type, the default ownership modifiers of $T$ are the same as the ownership modifiers of the type of $E$.

  In a cast expression of the form `(`$T$`)`$E$, where $T$ is a primitive value type, there is no ownership modifier attached to $T$.
- In an `instanceof` expression of the form $E$ `instanceof` $T$, where $T$ is a reference type that is not an array type, the default ownership modifier of $T$ is the ownership modifier of the type of $E$; in this case, if the type of $E$ is an array type, this is the ownership modifier of the array object itself, not the ownership modifier of the elements.

  In an `instanceof` expression of the form $E$ `instanceof` $T$, where $T$ is an array type, the default ownership modifiers of $T$ are the same as the ownership modifiers of the type of $E$.

The defaults for casts and instanceof expressions allow one to only test for Java types, if the ownership modifiers are omitted [Dietl-Mueller05]. See Section 18.7 [Casts and Ownership Types], page 138, for more details on these expressions and their interaction with the Universe type system.

## 18.6 Ownership Type Rules

This section explains details of how the Universe type system does type checking.

### 18.6.1 Ownership Subtyping

Type checking in the Universe type system uses a notion of subtyping that extends Java's rules to take *ownership-modifiers* into account (see section 3 of [Dietl-Mueller05]).

If two types have the same ownership modifiers, then they are subtypes if the underlying Java types are subtypes. For example, `rep Stack` is a subtype of `rep Object`, because `Stack` is a subtype of `Object`.

If *S* is a reference type, then both `peer` *S* and `rep` *S* are subtypes of the type `readonly` *S*. Moreover, both `peer` *om* *S*`[]` and `rep` *om* *S*`[]` are subtypes of the type `readonly` *om* *S*`[]`, where *om* is any ownership modifier. For instance, `peer peer Natural[]` is a subtype of `readonly peer Natural[]`.

The types `peer` *S* and `rep` *S* as well as the array types `peer` *om* *S*`[]` and `rep` *om* *S*`[]` are incomparable—neither is a subtype of the other.

Like Java, the Universe type system has covariant array subtyping: "two array types with the same ownership modifier are subtypes if their element types are subtypes. . . . For instance, `rep peer Object[]` is a subtype of `rep readonly Object[]` because the element type `peer Object` is a subtype of the element type `readonly Object`" (Section 3 of [Dietl-Mueller05]).

### 18.6.2 Ownership Typing for Expressions

Most of the typing rules for the Universe type system are unchanged from standard Java (and JML) rules. For example, to type check an assignment expression, one checks that the type of the right hand side expression is a subtype of the type of the left hand side.

A small, but important change, is that the type given in a `new` expression must be a `rep` or `peer` type. The result type of the `new` expression has the given ownership modifier.

The main difference is that the type of field accesses, method parameters, and method results is determined by combining the type of the receiver, *R*, and the type of the field, the return type of the method, or the type of the formal parameter, *F*. The Java type is taken from the type *F*, and the modifier is determined by the following cases (see Section 3 of [Dietl-Mueller05]):

1. If both *R* and *F* are `peer` types, then the combination is also a `peer` type. For example, if `myList` has type `peer List` and the field `head` has type `peer Node`, then `myList.head` has type `peer Node`.
2. If the receiver is `this` and *F* is a `rep` type, then the combination is a `rep` type. For example, if a `Set` class has an instance field `elems` of type `rep List`, then in its instance methods, `this.elems` has type `rep List`.
3. If *R* is a `rep` type and *F* is a `peer` type, then the combination is a `rep` type. For example, `(this.elems).head` has type `rep Node`, because the receiver `this.elems` has type `rep List`, and the type of field `head` is `peer Node`.
4. Otherwise, the combination is a `readonly` type. For example, if `e` has type `readonly List`, then `e.head` has type `readonly Node`.

One can also illustrate these rules using method calls. For example, consider a method `lastNode` with the following signature.

```
public peer Node lastNode()
```

In this example, if `elems` has type `rep List`, then a call such as `elems.lastNode()` has type `rep Node` (by case 3).

As another example, consider a method `addNode` with the following signature.

```
public void addNode(peer Node n)
```

Still assuming that `elems` has type `rep List`, a call such as `elems.addNode(p)`, requires that `p` has type `rep Node` (also by case 3), because the argument, `p`, has to have the same owner as the receiver of call, `elems`, namely `this`.

The rules are analogous for arrays. For example, suppose that an instance field `a` has type `rep readonly Object[]`. Then the expression `this.a` has the same type, `rep readonly Object[]` (by case 2). Similarly, if `r` has a `readonly` type, then `r.a` would have type `readonly readonly Object[]` (by case 4).

Finally, consider a static method that returns a `peer` object, such as the following, in a class `Cache`.

```
public static peer int[] getInstance()
```

A call such as `peer Cache.getInstance()` has type `peer int[]` (by case 1).

## 18.7 Casts and Ownership Types

Since `readonly` types are supertypes of the corresponding `rep` and `peer` types, it is possible to do a downcast. Such a downcast will succeed when the object is in the context specified by the peer or rep type. For example, suppose `ro` has type `readonly List`. Then the cast `(rep List) ro` will succeed only if the object referenced by `ro` is owned by `this`. The cast `(peer List) ro` will succeed only if the object referenced by `ro` is owned by the owner of `this`.

Instanceof expressions of the form $E$ `instanceof` $T$ yield true when the value of $E$ is not `null` and the corresponding cast would succeed. For example, suppose `ro` has type `readonly List`. Then `ro instanceof rep List` yields true only if `ro` references an object that is owned by `this`.

Both casts and instanceof expressions have runtime overhead, in general. (Furthermore, as in Java, array updates also generate runtime checks.)

See [Dietl-Drossopoulou-Mueller07] [Dietl-Mueller-Schregenberger08] for a complete list of the Universe type system rules and the different supported compiler options.

# 19 Safe Math Extensions

## 19.1 \bigint

[[[ needs discussion ]]]

## 19.2 \real

[[[ needs discussion ]]]

# 20 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.

## 20.1 Deprecated Syntax

The following syntax is deprecated.

## 20.2 Replaced Syntax

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 A Grammar Summary

The following is a summary of the context-free grammar for JML. See Chapter 3 [Syntax Notation], page 25, for the notation used. In the first section below, grammatical productions are to be understood lexically. That is, no white space (see Section 4.1 [White Space], page 26) may intervene between the characters of a token.

## A.1 Lexical Conventions

*microsyntax* ::= *lexeme* [ *lexeme* ] . . .
*lexeme* ::= *white-space* | *lexical-pragma* | *comment*
    | *annotation-marker* | *doc-comment* | *token*
*token* ::= *ident* | *keyword* | *special-symbol*
    | *java-literal* | *informal-description*
*white-space* ::= *non-nl-white-space* | *end-of-line*
*non-nl-white-space* ::= a blank, tab, or formfeed character
*end-of-line* ::= *newline* | *carriage-return*
    | *carriage-return newline*
*newline* ::= a newline character
*carriage-return* ::= a carriage return character
*lexical-pragma* ::= *nowarn-pragma*
*nowarn-pragma* ::= `nowarn` [ *spaces* ] [ *nowarn-label-list* ] `;`
*spaces* ::= *non-nl-white-space* [ *non-nl-white-space* ] . . .
*nowarn-label-list* ::= *nowarn-label* [ *spaces* ]
        [ `,` [ *spaces* ] *nowarn-label* [ *spaces* ] ] . . .
*nowarn-label* ::= *letter* [ *letter* ] . . .
*comment* ::= *C-style-comment* | *C++-style-comment*
*C-style-comment* ::= `/*` [ *C-style-body* ] *C-style-end*
*C-style-body* ::= *non-at-plus-star* [ *non-stars-slash* ] . . .
    | `+` *non-at* [ *non-stars-slash* ] . . .
    | *stars-non-slash* [ *non-stars-slash* ] . . .
*non-stars-slash* ::= *non-star*
    | *stars-non-slash*
*stars-non-slash* ::= `*` [ `*` ] . . . *non-star-slash*
*non-at-plus-star* ::= any character except `@`, `+`, or `*`
*non-at* ::= any character except `@`
*non-star* ::= any character except `*`
*non-slash* ::= any character except `/`
*non-star-slash* ::= any character except `*` or `/`
*C-style-end* ::= [ `*` ] . . . `*/`
*C++-style-comment* ::= `//` [ `+` ] *end-of-line*
    | `//` *non-at-plus-end-of-line* [ *non-end-of-line* ] . . . *end-of-line*
    | `//+` *non-at-end-of-line* [ *non-end-of-line* ] . . . *end-of-line*
*non-end-of-line* ::= any character except a newline or carriage return
*non-at-plus-end-of-line* ::= any character except `@`, `+`, newline, or carriage return
*non-at-end-of-line* ::= any character except `@`, newline, or carriage return

