# Table of Contents

## 1 Introduction .................................................. 1
  1.1 Behavioral Interface Specifications ............................. 1
  1.2 A First Example .............................................. 2
  1.3 What is JML Good For? .......................................... 6
  1.4 Status and Plans for JML ....................................... 7
  1.5 Historical Precedents .......................................... 8
  1.6 Acknowledgments ............................................. 9

## 2 Fundamental Concepts ................................. 11
  2.1 Types can be Classes and Interfaces ......................... 11
  2.2 Model and Ghost .............................................. 11
  2.3 Lightweight and Heavyweight Specifications .................. 12
  2.4 Privacy Modifiers and Visibility ............................... 12
  2.5 Instance vs. Static ........................................... 15
  2.6 Locations and Aliasing ......................................... 15
  2.7 Expression Evaluation and Undefinedness .................... 15
  2.8 Null is Not the Default ...................................... 16
  2.9 Language Levels ............................................ 17
    2.9.1 Level 0 Features ........................................ 18
    2.9.2 Level 1 Features ....................................... 21
    2.9.3 Level 2 Features ....................................... 22
    2.9.4 Level 3 Features ....................................... 24
    2.9.5 Level C Features ....................................... 24
    2.9.6 Level X Features ....................................... 24

## 3 Syntax Notation ........................................ 25

## 4 Lexical Conventions ................................. 26
  4.1 White Space ................................................. 26
  4.2 Lexical Pragmas .............................................. 26
  4.3 Comments ................................................... 27
  4.4 Annotation Markers ........................................... 27
  4.5 Documentation Comments ..................................... 28
  4.6 Tokens ...................................................... 29

## 5 Compilation Units ................................. 35
  5.1 Package Declarations ........................................ 36
  5.2 Import Declarations ........................................ 36
9 Method Specifications.......................... 63
  9.1 Basic Concepts in Method Specification .............. 63
  9.2 Organization of Method Specifications .................. 63
  9.3 Access Control in Specification Cases .................. 64
  9.4 Lightweight Specification Cases ....................... 65
  9.5 Heavyweight Specification Cases ....................... 67
  9.6 Behavior Specification Cases ......................... 67
    9.6.1 Semantics of flat behavior specification cases .... 68
    9.6.2 Semantics of non-helper methods .................. 68
    9.6.3 Semantics of non-helper constructors ............... 71
    9.6.4 Semantics of helper methods and constructors ...... 71
    9.6.5 Semantics of nested behavior specification cases .. 71
  9.7 Normal Behavior Specification Cases ................. 72
  9.8 Exceptional Behavior Specification Cases ............. 73
    9.8.1 Pragmatics of Exceptional Behavior Specifications Cases 73
  9.9 Method Specification Clauses ......................... 75
    9.9.1 Specification Variable Declarations ............... 75
      9.9.1.1 Forall Variable Declarations .................. 75
      9.9.1.2 Old Variable Declarations ..................... 75
    9.9.2 Requires Clauses ................................ 76
    9.9.3 Ensures Clauses ................................ 77
    9.9.4 Signals Clauses ................................ 77
    9.9.5 Signals-Only Clauses .............................. 79
    9.9.6 Parameters in Postconditions ...................... 80
    9.9.7 Diverges Clauses ................................ 81
    9.9.8 When Clauses ................................ 82
    9.9.9 Assignable Clauses ................................ 83
    9.9.10 Accessible Clauses ................................ 83
    9.9.11 Callable Clauses ................................ 84
    9.9.12 Measured By Clauses ............................... 84
    9.9.13 Captures Clauses ................................ 85
    9.9.14 Working Space Clauses ............................ 85
    9.9.15 Duration Clauses ................................ 85
  10 Data Groups .................................... 87
    10.1 Static Data Group Inclusions ......................... 87
    10.2 Dynamic Data Group Mappings ......................... 88

11 Specification Inheritance ......................... 89
## 12 Predicates and Specification Expressions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1</td>
<td>Predicates</td>
<td>90</td>
</tr>
<tr>
<td>12.2</td>
<td>Specification Expressions</td>
<td>90</td>
</tr>
<tr>
<td>12.3</td>
<td>Expressions</td>
<td>90</td>
</tr>
<tr>
<td>12.4</td>
<td>JML Primary Expressions</td>
<td>92</td>
</tr>
<tr>
<td>12.4.1</td>
<td>\result</td>
<td>93</td>
</tr>
<tr>
<td>12.4.2</td>
<td>\old and \pre</td>
<td>93</td>
</tr>
<tr>
<td>12.4.3</td>
<td>\not_assigned</td>
<td>94</td>
</tr>
<tr>
<td>12.4.4</td>
<td>\not_modified</td>
<td>95</td>
</tr>
<tr>
<td>12.4.5</td>
<td>\only_accessed</td>
<td>95</td>
</tr>
<tr>
<td>12.4.6</td>
<td>\only_assigned</td>
<td>96</td>
</tr>
<tr>
<td>12.4.7</td>
<td>\only_called</td>
<td>96</td>
</tr>
<tr>
<td>12.4.8</td>
<td>\only_captured</td>
<td>97</td>
</tr>
<tr>
<td>12.4.9</td>
<td>\fresh</td>
<td>97</td>
</tr>
<tr>
<td>12.4.10</td>
<td>\reach</td>
<td>97</td>
</tr>
<tr>
<td>12.4.11</td>
<td>\duration</td>
<td>98</td>
</tr>
<tr>
<td>12.4.12</td>
<td>\space</td>
<td>98</td>
</tr>
<tr>
<td>12.4.13</td>
<td>\working_space</td>
<td>98</td>
</tr>
<tr>
<td>12.4.14</td>
<td>\nonnull_elements</td>
<td>99</td>
</tr>
<tr>
<td>12.4.15</td>
<td>Informal Predicates</td>
<td>99</td>
</tr>
<tr>
<td>12.4.16</td>
<td>\typeof</td>
<td>99</td>
</tr>
<tr>
<td>12.4.17</td>
<td>\elemtype</td>
<td>99</td>
</tr>
<tr>
<td>12.4.18</td>
<td>\type</td>
<td>100</td>
</tr>
<tr>
<td>12.4.19</td>
<td>\lockset</td>
<td>100</td>
</tr>
<tr>
<td>12.4.20</td>
<td>\max</td>
<td>100</td>
</tr>
<tr>
<td>12.4.21</td>
<td>\is_initialized</td>
<td>100</td>
</tr>
<tr>
<td>12.4.22</td>
<td>\Invariant_for</td>
<td>101</td>
</tr>
<tr>
<td>12.4.23</td>
<td>\lblrneg and \lbldpos</td>
<td>101</td>
</tr>
<tr>
<td>12.4.24</td>
<td>Quantified Expressions</td>
<td>101</td>
</tr>
<tr>
<td>12.4.24.1</td>
<td>Universal and Existential Quantifiers</td>
<td>102</td>
</tr>
<tr>
<td>12.4.24.2</td>
<td>Generalized Quantifiers</td>
<td>102</td>
</tr>
<tr>
<td>12.4.24.3</td>
<td>Numerical Quantifier</td>
<td>103</td>
</tr>
<tr>
<td>12.4.24.4</td>
<td>Executability of Quantified Expressions</td>
<td>103</td>
</tr>
<tr>
<td>12.4.24.5</td>
<td>Modifiers for Bound Variables</td>
<td>103</td>
</tr>
<tr>
<td>12.4.24.6</td>
<td>Quantifying over Reference Types</td>
<td>104</td>
</tr>
<tr>
<td>12.5</td>
<td>Set Comprehensions</td>
<td>104</td>
</tr>
<tr>
<td>12.6</td>
<td>JML Operators</td>
<td>105</td>
</tr>
<tr>
<td>12.6.1</td>
<td>Subtype operator</td>
<td>105</td>
</tr>
<tr>
<td>12.6.2</td>
<td>Equivalence and Inequivalence Operators</td>
<td>105</td>
</tr>
<tr>
<td>12.6.3</td>
<td>Forward and Reverse Implication Operators</td>
<td>105</td>
</tr>
<tr>
<td>12.6.4</td>
<td>Lockset Ordering</td>
<td>106</td>
</tr>
<tr>
<td>12.7</td>
<td>Store Refs</td>
<td>106</td>
</tr>
</tbody>
</table>
13 Statements and Annotation Statements . . 108
  13.1 Local Declaration Statements ......................... 108
  13.1.1 Modifiers for Local Declarations ................... 109
  13.2 Loop Statements .................................. 109
  13.2.1 Loop Invariants .................................. 111
  13.2.2 Loop Variant Functions ............................ 112
  13.3 Assert Statements .................................. 113
  13.4 JML Annotation Statements ............................ 113
  13.4.1 Assume Statements ................................ 114
  13.4.2 Set Statements .................................. 114
  13.4.3 Refining Statements ............................... 114
  13.4.4 Unreachable Statements ............................ 115
  13.4.5 Debug Statements ................................ 116
  13.4.6 Hence By Statements .............................. 116

14 Redundancy ............................................ 118
  14.1 Redundant Implications and Redundantly Clauses ........ 118
  14.2 Redundant Examples .................................. 120

15 Model Programs ......................................... 122
  15.1 Ideas Behind Model Programs ......................... 122
  15.2 Extracting Model Program Specifications ............... 124
  15.3 Details of Model Programs ............................ 124
  15.4 Nondeterministic Choice Statement .................... 124
  15.5 Nondeterministic If Statement ......................... 124
  15.6 Specification Statements ............................. 125
    15.6.1 Continues Clause ................................ 126
    15.6.2 Breaks Clause .................................. 126
    15.6.3 Returns Clause .................................. 126

16 Specification for Subtypes .............................. 127
  16.1 Method of Specifying for Subclasses .................. 127
  16.2 Code Contracts .................................... 127

17 Separate Files for Specifications .............. 129
  17.1 File Name Suffixes .................................. 129
  17.2 Using Separate Files ................................ 129
  17.3 Type Checking Separate Files ......................... 129
  17.4 Default Constructors and Separate Files ............. 131
Appendix E  Type Checking Summary...... 166
Appendix F  Verification Logic Summary ... 167
Appendix G  Differences ..................... 168
  G.1 Differences Between JML and Other Tools.................. 168
  G.1.1 Differences Between JML and ESC/Java2 ............ 168
  G.2 Differences Between JML and Java....................... 169
    G.2.1 Non-null by Default .................................. 169
Appendix H  What’s Missing .................. 170
Bibliography ........................................ 171
Index ................................................ 182
Chapter 1: Introduction

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 9), 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 parts of a program. In the case of JML, this is the Java syntax and type information needed to call a method or use a field or type. For a method the interface includes such things as the name of the method, its modifiers (including its visibility and whether it is final) its number of arguments, its return type, what 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, should ship with the JML tools, or you can find it online

Chapter 1: Introduction
package org.jmlspecs.samples.jmlrefman; // line 1
public abstract class IntHeap { // line 2
  // 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 &amp; j < elements.length; // line 12
  @ elements[j]); // line 13
  @*/ // line 14
  public abstract /*@ pure @*/ int largest(); // line 15
  // line 16
  // line 17
  //@ ensures \result == elements.length; // line 18
  public abstract /*@ pure @*/ int size(); // line 19
}; // line 20

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'.\footnote{\textnormal{This is the usual way to declare a specification-only field; it is also possible to use the \texttt{ghost} modifier (see Section 2.2 [Model and Ghost], page 11).}} 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 60, 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 `\texttt{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 `\texttt{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 `\texttt{non_null}'. This just says that in any publicly-visible state, the value of `\texttt{elements}' must not be `null'. It is thus a simple kind of invariant (see Section 8.2 [Invariants], page 52).

In the above specification of `\texttt{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 63), but we will usually not to do that in this document.

The specification of the method `\texttt{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 `\texttt{public}' says that the specification is intended for use by clients. (If the privacy modifier had been `\texttt{protected}', for example, then the specification would have been intended for subclasses.)

The keyword `\texttt{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 \textit{heavyweight} specification uses one of JML’s behavior keywords, like `\texttt{normal_behavior}', which tells JML that the method specification is intended to be complete. By contrast, a \textit{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.\footnote{\textnormal{Lightweight specifications come from ESC/Java.}} Second, the keyword `\texttt{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 `\texttt{false}', which prohibits the method from throwing an exception when the precondition holds. (Third, it says that the divergence condition defaults to `\texttt{false}'. See Chapter 9 [Method Specifications], page 63, for more details.)