*annotation-marker* ::= `//@` [ `@` ] . . . | `//+@` [ `@` ] . . .
    | `/*@` [ `@` ] . . . | `/*+@` [ `@` ] . . . | [ `@` ] . . . `@+*/` | [ `@` ] . . . `*/`
*ignored-at-in-annotation* ::= `@`
*doc-comment* ::= `/**` [ `*` ] . . . *doc-comment-body* `*/`
*doc-comment-ignored* ::= *doc-comment*
*doc-comment-body* ::= [ *description* ] . . .
    [ *tagged-paragraph* ] . . .
    [ *jml-specs* ] [ *description* ]
*description* ::= *doc-non-empty-textline*
*tagged-paragraph* ::= *paragraph-tag* [ *doc-non-nl-ws* ] . . .
    [ *doc-atsign* ] . . . [ *description* ] . . .
*jml-specs* ::= *jml-tag* [ *method-specification* ] *end-jml-tag*
    [ *jml-tag* [ *method-specification* ] *end-jml-tag* ] . . .
*paragraph-tag* ::= `@author` | `@deprecated` | `@exception`
    | `@param` | `@return` | `@see`
    | `@serial` | `@serialdata` | `@serialfield`
    | `@since` | `@throws` | `@version`
    | `@` *letter* [ *letter* ] . . .
*doc-atsign* ::= `@`
*doc-nl-ws* ::= *end-of-line*
    [ *doc-non-nl-ws* ] . . . [ `*` [ `*` ] . . . [ *doc-non-nl-ws* ] . . . ]
*doc-non-nl-ws* ::= *non-nl-white-space*
*doc-non-empty-textline* ::= *non-at-end-of-line* [ *non-end-of-line* ] . . .
*jml-tag* ::= <jml> | <JML> | <esc> | <ESC>
*end-jml-tag* ::= </jml> | </JML> | </esc> | </ESC>
*ident* ::= *letter* [ *letter-or-digit* ] . . .
*letter* ::= `_`, `$`, `a` through `z`, or `A` through `Z`
*digit* ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
*letter-or-digit* ::= *letter* | *digit*
*keyword* ::= *java-reserved-word*
    | *jml-predicate-keyword* | *jml-keyword*
*java-reserved-word* ::= `abstract` | `assert`
    | `boolean` | `break` | `byte`
    | `case` | `catch` | `char`
    | `class` | `const` | `continue`
    | `default` | `do` | `double`
    | `else` | `extends` | `false`
    | `final` | `finally` | `float`
    | `for` | `goto` | `if`
    | `implements` | `import` | `instanceof`
    | `int` | `interface` | `long`
    | `native` | `new` | `null`
    | `package` | `private` | `protected`
    | `public` | `return` | `short`
    | `static` | `strictfp` | `super`
    | `switch` | `synchronized` | `this`
    | `throw` | `throws` | `transient`

| `true` | `try` | `void`
| `volatile` | `while`
| *multijava-reserved* // *When the MultiJava option is on*
| *java-universe-reserved* // *When the Universe option is on*

*multijava-reserved* ::= `resend`

*java-universe-reserved* ::= `peer` | `pure`
| `readonly` | `rep`

*jml-predicate-keyword* ::= `\TYPE`
| `\bigint` | `\bigint_math` | `\duration`
| `\elemtype` | `\everything` | `\exists`
| `\forall` | `\fresh`
| `\into` | `\invariant_for` | `\is_initialized`
| `\java_math` | `\lblneg` | `\lblpos`
| `\lockset` | `\max` | `\min`
| `\nonnullelements` | `\not_assigned`
| `\not_modified` | `\not_specified`
| `\nothing` | `\nowarn` | `\nowarn_op`
| `\num_of` | `\old` | `\only_accessed`
| `\only_assigned` | `\only_called`
| `\only_captured` | `\pre`
| `\product` | `\reach` | `\real`
| `\result` | `\same` | `\safe_math`
| `\space` | `\such_that` | `\sum`
| `\typeof` | `\type` | `\warn_op`
| `\warn` | `\working_space`
| *jml-universe-pkeyword*

*jml-universe-pkeyword* ::= `\peer` | `\readonly` | `\rep`

*jml-keyword* ::= `abrupt_behavior` | `abrupt_behaviour`
| `accessible` | `accessible_redundantly`
| `also` | `assert_redundantly`
| `assignable` | `assignable_redundantly`
| `assume` | `assume_redundantly` | `axiom`
| `behavior` | `behaviour`
| `breaks` | `breaks_redundantly`
| `callable` | `callable_redundantly`
| `captures` | `captures_redundantly`
| `choose` | `choose_if`
| `code` | `code_bigint_math` |
| `code_java_math` | `code_safe_math`
| `constraint` | `constraint_redundantly`
| `constructor` | `continues` | `continues_redundantly`
| `decreases` | `decreases_redundantly`
| `decreasing` | `decreasing_redundantly`
| `diverges` | `diverges_redundantly`
| `duration` | `duration_redundantly`
| `ensures` | `ensures_redundantly` | `example`
| `exceptional_behavior` | `exceptional_behaviour`

| exceptional_example
| exsures | exsures_redundantly | extract
| field | forall
| for_example | ghost
| helper | hence_by | hence_by_redundantly
| implies_that | in | in_redundantly
| initializer | initially | instance
| invariant | invariant_redundantly
| loop_invariant | loop_invariant_redundantly
| maintaining | maintaining_redundantly
| maps | maps_redundantly
| measured_by | measured_by_redundantly
| method | model | model_program
| modifiable | modifiable_redundantly
| modifies | modifies_redundantly
| monitored | monitors_for | non_null
| normal_behavior | normal_behaviour
| normal_example | nowarn
| nullable | nullable_by_default
| old | or
| post | post_redundantly
| pre | pre_redundantly
| pure | readable
| refine | refines | refining
| represents | represents_redundantly
| requires | requires_redundantly
| returns | returns_redundantly
| set | signals | signals_only
| signals_only_redundantly | signals_redundantly
| spec_bigint_math | spec_java_math
| spec_protected | spec_public | spec_safe_math
| static_initializer | uninitialized
| unreachable | weakly
| when | when_redundantly
| working_space | working_space_redundantly
| writable
| *jml-universe-keyword*
*jml-universe-keyword* ::= peer | readonly | rep
*special-symbol* ::= *java-special-symbol* | *jml-special-symbol*
*java-special-symbol* ::= *java-separator* | *java-operator*
*java-separator* ::= ( | ) | { | } | '[' | ']' | ; | , | .
| *multijava-separator* // *When the MultiJava option is on*
*multijava-separator* ::= @ | @@
*java-operator* ::= = | < | > | ! | ~ | ? | :
| == | <= | >= | != | && | '||' | ++ | --
| + | - | * | / | & | '|' | ^ | % | << | >> | >>>
| += | -= | *= | /= | &= | '|=' | ^= | %=

| <<= | >>= | >>>=
*jml-special-symbol* ::= ==> | <== | <==> | <=!=>
| -> | <- | <: | .. | '{|' | '|}'
*java-literal* ::= *integer-literal*
| *floating-point-literal* | *boolean-literal*
| *character-literal* | *string-literal* | *null-literal*
*integer-literal* ::= *decimal-integer-literal*
| *hex-integer-literal* | *octal-integer-literal*
*decimal-integer-literal* ::= *decimal-numeral* [ *integer-type-suffix* ]
*decimal-numeral* ::= 0 | *non-zero-digit* [ *digits* ]
*digits* ::= *digit* [ *digit* ] . . .
*digit* ::= 0 | *non-zero-digit*
*non-zero-digit* ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
*integer-type-suffix* ::= l | L
*hex-integer-literal* ::= *hex-numeral* [ *integer-type-suffix* ]
*hex-numeral* ::= 0x *hex-digit* [ *hex-digit* ] . . .
| 0X *hex-digit* [ *hex-digit* ] . . .
*hex-digit* ::= *digit* | a | b | c | d | e | f
| A | B | C | D | E | F
*octal-integer-literal* ::= *octal-numeral* [ *integer-type-suffix* ]
*octal-numeral* ::= 0 *octal-digit* [ *octal-digit* ] . . .
*octal-digit* ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
*floating-point-literal* ::= *digits* . [ *digits* ]
[ *exponent-part* ] [ *float-type-suffix* ]
| . *digits* [ *exponent-part* ] [ *float-type-suffix* ]
| *digits exponent-part* [ *float-type-suffix* ]
| *digits* [ *exponent-part* ] *float-type-suffix*
*exponent-part* ::= *exponent-indicator signed-integer*
*exponent-indicator* ::= e | E
*signed-integer* ::= [ *sign* ] *digits*
*sign* ::= + | -
*float-type-suffix* ::= f | F | d | D
*boolean-literal* ::= true | false
*character-literal* ::= ' *single-character* ' | ' *escape-sequence* '
*single-character* ::= any character except ', \, carriage return, or newline
*escape-sequence* ::= \b // *backspace*
| \t // *tab*
| \n // *newline*
| \r // *carriage return*
| \' // *single quote*
| \" // *double quote*
| \\ // *backslash*
| *octal-escape*
| *unicode-escape*
*octal-escape* ::= \ *octal-digit* [ *octal-digit* ]
| \ *zero-to-three octal-digit octal-digit*
*zero-to-three* ::= 0 | 1 | 2 | 3

*unicode-escape* ::= `\u` *hex-digit hex-digit hex-digit hex-digit*
*string-literal* ::= `"` [ *string-character* ] . . . `"`
*string-character* ::= *escape-sequence*
    `|` any character except `"`, `\`, carriage return, or newline
*null-literal* ::= `null`
*informal-description* ::= `(*` *non-stars-close* [ *non-stars-close* ] . . . `*)`
*non-stars-close* ::= *non-star*
    `|` *stars-non-close*
*stars-non-close* ::= `*` [ `*` ] . . . *non-star-close*
*non-star-close* ::= any character except `)` or `*`

## A.2 Compilation Units

*compilation-unit* ::= [ *package-definition* ]
    [ *refine-prefix* ]
    [ *import-definition* ] . . .
    [ *top-level-definition* ] . . .
*top-level-definition* ::= *type-definition*
    `|` *multijava-top-level-declaration* // *When parsing MultiJava*
*package-definition* ::= `package` *name* `;`
*name* ::= *ident* [ `.` *ident* ] . . .
*import-definition* ::= [ `model` ] `import` *name-star* `;`
*name-star* ::= *ident* [ `.` *ident* ] . . . [ `. *` ]