The heart of the method specification of `\texttt{largest}' is found on lines 7 through 13. This part of the specification gives the method’s precondition, frame axiom, and normal postcondition. The precondition is contained in the `\texttt{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. One 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

```java
//@ 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 63, 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 83). 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 description 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 [Rodriguez-etal05], 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.

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 ease and accuracy of tool support. One specific goal that has emerged over time is that JML should be able to unify several different tool-building efforts in the area of formal methods.

The most basic tool support for JML – simply parsing and 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 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 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]. 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 \texttt{IntHeap} using abstract values of the form \texttt{empty} and \texttt{add(i,h)}, where \texttt{i} is an \texttt{int} and \texttt{h} is an \texttt{IntHeap}. These notations form a mathematical vocabulary used in the rest of the specification.

There are two advantages to writing specifications with a mathematically-defined abstract values instead of directly using Java variables and data structures. The first is that by using abstract values, the specification does not have to be changed when the particular data structure used in the program is changed. This permits different implementations of the same specification to use different data structures. Therefore the specification forms a contract between the rest of the program 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 \texttt{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 (i.e., pre- and postconditions for methods). In Z one builds abstract values using tuples, sets, relations, functions, sequences, and bags; these all come with pre-defined operations that can be used in assertions. In VDM one has a similar collection of mathematical tools to describe abstract values, and another set of pre-defined operations. In the Larch approach, there are some pre-defined kinds of abstract values (found in Guttag and Horning’s LSL Handbook, Appendix A of [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 “\texttt{old}” notation from Eiffel, which appears in JML as \texttt{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 University), 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, Jesus Ravelo, and Faraz Hussain for comments and questions on earlier versions of this document. Thanks to the many who have worked on the JML checker used to check the specifications in this document. Leavens thanks Iowa State University and its computer science department for helping foster and support the initial work on JML.
See the “Preliminary Design of JML” [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 as “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 model modifier has two meanings. The first 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 JML annotations. The second meaning of model depends on the type of feature being declared.

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 60). (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 87).

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 JML 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 13 [Statements and Annotation Statements], page 108).

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 (see Chapter 17 [Separate Files for Specifications], page 129). 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 eventual goal, then 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 \texttt{spec\_public}. See Section 2.4 [Privacy Modifiers and Visibility], page 12, for more about \texttt{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 \textit{lightweight}, in which the user only says what interests them. On the other hand, in a \textit{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 63, for details, but in essence in a method specification case that uses one of the behavior keywords (such as \texttt{normal\_behavior}, \texttt{exceptional\_behavior}, or \texttt{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 a JML 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 \texttt{P} and type \texttt{P.T} may be referenced from outside \texttt{P} only if it is declared as \texttt{public}, or if it is declared as \texttt{protected} and the reference occurs within a subclass of \texttt{P.T}. This name may be referenced from within \texttt{P} but outside of \texttt{P.T} only if it is declared as \texttt{public}, default access, or \texttt{protected}. Such a name may always be referenced from within \texttt{P.T}, even if it is declared as \texttt{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 \textit{annotation context} of a reference to a name is the smallest grammatical unit with an attached (or implicit) visibility. For example, this annotation context could be a method specification case, an invariant, a history constraint, or a field declaration. The visibility level of such an annotation context can be \texttt{public}, \texttt{protected}, \texttt{private}, or default (package) visibility.

JML has two rules governing visibility that differ from Java. The first is that an annotation context cannot refer to names that are more hidden than the context’s own visibility. That is, for a reference to a name \texttt{x} to be legal, the visibility of the annotation context that contains the reference to \texttt{x} must be at least as permissive as the declaration of \texttt{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 declara-
tions of names in publicly visible annotations, hence public annotations should not contain protected, default access, or private names.

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

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

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

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

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

    //@ 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].
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 a similar name. That is, the declaration of length above can be thought of as equivalent to the following declarations, together with a rewrite of the Java code that uses length to use _length instead (where we assume _length is fresh, i.e., not used elsewhere in the class).

```
//@ 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.

The second rule for visibility prohibits an annotation context from writing specifications in an annotation context that constrain fields that are visible to more clients than the specifications (see section 3 of [Leavens-Mueller07]). In particular, this applies to invariants and history constraints. Thus, for example, a private invariant cannot mention a public field, since clients could see the public field without seeing the invariant, and thus would not know when they might violate the private invariant by assigning to the public field. Thus, for example, the invariants in the following example are all illegal, since they constrain fields that are more visible than the invariant itself.

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

    //@ protected invariant pub > 1;  // illegal!
    //@ invariant pub > 1;           // illegal!
    //@ invariant prot > 1;         // illegal!
    //@ private invariant pub > 1;  // illegal!
    //@ private invariant prot > 1; // illegal!
    //@ private invariant def > 1;  // illegal!
```
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 Declarations], 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 56, 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 block (such as a method body) 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 generally have the values that are defined in the Java Language Specification [Gosling-etal00]. This has consequences on the interpretation of assertion expressions [Chalin07] [Riou-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,

---

\(^1\) By an identifier, we technically mean an ident in the Java grammar. See Section 4.6 [Tokens], page 29, for details.
assertions were interpreted using a classical definition 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 evaluation.

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-circuit nature of the Java operators && and ||. JML also evaluates its two implication operators, => and <= in short-circuit fashion from left to right. 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 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 49),
- formal parameter, (see Section 7.1.1.1 [Formal Parameters], page 46),
- method return type (see Section 7.1.1 [Method and Constructor Declarations], page 45), or
- bound variable (see Section 12.4.24.5 [Modifiers for Bound Variables], page 103)

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 6.2.13 [Nullity Modifiers], page 44, 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.\textsuperscript{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 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

\textsuperscript{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.
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.

Synonyms for the keywords used in level 0 features are also considered to be part of level 0’s lexical syntax. For example, since assignable is a keyword used in level 0, its synonyms modifiable and modifies are also included in the lexical syntax for level 0.

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 6.2.3 [Spec Public], page 41).
- The modifier `spec_protected` (see Section 6.2.4 [Spec Protected], page 41).
- The modifier `instance` (see Section 6.2.8 [Instance], page 42).
- The modifier `model` (see Section 6.2.6 [Model], page 41), as applied to field declarations (see Section 7.1.2.1 [JML Modifiers for Fields], page 49). Note that this modifier as applied to other declarations is not a level 0 feature.
- The modifier `ghost` (see Section 6.2.7 [Ghost], page 42), as applied to both field and variable declarations (see Section 7.1.2 [Field and Variable Declarations], page 49).
- The modifier `helper` (see Section 6.2.9 [Helper], page 42).

Type specifications (see Chapter 8 [Type Specifications], page 52) 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 52) that is either written in an interface using the modifier `instance` (see Section 6.2.8 [Instance], page 42) 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 56).
- The functional form of a represents-clause (see Section 8.4 [Represents Clauses], page 60). That is, a represents clause that uses = and (not \such\_that).
• The *initially-clause* (see Section 8.5 [Initially Clauses], page 61).

• The *type-spec* \( \text{\textbackslash TYPE} \) (optionally, as a type of array element). See Section 7.1.2.2 [Type-Specs], page 50, for more details.

Method specifications (see Chapter 9 [Method Specifications], page 63) 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 71) and specification inheritance [Dhara-Leavens96] [Leavens-Naumann06] [Leavens06b]. It also includes the use of \( \text{\textbackslash not\_specified} \) 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 65), 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 67) that do not use the keyword *code*. This includes *behavior-spec-case* (see Section 9.6 [Behavior Specification Cases], page 67), *normal-behavior-spec-case* (see Section 9.7 [Normal Behavior Specification Cases], page 72), and *exceptional-behavior-spec-case* (see Section 9.8 [Exceptional Behavior Specification Cases], page 73). 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 76). The redundant form of this clause (\( \text{\textbackslash requires\_redundantly} \), \( \text{\textbackslash pre\_redundantly} \)) is a level 1 feature.

• The *ensures-clause* (see Section 9.9.3 [Ensures Clauses], page 77). The redundant form of this clause (\( \text{\textbackslash ensures\_redundantly} \), \( \text{\textbackslash post\_redundantly} \)) is a level 1 feature.

• The *signals-clause* (see Section 9.9.4 [Signals Clauses], page 77). The redundant form of this clause (\( \text{\textbackslash signals\_redundantly} \), \( \text{\textbackslash ensures\_redundantly} \)) is a level 1 feature.

• The *signals-only-clause* (see Section 9.9.5 [Signals-Only Clauses], page 79). The redundant form of this clause (\( \text{\textbackslash signals\_only\_redundantly} \)) is a level 1 feature.

• The *assignable-clause* (see Section 9.9.9 [Assignable Clauses], page 83). The redundant form of this clause (\( \text{\textbackslash assignable\_redundantly} \), \( \text{\textbackslash modifiable\_redundantly} \), \( \text{\textbackslash modifies\_redundantly} \)) is a level 1 feature.

Only static data groups (see Chapter 10 [Data Groups], page 87) are part of level 0.

• The *in-group-clause* (see Section 10.1 [Static Data Group Inclusions], page 87) kind of \( \text{\textbackslash jml\_data\_group\_clause} \) that attaches to field declarations (see Section 7.1.2 [Field and Variable Declarations], page 49).

Some of JML’s extensions to Java’s *expression* syntax (see Chapter 12 [Predicates and Specification Expressions], page 90), but not all of them, can be used at level 0. Note that calls to pure methods and constructors in *spec-expressions* 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 12.4.1 [Backslash result], page 93).

• The *old-expression* (see Section 12.4.2 [Backslash old and Backslash pre], page 93).

• The *fresh-expression* (see Section 12.4.9 [Backslash fresh], page 97).

• The *nonnullelements-expression* (see Section 12.4.14 [Backslash nonnullelements], page 99).
Chapter 2: Fundamental Concepts

- The *informal-description* (see Section 12.4.15 [Informal Predicates], page 99).
- The *typeof-expression* (see Section 12.4.16 [Backslash typeof], page 99).
- The *elatatype-expression* (see Section 12.4.17 [Backslash elatatype], page 99).
- The *type-expression* (see Section 12.4.18 [Backslash type], page 100).
- The *spec-quantified-expr* (see Section 12.4.24 [Quantified Expressions], page 101) forms that use the *quantifier* keywords `\forall` and `\exists` (see Section 12.4.24.1 [Universal and Existential Quantifiers], page 102).
  (The *quantifier* keywords `\max`, `\min`, `\product`, and `\sum` (see Section 12.4.24.2 [Generalized Quantifiers], page 102), as well as `\num_of` (see Section 12.4.24.3 [Numerical Quantifier], page 103, are all level 1 features.)
- The `<:` operator (see Section 12.6.1 [Subtype operator], page 105).
- The `<>` and `<=!=>` operators (see Section 12.6.2 [Equivalence and Inequivalence Operators], page 105).
- The `==>` and `<=` operators (see Section 12.6.3 [Forward and Reverse Implication Operators], page 105).
- The syntax for *store-ref*s (see Section 12.7 [Store Refs], page 106).

All of the Java statements and most of the JML extensions for adding assertions to statements and annotation statements (see Chapter 13 [Statements and Annotation Statements], page 108) 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 extensions to Java statements below.

- Using the modifier *ghost* in *local-declarations* (see Section 13.1.1 [Modifiers for Local Declarations], page 109).
- The *possibly-annotated-loop* statement (see Section 13.2 [Loop Statements], page 109), with a *loop-invariant* (see Section 13.2.1 [Loop Invariants], page 111). The redundant forms of *loop-invariants*, 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 13.3 [Assert Statements], page 113). 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 13.4.1 [Assume Statements], page 114). Use of the keyword *assume_redundantly* is a level 1 feature.
- The *set-statement* (see Section 13.4.2 [Set Statements], page 114).

The ability to use a `.jml` file (see Section 17.1 [File Name Suffixes], page 129) 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.

Some syntax from the Universe type system (see Chapter 18 [Universe Type System], page 133) 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-modifiers (see Section 18.2 [Rep and Peer], page 134).
• The \textit{peer} and \textit{peer} ownership-modifiers (see Section 18.2 [Rep and Peer], page 134).

\subsection*{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 14 [Redundancy], page 118), which are included here for all level 0 features and for other features at level 1.

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

• Method or constructor declarations that use the \textit{modifier} \texttt{model} (see Section 7.1.1.2 [Model Methods and Constructors], page 46). However, note that using \texttt{model} on a field declarations is a level 0 feature and that using \texttt{model} on a type declaration is a level 3 feature.

• \textit{import-declarations} that use the \textit{modifier} \texttt{model} (see Section 5.2 [Import Declarations], page 36).

• The \texttt{modifier} \texttt{pure} (see Section 6.2.5 [Pure], page 41).

• The \textit{modifier} \texttt{uninitialized} (see Section 6.2.11 [Uninitialized], page 43).

The following type-level clauses (see Chapter 8 [Type Specifications], page 52) are included in level 1.

• Attaching a \textit{method-specification} to a \textit{class-initializer-decl} (see Section 7.2 [Class Initializer Declarations], page 50).

• Static invariants, that is an \textit{invariant} (see Section 8.2 [Invariants], page 52) that is either written in an interface without using the \texttt{modifier} \texttt{instance} (see Section 6.2.8 [Instance], page 42), or one that is written in a class and that uses the \textit{modifier} \texttt{static} (see Section 8.2.1 [Static vs. instance invariants], page 56).

• Both instance (object) and static \textit{history-constraints} (see Section 8.3 [Constraints], page 57).

• The \textit{axiom-clause} (see Section 8.6 [Axioms], page 61).

• The \textit{maps-into-clause} (see Section 10.2 [Dynamic Data Group Mappings], page 88) kind of \textit{jml-data-group-clause} that attaches to field declarations (see Section 7.1.2 [Field and Variable Declarations], page 49).

The following features of method specifications (see Chapter 9 [Method Specifications], page 63) are included in level 1.

• The \textit{spec-var-decls} that may occur in a specification case (see Section 9.9.1 [Specification Variable Declarations], page 75).

• The \textit{redundant-spec} parts of a method specification (see Chapter 14 [Redundancy], page 118) are also included in level 1. The following describes these parts.
  • The \textit{implications} (\texttt{implies-that}) part of a \textit{redundant-spec} (see Section 14.1 [Redundant Implications and Redundantly Clauses], page 118).
  • The \textit{examples} (\texttt{for-example}) part of a \textit{redundant-spec}.

The following extensions to Java’s \textit{expression} syntax (see Chapter 12 [Predicates and Specification Expressions], page 90) are included in level 1.
• The `spec-quantified-expr` (see Section 12.4.24 [Quantified Expressions], page 101) forms that use the `quantifier` keywords `\max`, `\min`, `\product`, and `\sum` (see Section 12.4.24.2 [Generalized Quantifiers], page 102), as well as `\num_of` (see Section 12.4.24.3 [Numerical Quantifier], page 103).

(Note that the `\max` quantifier is distinct from the `max-expression` (see Section 12.4.20 [Backslash max], page 100), 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 46) in `spec-expressions` (see Chapter 12 [Predicates and Specification Expressions], page 90).

• The `set-comprehension` expression (see Section 12.5 [Set Comprehensions], page 104).

The following additions to Java’s statement syntax (see Chapter 13 [Statements and Annotation Statements], page 108) are included in level 1.

• The use of redundant forms of `loop-invariants` (see Section 13.2.1 [Loop Invariants], page 111) 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 13.2 [Loop Statements], page 109), with a `variant-function` (see Section 13.2.2 [Loop Variant Functions], page 112).

• The `assert-redundantly-statement` (see Section 13.3 [Assert Statements], page 113); 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 13.4.1 [Assume Statements], page 114); that is, assume statements that use the keyword `assume_redundantly`. The non-redundant `assume-statements` are a level 0 feature.

The `\bigint` type (see Section 19.1 [Backslash bigint], page 140) from the safe math extensions (see Chapter 19 [Safe Math Extensions], page 140) 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 52) are included in level 2.

• The relational form of a `represents-clause` (see Section 8.4 [Represents Clauses], page 60). 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 61).

• The `writable-if-clause` clause (see Section 8.8 [Writable If Clauses], page 62).

The following features of method specifications (see Chapter 9 [Method Specifications], page 63) are included in level 2.

• The `diverges-clause` (see Section 9.9.7 [Diverges Clauses], page 81).
• The accessible-clause (see Section 9.9.10 [Accessible Clauses], page 83).
• The callable-clause (see Section 9.9.11 [Callable Clauses], page 84).
• The measured-by-clause (see Section 9.9.12 [Measured By Clauses], page 84).
• The captures-clause (see Section 9.9.13 [Captures Clauses], page 84).
• The working-space-clause (see Section 9.9.14 [Working Space Clauses], page 85).
• The duration-clause (see Section 9.9.15 [Duration Clauses], page 85).
• The model-program style of method specification (see Chapter 15 [Model Programs], page 122).
• The refining-statement (see Section 13.4.3 [Refining Statements], page 114).
• The extract modifier (see Section 15.2 [Extracting Model Program Specifications], page 124).

The following extensions to Java’s expression syntax (see Chapter 12 [Predicates and Specification Expressions], page 90) are included in level 2.
• The not-assigned-expression (see Section 12.4.3 [Backslash not_assigned], page 94).
• The not-modified-expression (see Section 12.4.4 [Backslash not_modified], page 95).
• The only-accessed-expression (see Section 12.4.5 [Backslash only_accessed], page 95).
• The only-assigned-expression (see Section 12.4.6 [Backslash only_assigned], page 96).
• The only-called-expression (see Section 12.4.7 [Backslash only_called], page 96).
• The only-captured-expression (see Section 12.4.8 [Backslash only_captured], page 97).
• The reach-expression (see Section 12.4.10 [Backslash reach], page 97).
• The is-initialized-expression (see Section 12.4.21 [Backslash is_initialized], page 100).
• The invariant-for-expression (see Section 12.4.22 [Backslash invariant_for], page 101).
• The lblneg-expression and the lblpos-expression (see Section 12.4.23 [Backslash lblneg and lblpos], page 101).

The following additions to Java’s statement syntax (see Chapter 13 [Statements and Annotation Statements], page 108) are included in level 2.
• The unreachable-statement (see Section 13.4.4 [Unreachable Statements], page 115).
• The debug-statement (see Section 13.4.5 [Debug Statements], page 116)
• The hence-by-statement (see Section 13.4.6 [Hence By Statements], page 116).

Note that all the model-prog-statements (see Chapter 15 [Model Programs], page 122) 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 140), which is a level 1 feature, the rest of the safe math extensions (see Chapter 19 [Safe Math Extensions], page 140) are level 2 features. This includes the following particulars.
• The \real type (see Section 19.2 [Backslash real], page 140).
• The modifiers code_bigint_math, code_java_math, code_safe_math, spec_bigint_math, spec_java_math, and spec_safe_math (see Section 6.2.12 [Math Modifiers], page 43).
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-declarations that use the modifier model (see Section 6.1.2 [Modifiers for Type Declarations], page 38).
- The duration-expression (see Section 12.4.11 [Backslash duration], page 98).
- The space-expression (see Section 12.4.12 [Backslash space], page 98).
- The working-space-expression (see Section 12.4.13 [Backslash working space], page 98).

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 62).
- The when-clause (see Section 9.9.8 [When Clauses], page 82).
- The lockset-expression (see Section 12.4.19 [Backslash lockset], page 100).
- The max-expression (see Section 12.4.20 [Backslash max], page 100). Note that this is not the quantifier \max (see Section 12.4.24.2 [Generalized Quantifiers], page 102), which is a level 1 feature.
- The <# and <=# operators applied to test ordering of locks (see Section 12.6.4 [Lockset Ordering], page 106).

2.9.6 Level X Features

The features in this level are experimental. Some of the ones we know about are as follows.

- The \readonly and \readonly ownership-modifiers from the Universe type system (see Chapter 18 [Universe Type System], page 133). 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-declarators, separated by commas.

\[\text{init-declarator-list ::= init-declarator [ , init-declarator ] ...}\]

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.

\[\text{modifiers ::= [ modifier ] ...}\]

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

\[\text{spec-case-seq ::= spec-case [ also spec-case ] ...}\]

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 C [Grammar Summary], page 144). 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 182). Every nonterminal and terminal symbol in the grammar is found in the index, and each definition and use is noted.
Chapter 4: Lexical Conventions

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].

\[
\text{microsyntax ::= lexeme [ lexeme ] \ldots}
\]
\[
\text{lexeme ::= white-space | lexical-pragma | comment}
\]
\[
\text{ | annotation-marker | doc-comment | token}
\]
\[
\text{token ::= ident | keyword | special-symbol}
\]
\[
\text{ | java-literal | informal-description}
\]

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).

\[
\text{white-space ::= non-nl-white-space | end-of-line}
\]
\[
\text{non-nl-white-space ::= a blank, tab, or formfeed character}
\]
\[
\text{end-of-line ::= newline | carriage-return}
\]
\[
\text{ | carriage-return newline}
\]
\[
\text{newline ::= a newline character}
\]
\[
\text{carriage-return ::= a carriage return character}
\]

4.2 Lexical Pragmas

ESC/Java [Leino-et al00] 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.

\[
\text{lexical-pragmas ::= nowarn-pragmas}
\]
\[
\text{nowarn-pragmas ::= nowarn [ spaces ] [ nowarn-label-list ] ;}
\]
\[
\text{spaces ::= non-nl-white-space [ non-nl-white-space ] \ldots}
\]
\[
\text{nowarn-label-list ::= nowarn-label [ spaces ]}
\]
\[
\text{ | [ , [ spaces ] nowarn-label [ spaces ] ] \ldots}
\]
\[
\text{nowarn-label ::= letter [ letter ] \ldots}
\]

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 sequence of annotation-keys and an at-sign (@), then JML considers it to be the start of an annotation (see Section 4.4 [Annotation Markers], page 27), 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-minus-star [ non-stars-slash ] . . .
   | + non-letter [ non-stars-slash ] . . .
   | - non-letter [ non-stars-slash ] . . .
   | stars-non-slash [ non-stars-slash ] . . .
non-letter ::= any character except _, a through z, or A through Z
non-stars-slash ::= non-star
   | stars-non-slash
stars-non-slash ::= * [ * ] . . . non-star-slash
non-at-plus-minus-star ::= any character except @, +, -, or *
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-minus-end-of-line [ non-end-of-line ] . . . end-of-line
   | //+ non-letter-end-of-line [ non-end-of-line ] . . . end-of-line
   | //- non-letter-end-of-line [ non-end-of-line ] . . . end-of-line
non-letter-end-of-line ::= any character except _, a through z, A through Z, a newline, or a carriage return
non-end-of-line ::= any character except a newline or carriage return
non-at-plus-minus-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, or starts with a sequence of annotation-keys followed by an at-sign, 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 (/@) and end (*/ or @*/ ) markers as annotations. The definition of an annotation marker is given below.

```
annotation-marker ::= // [ annotation-key ]...@ [ ignored-at-in-annotation ] . . .
   | /* [ annotation-key ]...@ [ ignored-at-in-annotation ] . . .
   | [ ignored-at-in-annotation ] . . . @*/
   | [ ignored-at-in-annotation ] . . . */
annotation-key ::= positive-key | negative-key
positive-key ::= + ident
```
negative-key ::= - ident
ignored-at-in-annotation ::= @

Within annotations, on each line, initial white-space and any immediately following at-signs (@) are ignored.

Note that JML annotations are not the same as Java annotations (see Section 6.2.2 [Java Annotations], page 41). Besides the syntactic differences, JML annotations can appear anywhere a comment may appear, not just attached to declarations.

An annotation-key is a + or - sign followed by an ident (see Section 4.6 [Tokens], page 29). Note that no white space can appear within, before, or after the annotation-key. Tools will provide a way to enable a selection of annotation-key identifiers. These identifiers, hereafter called “keys” provide for conditional inclusion of JML annotations as follows:

- a JML annotation with no keys is always included,
- a JML annotation with at least one positive-key is only included if at least one of these positive keys is enabled and there are no negative-keys in the annotation that have enabled keys, and
- a JML annotation with an enabled negative-key is ignored (even if there are enabled positive-keys).

For example, a comment beginning with //+ESC@ is included as a JML annotation only if the ESC key is enabled; a comment beginning with //-ESC@ is included except when the ESC key is enabled.

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;

However, implementations are not required to check for such errors. On the other hand, 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, as well as the ignored-at-in-annotations. However, note that this technique does not properly check for annotations that do not contain entire grammatical units of JML specifications, as described in the previous paragraph.

Note that JML will recognize jml-keywords only within JML annotations.

### 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 ] . . .
jml-specs ::= 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.

```
doc-at-sign ::= @
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.
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 supported by the tool in question, hopefully the latest official release).

The `jml-keyword` s 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. For example, 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
java-universe-reserved // When the Universe option is on

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 | \lbneg | \lbpos
| \lockset | \max | \min
| \nonnullelements | \not_assigned
| \not_modified | \not_specified
| \nothing | \nowarn | \nowarn_op
| \num_of | \old | \only_accessed
| \only_assigned | \onlyCalled
| \only_captured | \pre
| \product | \reach | \real
| \result | \same | \safe_math
| \space | \such_that | \sum
| \typeof | \type | \warn_op
| \warn | \working_space

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
| ensures | ensures_redundantly | extract
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 ::= ( | ) | { | } | "[' | ']'" | ; | , | . | @
java-operator ::= == | <= | >= | != | && | || | ++ | -- |
                | + | - | * | / | & | | | | | | |
                | += | -= | *= | /= | &= | ^= | %= |
                | <<= | >>= | >>>=
```

```
jml-universe-keyword ::= peer | readonly | rep
```

```
| 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 |
| 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 |
| when | when_redundantly |
| working_space | working_space_redundantly |
| writable |
| jml-universe-keyword |

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].
jml-special-symbol ::= ==> | <= | <==> | <=!=>
| -> | <- | <: | . . . | ‘{‘ | ‘|’}
| <# | <=#

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

\textit{java-literal} ::= integer-literal
\hspace{1em} | floating-point-literal | boolean-literal
\hspace{1em} | character-literal | string-literal | null-literal

\textit{integer-literal} ::= decimal-integer-literal
\hspace{1em} | hex-integer-literal | octal-integer-literal
\textit{decimal-integer-literal} ::= non-zero-digit \[ digits \] \[ integer-type-suffix \]
digits ::= digit \[ digit \] \ldots
digit ::= 0 \| non-zero-digit
non-zero-digit ::= 1 \| 2 \| 3 \| 4 \| 5 \| 6 \| 7 \| 8 \| 9
\textit{integer-type-suffix} ::= 1 \| L
\textit{hex-integer-literal} ::= hex-numeral \[ integer-type-suffix \]
\textit{hex-numeral} ::= 0x hex-digit \[ hex-digit \] \ldots
\hspace{1em} | 0X hex-digit \[ hex-digit \] \ldots
\textit{hex-digit} ::= digit | a | b | c | d | e | f
\hspace{1em} | A | B | C | D | E | F
\textit{octal-integer-literal} ::= octal-numeral \[ integer-type-suffix \]
octal-numeral ::= 0 octal-digit \[ octal-digit \] \ldots
octal-digit ::= 0 \| 1 \| 2 \| 3 \| 4 \| 5 \| 6 \| 7