## A.3 Type Definitions

*type-definition* ::= *class-definition*
    `|` *interface-definition*
    `|` `;`
*class-definition* ::= [ *doc-comment* ] *modifiers* `class` *ident*
    [ *class-extends-clause* ] [ *implements-clause* ]
    *class-block*
*class-block* ::= `{` [ *field* ] . . . `}`
*interface-definition* ::= [ *doc-comment* ] *modifiers* `interface` *ident*
    [ *interface-extends* ]
    *class-block*
*class-extends-clause* ::= [ `extends` *name* [ `weakly` ] ]
*implements-clause* ::= `implements` *name-weakly-list*
*name-weakly-list* ::= *name* [ `weakly` ] [ `,` *name* [ `weakly` ] ] . . .
*interface-extends* ::= `extends` *name-weakly-list*
*modifiers* ::= [ *modifier* ] . . .
*modifier* ::= `public` `|` `protected` `|` `private`
    `|` `abstract` `|` `static` `|`
    `|` `final` `|` `synchronized`
    `|` `transient` `|` `volatile`
    `|` `native` `|` `strictfp`

```
      | const          // reserved but not used in Java
      | jml-modifier
jml-modifier ::= spec_public | spec_protected
      | model | ghost | pure
      | instance | helper
      | uninitialized
      | spec_java_math | spec_safe_math | spec_bigint_math
      | code_java_math | code_safe_math | code_bigint_math
      | non_null | nullable | nullable_by_default
      | extract
```

## A.4 Class and Interface Member Declarations

```
field ::= member-decl
      | jml-declaration
      | class-initializer-decl
      | ;
member-decl ::= method-decl
      | variable-definition
      | class-definition
      | interface-definition
method-decl ::= [ doc-comment ] . . .
            method-specification
            modifiers [ method-or-constructor-keyword ]
            [ type-spec ] method-head
            method-body
      | [ doc-comment ] . . .
        modifiers method-or-constructor-keyword
        [ type-spec ] method-head
        [ method-specification ]
        method-body
method-or-constructor-keyword ::= method | constructor
method-head ::= ident formals [ dims ] [ throws-clause ]
method-body ::= compound-statement | ;
throws-clause ::= throws name [ , name ] . . .
formals ::= ( [ param-declaration-list ] )
param-declaration-list ::= param-declaration
                    [ , param-declaration ] . . .
param-declaration ::= [ param-modifier ] . . . type-spec ident [ dims ]
      | multijava-param-declaration   // When MultiJava parsing is on
param-modifier ::= final | non_null | nullable
variable-definition ::= [ doc-comment ] . . . modifiers variable-decls
variable-decls ::= [ field ] type-spec variable-declarators ;
              [ jml-data-group-clause ] . . .
variable-declarators ::= variable-declarator
                    [ , variable-declarator ] . . .
```

*variable-declarator* ::= *ident* [ *dims* ] [ `=` *initializer* ]
*initializer* ::= *expression* `|` *array-initializer*
*array-initializer* ::= `{` [ *initializer-list* ] `}`
*initializer-list* ::= *initializer* [ `,` *initializer* ] . . . [ `,` ]
*type-spec* ::= [ *ownership-modifiers* ] *type* [ *dims* ]
  `|` `\TYPE` [ *dims* ]
*type* ::= *reference-type* `|` *built-in-type*
*reference-type* ::= *name*
*dims* ::= '`[`' '`]`' [ '`[`' '`]`' ] . . .
*class-initializer-decl* ::= [ *method-specification* ]
  [ `static` ] *compound-statement*
  `|` *method-specification* `static_initializer`
  `|` *method-specification* `initializer`

## A.5 Type Specifications

*jml-declaration* ::= *modifiers invariant*
  `|` *modifiers history-constraint*
  `|` *modifiers represents-clause*
  `|` *modifiers initially-clause*
  `|` *modifiers monitors-for-clause*
  `|` *modifiers readable-if-clause*
  `|` *modifiers writable-if-clause*
  `|` *axiom-clause*
*invariant* ::= *invariant-keyword predicate* `;`
*invariant-keyword* ::= `invariant` `|` `invariant_redundantly`
*history-constraint* ::= *constraint-keyword predicate*
  [ `for` *constrained-list* ] `;`
*constraint-keyword* ::= `constraint` `|` `constraint_redundantly`
*constrained-list* ::= *method-name-list* `|` `\everything`
*method-name-list* ::= *method-name* [ `,` *method-name* ] . . .
*method-name* ::= *method-ref* [ `(` [ *param-disambig-list* ] `)` ] `|` *method-ref-start* `. *`
*method-ref* ::= *method-ref-start* [ `.` *method-ref-rest* ] . . .
  `|` `new` *reference-type*
*method-ref-start* ::= `super` `|` `this` `|` *ident*
*method-ref-rest* ::= `this` `|` *ident*
*param-disambig-list* ::= *param-disambig* [ `,` *param-disambig* ] . . .
*param-disambig* ::= *type-spec* [ *ident* [ *dims* ] ]
*represents-clause* ::= *represents-keyword store-ref-expression*
  *l-arrow-or-eq spec-expression* `;`
  `|` *represents-keyword store-ref-expression* `\such_that`
  *predicate* `;`
*represents-keyword* ::= `represents` `|` `represents_redundantly`
*l-arrow-or-eq* ::= `<-` `|` `=`
*initially-clause* ::= `initially` *predicate* `;`
*axiom-clause* ::= `axiom` *predicate* `;`

*readable-if-clause* ::= `readable` *ident* `if` *predicate* `;`
*writable-if-clause* ::= `writable` *ident* `if` *predicate* `;`
*monitors-for-clause* ::= `monitors_for` *ident*
*l-arrow-or-eq* *spec-expression-list* `;`

## A.6 Method Specifications

*method-specification* ::= *specification* | *extending-specification*
*extending-specification* ::= `also` *specification*
*specification* ::= *spec-case-seq* [ *redundant-spec* ]
| *redundant-spec*
*spec-case-seq* ::= *spec-case* [ `also` *spec-case* ] . . .
*spec-case* ::= *lightweight-spec-case* | *heavyweight-spec-case*
| *model-program*
*privacy* ::= `public` | `protected` | `private`
*lightweight-spec-case* ::= *generic-spec-case*
*generic-spec-case* ::= [ *spec-var-decls* ]
*spec-header*
[ *generic-spec-body* ]
| [ *spec-var-decls* ]
*generic-spec-body*
*generic-spec-body* ::= *simple-spec-body*
| `{|` *generic-spec-case-seq* `|}`
*generic-spec-case-seq* ::= *generic-spec-case*
[ `also` *generic-spec-case* ] . . .
*spec-header* ::= *requires-clause* [ *requires-clause* ] . . .
*simple-spec-body* ::= *simple-spec-body-clause*
[ *simple-spec-body-clause* ] . . .
*simple-spec-body-clause* ::= *diverges-clause*
| *assignable-clause* | *captures-clause*
| *when-clause* | *working-space-clause*
| *duration-clause* | *ensures-clause*
| *signals-only-clause* | *signals-clause*
*heavyweight-spec-case* ::= *behavior-spec-case*
| *exceptional-behavior-spec-case*
| *normal-behavior-spec-case*
*behavior-spec-case* ::= [ *privacy* ] [ `code` ] *behavior-keyword*
*generic-spec-case*
*behavior-keyword* ::= `behavior` | `behaviour`
*normal-behavior-spec-case* ::= [ *privacy* ] [ `code` ] *normal-behavior-keyword*
*normal-spec-case*
*normal-behavior-keyword* ::= `normal_behavior` | `normal_behaviour`
*normal-spec-case* ::= *generic-spec-case*
*exceptional-behavior-spec-case* ::= [ *privacy* ] [ `code` ] `exceptional-behavior-keyword`
*exceptional-spec-case*
*exceptional-behavior-keyword* ::= `exceptional_behavior` | `exceptional_behaviour`