\textit{floating-point-literal} ::= digits . \[ digits \]
\hspace{1em} | exponent-part \[ float-type-suffix \]
\hspace{1em} | . digits exponent-part \[ float-type-suffix \]
\hspace{1em} | digits exponent-part \[ float-type-suffix \]
\hspace{1em} | digits exponent-part float-type-suffix
\textit{exponent-part} ::= exponent-indicator signed-integer
\textit{exponent-indicator} ::= e | E
\textit{signed-integer} ::= [ sign ] digits
\textit{sign} ::= + | -
\textit{float-type-suffix} ::= f | F | d | D

\textit{boolean-literal} ::= true | false

\textit{character-literal} ::= ‘ single-character ‘ | ‘ escape-sequence ‘
\textit{single-character} ::= any character except ‘, \\, carriage return, or newline
\textit{escape-sequence} ::= \b \hspace{1em} // backspace
\hspace{1em} \t \hspace{1em} // tab
\hspace{1em} \n \hspace{1em} // newline
\hspace{1em} \r \hspace{1em} // carriage return
\hspace{1em} \’ \hspace{1em} // single quote
\hspace{1em} \” \hspace{1em} // double quote
Chapter 4: Lexical Conventions

```
| \ \                       // 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 12.1 [Predicates], page 90) and in store-ref expressions (see Section 12.7 [Store Refs], page 106) 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 *
```
Chapter 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-declaration ]
  [ import-declaration ] . . .
  [ top-level-declaration ] . . .

top-level-declaration ::= type-declaration
```

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 Declarations], page 37, for the syntax and semantics of `type-declarations`.

See Chapter 17 [Separate Files for Specifications], page 129, for a discussion of how you can place JML specification in separate (.jml) files.

Some JML tools may support various optional extensions to JML. This manual partially describes one such extension: 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 the Universe type system, selected by any options); in particular it must obey all of Java's static restrictions. For example, at most one of the type declarations in a compilation unit may be declared `public`. See the Java Language Specification [Gosling-etal05] 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 17.1 [File Name Suffixes], page 129, for details on the file name suffixes allowed in JML and how the extra files determine the specification for the compilation unit.

The specification of the compilation unit consists of the specifications of the `top-level-declarations` it contains, placed in the declared package (if any). The interface part of this specification is determined as in Java [Gosling-etal05] (or as in the Java extension used). The specifications of each `type-declaration` 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-etal05]. In JML, not only is the package `java.lang` implicitly imported, but also there is an implicit model import of `org.jmle specs.lang`. (See Section 5.2 [Import Declarations], page 36, for the meaning of a model import.)

A Java compilation unit satisfies such a JML specification if it satisfies the specified `package-declaration` (if any), and if for each specified `type-declaration`, there is a corresponding Java `type-declaration` 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-declarations` and `import-declarations` are discussed in the subsections below.

### 5.1 Package Declarations

The syntax of a `package-declaration` is as in Java (see Section 7.4 of [Gosling-etal05]). See Section 6.2.2 [Java Annotations], page 41, for the syntax of `java-annotations`.

```
package-declaration ::= [ java-annotations ] package name ;
name ::= ident [ . ident ] . . .
```

A Java package declaration satisfies the JML specification only if its `java-annotations` are satisfied by the declaration and if the remainder of the package declaration is the same as that specified. That is, the Java code has to be the same (modulo whitespace) as the JML specification.

### 5.2 Import Declarations

The syntax of a `import-declaration` is as follows. The only difference from the Java syntax [Gosling-etal05] is the optional `model` modifier.

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

An `import-declaration` may use the `model` modifier if and only if the whole `import-declaration` 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-declaration` without the `model` keyword is the same as in Java (Section 7.5 of [Gosling-etal05]). Checking of such import declarations is done exactly as in Java. Such import declarations 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 Declarations

The following is the syntax of type declarations.

\[
\text{type-declaration ::= class-declaration} \\
| \text{interface-declaration} \\
| ;
\]

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

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

6.1 Class and Interface Declarations

Class and interface declarations are quite similar, as interfaces may be seen as a special kind of class declaration that only allows the declaration of abstract instance methods and final static fields (in the Java code, see Chapter 9 of [Gosling-etal05]). Their syntax is also similar.

\[
\text{class-declaration ::= [ doc-comment ] modifiers class ident} \\
| [ class-extends-clause ] [ implements-clause ] \\
| class-block
\]

\[
\text{class-block ::= \{ [ field ] \ldots \}}
\]

\[
\text{interface-declaration ::= [ doc-comment ] modifiers interface ident} \\
| [ interface-extends ] \\
| class-block
\]

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 45, 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 Declarations

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

\[
\text{class-extends-clause ::= [ extends name ]}
\]

\[
\text{implements-clause ::= implements name-list}
\]

\[
\text{name-list ::= name [ , name ] ...}
\]

\[
\text{interface-extends ::= extends name-list}
\]

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 addition, every type in Java is a subtype of `Object`. In particular every class $S$ and every interface $U$ is a subtype of `Object`.

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 along with 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].

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, specifications of supertypes are inherited by subtypes, and thus must be obeyed by subtypes. This forces subtypes to be behavioral subtypes [Dhara-Leavens96] [Leavens-Naumann06] [Leavens06b]. See Chapter 11 [Specification Inheritance], page 89, for details about specification inheritance and behavioral subtyping.

### 6.1.2 Modifiers for Type Declarations

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

```plaintext
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.

The modifiers `spec_java_math`, `spec_safe_math`, and `spec_bigint_math` are mutually exclusive. They declare that all the math used in the type’s specifications uses the rules of Java’s math, safe math, or bigint math, respectively. The modifiers `code_java_math`, `code_safe_math`, and `code_bigint_math` are also mutually exclusive. They say that the math used in the type’s Java code uses the rules of Java’s math, safe math, or bigint math, respectively. See Section 6.2.12 [Math Modifiers], page 43, for more details on these modifiers.

We discuss the use of `pure` and `model` on type declarations below.
6.1.2.1 Pure Type Declarations

A type declaration 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 46, for more about pure methods). However, its static methods may still have side effects in a type declared with pure, as the pure does not apply to the static methods declared in a type. So, in essence, declaring a class pure is merely a shorthand for declaring all of the constructors and instance methods declared in that class pure.

Although an override of a pure method must be pure, instance methods declared in subtypes that do not override a pure supertype’s methods need not be pure. Hence, some methods of such a subtype object may mutate the objects of such a subtype. In other words, such a subtype does not necessarily have immutable objects.

However, if one is careful, once an object of class declared to be pure is created, such an object will be immutable, since none of its instance methods will have any side effects. Being careful to avoid problems means first that the class’s fields must be either final or encapsulated (e.g., declared as private, to avoid direct mutation by clients), and second that the type’s constructors and methods, must either avoid representation exposure (see Section 18.1 [Basic Concepts of Universes], page 134) or all of its fields must be immutable objects or values (such as integers).

6.1.2.2 Model Type Declarations

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 declaration must be contained within an annotation comment, and consequently annotations within the type declaration do not need to be separately enclosed in annotation comments.

The scope rules for a model type declaration are the same as for Java type definitions, except that a model type declaration is not in scope for any Java code, only for annotations.

Types declared with the keyword model are seldom used in JML. If a tool does not support such types, one can always just define a Java type, which will also be useful in runtime assertion checking.

Various authors refer to “model types” when they really mean “types with modifier pure that are used for modeling.” Such a usage is contrary to JML’s notion of a type with a model modifier.

6.2 Modifiers

The following is the syntax of modifiers.

\[
\text{modifiers ::= [ modifier ] \ldots}
\]

\[
\text{modifier ::= public | protected | private}
\]

\[
| \text{abstract | static | final | synchronized}
\]

\[
| \text{transient | volatile | native | strictfp}
\]

\[
| \text{const} \quad \text{// reserved but not used in Java}
\]
Chapter 6: Type Declarations

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 D [Modifier Summary], page 163.

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 (see, for example section 8 of [Gosling-etal00], which is enforced by Checkstyle). 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).

```
| java-annotation
| 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
```

### 6.2.2 JML-specific Modifiers

- `spec_public`: Public annotations (not in JML).
- `spec_protected`: Protected annotations (not in JML).
- `spec_java_math`: Java math annotations.
- `spec_safe_math`: Safe math annotations.
- `spec_bigint_math`: Big integer math annotations.
- `code_java_math`: Code annotations for Java math.
- `code_safe_math`: Code annotations for safe math.
- `code_bigint_math`: Code annotations for big integer math.
- `non_null`: Non-null annotations.
- `nullable`: Nullable annotations.
- `nullable_by_default`: Nullable by default annotations.
- `extract`: Extract annotations.
peer rep readonly

6.2.2 Java Annotations

A Java annotation (see section 9.7 of [Gosling-etal05]) has the following syntax. Note that these are quite different from JML annotations (see Section 4.4 [Annotation Markers], page 27).

```
java-annotations ::= java-annotation [ java-annotation ] . . .
java-annotation ::= @ name ( [ element-value-pairs ] . . . )
                 | @ name ( element-values )
element-value-pairs ::= element-value [ , element-value ]
element-value-pair ::= ident = element-value
element-value ::= conditional-expr
                 | annotation
                 | element-value-array-initializer
element-value-array-initializer ::= `{ element-values }` `{`
element-values ::= element-value [ , element-value ] . . . [ , ]
```

A Java annotation forms part of the Java interface of a declaration, and as such is only satisfied by an implementation if the implementation contains the same annotation. Its semantics and checking is exactly as in Java.

6.2.3 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.

6.2.4 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.

6.2.5 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 declaration is shorthand for applying that modifier to all constructors and instance methods in the type (see Section 6.1.2 [Modifiers for Type Declarations], page 38).

See Section 7.1.1.3 [Pure Methods and Constructors], page 46, for the exact semantics of pure methods and constructors.

6.2.6 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 6.2.7 [Ghost], page 42.

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.

### 6.2.7 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.

### 6.2.8 Instance

The `instance` modifier says that a field is not static. See Section 2.5 [Instance vs. Static], page 15.

### 6.2.9 Helper

The `helper` modifier may be used on a method that is either pure or private or on a private constructor to say that its specification is not augmented by invariants and history constraints that would otherwise be relevant. That is, when a method or constructor is declared with the `helper` modifier, no invariants or history constraints apply to it (see Section 7.1.1.4 [Helper Methods and Constructors], page 48). Thus `helper` makes such a method or constructor an exception to the general rule that each invariant must be obeyed by all methods in a class or interface and its subtypes. (see Section 8.2 [Invariants], page 52). Similar remarks apply to helper methods and history constraints.

The paper “Information Hiding and Visibility in Interface Specifications” [Leavens-Mueller07] describes why helper methods must be `private`. Essentially, the reason is that a helper method (or constructor) may violate various invariants, and all of the potentially violated invariants “must be visible wherever the helper method is visible” (Rule 7 in Section 5.2 of [Leavens-Mueller07]). The only way to guarantee that in all cases all such invariants are visible is to force all helper methods to be private. However, the discussion in that
paper did not consider pure methods, and it is sometimes helpful to make a pure method a
helper method, so this case is also allowed.

6.2.10 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].

6.2.11 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].

6.2.12 Math Modifiers
The modifiers spec_java_math, spec_safe_math, spec_bigint_math, code_java_math,
code_safe_math, and code_bigint_math describe what math modes are used for specifi-
cations (spec_. . . ) and for Java code (code_. . . ) [Chalin04]. For each of these two dimensions
of specifications and code, there are three kinds of math modes.
• In Java math mode, integral arithmetic can cause silent wrap-around of computations.
  For example 1+Integer.MAX_VALUE will equal 1+Integer.MIN_VALUE. This is the de-
  fault math mode.
• In safe math mode, wrap-around of integral arithmetic causes exceptions. For example
  1+Integer.MAX_VALUE will throw an exception.
• In bigint math mode, integral arithmetic is considered to use infinite precision integers,
  and thus there is no wrap-around. (This semantics will be approximated in runtime
  assertion checking by using integers that can grow to very large numbers, limited only
  by the size of a computer’s memory.) For example 1+Integer.MAX_VALUE will be the
  next integer past Integer.MAX_VALUE.

The spec_ modifiers spec_java_math, spec_safe_math, and spec_bigint_math de-
termine the kind of mathematics used in specifications at the level in which the modifier
appears. For example, if the modifier spec_java_math is used on a type declaration, all
arithmetic used in specifications written in that type use Java math mode. Similarly, if the
modifier spec_java_math is used on a method declaration, then Java math mode will be
used for the specifications written in that file for that method. The mode spec_java_math is
the default math used in specifications, used if neither spec_safe_math, nor spec_bigint_
math are given. Within a type marked with one of these modifiers, individual method or
constructors can have one of the other modifiers, which is used for that method or con-
structor’s specifications in that file. Similarly, spec_safe_math specifies that safe math
mode will be used for the specifications in the type or method to which it is attached, and
spec_bigint_math specifies that bigint math mode will be used for the specifications in
the type or method to which it is attached.

The modifiers code_java_math, code_safe_math, and code_bigint_math are similar
to the specification math modes, but describe the way that the Java code used in an im-
plementation is compiled. For example, code_java_math specifies that the type or method
to which the modifier is attached is to be compiled using the default Java math mode.
For example, `code_safe_math` specifies that the type or method to which the modifier is attached is to be compiled using the safe math mode, and `code_bigint_math` specifies that the type or method to which the modifier is attached is to be compiled using bigint math mode.

These modes are level 2 features of JML. See Chalin’s paper [Chalin04] for more details on the use of these modes.

### 6.2.13 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 16).

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 Declarations], page 37).

\[
\text{field} ::= \text{member-decl} \\
| \text{jml-declaration} \\
| \text{class-initializer-decl} \\
| ;
\]

Also see Section G.2.1 [Non-null by Default], page 169. 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 52, 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.

\[
\text{member-decl} ::= \text{method-decl} \\
| \text{variable-definition} \\
| \text{class-declaration} \\
| \text{interface-declaration}
\]

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

7.1.1 Method and Constructor Declarations

The following is the syntax of a method declaration.

\[
\text{method-decl} ::= [ \text{doc-comment} ] . . . \\
\text{method-specification} \\
\text{modifiers} [ \text{method-or-constructor-keyword} ] \\
[ \text{type-spec} ] \text{method-head} \\
\text{method-body} \\
| [ \text{doc-comment} ] . . . \\
\text{modifiers} [ \text{method-or-constructor-keyword} ] \\
[ \text{type-spec} ] \text{method-head} \\
[ \text{method-specification} ] \\
\text{method-body}
\]

\[
\text{method-or-constructor-keyword} ::= \text{method} | \text{constructor}
\]

Notice that the specification of a method (see Chapter 9 [Method Specifications], page 63) 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 15.2 [Extracting Model Program Specifications], page 124, for an explanation of this modifier.

7.1.1.1 Formal Parameters

\[
\begin{align*}
formals & ::= ( [ \text{param-declaration-list} ] ) \\
\text{param-declaration-list} & ::= \text{param-declaration} \\
& \quad [ , \text{param-declaration} ] \ldots \\
\text{param-declaration} & ::= [ \text{param-modifier} ] \ldots \text{type-spec} \ \text{ident} \ [ \text{dims} ] \\
\text{param-modifier} & ::= \text{final} | \text{non_null} | \text{nullable}
\end{align*}
\]

See Section 7.1.2.2 [Type-Specs], page 50, for more about the nonterminals `type-spec` and `dims`.

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 separate file (see Chapter 17 [Separate Files for Specifications], page 129), and thus need not appear originally in the Java source code. 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 (although there is no particular purpose served by making a constructor final, since constructors are not overridden in any case).

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 15 [Model Programs], page 122). 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

\[
\text{diverges } false; \\
\text{assignable } \text{nothing};
\]

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 83), although doing so may improve the specification’s clarity.

A pure constructor has the clauses

\[
\text{diverges } false; \\
\text{assignable } \text{this.}*;
\]

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 theorem provers by making pure methods more like mathematical functions.

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

\[
\text{private behavior} \\
\text{requires } true; \\
\text{diverges } false; \\
\text{assignable } \text{nothing};
\]

(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 \texttt{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 \texttt{pure} on each method declared in a class and interface, one can use the modifier \texttt{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 \texttt{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 \texttt{java.lang.Object}, which has some methods, such as \texttt{notify} and \texttt{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 \texttt{pure} modifier as \texttt{pure}.

In JML the modifiers \texttt{model} and \texttt{pure} are orthogonal. (Recall something declared with the modifier \texttt{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 \texttt{void}, then it can essentially only do nothing. So, while pure methods with \texttt{void} as their return type are not illegal, they are useless.

\textbf{7.1.1.4 Helper Methods andConstructors}\n
The \texttt{helper} modifier may only be used on a private method or constructor [Leavens-Mueller07]. See Section 6.2.9 [Helper], page 42, for more on why such methods and constructors must be \texttt{private}. 

A method or constructor with the helper modifier, 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 an abbreviation device. However, whatever specifications are given explicitly for such a method or constructor still apply. See Section 8.2 [Invariants], page 52, for more details.

### 7.1.2 Field and Variable Declarations

The following is the syntax of field and variable declarations.

\[
\text{variable-definition ::= } [ \text{doc-comment} ] \ldots \text{modifiers variable-decls} \\
\text{variable-decls ::= } [ \text{field} ] \text{type-spec variable-declarators} ; \\
\text{variable-declarators ::= } \text{variable-declarator} \\
\hspace{1em} [ , \text{variable-declarator} ] \ldots \\
\text{variable-declarator ::= } \text{ident} [ \text{dims} ] [ = \text{initializer} ] \\
\text{initializer ::= expression | array-initializer} \\
\text{array-initializer ::= } \{ [ \text{initializer-list} ] \} \\
\text{initializer-list ::= } \text{initializer} [ , \text{initializer} ] \ldots [ , ]
\]

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 ‘.jml’ (see Chapter 17 [Separate Files for Specifications], page 129), 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 87, for more about jml-data-group-clauses. See Section 12.2 [Specification Expressions], page 90, for the syntax of expression. In the following we discuss the modifiers for field and variable declarations and type-specs.

### 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 13 [Statements and Annotation Statements], page 108), 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 60). 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 52, 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 15.

To declare a static field in an interface, one omits the instance modifier; such a field, as in Java is both static and final.

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 133, for how to do that, and for the syntax and semantics of ownership-modifiers.

\[
type-spec ::= [ ownership-modifiers ] type [ dims ] \\
\ | \TYPE [ dims ]
\]

\[
type ::= reference-type | built-in-type
\]

\[
reference-type ::= name
\]

\[
\]

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 63), which treats the initializer as a private helper method with return type void, whose body is given by the compound-statement (see Chapter 13 [Statements and Annotation Statements], page 108).

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 17 [Separate Files for Specifications], page 129). 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 63).

Note that, due to this semantics, the `method-specifications` 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. - DRCOek ]]]
Chapter 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 45); 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-declarations to further refine the logical model defined by these fields. These jml-declarations can also be used to record various design and implementation decisions.

The syntax of these jml-declarations is as follows.

\[ jml-declaration ::= \begin{align*}
&\text{modifiers} \quad \text{invariant} \\
&| \text{modifiers} \quad \text{history-constraint} \\
&| \text{modifiers} \quad \text{represents-clause} \\
&| \text{modifiers} \quad \text{initially-clause} \\
&| \text{modifiers} \quad \text{monitors-for-clause} \\
&| \text{modifiers} \quad \text{readable-if-clause} \\
&| \text{modifiers} \quad \text{writable-if-clause} \\
&| \text{axiom-clause}
\end{align*} \]

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.

\[ invariant ::= \text{invariant-keyword} \quad \text{predicate} \ ; \]

\[ \text{invariant-keyword} ::= \text{invariant} \quad | \text{invariant\_redundantly} \]

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.

```java
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, which are constructors or methods declared with the helper modifier (see Section 9.6.4 [Semantics of helper methods and constructors], page 71). 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. Note that objects of subtypes of type T also have 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 48.

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 56.
- 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 57.
- 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 56.
- A class inherits all visible invariants specified in its superclasses and superinterfaces; see Section 8.2.4 [Invariants and Inheritance], page 57.
- 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 63.
- 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 language. 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 52), 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:

<table>
<thead>
<tr>
<th>static</th>
<th>non-helper static method</th>
<th>non-helper constructor</th>
<th>non-helper instance method</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>static invariant</td>
<td>establish</td>
<td>preserve</td>
<td>preserve</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instance</td>
<td>(irrelevant)</td>
<td>(irrelevant)</td>
<td>establish</td>
</tr>
</tbody>
</table>

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

---

1 Thanks to Peter Müller for clarifying this paragraph.
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 seem excessively strong. However, it is the only sound option in the long run. After 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.

As noted in See Section 2.4 [Privacy Modifiers and Visibility], page 12, there are restrictions on the visibility of fields that can be referenced in invariants to prevent specifications that clients cannot understand and to prevent invariants that clients cannot preserve. Thus, for example, private invariants cannot mention public fields [Leavens-Mueller07].

### 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.
Chapter 8: Type Specifications

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.

```java
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 59.
- Constraints can be declared with the access modifiers public, protected, and private; see Section 8.3.2 [Access Modifiers for Constraints], page 60.
- 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 60.
- 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 (i.e. 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 63.
- 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:

<table>
<thead>
<tr>
<th>static</th>
<th>non-helper</th>
<th>non-helper</th>
<th>non-helper</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>static method</td>
<td>constructor</td>
<td>instance method</td>
</tr>
<tr>
<td>static</td>
<td>(irrelevant)</td>
<td>respect</td>
<td>respect</td>
</tr>
</tbody>
</table>
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 57.

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 = spec-expression ;
| represents-keyword store-ref-expression \such_that predicate ;
represents-keyword ::= represents | represents_redundantly

The first form of represents clauses 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 =.

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 45, 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.

For each type and model field, there can be at most one non-redundant represents-clause that in the type that has the given model field in its left-hand side. A represents clause is redundant if it is introduced using the keyword represents_redundantly.
A *represents-clause* can be declared as *static* (See Chapter 6 [Type Declarations], 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 Declarations], 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.

\[
\text{initially-clause ::= } \text{initially } \text{predicate} ;
\]

The meaning of an *initially-clause* is that each non-helper (see Section 6.2.9 [Helper], page 42) 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 of such a type and all of its subtypes.

### 8.6 Axioms

An *axiom-clause* has the following syntax.

\[
\text{axiom-clause ::= } \text{axiom } \text{predicate} ;
\]

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.

\[
\text{readable-if-clause ::= } \text{readable } \text{ident if } \text{predicate} ;
\]

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.

\[\text{writable-if-clause ::= writable ident if predicate ;}\]

Such a clause gives a condition that must be true before the field named by \text{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 \text{monitors-for-clause} is adapted from ESC/Java [Leino-Nelson-Saxe00] [Rodriguez-etal05]. It has the following syntax.

\[\text{monitors-for-clause ::= monitors_for ident = spec-expression-list ;}\]

A \text{monitors-for-clause} such as \text{monitors_for f <- e1, e2;} specifies a relationship between the field, \text{f} and a set of objects, denoted by a specification expression list \text{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 \text{e1} and \text{e2}, must be locked to read the field (\text{f} in the example) in this object.

Note that the righthand-side of the \text{monitors-for-clause} is not just a \text{store-ref-list}, but is in fact a \text{spec-expression-list}, where each \text{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 15 [Model Programs], page 122.

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 14 [Redundancy], page 118.

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 satisfied by all implementations.

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

\[
\text{spec-case ::= lightweight-spec-case | heavyweight-spec-case | model-program}
\]

Model programs are discussed in Chapter 15 [Model Programs], page 122. 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.

\[
\text{privacy ::= public | protected | private}
\]

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.

\[
\begin{align*}
\text{lightweight-spec-case} & \ ::= \text{generic-spec-case} \\
\text{generic-spec-case} & \ ::= \ [ \text{spec-var-decls} ] \\
& \quad \text{spec-header} \\
& \quad \ [ \text{generic-spec-body} ] \\
\text{spec-header} & \ ::= \text{requires-clause} \\
\text{generic-spec-body} & \ ::= \text{simple-spec-body} \\
& \quad \ {\} \text{generic-spec-case-seq} \ {\} \\
\text{generic-spec-case-seq} & \ ::= \text{generic-spec-case} \\
& \quad \text{also} \text{generic-spec-case} \ ... \\
\text{simple-spec-body} & \ ::= \text{simple-spec-body-clause} \\
& \quad \text{diverges-clause} \\
& \quad \text{assignable-clause} \ | \text{accessible-clause} \\
& \quad \text{captures-clause} \ | \text{callable-clause} \\
& \quad \text{when-clause} \ | \text{working-space-clause} \\
& \quad \text{duration-clause} \ | \text{ensures-clause} \\
& \quad \text{signals-only-clause} \ | \text{signals-clause} \\
& \quad \text{measured-clause} \\
\end{align*}
\]