*exceptional-spec-case* ::= *generic-spec-case*
*spec-var-decls* ::= *forall-var-decls* [ *old-var-decls* ]
  | *old-var-decls*
*forall-var-decls* ::= *forall-var-declarator* [ *forall-var-declarator* ] . . .
*forall-var-declarator* ::= `forall` [ *bound-var-modifiers* ] *quantified-var-declarator* `;`
*old-var-decls* ::= *old-var-declarator* [ *old-var-declarator* ] . . .
*old-var-declarator* ::= `old` [ *bound-var-modifiers* ] *type-spec spec-variable-declarators* `;`
*requires-clause* ::= *requires-keyword pred-or-not* `;`
  | *requires-keyword* `\same ;`
*requires-keyword* ::= `requires` | `pre`
  | `requires_redundantly` | `pre_redundantly`
*pred-or-not* ::= *predicate* | `\not_specified`
*ensures-clause* ::= *ensures-keyword pred-or-not* `;`
*ensures-keyword* ::= `ensures` | `post`
  | `ensures_redundantly` | `post_redundantly`
*signals-clause* ::= *signals-keyword* `(` *reference-type* [ *ident* ] `)`
    [ *pred-or-not* ] `;`
*signals-keyword* ::= `signals` | `signals_redundantly`
  | `exsures` | `exsures_redundantly`
*signals-only-clause* ::= *signals-only-keyword reference-type* [ `,` *reference-type* ] . . . `;`
  | *signals-only-keyword* `\nothing ;`
*signals-only-keyword* ::= `signals_only` | `signals_only_redundantly`
*diverges-clause* ::= *diverges-keyword pred-or-not* `;`
*diverges-keyword* ::= `diverges` | `diverges_redundantly`
*when-clause* ::= *when-keyword pred-or-not* `;`
*when-keyword* ::= `when` | `when_redundantly`
*assignable-clause* ::= *assignable-keyword store-ref-list* `;`
*assignable-keyword* ::= `assignable` | `assignable_redundantly`
  | `modifiable` | `modifiable_redundantly`
  | `modifies` | `modifies_redundantly`
*accessible-clause* ::= *accessible-keyword store-ref-list* `;`
*accessible-keyword* ::= `accessible` | `accessible_redundantly`
*callable-clause* ::= *callable-keyword callable-methods-list* `;`
*callable-keyword* ::= `callable` | `callable_redundantly`
*callable-methods-list* ::= *method-name-list* | *store-ref-keyword*
*measured-clause* ::= *measured-by-keyword* `\not_specified ;`
  | *measured-by-keyword spec-expression* [ `if` *predicate* ] `;`
*measured-by-keyword* ::= `measured_by` | `measured_by_redundantly`
*captures-clause* ::= *captures-keyword store-ref-list* `;`
*captures-keyword* ::= `captures` | `captures_redundantly`
*working-space-clause* ::= *working-space-keyword* `\not_specified ;`
  | *working-space-keyword spec-expression* [ `if` *predicate* ] `;`
*working-space-keyword* ::= `working_space` | `working_space_redundantly`
*duration-clause* ::= *duration-keyword* `\not_specified ;`
  | *duration-keyword spec-expression* [ `if` *predicate* ] `;`
*duration-keyword* ::= `duration` | `duration_redundantly`

## A.7 Data Groups

*jml-data-group-clause* ::= *in-group-clause* `|` *maps-into-clause*
*in-group-clause* ::= *in-keyword group-list* `;`
*in-keyword* ::= `in` `|` `in_redundantly`
*group-list* ::= *group-name* [ `,` *group-name* ] . . .
*group-name* ::= [ *group-name-prefix* ] *ident*
*group-name-prefix* ::= `super .` `|` `this .`
*maps-into-clause* ::= *maps-keyword member-field-ref* `\into` *group-list* `;`
*maps-keyword* ::= `maps` `|` `maps_redundantly`
*member-field-ref* ::= *ident* `.` *maps-member-ref-expr*
  `|` *maps-array-ref-expr* [ `.` *maps-member-ref-expr* ]
*maps-member-ref-expr* ::= *ident* `|` `*`
*maps-array-ref-expr* ::= *ident maps-spec-array-dim*
  [ *maps-spec-array-dim* ] . . .
*maps-spec-array-dim* ::= '`[`' *spec-array-ref-expr* '`]`'

## A.8 Predicates and Specification Expressions

*predicate* ::= *spec-expression*
*spec-expression-list* ::= *spec-expression*
  [ `,` *spec-expression* ] . . .
*spec-expression* ::= *expression*
*expression-list* ::= *expression* [ `,` *expression* ] . . .
*expression* ::= *assignment-expr*
*assignment-expr* ::= *conditional-expr*
  [ *assignment-op assignment-expr* ]
*assignment-op* ::= `=` `|` `+=` `|` `-=` `|` `*=` `|` `/=` `|` `%=` `|` `>>=`
  `|` `>>>=` `|` `<<=` `|` `&=` `|` '`|=`' `|` `^=`
*conditional-expr* ::= *equivalence-expr*
  [ `?` *conditional-expr* `:` *conditional-expr* ]
*equivalence-expr* ::= *implies-expr*
  [ *equivalence-op implies-expr* ] . . .
*equivalence-op* ::= `<==>` `|` `<=!=>`
*implies-expr* ::= *logical-or-expr*
  [ `==>` *implies-non-backward-expr* ]
  `|` *logical-or-expr* `<==` *logical-or-expr*
  [ `<==` *logical-or-expr* ] . . .
*implies-non-backward-expr* ::= *logical-or-expr*
  [ `==>` *implies-non-backward-expr* ]
*logical-or-expr* ::= *logical-and-expr* [ '`||`' *logical-and-expr* ] . . .
*logical-and-expr* ::= *inclusive-or-expr* [ `&&` *inclusive-or-expr* ] . . .
*inclusive-or-expr* ::= *exclusive-or-expr* [ '`|`' *exclusive-or-expr* ] . . .
*exclusive-or-expr* ::= *and-expr* [ `^` *and-expr* ] . . .
*and-expr* ::= *equality-expr* [ `&` *equality-expr* ] . . .
*equality-expr* ::= *relational-expr* [ `==` *relational-expr*] . . .

`|` *relational-expr* [ `!=` *relational-expr*] . . .
*relational-expr* ::= *shift-expr* `<` *shift-expr*
`|` *shift-expr* `>` *shift-expr*
`|` *shift-expr* `<=` *shift-expr*
`|` *shift-expr* `>=` *shift-expr*
`|` *shift-expr* `<:` *shift-expr*
`|` *shift-expr* [ `instanceof` *type-spec* ]
*shift-expr* ::= *additive-expr* [ *shift-op* *additive-expr* ] . . .
*shift-op* ::= `<<` `|` `>>` `|` `>>>`
*additive-expr* ::= *mult-expr* [ *additive-op* *mult-expr* ] . . .
*additive-op* ::= `+` `|` `-`
*mult-expr* ::= *unary-expr* [ *mult-op* *unary-expr* ] . . .
*mult-op* ::= `*` `|` `/` `|` `%`
*unary-expr* ::= `(` *type-spec* `)` *unary-expr*
`|` `++` *unary-expr*
`|` `--` *unary-expr*
`|` `+` *unary-expr*
`|` `-` *unary-expr*
`|` *unary-expr-not-plus-minus*
*unary-expr-not-plus-minus* ::= `~` *unary-expr*
`|` `!` *unary-expr*
`|` `(` *built-in-type* `)` *unary-expr*
`|` `(` *reference-type* `)` *unary-expr-not-plus-minus*
`|` *postfix-expr*
*postfix-expr* ::= *primary-expr* [ *primary-suffix* ] . . . [ `++` ]
`|` *primary-expr* [ *primary-suffix* ] . . . [ `--` ]
`|` *built-in-type* [ ‘`[`’ ‘`]`’ ] . . . `. class`
*primary-suffix* ::= `.` *ident*
`|` `. this`
`|` `. class`
`|` `.` *new-expr*
`|` `. super (` [ *expression-list* ] `)`
`|` `(` [ *expression-list* ] `)`
`|` ‘`[`’ *expression* ‘`]`’
`|` [ ‘`[`’ ‘`]`’ ] . . . `. class`
*primary-expr* ::= *ident* `|` *new-expr*
`|` *constant* `|` `super` `|` `true`
`|` `false` `|` `this` `|` `null`
`|` `(` *expression* `)`
`|` *jml-primary*
*built-in-type* ::= `void` `|` `boolean` `|` `byte`
`|` `char` `|` `short` `|` `int`
`|` `long` `|` `float` `|` `double`
*constant* ::= *java-literal*
*new-expr* ::= `new` *type* *new-suffix*
*new-suffix* ::= `(` [ *expression-list* ] `)` [ *class-block* ]
`|` *array-decl* [ *array-initializer* ]

| *set-comprehension*
*array-decl* ::= *dim-exprs* [ *dims* ]
*dim-exprs* ::= '`[`' *expression* '`]`' [ '`[`' *expression* '`]`' ] . . .
*array-initializer* ::= `{` [ *initializer* [ `,` *initializer* ] . . . [ `,` ] ] `}`
*initializer* ::= *expression*
| *array-initializer*
*jml-primary* ::= *result-expression*
| *old-expression*
| *not-assigned-expression*
| *not-modified-expression*
| *only-accessed-expression*
| *only-assigned-expression*
| *only-called-expression*
| *only-captured-expression*
| *fresh-expression*
| *reach-expression*
| *duration-expression*
| *space-expression*
| *working-space-expression*
| *nonnullelements-expression*
| *informal-description*
| *typeof-expression*
| *elemtype-expression*
| *type-expression*
| *lockset-expression*
| *max-expression*
| *is-initialized-expression*
| *invariant-for-expression*
| *lblneg-expression*
| *lblpos-expression*
| *spec-quantified-expr*
*result-expression* ::= `\result`
*old-expression* ::= `\old` `(` *spec-expression* [ `,` *ident* ] `)`
| `\pre` `(` *spec-expression* `)`
*not-assigned-expression* ::= `\not_assigned` `(` *store-ref-list* `)`
*not-modified-expression* ::= `\not_modified` `(` *store-ref-list* `)`
*only-accessed-expression* ::= `\only_accessed` `(` *store-ref-list* `)`
*only-assigned-expression* ::= `\only_assigned` `(` *store-ref-list* `)`
*only-called-expression* ::= `\only_called` `(` *method-name-list* `)`
*only-captured-expression* ::= `\only_captured` `(` *store-ref-list* `)`
*fresh-expression* ::= `\fresh` `(` *spec-expression-list* `)`
*reach-expression* ::= `\reach` `(` *spec-expression* `)`
*duration-expression* ::= `\duration` `(` *expression* `)`
*space-expression* ::= `\space` `(` *spec-expression* `)`
*working-space-expression* ::= `\working_space` `(` *expression* `)`
*nonnullelements-expression* ::= `\nonnullelements` `(` *spec-expression* `)`
*typeof-expression* ::= `\typeof` `(` *spec-expression* `)`