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 67) is that the latter has the keyword \textit{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 64. 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 \texttt{m} in class \texttt{Lightweight} below

\begin{verbatim}
package org.jmlspecs.samples.jmlrefman;

public abstract class Lightweight {

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

    /*@ requires P;*/
\end{verbatim}
has a specification that is equivalent to that of method \texttt{m} in class \texttt{Heavyweight} below.

```java
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;
     * @ 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 \texttt{\not_specified} for all clauses, except \texttt{diverges}, which has a default of \texttt{false}, and
\texttt{signals} [Leavens-Baker-Ruby06]. The default for an omitted \texttt{signals} clause is to only
permit the exceptions declared in the method's header to be thrown. Thus, if the method
declares that exceptions \texttt{DE1} and \texttt{DE2} may be thrown, then the default for an omitted
\texttt{signals} clause is

\[
\text{signals (Exception e) e instanceof DE1 || e instanceof DE2;}
\]

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

\[
\text{heavyweight-spec-case ::= behavior-spec-case} \\
| \text{exceptional-behavior-spec-case} \\
| \text{normal-behavior-spec-case}
\]

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.

\[
\text{behavior-spec-case ::= [privacy] [code] behavior-keyword} \\
\text{generic-spec-case}
\]

\[
\text{behavior-keyword ::= behavior | behaviour}
\]

See Section 16.2 [Code Contracts], page 127, 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
  \[
  \text{behavior} \left[ \text{spec-var-decls} \right] \left[ \text{spec-header} \right] \text{simple-spec-body}
  \]
  
  A flat specification case is just made up of a sequence of method specification clauses, i.e. `require`, `ensures`, etc. clauses, and its semantics is explained directly in Section 9.6.1 [Semantics of flat behavior specification cases], page 68.

- **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 71.

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 52 and Section 8.3 [Constraints], page 57. 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 `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 75, any `behavior` specification case is equivalent with a `behavior` specification case of this form.

9.6.2 Semantics of non-helper methods

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

\[
\text{behavior}
\begin{align*}
  & \text{forall } T1 \ x1; \ldots \ forall \ Tn \ xn; \\
  & \text{old } U1 \ y1 = F1; \ldots \ old \ Uk \ yk = Fk; \\
  & \text{requires } P; \\
  & \text{measured_by } Mbe \ if \ Mbp; \\
  & \text{diverges } D; \\
  & \text{when } W; \\
  & \text{accessible } R; \\
  & \text{assignable } A; \\
  & \text{callable } p1(\ldots), \ldots, p1(\ldots); \\
  & \text{captures } Z;
\end{align*}
\]
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, \( x_1, \ldots, x_n \), 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 \( y_1 \) is bound to the pre-state value of \( F_1 \) 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 \( y_k \) is bound to the value of \( F_k \) 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, \( \text{Caller} \), 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 \( \text{Caller} \). (If the measured_by clause is omitted, there is no such requirement.) For example, consider a method \( \text{fib} \) 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 \( \text{fib} \) that appear in the body of \( \text{fib} \) must have actual argument expressions whose value is (non-negative and) strictly less than \( n \), such as \( n-1 \) and \( n-2 \).

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 87) 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 87) 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 $p1, \ldots, pl$. (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 $E1, \ldots, Ea$, 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 \texttt{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 $\texttt{old()}$ to refer to pre-state values. If the \texttt{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 \texttt{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 $\texttt{old()}$ to refer to pre-state values. If the \texttt{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 80.

9.6.3 Semantics of 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 52 and Section 8.3 [Constraints], page 57.

9.6.4 Semantics of 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 52 and Section 8.3 [Constraints], page 57.

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 \texttt{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.

\[
\text{spec-var-decls} \\
\text{spec-header} \\
\{ \\
\begin{array}{l}
\text{GenSpecCase1} \\
\text{also} \\
\text{...} \\
\text{also} \\
\text{GenSpecCasen}
\end{array} \\
\}
\]

The above desugars to the following.

\[
\text{spec-var-decls} \\
\text{spec-header} \\
\text{GenSpecCase1} \\
\text{also} \\
\text{...} \\
\text{also} \\
\text{spec-var-decls} \\
\text{spec-header} \\
\text{GenSpecCasen}
\]

In the above desugaring either the \text{spec-var-decls} or the \text{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 63. The meaning of a single simple specification case is explained in Section 9.6.1 [Semantics of flat behavior specification cases], page 68.

### 9.7 Normal Behavior Specification Cases

A \text{normal\_behavior} specification case is just syntactic sugar for a \text{behavior} specification case with an implicit \text{signals} clause

\[
\text{signals (java.lang.Exception) false;}
\]

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

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

\[
\text{normal\_behavior\_spec\_case ::= [ privacy ] [ code ] normal\_behavior\_keyword} \\
\text{normal\_spec\_case} \\
\text{normal\_behavior\_keyword ::= normal\_behavior | normal\_behaviour} \\
\text{normal\_spec\_case ::= generic\_spec\_case}
\]

As far as syntax is concerned, the only difference between a \text{normal\_spec\_case} and a \text{generic\_spec\_case} is that normal behavior specification cases cannot include \text{signals\_clauses} or \text{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 67) with the addition of the following signals-clause

```java
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.

```
excepiton-behavior-spec-case ::= [privacy] [code] exceptional-behavior-keyword

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 67) with the addition of the following ensures clause.

```java
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 must 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

```java
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 \( z \leq 99 \) 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

```java
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.

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

public abstract class SignalsClause {

    /*@ signals (IllegalArgumentEception) 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 65, for the overall syntax that ties these clauses together.

### 9.9.1 Specification Variable Declarations

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

\[
\text{spec-var-decls} ::= \text{forall-var-decls} \mid \text{old-var-decls} 
\]

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.

\[
\text{forall-var-decls} ::= \text{forall-var-declarator} \mid \text{forall-var-declarator} \ldots
\]

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. See Section 12.4.24 [Quantified Expressions], page 101, for the syntax of quantified-var-declarator.

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 have the same name as earlier bound variables in the specification, nor may it shadow 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 50, for the syntax of type-spec. \([[\text{Give cross ref for spec-variable-declarators when ready.}]]\)

\[
\text{old-var-decls} ::= \text{old-var-declarator} \mid \text{old-var-declarator} \ldots
\]

\[
\text{old-var-declarator} ::= \text{old} \mid \text{bound-var-modifiers} \mid \text{type-spec spec-variable-declarators} ;
\]
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.

```plaintext
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,

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

means the same thing as:

```plaintext
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 (including overriding a specification from an interface). 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 71).

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 (with |) 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). The order of this disjunction has, from left to right, inherited preconditions before each of the preconditions from the specification of any method that overrides it.
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 76, for the syntax of \texttt{pred-or-not}.

\[
\text{ensures-clause ::= \texttt{ensures-keyword \texttt{pred-or-not ;}}}
\]

\[
\text{ensures-keyword ::= \texttt{ensures | post}}
\]

\[
| \texttt{ensures_redundantly | post_redundantly}
\]

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

Informally,

\[
\text{ensures } Q;
\]

means

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

In an ensures clause, \texttt{\textbackslash result} stands for the result that is returned by the method. The postcondition \( Q \) may contain expressions of the form \texttt{\textbackslash 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 \texttt{\textbackslash old(x)}. For a detailed explanation of this see Section 9.9.6 [Parameters in Postconditions], page 80.

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 \textit{conjunction} of the postcondition predicates in the given ensures clauses. So

\[
\text{ensures } P;
\]

\[
\text{ensures } Q;
\]

means the same as

\[
\text{ensures } P \land Q;
\]

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 \texttt{\textbackslash not_specified}. The default precondition for a heavyweight specification case is \texttt{true}.

9.9.4 Signals Clauses

In a specification case a \texttt{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 76, for the syntax of pred-or-not.

\[
\text{signals-clause} ::= \text{signals-keyword ( reference-type [ ident ] ) [ pred-or-not ] ;}
\]

\[
\text{signals-keyword ::= \text{signals} | \text{signals_redundantly}
\]

| exsures | exsures_redundantly

In a signals-clause of the form

\[
\text{signals (E e) P ;}
\]

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,

\[
\text{signals (E e) P ;}
\]

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

\[
\text{signals (E e) R ;}
\]

is equivalent to the signals clause

\[
\text{signals (java.lang.Exception e) (e instanceof E) \Rightarrow R ;}
\]

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,

\[
\text{signals (E1 e) R1 ;}
\text{signals (E2 e) R2 ;}
\]

means the same as

\[
\text{signals (Exception e) ((e instanceof E1) \Rightarrow R1)
\quad \&\& ((e instanceof E2) \Rightarrow R2) ;}
\]

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 79).
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 77) 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-types` 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 77).

```
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-clauses` 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 \(E_j\) that are subtypes of some type named in all of the given `signals-only-clauses`. 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`).
Chapter 9: Method Specifications

If the \texttt{signals\_only} is omitted from a specification case, a default \texttt{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 \texttt{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 \texttt{signals\_only \nothing}; (which means that the method cannot throw any exceptions). However, if the method header declares that the method may throw exceptions \texttt{DE\_1, \ldots, DE\_n, Err\_1, \ldots, Err\_m}, where each \texttt{DE\_i} is a subtype of \texttt{java.lang.Exception}, and each \texttt{Err\_j} is not a subtype of \texttt{java.lang.Exception}, then the default \texttt{signals\_only} clause is as follows.

\begin{verbatim}
signals\_only DE\_1, \ldots, DE\_n
\end{verbatim}

For example, if the method has the header

\begin{verbatim}
public void foo() throws E1, E2
\end{verbatim}

then the default \texttt{signals\_only} clause would be

\begin{verbatim}
signals\_only E1, E2;
\end{verbatim}

It is important to note that the set of exceptions included in the default \texttt{signals} clause described above never includes \texttt{java.lang.Throwable}, and does not include \texttt{java.lang.Error} or any of its subtypes. Furthermore, this default would not normally include \texttt{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 \texttt{throws} clause. Since such unchecked, runtime exceptions are not usually listed in the method header, they would not find their way into the default \texttt{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 \texttt{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:

\begin{itemize}
  \item The parameters of a method or constructor can never be listed in its assignable clause.
  \item If parameters of a method (or constructor) are used in a normal or exceptional post-condition for that method (or constructor), i.e., in an \texttt{ensures} or \texttt{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 \texttt{\old()} placed around any occurrence of a formal parameter in a postcondition.
\end{itemize}

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 can be used to specify when a method may either loop forever or not return normally to its caller. The syntax is as follows See Section 9.9.2 [Requires Clauses], page 76, for the syntax of pred-or-not.

\[
diverges-clause ::= diverges-keyword pred-or-not \\
\text{diverges-keyword ::= diverges | diverges_redundantly}
\]
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 abort method in the following class. (This example is simplified from the specification of Java's System.exit method. This specification says that the method can always be called (the implicit precondition is true), 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 76, 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 87, 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 12.7 [Store Refs], page 106, for the syntax of store-ref-list.

\[
\text{assignable-clause ::= assignable-keyword store-ref-list ;}
\]

\[
\text{assignable-keyword ::= assignable | assignable_redundantly}
\quad \mid \text{modifiable | modifiable_redundantly}
\quad \mid \text{modifies | modifies_redundantly}
\]

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 \( \not\text{specified} \). The default store-ref-list for a heavyweight specification case is \( \text{everything} \).

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:

\[
\text{assignable \ nothing;}
\]

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 83), it says what (pre-existing) locations a method may read during its execution. It has the following syntax.

\[
\text{accessible-clause ::= accessible-keyword store-ref-list ;}
\]

\[
\text{accessible-keyword ::= accessible | accessible_redundantly}
\]

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 87) 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 16 [Specification for Subtypes], page 127, 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.

\[
\text{callable-clause} ::= \text{callable-keyword callable-methods-list} ;
\]

\[
\text{callable-keyword} ::= \text{callable} \mid \text{callable_redundantly}
\]

\[
\text{callable-methods-list} ::= \text{method-name-list} \mid \text{store-ref-keyword}
\]

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 16 [Specification for Subtypes], page 127, 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.

\[
\text{measured-clause} ::= \text{measured-by-keyword} \text{ not_specified} ;
\]

\[
\mid \text{measured-by-keyword spec-expression} [\text{ if predicate}] ;
\]

\[
\text{measured-by-keyword} ::= \text{measured_by} \mid \text{measured_by_redundantly}
\]

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.

\[
\text{measured_by} \text{ not_specified};
\]

### 9.9.13 Captures Clauses

The captures clause has the following syntax.

\[
\text{captures-clause} ::= \text{captures-keyword store-ref-list} ;
\]

\[
\text{captures-keyword} ::= \text{captures} \mid \text{captures_redundantly}
\]

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].

\[
\text{working-space-clause ::= working-space-keyword \not_specified ;} \\
| \text{working-space-keyword spec-expression [ if predicate ] ;}
\]

\[
\text{working-space-keyword ::= working_space | working_space_redundantly}
\]

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.

\[
\text{working_space \not_specified;}
\]

See Section 12.4.13 [Backslash working space], page 98, for information about the `\working_space` expression that can be used to describe the working space needed by a method call. See Section 12.4.12 [Backslash space], page 98, 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 12.4.11 [Backslash duration], page 98, 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 83) to name such sets of locations in a way that does not expose representation details [Leino98].

Each location (field or array element) in a program defines a data group, whose name is the same as that of the field or array element.

The main purpose for putting locations into data groups is so that these sets of locations may described succinctly in data groups by assignable clauses (see Section 9.9.9 [Assignable Clauses], page 83) For example, if locations x.f and x.y are in data group x.d, then an assignable clause of the form

\[
\text{assignable x.d;}
\]

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.

JML requires users to put fields that are used to compute the value of a model field (see Section 8.4 [Represents Clauses], page 60) into the data group of that model field. This is especially useful for private and protected fields, since they would not be visible to clients who can see the public 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.lang.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.

```java
// 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. Fields are not automatically put into `objectState` by JML, (indeed, there is no default datagroup for a field), but it is often convenient to put fields into this datagroup.

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]]]
Chapter 10: Data Groups

10.2 Dynamic Data Group Mappings

See Section 12.7 [Store Refs], page 106, for the definition of \textit{spec-array-ref-expr}.

\begin{verbatim}
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 ']
\end{verbatim}

The \textit{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 meaning of a \textit{member-field-ref} of the form \textit{E.*} depends on the denotation of the \textit{ident} or \textit{maps-array-ref-expr} \textit{E}. If \textit{E} denotes a data group whose locations are objects, then \textit{E.*} denotes the union of the data groups of all visible instance fields in \textit{E}'s (static) type. On the other hand, if \textit{E} names a class or interface, then \textit{E.*} denotes the union of the data groups of all visible static fields of the named class or interface.