*elemtype-expression* ::= `\elemtype` `(` *spec-expression* `)`
*type-expression* ::= `\type` `(` *type* `)`
*lockset-expression* ::= `\lockset`
*max-expression* ::= `\max` `(` *spec-expression* `)`
*is-initialized-expression* ::= `\is_initialized` `(` *reference-type* `)`
*invariant-for-expression* ::= `\invariant_for` `(` *spec-expression* `)`
*lblneg-expression* ::= `(` `\lblneg` *ident spec-expression* `)`
*lblpos-expression* ::= `(` `\lblpos` *ident spec-expression* `)`
*spec-quantified-expr* ::= `(` *quantifier quantified-var-decls* `;`
  [ [ *predicate* ] `;` ]
  *spec-expression* `)`
*quantifier* ::= `\forall` | `\exists`
  | `\max` | `\min`
  | `\num_of` | `\product` | `\sum`
*quantified-var-decls* ::= [ *bound-var-modifiers* ] *type-spec quantified-var-declarator*
  [ `,` *quantified-var-declarator* ] . . .
*quantified-var-declarator* ::= *ident* [ *dims* ]
*spec-variable-declarators* ::= *spec-variable-declarator*
  [ `,` *spec-variable-declarator* ] . . .
*spec-variable-declarator* ::= *ident* [ *dims* ]
  [ `=` *spec-initializer* ]
*spec-array-initializer* ::= `{` [ *spec-initializer*
  [ `,` *spec-initializer* ] . . . [ `,` ] ] `}`
*spec-initializer* ::= *spec-expression*
  | *spec-array-initializer*
*bound-var-modifiers* ::= `non_null` | `nullable`
*set-comprehension* ::= `{` [ *bound-var-modifiers* ] *type-spec*
  *quantified-var-declarator* '`|`'
  *postfix-expr* `&&` *predicate* `}`
*store-ref-list* ::= *store-ref-keyword* | *store-ref* [ `,` *store-ref* ] . . .
*store-ref* ::= *store-ref-expression*
  | *informal-description*
*store-ref-expression* ::= *store-ref-name* [ *store-ref-name-suffix* ] . . .
*store-ref-name* ::= *ident* | `super` | `this`
*store-ref-name-suffix* ::= `.` *ident* | `.` `this` | '`[`' *spec-array-ref-expr* '`]`' | `.` `*`
*spec-array-ref-expr* ::= *spec-expression*
  | *spec-expression* `..` *spec-expression*
  | `*`
*store-ref-keyword* ::= `\nothing` | `\everything` | `\not_specified`

## A.9 Statements and Annotation Statements

*compound-statement* ::= `{` *statement* [ *statement* ] . . . `}`
*statement* ::= *compound-statement*
  | *local-declaration* `;`
  | *ident* `:` *statement*

| *expression* `;`
| `if` `(` *expression* `)`
*statement* [ `else` *statement* ]
| *possibly-annotated-loop*
| `break` [ *ident* ] `;`
| `continue` [ *ident* ] `;`
| `return` [ *expression* ] `;`
| *switch-statement*
| *try-block*
| `throw` *expression* `;`
| `synchronized` `(` *expression* `)` *statement*
| `;`
| *jml-annotation-statement*
| *assert-statement*
| *jml-annotation-statement*
| *model-prog-statement // only allowed in model programs*

*switch-statement* ::= `switch` `(` *expression* `)` `{`
[ *switch-body* ] . . . `}`
*switch-body* ::= *switch-label-seq* [ *statement* ] . . .
*switch-label-seq* ::= *switch-label* [ *switch-label* ] . . .
*switch-label* ::= `case` *expression* `:` | `default` `:`
*try-block* ::= `try` *compound-statement*
[ *handler* ] . . .
[ `finally` *compound-statement* ]
*handler* ::= `catch` `(` *param-declaration* `)` *compound-statement*
*local-declaration* ::= *local-modifiers variable-decls*
*local-modifiers* ::= [ *local-modifier* ] . . .
*local-modifier* ::= `ghost` | `final` `uninitialized` | `non_null` | `nullable`
| *ownership-modifier // when the Universe type system is on*
*possibly-annotated-loop* ::=
[ *loop-invariant* ] . . .
[ *variant-function* ] . . .
[ *ident* `:` ] *loop-stmt*
*loop-stmt* ::= `while` `(` *expression* `)` *statement*
| `do` *statement* `while` `(` *expression* `)` `;`
| `for` `(` [ *for-init* ] `;` [ *expression* ] `;` [ *expression-list* ] `)`
*statement*
*for-init* ::= *local-declaration* | *expression-list*
*loop-invariant* ::= *maintaining-keyword predicate* `;`
*maintaining-keyword* ::= `maintaining` | `maintaining_redundantly`
| `loop_invariant` | `loop_invariant_redundantly`
*variant-function* ::= *decreasing-keyword spec-expression* `;`
*decreasing-keyword* ::= `decreasing` | `decreasing_redundantly`
| `decreases` | `decreases_redundantly`
*assert-statement* ::= `assert` *expression* [ `:` *expression* ] `;`
| `assert` *predicate* [ `:` *expression* ] `;`
*assert-redundantly-statement* ::= `assert_redundantly` *predicate*

[ : *expression* ] ;
*jml-annotation-statement* ::= *assert-redundantly-statement*
| *assume-statement*
| *hence-by-statement*
| *set-statement*
| *refining-statement*
| *unreachable-statement*
| *debug-statement*
*assume-statement* ::= *assume-keyword* *predicate*
[ : *expression* ] ;
*assume-keyword* ::= `assume` | `assume_redundantly`
*set-statement* ::= `set` *assignment-expr* ;
*refining-statement* ::= `refining` *spec-statement* *statement*
| `refining` *generic-spec-statement-case* *statement*
*unreachable-statement* ::= `unreachable` ;
*debug-statement* ::= `debug` *expression* ;
*hence-by-statement* ::= *hence-by-keyword* *predicate* ;
*hence-by-keyword* ::= `hence_by` | `hence_by_redundantly`

## A.10 Redundancy

*redundant-spec* ::= *implications* [ *examples* ] | *examples*
*implications* ::= `implies_that` *spec-case-seq*
*examples* ::= `for_example` *example* [ `also` *example* ] . . .
*example* ::= [ [ *privacy* ] `example` ]
[ *spec-var-decls* ]
[ *spec-header* ]
*simple-spec-body*
| [ *privacy* ] `exceptional_example`
[ *spec-var-decls* ]
*spec-header*
[ *exceptional-example-body* ]
| [ *privacy* ] `exceptional_example`
[ *spec-var-decls* ]
*exceptional-example-body*
| [ *privacy* ] `normal_example`
[ *spec-var-decls* ]
*spec-header*
[ *normal-example-body* ]
| [ *privacy* ] `normal_example`
[ *spec-var-decls* ]
*normal-example-body*
*exceptional-example-body* ::= *exceptional-spec-clause*
[ *exceptional-spec-clause* ] . . .
*normal-example-body* ::= *normal-spec-clause*
[ *normal-spec-clause* ] . . .

## A.11 Model Programs

*model-program* ::= [ *privacy* ] [ `code` ] `model_program`
    *jml-compound-statement*
*jml-compound-statement* ::= *compound-statement*
*jml-statement* ::= *statement*
*model-prog-statement* ::= *nondeterministic-choice*
    | *nondeterministic-if*
    | *spec-statement*
    | *invariant*
*nondeterministic-choice* ::= `choose` *alternative-statements*
*alternative-statements* ::= *jml-compound-statement*
    [ `or` *jml-compound-statement* ] . . .
*nondeterministic-if* ::= `choose_if` *guarded-statements*
    [ `else` *jml-compound-statement* ]
*guarded-statements* ::= *guarded-statement*
    [ `or` *guarded-statement* ] . . .
*guarded-statement* ::= `{`
    *assume-statement*
    *jml-statement* [ *jml-statement*] . . . `}`
*spec-statement* ::= [ *privacy* ] *behavior-keyword*
    *generic-spec-statement-case*
    | [ *privacy* ] *exceptional-behavior-keyword*
    *exceptional-spec-case*
    | [ *privacy* ] *normal-behavior-keyword*
    *normal-spec-case*
    | [ *privacy* ] *abrupt-behavior-keyword*
    *abrupt-spec-case*
*generic-spec-statement-case* ::= [ *spec-var-decls* ]
    *generic-spec-statement-body*
    | [ *spec-var-decls* ]
    *spec-header*
    [ *generic-spec-statement-body* ]
*generic-spec-statement-body* ::= *simple-spec-statement-body*
    | `{|` *generic-spec-statement-case-seq* `|}`
*generic-spec-statement-body-seq* ::= *generic-spec-statement-case*
    [ `also` *generic-spec-statement-case* ] . . .
*simple-spec-statement-body* ::= *simple-spec-statement-clause*
    [ *simple-spec-statement-clause* ] . . .
*simple-spec-statement-clause* ::= *diverges-clause*
    | *assignable-clause*
    | *when-clause* | *working-space-clause* | *duration-clause*
    | *ensures-clause* | *signals-only-clause* | *signals-clause*
    | *continues-clause* | *breaks-clause* | *returns-clause*
*abrupt-behavior-keyword* ::= `abrupt_behavior` | `abrupt_behaviour`
*abrupt-spec-case* ::= *generic-spec-statement-case*