Similarly, when \textit{E} denotes a set of locations containing arrays, then \textit{E[\star]} denotes the union of all data groups of all elements in all the arrays denoted by \textit{E}. Also, when \textit{E} denotes a set of locations containing arrays, then \textit{E[L..H]} denotes the union of all data groups of all elements in the arrays denoted by \textit{SR} whose indexes are between \textit{L} and \textit{H} inclusive. In the case where \textit{E} denotes a set of locations containing arrays, then \textit{E[J]} denotes the union of all data groups of those arrays at the index denoted by the \textit{spec-expression J}.

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 Specification Inheritance

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].

In JML, specification inheritance means that 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].

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

[[[This needs more work..., examples, etc.]]]
12 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.

12.1 Predicates

A **predicate** is defined as the following gives the syntax of predicates, which are simply *spec-expressions* that must have a boolean value. See Section 12.2 [Specification Expressions], page 90, for the syntax of specification expressions.

\[
\text{predicate} ::= \text{spec-expression}
\]

12.2 Specification Expressions

The following gives the syntax of specification expressions in JML. See Section 12.3 [Expressions], page 90, for the syntax of expression.

\[
\text{spec-expression-list} ::= \text{spec-expression} \\mid, \text{spec-expression} \ldots
\]

\[
\text{spec-expression} ::= \text{expression}
\]

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.

12.3 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.

<table>
<thead>
<tr>
<th>precedence level</th>
<th>operators</th>
</tr>
</thead>
</table>
| highest          | `new ()` `\forall` `\exists` `\max` `\min` `\num_of` `\product` `\sum` `informal-description` `[]` . and method calls `+` `&` `!` `*` `/` `%` `^` `|` `&` `^` `|` `==` `!=` `=>` `<>` `==>` `>=` `<=` `instanceof` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>>` `<<` `>>` `>>`
The following is the syntax of Java expressions, with JML additions. The additions are the operators $\Rightarrow$, $\Leftarrow$, $\Leftarrow\Rightarrow$, and $:\Leftarrow$; and the syntax found under the nonterminals $jml-primary$ (see Section 12.4 [JML Primary Expressions], page 92) and $set-comprehension$ (see Section 12.5 [Set Comprehensions], page 104). 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 ::= $\Leftarrow\Rightarrow$ | $\Leftarrow
implies-expr ::= logical-or-expr
  [ $\Rightarrow$ implies-non-backward-expr ]
logical-or-expr ::= $\Leftarrow\Rightarrow$ logical-or-expr
  [ $\Leftarrow$ logical-or-expr ] . . .
implies-non-backward-expr ::= $\Rightarrow$
logical-or-expr ::= logical-and-expr
  [ $\Rightarrow$ logical-and-expr ] . . .
logical-and-expr ::= inclusive-or-expr
  [ $\&$ inclusive-or-expr ] . . .
inclusive-or-expr ::= exclusive-or-expr
  [ $\mid$ exclusive-or-expr ] . . .
exclusive-or-expr ::= and-expr
  [ $\mid$ and-expr ] . . .
and-expr ::= equality-op
  [ $\mid$ equality-op ] . . .
equality-op ::= relational-op
  [ $\Rightarrow$ relational-op ] . . .
relational-op ::= shift-op
  [ $\Rightarrow$ shift-op ] . . .
shift-op ::= additive-op
  [ shift-op additive-op ] . . .
additive-op ::= mult-op
  [ additive-op mult-op ] . . .
mult-op ::= unary-op
  [ mult-op unary-op ] . . .
additive-op ::= mult-op
  [ additive-op mult-op ] . . .
additive-op ::= mult-op
  [ additive-op mult-op ] . . .
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 ] . . . [ , ] } 
| array-initializer 

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

12.4 JML Primary Expressions

The following is the syntax of jml-primary.

jml-primary ::= result-expression 
| old-expression
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].

12.4.1 \result
The syntax of a result-expression is as follows.

\begin{verbatim}
result-expression ::= \result
\end{verbatim}

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.

12.4.2 \old and \pre
An old-expression has the following syntax. See Section 12.2 [Specification Expressions], page 90, for the syntax of spec-expression.

\begin{verbatim}
old-expression ::= \old ( spec-expression [ , ident ] )
| \pre ( spec-expression )
\end{verbatim}

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 $\text{old}(v)$ denotes the pre-state value of v, which is the same reference, i.e., it is the address 0x952ab340. Note that $\text{old}(v)$ 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 $\text{old}(v)$; for example, $\text{old}(v).size()$ 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 $\text{old}(v.size())$. Indeed as a general rule, it is always safest to use $\text{old}()$ 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 63, 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 13 [Statements and Annotation Statements], page 108, 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 $\text{old}(Expr)$ 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 $\text{pre}(Expr)$ 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 $\text{old}(Expr, Label)$ 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 last reached the statement the label is attached to. Expressions of this form may only be used when the Label has been declared in a surrounding context. Thus such expressions may be used in assert and assume statements, and in loop invariants and variant functions, where such a label has been declared. They may also be used in specification cases that occur in model program spec-statement (see Section 15.6 [Specification Statements], page 125) and in the refining-statements (see Section 13.4.3 [Refining Statements], page 114).

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

12.4.3 $\text{not\_assigned}$

The syntax of a not-assigned-expression is as follows. See Section 12.7 [Store Refs], page 106, for the syntax of 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 87) 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 12.4.4 [Backslash not_modified], page 95, 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`.

12.4.4 \not_modified

The syntax of a `not-modified-expression` is as follows. See Section 12.7 [Store Refs], page 106, for the syntax of `store-ref-list`.

\[
\text{not-modified-expression ::= } \not_modified \ (\text{store-ref-list})
\]

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 12.4.3 [Backslash not_assigned], page 94, 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`.

12.4.5 \only_accessed

The syntax of an `only-accessed-expression` is as follows. See Section 12.7 [Store Refs], page 106, for the syntax of `store-ref-list`.

\[
\text{only-accessed-expression ::= } \only_accessed \ (\text{store-ref-list})
\]

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, \texttt{\textbackslash only\_accessed(xval,yval)} says that no fields, outside of the data groups
of \texttt{xval} and \texttt{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 \texttt{\textbackslash only\_accessed(x.f)} refers to the entire data group named by \texttt{x.f}
not just to the location \texttt{x.f} itself.

The \texttt{\textbackslash 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 \texttt{\textbackslash only\_accessed} expression is \texttt{boolean}.

12.4.6 \texttt{\textbackslash only\_assigned}

The syntax of an \texttt{\textbackslash only\_assigned-expression} is as follows. See Section 12.7 [Store Refs],
page 106, for the syntax of \texttt{store-ref-list}.

\texttt{\textbackslash only\_assigned-expression := \textbackslash only\_assigned ( store-ref-list )}

The JML operator \texttt{\textbackslash only\_assigned} can be used in both normal and exceptional precondi-
tions (i.e., in \texttt{ensures} and \texttt{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, \texttt{\textbackslash only\_assigned(xval,yval)} says that no fields, outside of the data groups
of \texttt{xval} and \texttt{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 \texttt{\textbackslash only\_assigned(x.f)} refers to the entire data group named by \texttt{x.f}
not just to the location \texttt{x.f} itself.

The \texttt{\textbackslash 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 \texttt{\textbackslash only\_assigned} expression is \texttt{boolean}.

12.4.7 \texttt{\textbackslash only\_called}

The syntax of an \texttt{\textbackslash only\_called-expression} is as follows. See Section 8.3 [Constraints], page 57,
for the syntax of \texttt{method-name-list}.

\texttt{\textbackslash only\_called-expression := \textbackslash only\_called ( method-name-list )}

The JML operator \texttt{\textbackslash only\_called} can be used in both normal and exceptional precondi-
tions (i.e., in \texttt{ensures} and \texttt{signals} clauses), and in history constraints. Used in a method’s
postcondition (perhaps implicitly in a history constraint), it asserts that the method’s execu-
tion only called from a subset of methods given in the \texttt{method-name-list}. For example,
\texttt{\textbackslash only\_called(p,q)} says that methods, apart from \texttt{p} and \texttt{q}, were called during this
method’s execution.

The type of an \texttt{\textbackslash only\_called} expression is \texttt{boolean}.
12.4.8 \texttt{\only\_captured}

The syntax of an \texttt{only-captured-expression} is as follows. See Section 12.7 [Store Refs], page 106, for the syntax of \texttt{store-ref-list}.

\texttt{only-captured-expression ::= \only\_captured \ ( store-ref-list )}

The JML operator \texttt{\only\_captured} can be used in both normal and exceptional pre-conditions (i.e., in \texttt{ensures} and \texttt{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, \texttt{\only\_captured(xv,yv)} says that no references, outside of the data groups of \texttt{xv} and \texttt{yv} were captured by the method.

A reference is \texttt{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 \texttt{this} object), a field in some object (or to an array element), or to a static field.

A predicate such as \texttt{\only\_captured(x.f)} refers to the references stored in the entire data group named by \texttt{x.f} in the pre-state, not just to those stored in the location \texttt{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 \texttt{\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 \texttt{\only\_captured} expression is \texttt{boolean}.

12.4.9 \texttt{\fresh}

The syntax of a \texttt{fresh-expression} is as follows. See Section 12.2 [Specification Expressions], page 90, for the syntax of \texttt{spec-expression-list}.

\texttt{fresh-expression ::= \fresh \ ( spec-expression-list )}

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

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

12.4.10 \texttt{\reach}

The syntax of a \texttt{reach-expression} is as follows. See Section 12.2 [Specification Expressions], page 90, for the syntax of \texttt{spec-expression}.

\texttt{reach-expression ::= \reach \ ( spec-expression )}

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

### 12.4.11 \( \text{duration} \)

The syntax of a duration-expression is as follows. See Section 12.3 [Expressions], page 90, for the syntax of expression.

\[
\text{duration-expression} ::= \text{duration} \ ( \text{expression} )
\]

\( \text{duration} \), 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., \( \text{duration}(\text{myStack}\.\text{push}(o)) \) is the maximum number of virtual machine cycles needed to execute the call \text{myStack}\.\text{push}(o)\), according to the contract of the static type of \text{myStack}’s type’s \text{push} method, when passed argument \( o \). Note that the expression used as an argument to \( \text{duration} \) 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 \( \text{duration} \) 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 \( \text{duration} \) must be a method call or explicit constructor invocation expression; the type of a \( \text{duration} \) expression is \text{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 \( i \).

### 12.4.12 \( \text{space} \)

The syntax of a space-expression is as follows. See Section 12.2 [Specification Expressions], page 90, for the syntax of spec-expression. [[Shouldn’t this take an expression instead of a spec-expression? - DRC]]

\[
\text{space-expression} ::= \text{space} \ ( \text{spec-expression} )
\]

\( \text{space} \), which describes the amount of heap space, in bytes, allocated to the object referred to by its argument [Krone-Ogden-Sitaraman03]; e.g., \( \text{space}(\text{myStack}) \) is number of bytes in the heap used by \text{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 \( \text{space} \) expression is \text{long}.

### 12.4.13 \( \text{working_space} \)

\[
\text{working-space-expression} ::= \text{working_space} \ ( \text{expression} )
\]

\( \text{working_space} \), 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., \( \text{working_space}(\text{myStack}\.\text{push}(o)) \) is the maximum number of bytes needed on the heap to execute the call \text{myStack}\.\text{push}(o)\), according to the contract of the static type of \text{myStack}’s type’s \text{push} method, when passed argument \( o \). Note that the expression used as an argument to \( \text{working_space} \) 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`.

### 12.4.14 `\textbackslash:\textbackslash nonnullelements`  

The syntax of a `nonnullelements`-expression is as follows. See Section 12.2 [Specification Expressions], page 90, for the syntax of `spec-expression`.

\[
\textbackslash:\textbackslash nonnullelements \text{ ( spec-expression )}
\]

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

\[
\text{myArray \neq null } \&\& \\
(\forall \text{ int } \text{i}; \text{ } 0 \leq \text{i } \&\& \text{i < myArray.length}; \text{myArray}[\text{i}] \neq \text{null})
\]

### 12.4.15 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.

### 12.4.16 `\textbackslash:\textbackslash typeof`  

The syntax of a `typeof`-expression is as follows. See Section 12.2 [Specification Expressions], page 90, for the syntax of `spec-expression`.

\[
\textbackslash:\textbackslash typeof \text{ ( spec-expression )}
\]

The operator `\textbackslash:\textbackslash typeof` returns the most-specific dynamic type of an expression’s value [Leino-Nelson-Saxe00]. The meaning of `\textbackslash:\textbackslash typeof(E)` is unspecified if `E` is null. If `E` has a static type that is a reference type, then `\textbackslash:\textbackslash 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 `\textbackslash:\textbackslash typeof(c)` is `HashSet.class`, which is the same thing as `\textbackslash:\textbackslash type(HashSet)`. If `E` has a static type that is not a reference type, then `\textbackslash:\textbackslash typeof(E)` means the instance of `java.lang.Class` that represents its static type. For example, `\textbackslash:\textbackslash typeof(true)` is `Boolean.TYPE`, which is the same as `\textbackslash:\textbackslash type(boolean)`. Thus an expression of the form `\textbackslash:\textbackslash typeof(E)` has type `\text{\textbackslash:\textbackslash TYPE}`, which JML considers to be the same as `java.lang.Class`.