*continues-clause* ::= *continues-keyword* [ *target-label* ]
[ *pred-or-not* ] `;`
*continues-keyword* ::= `continues` | `continues_redundantly`
*target-label* ::= `->` `(` *ident* `)`
*breaks-clause* ::= *breaks-keyword* [ *target-label* ]
[ *pred-or-not* ] `;`
*breaks-keyword* ::= `breaks` | `breaks_redundantly`
*returns-clause* ::= *returns-keyword* [ *pred-or-not* ] `;`
*returns-keyword* ::= `returns` | `returns_redundantly`

## A.12 Specification for Subtypes

## A.13 Refinement

*refine-prefix* ::= *refine-keyword string-literal* `;`
*refine-keyword* ::= `refine` | `refines`

## A.14 MultiJava Extensions to JML

*multijava-top-level-declaration* ::= *multijava-top-level-method*
*multijava-top-level-method* ::= [ *method-specification* ]
*modifiers* [ `method` ]
[ *type-spec* ] *extending-method-head method-body*
*extending-method-head* ::= *name* `.` `ident` *formals* [*dims* ]
[ *throws-clause* ]
*multijava-param-declaration* ::= [ *param-modifier* ] . . .
*type-spec specializer ident* [ *dims* ]
*specializer* ::= `@` *type-spec*
| `@@` *value-specializer*
*value-specializer* ::= *expression*

## A.15 Universe Type System

*ownership-modifiers* ::= *ownership-modifier* [ *ownership-modifier* ]
*ownership-modifier* ::= `\rep` | `\peer` | `\readonly`
| *reserved-ownership-modifier // with –universesx parse or –universesx full*
*reserved-ownership-modifier* ::= `rep` | `peer` | `readonly`

## A.16 Safe Math Extensions

## A.17 Deprecated and Replaced Syntax

# Appendix B Modifier Summary

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

| Grammatical construct | Java modifiers | JML modifiers |
|---|---|---|
| All modifiers | `public` `protected` `private` `abstract` `static` `final` `synchronized` `transient` `volatile` `native` `strictfp` | `spec_public` `spec_protected` `model` `ghost` `pure` `instance` `helper` `non_null` `nullable` `nullable_by_default` `monitored` `uninitialized` |
| 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` |
| 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` |
| 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` | `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` |

Note that within interfaces, fields are implicitly public, static and final [Gosling-etal00]. In an interface, ghost and model fields are implicitly public and static, though they may be declared as `instance` fields, which makes them not static.

Also within an interface, methods may not be static and are implicitly abstract. Model methods in interfaces, however, are not implicitly abstract and may be declared static.

# Appendix C Type Checking Summary

[[[Hope to generate this automatically]]]

# Appendix D Verification Logic Summary

[[[Hope to generate this automatically]]]

# Appendix E Differences

The subsections below detail the differences between the JML Common Tools release of JML and other tools and between JML and Java itself.

## E.1 Differences Between JML and Other Tools

ESC/Java [Leino-Nelson-Saxe00] and JML share a common syntax; this is even more true of ESC/Java2 and JML. The initial efforts to merge syntaxes were due to the efforts of Raymie Stata. After a long process, the syntax of ESC/Java and JML were both changed and JML was nearly a superset of ESC/Java when work on ESC/Java stopped with ESC/Java 1.2.4. Following the open-source release of ESC/Java, Kiniry and Cok began work on ESC/Java2, which is now very compatible with JML's syntax [Kiniry-Cok04]. Users can thus use both tools with little or no changes to their files.

Similarly the Daikon tool [Ernst-etal01] also uses a variant of JML's syntax, as do several other tools [Burdy-etal03]. While efforts are ongoing to avoid differences, some differences are unavoidable, as research is ongoing (and people have other things to do).

We discuss the differences between the JML language described in this manual and the variants used in these other tools below.

### E.1.1 Differences Between JML and ESC/Java2

This section discusses the current state of affairs of ESC/Java2 compatibility with JML's syntax.

The following differences remain between ESC/Java2 and JML.

- ESC/Java2 is tolerant (with a suppressible warning) of missing semicolons at the ends of annotations, in many circumstances.
- ESC/Java2 does not enforce the visibility modifiers.
- ESC/Java2 strictly requires whole syntactic constructs within a single annotation comment; JML tools are more lenient.
- JML and ESC/Java2 differ in the search order for refinement files in the classpath.
- JML and ESC/Java2 differ in where `helper` annotations are permitted.
- JML does not support model classes (at least in runtime assertion checking).
- ESC/Java2 reads but ignores model programs.

The following differences between ESC/Java2 and JML are designed to remain differences. While the plan is for ESC/Java2 to parse all of JML's syntax, there are times when one needs to write annotations for one of these tool that are not understood by the other. Thus these differences are intended to allow users of both tools to write such annotations.

- JML supports annotation forms `//+@` and `/*+@` ... `@+*/`, so that annotations that JML understands but ESC/Java doesn't can be written.
- ESC/Java2 supports annotation forms `//-@` and `/*-@` ... `@-*/`, so that annotations that ESC/Java2 understands but JML doesn't can be written.

## E.2 Differences Between JML and Java

This section describes differences between JML and Java without JML. Currently the major differences are the way that JML treats `null`.

### E.2.1 Non-null by Default

As described earlier (see Section 2.8 [Null is Not the Default], page 15), JML does not, by default, allow `null` to be a value in a field, formal parameter, method or a bound variable (see Section 11.4.24.5 [Modifiers for Bound Variables], page 101). To allow `null` as a value, one has to use the `nullable` modifier on the declaration, or the `nullable_by_default` modifier on the type where the declaration occurs See Section 6.2.12 [Nullity Modifiers], page 42, for more details.

# Appendix F What's Missing

What is missing from this reference manual?

The following constructs are not discussed at all:

- `abrupt_behavior`
- `breaks` and `breaks_redundantly`
- `choose` and `choose_if`
- `continues` and `continues_redundantly`
- `example` and `exceptional_example`
- `implies_that`
- `hence_by` and `hence_by_redundantly`
- `model_program`
- `returns` and `returns_redundantly`
- `weakly` xxx

Other stuff not to forget - DRCok

- \not_specified
- \nothing
- \everything
- nowarn annotation
- methods and constructors without bodies in java files
- methods and constructors with bodies in specification files
- methods and constructors in annotation expressions - purity - modifies clauses - various checking
- anonymous and block-level classes
- field, method, constructor keywords
- exceptions in annotation expressions

# Bibliography

[America87]
Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In Jean Bezivin and others (eds.), *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*. Lecture Notes in Computer Science, Vol. 276 (Springer-Verlag, NY), pages 234-242.

[Arnold-Gosling-Holmes00]
Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language Third Edition*. The Java Series. Addison-Wesley, Reading, MA, 2000.

[ANSI95] *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++*. CBEMA, 1250 Eye Street NW, Suite 200, Washington DC 20005, April 28, 1995. (Obtained by anonymous ftp to research.att.com, directory dist/c++std/WP.)

[Back88] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, **25**(6):593-624, August 1988.

[Back-vonWright89a]
R. J. R. Back and J. von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J. W. de Bakker, et al, (eds.), *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*, Mook, The Netherlands, May/June 1989, pages 42-66. Volume 430 of *Lecture Notes Computer Science*, Spring-Verlag, 1989.

[Back-vonWright98]
Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[Borgida-etal95]
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.

[Boyland00]
John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, **31**(6):533-553, May 2001.

[Buechi-Weck00]
Martin Büchi and Wolfgang Weck. The Greybox Approach: When Blackbox Specifications Hide Too Much. Technical Report 297, Turku Centre for Computer Science, August 1999.
'`http://www.tucs.abo.fi/publications/techreports/TR297.html`'.

[Buechi00] Martin Büchi. Safe Language Mechanisms for Modularization and Concurrency. Ph.D. Thesis, Turku Center for Computer Science, May 2000. TUCS Dissertations No. 28.

[Burdy-etal03]
Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools

and applications. Dept. of Computer Science, University of Nijmegen, TR NIII-R0309, 2003.
'http://www.eecs.ucf.edu/~leavens/JML/OldReleases/jml-white-paper.pdf'.

[Chalin07] Patrice Chalin. A Sound Assertion Semantics for the Dependable Systems Evolution Verifying Compiler. *Proceedings of the International Conference on Software Engineering (ICSE)*, Minneapolis, MN, USA, 2007.

[Chalin-Rioux05]
Patrice Chalin and Frederic Rioux. Non-null References by Default in the Java Modeling Language. In Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS'05), Lisbon, Portugal. September, 2005. An updated version is available as Department of Computer Science, Concordia University, ENCS-CSE TR 2005-004, December 2005, which is available from the URL
'http://www.cs.concordia.ca/~chalin/papers/TR-2005-004-r3.2.pdf'.

[Cheon-Leavens02]
Yoonsik Cheon and Gary T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *ECOOP 2002 – Object-Oriented Programming, 16th European Conference, Malaga, Spain*, pages 231–255. Springer-Verlag, June 2002. Also Department of Computer Science, Iowa State University, TR #01-12a, November 2001, revised March 2002, which is available from the URL
'ftp://ftp.cs.iastate.edu/pub/techreports/TR01-12/TR.pdf'.

[Cheon-Leavens02b]
Yoonsik Cheon and Gary T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun (eds.), *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA*, pages 322–328. CSREA Press, June 2002. Also Department of Computer Science, Iowa State University, TR #02-05, March 2002, which is available from the URL
'ftp://ftp.cs.iastate.edu/pub/techreports/TR02-05/TR.pdf'.

[Cheon-etal05]
Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model Variables: Cleanly Supporting Abstraction in Design By Contract. *Software—Practice and Experience*, **35**(6):583-599, May 2005. Also Department of Computer Science, Iowa State University, TR 03-10, March 2003.
'ftp://ftp.cs.iastate.edu/pub/techreports/TR03-10/TR.pdf'.

[Cheon03] Yoonsik Cheon. A Runtime Assertion Checker for the Java Modeling Language. Department of Computer Science, Iowa State University, TR 03-09, April, 2003.
'ftp://ftp.cs.iastate.edu/pub/techreports/TR03-09/TR.pdf'

[Clifton-etal00]
Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In

*OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota* (*ACM SIGPLAN Notices*, **35**(10):130-145, October 2000).

[Cohen90] Edward Cohen. *Programming in the 1990s: An Introduction to the Calculation of Programs*. Springer-Verlag, New York, N.Y., 1990.

[Corbett-etal00]
James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In S. Brookes and M. Main and A. Melton and M. Mislove (eds.), *Proceedings of the 22nd International Conference on Software Engineering*, pp. 439-448, ACM Press, 2000.

[Dhara-Leavens94b]
Krishna Kishore Dhara and Gary T. Leavens. Weak Behavioral Subtyping for Types with Mutable Objects. In S. Brookes and M. Main and A. Melton and M. Mislove (eds.), *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, Volume 1 of *Electronic Notes in Computer Science*, Elsevier, 1995. '`http://www.sciencedirect.com/science/journal/15710661`'.

[Dhara-Leavens96]
Krishna Kishore Dhara and Gary T. Leavens. Forcing Behavioral Subtyping Through Specification Inheritance. In *Proceedings 18th International Conference on Software Engineering*, Berlin, Germany, pages 258-267. IEEE 1996. An extended version is Department of Computer Science, Iowa State University, TR #95-20b, December 1995, which is available from the URL '`ftp://ftp.cs.iastate.edu/pub/techreports/TR95-20/TR.ps.Z`'.

[Dhara97] Krishna Kishore Dhara. Behavioral Subtyping in Object-Oriented Languages. Ph.D. Thesis, Department of Computer Science, Iowa State University. Also Technical Report TR #97-09, May 1997. Available from the URL '`ftp://ftp.cs.iastate.edu/pub/techreports/TR97-09/TR.ps.gz`'.

[Dietl-Drossopoulou-Mueller07]
Werner Dietl, Sophia Drossopoulou and Peter Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)* pages 28–53, 2007. Available from '`http://sct.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=DietlDrossopoulouMu`

[Dietl-Mueller04]
Werner Dietl and Peter Müller. Exceptions in ownership type systems. In E. Poll, editor, *Formal Techniques for Java-like Programs* pages 49–54, 2004. Available from '`http://sct.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=DietlMueller04.pdf`'

[Dietl-Mueller05]
Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, **4**(8):5–32, October 2005. Available from '`http://www.jot.fm/issues/issue_2005_10/article1.pdf`'.

[Dietl-Mueller-Schregenberger08]
Werner Dietl, Peter Müller and Daniel Schregenberger. Universe Type System — Quick-Reference. Available from ‘`http://sct.inf.ethz.ch/research/universes/tools/juts-quickref.pdf`’.

[Dijkstra76]
Edsger W. Dijkstra. *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, N.J., 1976).

[Edwards-etal94]
Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, Murali Sitaraman, and Bruce W. Weide. Part II: Specifying Components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, **19**(4):29-39, October 1994.

[Ernst-etal01]
Michael D. 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):1-25, February 2001.

[Fitzgerald-Larsen98]
John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, Cambridge, UK, 1998.

[Gosling-etal00]
James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, MA, 2000.

[Gries-Schneider95]
David Gries and Fred B. Schneider. Avoiding the Undefined by Underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer-Verlag, New York, N.Y., 1995.

[Guttag-Horning-Wing85b]
John V. Guttag and James J. Horning and Jeannette M. Wing. The Larch Family of Specification Languages. *IEEE Software*, **2**(5):24-36, September 1985.

[Guttag-Horning93]
John V. Guttag and James J. Horning with S.J. Garland, K.D. Jones, A. Modet and J.M. Wing. *Larch: Languages and Tools for Formal Specification* (Springer-Verlag, NY, 1993).

[Hall90] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, **7**(5):11-19, September 1990.

[Hayes93] I. Hayes (ed.), *Specification Case Studies*, second edition (Prentice-Hall, Englewood Cliffs, N.J., 1990).

[Hesselink92]
Wim H. Hesselink. *Programs, Recursion, and Unbounded Choice* (Cambridge University Press, Cambridge, UK, 1992).

[Hoare69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, **12**(10):576-583, October 1969.

[Hoare72a]
C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, **1**(4):271-281, 1972.

[Huisman01]
Marieke Huisman. Reasoning about JAVA programs in higher order logic with PVS and Isabelle. IPA dissertation series, 2001-03. Ph.D. dissertation, University of Nijmegen, 2001.

[ISO96] International Standards Organization. *Information Technology - Programming Languages, Their Environments and System Software Interfaces - Vienna Development Method - Specification Language - Part 1: Base language*. International Standard ISO/IEC 13817-1, December, 1996.

[Khurshid-Marinov-Jackson02]
Sarfraz Khurshid and Darko Marinov and Daniel Jackson. An Analyzable Annotation Language. In *Proceedings of OOPSLA '02 Conference on Object-Oriented Programming, Languages, Systems, and Applications*. (*ACM SIGPLAN Notices*, **37**(11):231–245, October 2002).

[Jacobs-etal98]
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 Proceedings* (*ACM SIGPLAN Notices*, **33**(10):329-490, October 1998).

[Jones90] Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[Jones95e] C.B. Jones, Partial functions and logics: A warning. *Information Processing Letters*, **54**(2):65-67, 1995.

[Kiczales-Lamping92]
Gregor Kiczales and John Lamping. Issues in the Design and Documentation of Class Libraries. In Andreas Paepcke (ed.), *OOPSLA '92 Proceedings* (*ACM SIGPLAN Notices*, **27**(10):435-451, October 1992).

[Kiniry-Cok04]
Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In Marieke Huisman (ed.), *CASSIS 2004 - Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille, France, 2004, Proceedings*, volume 3362 of *Lecture Notes in Computer Science*, pages 108-128. Springer-Verlag, 2004.

[Krone-Ogden-Sitaraman03]
Joan Krone, William F. Ogden, Murali Sitaraman. Modular Verification of Performance Constraints. Technical Report RSRG-03-04, Department

of Computer Science, Clemson University, May, 2003. Available from 'http://www.cs.clemson.edu/~resolve/reports/RSRG-03-04.pdf'

[Lamport89]
Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. *CACM*, **32**(1):32-45, January 1989.

[LeavensLarchFAQ]
Gary T. Leavens. Larch frequently asked questions. Version 1.110. Available in 'http://www.eecs.ucf.edu/~leavens/larch-faq.html', May 2000.

[Leavens-Baker99]
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.

[Leavens-Baker-Ruby99]
Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175-188.

[Leavens-Baker-Ruby06]
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.
'http://doi.acm.org/10.1145/1127878.1127884'. Also Iowa State University, Department of Computer Science, TR #98-06-rev29, January 2006, which is available from the URL
'ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.pdf'.

[Leavens-Cheon06]
Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. December, 2006, which is available from the URL
'http://www.jmlspecs.org/jmldbc.pdf'.

[Leavens-Dhara00]
Gary T. Leavens and Krishna Kishore Dhara. Concepts of Behavioral Subtyping and a Sketch of Their Extension to Component-Based Systems. In Gary T. Leavens and Murali Sitaraman (eds.), *Foundations of Component-Based Systems*, Cambridge University Press, 2000, pp. 113-135.
'http://www.eecs.ucf.edu/~leavens/FoCBS-book/06-leavens-dhara.pdf'

[Leavens-etal05]
G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification *Science of Computer Programming*, **55**(1-3):185-208, 2005.

[Leavens-Mueller07]
Gary T. Leavens and Peter Müller. Information Hiding and Visibility in Interface Specifications. In *International Conference on Software Engineering (ICSE)*, pages 385-395, IEEE, 2007. '`http://dx.doi.org/10.1109/ICSE.2007.44`'

[Leavens-Naumann06]
Gary T. Leavens and David A. Naumann. Behavioral Subtyping, Specification Inheritance, and Modular Reasoning. Department of Computer Science, TR \#06-20b, July 2006, revised August, September 2006. Available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR90-09/TR.pdf`'.

[Leavens-Weihl90]
Gary T. Leavens and William E. Weihl. Reasoning about Object-oriented Programs that use Subtypes (extended abstract). In N. Meyrowitz (ed.), *OOPSLA ECOOP '90 Proceedings* (*ACM SIGPLAN Notices*, **25**(10):212-223, October 1990).

[Leavens-Weihl95]
Gary T. Leavens and William E. Weihl. Specification and Verification of Object-Oriented Programs Using Supertype Abstraction. *Acta Informatica*, **32**(8):705-778, November 1995.

[Leavens-Wing98]
Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. *Formal Aspects of Computing*, **10**(1):590-75, January 1998.

[Leavens90]
Gary T. Leavens. Modular Verification of Object-Oriented Programs with Subtypes. Department of Computer Science, Iowa State University (Ames, Iowa, 50011), TR 90-09, July 1990. Available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR90-09/TR.ps.Z`'.

[Leavens91]
Gary T. Leavens. Modular Specification and Verification of Object-Oriented Programs. *IEEE Software*, **8**(4):72-80, July 1991.

[Leavens96b]
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* (Kluwer Academic Publishers, 1996), Chapter 8, pages 121-142. An extended version is Department of Computer Science, Iowa State University, TR #96-01c, July 1996, which is available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR96-01/TR.ps.Z`'.

[Leavens97c]
Gary T. Leavens. *Larch/C++ Reference Manual*. Version 5.14. Available in
'`http://www.eecs.ucf.edu/~leavens/larchc++.html`', October 1997.

[Leavens06b]
Gary T. Leavens. JML's Rich, Inherited Specifications for Behavioral Subtypes. In Zhiming Liu and He Jifeng (eds), *Proceedings, International Conference on Formal Engineering Methods (ICFEM'06), Macao, China*, pages 2-36. Volume 4260 of *Lecture Notes in Computer Science*, Springer-Verlag, 2006. Also Department of Computer Science, Iowa State University, TR \#06-22, August 2006.
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR06-22/TR.pdf`'

[Ledgard80]
Henry. F. Ledgard. A Human Engineered Variant of BNF. *ACM SIGPLAN Notices*, **15**(10):57-62, October 1980.

[Leino-Nelson-Saxe00]
K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java User's Manual. Technical Note 2000-02, Systems Research Center, October, 2000.

[Leino-etal00]
K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking. Web page at
'`http://research.compaq.com/SRC/esc/Esc.html`'.

[Leino95] K. Rustan M. Leino. Towards Reliable Modular Programs. PhD thesis, California Institute of Technology, January 1995. Available from the URL
'`ftp://ftp.cs.caltech.edu/tr/cs-tr-95-03.ps.Z`'.

[Leino95b] K. Rustan M. Leino. A myth in the modular specification of programs. KRML 63, November 1995. Obtained from the author (rustan@pa.dec.com).

[Leino98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. *OOPSLA '98 Conference Proceedings*. (*ACM SIGPLAN Notices*, **33**(10):144-153, October 1998).

[Lerner91] Richard Allen Lerner. Specifying Objects of Concurrent Systems. School of Computer Science, Carnegie Mellon University, CMU-CS-91-131, May 1991. Available from the URL
'`ftp://ftp.cs.cmu.edu/afs/cs.cmu.edu/project/larch/ftp/thesis.ps.Z`'.

[Liskov-Guttag86]
Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development* (MIT Press, Cambridge, Mass., 1986).

[Liskov-Wing93b]
Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. In Andreas Paepcke, editor, *OOPSLA '93 Proceedings*. (*ACM SIGPLAN Notices* **28**(10):16-28, October, 1993.)

[Liskov-Wing94]
Barbara Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, **16**(6):1811-1841, November 1994.

[Meyer92a]
Bertrand Meyer. Applying “design by contract”. *Computer*, **25**(10):40–51, October 1992.

[Meyer92b]
Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[Meyer97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[Morgan-Vickers94]
Carroll Morgan and Trevor Vickers. *On the refinement calculus*. Springer-Verlag, New York, N.Y., 1994.

[Morgan94]
Carroll Morgan. *Programming from Specifications*, second edition (Prentice-Hall, 1994).

[Morris87] Joseph~M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, **9**(3):287-306, December 1987.

[Mueller-Poetzsch-Heffter00]
Peter Müller and Arnd Poetzsch-Heffter. Modular Specification and Verification Techniques for Object-Oriented Software Components. In Gary T. Leavens and Murali Sitaraman (eds.), *Foundations of Component-Based Systems*, pages 137-159. Cambridge University Press, 2000.

[Mueller-Poetzsch-Heffter00a]
Peter Müller and Arnd Poetzsch-Heffter. A Type System for Controlling Representation Exposure in Java. In S. Drossopoulou, et al. (eds.), *Formal Techniques for Java Programs*, 2000. Technical Report 269, Fernuniversität Hagen, Available from
‘`http://www.informatik.fernuni-hagen.de/pi5/publications.html`’

[Mueller-Poetzsch-Heffter01a]
Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Alias and Dependency Control. Technical Report 279, Fernuniversität Hagen, 2001. Available from
‘`http://www.informatik.fernuni-hagen.de/pi5/publications.html`’

[Mueller-Poetzsch-Heffter-Leavens03]
Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular Specification of Frame Properties in JML. *Concurrency and Computation: Practice and Experience*, **15**(2):117-154, February 2003. Also Technical Report TR #02-02, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, February 2002. Available from
‘`ftp://ftp.cs.iastate.edu/pub/techreports/TR02-02/TR.pdf`’

[Mueller-Poetzsch-Heffter-Leavens06]
Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular Invariants for Layered Object Structures. *Science of Computer Programming*, **62**(3):253-

286, October 2006.
'http://dx.doi.org/10.1016/j.scico.2006.03.001' Also Technical Report 424, ETH Zürich, October 2003, revised March 2004, March 2005. Available from
'ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/4xx/424.pdf'

[Mueller02]
Peter Müller. Modular Specification and Verification of Object-Oriented Programs. Volume 2262 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002.

[Nelson89] Greg Nelson. A Generalization of Dijkstra's Calculus. *ACM Transactions on Programming Languages and Systems*, **11**(4):517-561, October 1989.

[Noble-Vitek-Potter98]
James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In Eric Jul (ed.), *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, pages volume 1445 of *Lecture Notes in Computer Science*, pages 158-185. Springer-Verlag, New York, N.Y., 1998.

[Parnas72] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACM*, **15**(12):1053-1058, December 1972.

[Poetzsch-Heffter97]
Arnd Poetzsch-Heffter. Specification and Verification of Object-Oriented Programs. Habilitationsschrift, Technische Universitaet Muenchen, 1997. Available from the URL
'http://wwweickel.informatik.tu-muenchen.de/persons/poetzsch/habil.ps.gz'.

[Jacobs-Poll01]
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*, Springer-Verlag, 2001. 'http://www.cs.kun.nl/~erikpoll/publications/jmllogic.html'

[Raghavan-Leavens05]
Arun D. Raghavan and Gary T. Leavens. Desugaring JML Method Specifications. Technical Report #00-03a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, April, 2000, revised May 2005. Available in
'ftp://ftp.cs.iastate.edu/pub/techreports/TR00-03/TR.ps.gz'.

[Rioux-Chalin07]
F. Rioux and P. Chalin. Effective and Efficient Runtime Assertion Checking for JML Through Strong Validity. *Proceedings of the 9th Workshop on Formal Techniques for Java-like Programs (FTfJP'07)*, Berlin, Germany, 2007.

[Rodriguez-etal05]
Edwin Rodriguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, Robby. Extending JML for Modular Specification and Verification of Multi-Threaded Programs. In Andrew P. Black (ed.), *ECOOP 2005 – Object-Oriented Programming 19th European Conference*, Glasgow, UK, pages 551-576. Volume 3586 of *Lecture Notes in Computer Science*, Springer Verlag, July 2005.

[Rosenblum95]
David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, **21**(1):19–31, January 1995.

[Ruby-Leavens00]
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*. (*ACM SIGPLAN Notices*, **35**(10):208-228, October, 2000.) Also Technical Report #00-05d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011. April 2000, revised April, June, July 2000. Available in '`ftp://ftp.cs.iastate.edu/pub/techreports/TR00-05/TR.ps.gz`'.

[Ruby06] Clyde Dwain Ruby. Modular subclass verification: safely creating correct subclasses without superclass code. Ph.D. Thesis, Department of Computer Science, Iowa State University. Also Technical Report #06-34, December 2006. Available from the URL '`ftp://ftp.cs.iastate.edu/pub/techreports/TR06-34/TR.pdf`'.

[Salcianu-Rinard05]
Alexandru Salcianu and Martin Rinard. Purity and Side Effect Analysis for Java Programs. In Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation. Paris, France January 2005. Available in '`http://www.mit.edu/~salcianu/publications/vmcai05-purity.pdf`'

[Shaner-Leavens-Naumann07]
Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular Verification of Higher-Order Methods with Mandatory Calls Specified by Model Programs Department of Computer Science, Iowa State University, TR #07-04a, March 2007, revised April 2007, which is available from the URL '`ftp://ftp.cs.iastate.edu/pub/techreports/TR07-04/TR.pdf`'.

[Spivey92] J. Michael Spivey. *The Z Notation: A Reference Manual*, second edition, (Prentice-Hall, Englewood Cliffs, N.J., 1992).

[Steyaert-etal96]
Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Issues in the Design and Documentation of Class Libraries. In *OOPSLA '96 Proceedings*. (*ACM SIGPLAN Notices*, **31**(10):268-285, October, 1996.)

[Tan95] Yang Meng Tan. *Formal Specification Techniques for Engineering Modular C Programs*. International Series in Software Engineering (Kluwer Academic Publishers, Boston, 1995). Also published as Formal Specification Techniques for Promoting Software Modularity, Enhancing Documentation, and Testing Specifications. Technical Report TR-619, MIT Lab. for Comp. Sci., June 1994.

[Watt91] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall, International Series in Computer Science, New York, 1991.

[Wills92b] Alan Wills. Specification in Fresco. In Susan Stepney and Rosalind Barden and David Cooper (eds.), *Object Orientation in Z*, chapter 11, pages 127-135.

Springer-Verlag, Workshops in Computing Series, Cambridge CB2 1LQ, UK, 1992.

[Wing83] Jeannette Marie Wing. *A Two-Tiered Approach to Specifying Programs* Technical Report TR-299, Mass. Institute of Technology, Laboratory for Computer Science, 1983.

[Wing87] Jeannette M. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, **9**(1):1-24, January 1987.

[Wing90a] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *Computer*, **23**(9):8-24, September 1990.

# Index

## <



## =



## >



## ?



## @



## [



## ]



## ^



## _



## ‘



## {



## }



## \

# B

# C

# D

# I

## J



## K

## L



## M

## N

# O



# P

# Q



# R

# S

## T



## U



## V

## W



## Z