### 12.4.17 `\textbackslash:\textbackslash elemtype`  

The syntax of a `elemtype`-expression is as follows.

\[
\textbackslash:\textbackslash elemtype \text{ ( spec-expression )}
\]

The `\textbackslash:\textbackslash elemtype` operator returns the most-specific static type shared by all elements of its array argument [Leino-Nelson-Saxe00]. For example, `\textbackslash:\textbackslash elemtype(\text{\textbackslash:\textbackslash type(int[])})` is
\texttt{\textbackslash type(int)}. The argument to \texttt{\textbackslash elemtype} must be an expression of type \texttt{\textbackslash TYPE}, which JML considers to be the same as \texttt{java.lang.Class}, and its result also has type \texttt{\textbackslash TYPE} (see Section 7.1.2.2 [Type-Specs], page 50). If the argument is not an array type, then the result is \texttt{null}. For example, \texttt{\textbackslash elemtype(\texttt{\textbackslash type(int)})} and \texttt{\textbackslash elemtype(\texttt{\textbackslash type(Object)})} are both \texttt{null}.

\textbf{12.4.18 \texttt{\textbackslash type}}

The syntax of a \textit{type-expression} is as follows. See Section 7.1.2.2 [Type-Specs], page 50, for the syntax of \texttt{type}.

\begin{align*}
\text{type-expression} &::= \texttt{\textbackslash type(\: type)}
\end{align*}

The operator \texttt{\textbackslash type} can be used to introduce literals of type \texttt{\textbackslash TYPE} in expressions. An expression of the form \texttt{\textbackslash type(T)}, where \texttt{T} is a type name, has the type \texttt{\textbackslash TYPE}. Since in JML \texttt{\textbackslash TYPE} is the same as \texttt{java.lang.Class}, an expression of the form \texttt{\textbackslash type(T)} means the same thing as \texttt{T.class}, if \texttt{T} is a reference type. If \texttt{T} is a primitive type, then \texttt{\textbackslash type(T)} is equivalent to the value of the \texttt{\textbackslash TYPE} field of the corresponding reference type. Thus \texttt{\textbackslash type(boolean)} equals \texttt{Boolean.TYPE}.

For example, in
\begin{align*}
\texttt{typeof(myObj) <: \textbackslash type(PlusAccount)}
\end{align*}
the use of \texttt{\textbackslash type(PlusAccount)} is used to introduce the type \texttt{PlusAccount} into this expression context.

\textbf{12.4.19 \texttt{\textbackslash lockset}}

The syntax of a \textit{lockset-expression} is as follows.

\begin{align*}
\text{lockset-expression} &::= \texttt{\textbackslash lockset}
\end{align*}

The \texttt{\textbackslash lockset} primitive denotes the set of locks held by the current thread. It is of type \texttt{JMLObjectSet}. (This is an adaptation from ESC/Java [Leino-etal00] [Leino-Nelson-Saxe00] for dealing with threads.)

\textbf{12.4.20 \texttt{\textbackslash max}}

The syntax of a \textit{max-expression} is as follows. See Section 12.2 [Specification Expressions], page 90, for the syntax of \texttt{spec-expression}.

\begin{align*}
\text{max-expression} &::= \texttt{\textbackslash max(\: spec-expression)}
\end{align*}

The \texttt{\textbackslash max} operator returns the "largest" (as defined by \texttt{<}) of a set of lock objects, given a lock set as an argument. The result is of type \texttt{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 12.4.24.2 [Generalized Quantifiers], page 102).

\textbf{12.4.21 \texttt{\textbackslash is_initialized}}

The syntax of the \textit{is-initialized-expression} is as follows. See Section 7.1.2.2 [Type-Specs], page 50, for the syntax of \texttt{reference-type}.

\begin{align*}
is-initialized-expression &::= \texttt{\textbackslash is_initialized(\: reference-type)}
\end{align*}

The \texttt{\textbackslash is_initialized} operator returns true just when its \texttt{reference-type} argument is a class that has finished its static initialization. It is of type \texttt{boolean}.
12.4.22 \invariant_for

\textit{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.

12.4.23 \lblneg and \lblpos

The syntax of the two kinds of labeled expressions is as follows. See Section 12.2 [Specification Expressions], page 90, for the syntax of spec-expression.

\textit{lblneg-expression ::= ( \lblneg ident spec-expression )}
\textit{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

\((\lblneg \text{indexInBounds } 0 \leq \text{index} \&\& \text{index} < \text{length})\)

the label is \text{indexInBounds} and the body is the expression \(0 \leq \text{index} \&\& \text{index} < \text{length}\). 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 \text{false} (e.g., when doing run-time checking of assertions), then a warning will be printed that includes the label \text{indexInBounds}. The form using \lblpos has a similar syntax, but should be used for warnings when the value of the enclosed expression is \text{true}.

12.4.24 Quantified Expressions

\textit{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 \textit{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 12.4.24.6 [Quantifying over Reference Types], page 104). 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.

12.4.24.1 Universal and Existential Quantifiers
The quantifiers \forall and \exists, are universal and existential quantifiers (respectively). For example, 
\[
(\forall \text{int } i,j; 0 \leq i \land i < j \land j < 10; a[i] < a[j])
\]
says that the values \(a[0] \ldots a[9]\) 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:
\[
(\forall \text{int } i; 0 < i \land i < 0; 0 < i) == \text{true} \\
(\exists \text{int } i; 0 < i \land i < 0; 0 < i) == \text{false}
\]

12.4.24.2 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 \text{int} or \text{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]):
\[
(\sum \text{int } i; 0 \leq i \land i < 5; i) == 0 + 1 + 2 + 3 + 4 \\
(\product \text{int } i; 0 < i \land i < 5; i) == 1 * 2 * 3 * 4 \\
(\max \text{int } i; 0 \leq i \land i < 5; i) == 4 \\
(\min \text{int } i; 0 \leq i \land i < 5; i-1) == -1
\]

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.
\[
(\product \text{float } f; 1.0e30f < f \land f < 1.0e38f; f) \\
== \text{Float.POSITIVE_INFINITY}
\]
\[
(\sum \text{int } i; i == \text{Integer.MAX_VALUE} || i == 1; i) \\
== \text{Integer.MAX_VALUE + 1} \\
== \text{Integer.MIN_VALUE}
\]

When the range predicate is not satisfiable, the sum is 0 and the product is 1; for example:
\[
(\sum \text{int } i; \text{false}; i) == 0
\]
\[
\prod \text{double } d; \ false; d \ast d = 1.0
\]

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. \[\text{[Or should this be undefined - DRC]}\]

12.4.24.3 Numerical Quantifier

The numerical quantifier, \(\text{\num_of}\), 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 \text{boolean}, and the entire quantified expression has type \text{long}. The meaning of this quantifier is defined by the following equation (see p. 57 of [Cohen90]).

\[
\text{\num_of } T \ x; R(x); P(x)) = (\sum T \ x; R(x) \& \& P(x); 1L)
\]

12.4.24.4 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 \text{boolean}, then it is always executable. Otherwise a \text{spec-quantified-exp} 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 \text{int}) or where the range predicate requires the quantified variable to be drawn from some set. Examples include the following. \[\text{[Make these examples be real examples in the samples directory]}\]

\[
\forall \text{int } x; 0 \leq x \& \& x < \text{somelimit}; \ldots
\]
\[
\forall \text{Object } x; \text{someSet.has(x); } \ldots
\]

You should get warnings from the \text{jmlc} tool when assertions are not executable, but you have to use the \text{-w2} flag to see them.

If a \text{spec-quantified-exp}, \(QE\), 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 \(QE\).

12.4.24.5 Modifiers for Bound Variables

\[
\text{bound-var-modifiers ::= non_null | nullable}
\]

Logical variables can be bound in

- quantified expressions (see Section 12.4.24 [Quantified Expressions], page 101),
- set comprehension expressions (see Section 12.5 [Set Comprehensions], page 104),
- forall clauses of method contracts (see Section 9.9.1.1 [Forall Variable Declarations], page 75), or
- old clauses of method contracts (see Section 9.9.1.2 [Old Variable Declarations], page 75).

Note that in JML, \text{non_null} and \text{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 \text{non_null} or \text{nullable} is necessary to provide an explicit nullity modifier. For example,
\( (\forall \text{non-null non-null nn; \ldots}) \)

where the first \text{non-null} is one of the \textit{bound-var-modifiers} and the second is the type \text{non-null}.

### 12.4.24.6 Quantifying over Reference Types

The range of values for a quantified variable that is declared to be of a reference type:

- Does not include \textit{null} unless the bound variable is declared \textit{nullable} (see Section G.2.1 [Non-null by Default], page 169).
- 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.

### 12.5 Set Comprehensions

The syntax of a \textit{set-comprehension} expression is as follows.

\[
\text{set-comprehension ::= \{ [ bound-var-modifiers ] type-spec quantified-var-declarator \text{"\'|"} postfix-expr \&\& predicate \}}
\]

The set comprehension notation can be used to succinctly define sets. The meaning of a \textit{new-expr} (see Section 12.3 [Expressions], page 90) with a \textit{set-comprehension} suffix, such as  
\[
\text{new ST \{ T x | s\text{.has}(x) \&\& P(x) \}}
\] is to form a subset, of type \textit{ST}, of the set \textit{s}, which contains just those elements \textit{x} that are both in \textit{s} and for which \textit{P(x)} is true.

For example, the following is the \texttt{JMLObjectSet} that is the subset of non-null \texttt{Integer} objects found in the set \texttt{myIntSet} whose values are between 0 and 10, inclusive.

\[
\text{new JMLObjectSet \{Integer i | myIntSet\text{.has}(i) \&\& i \neq \texttt{null} \&\& 0 < i\text{.intValue()} \&\& i\text{.intValue()} \leq 10 \}}
\]

The syntax of JML limits set comprehensions so that the \textit{postfix-expr} following the vertical bar (\texttt{|}) is always a method invocation with the bound variable declared in the \textit{quantified-var-declarator} as its parameter; the method may be either the \texttt{has} method of an \texttt{org.jmlspecs.models.JMLObjectSet} or \texttt{org.jmlspecs.models.JMLValueSet}, or the \texttt{contains} method of a \texttt{java.util.Collection}. This restriction is used to avoid Russell’s paradox [Whitehead-Russell25]. The bound variable, whose scope is the \textit{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 \textit{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 (given an existing \texttt{Collection s}) result in identical sets:

\[
\text{new JMLObjectSet \{Integer i | s\text{.contains}(i) \&\& 0 < i\text{.intValue()} \}}
\]
\[
\text{new JMLObjectSet \{Object i | s\text{.contains}(i) \&\& i\text{ instanceof Integer} \&\& 0 < ((\text{Integer}) i)\text{.intValue()} \}}
\]

In practice, one starts either from some relevant set at hand or from the sets found in \texttt{JMLObjectSet} and \texttt{JMLValueSet} containing the objects of primitive types. The type of a set comprehension is the type named following \texttt{new}, which must be \texttt{JMLObjectSet} or \texttt{JMLValueSet}. The bound variable type must be compatible with the set comprehension type; in particular, the bound variable type must be a subtype of \texttt{org.jmlspecs.models.JMLType} if the set comprehension type is \texttt{JMLValueSet}. 
12.6 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.

12.6.1 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.

12.6.2 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 \"\texttt{\textbackslash result} <=> \texttt{\textbackslash size} == 0\" means the same thing as \"\texttt{\textbackslash result} == (\texttt{\textbackslash 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 \"\texttt{\textbackslash result} <=!=> \texttt{\textbackslash size} == 0\" means the same thing as \"\texttt{\textbackslash result} != (\texttt{\textbackslash size} == 0)\".

The expressions on either side of these operators must be of type boolean, and the type of the result is also boolean.

12.6.3 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.
12.6.4 Lockset Ordering

JML uses `<#` and `<#=` to test order of locks. (The previously-used operators `<` and `<=` were deprecated because their use conflicts with the Java comparisons defined for those operators when autoboxing is available.)

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-clauses 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.

12.7 Store Refs

The syntax related to the store-ref production is used in several places, in particular in assignable clauses (see Section 9.9.9 [Assignable Clauses], page 83).

```
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
                      | *
store-ref-keyword ::= \nothing | \everything | \not_specified
```

A store-ref denotes a set of locations. These sets can be specified using data groups (see Chapter 10 [Data Groups], page 87), and if this is done then the set of locations denoted by a store-ref is the union of all the sets of locations in the specified set of data groups.

The set of locations denoted by a store-ref-list of the form store-ref, [ store-ref ] ... is the union of all the sets of locations denoted by each store-ref in the list.

The store-ref \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 a particular tool.

When SR denotes a set of locations contain objects, then SR.f, where f is an ident, denotes the union of the data groups of each field named f in each object in the denotation of SR.

The meaning of a store-ref-expression of the form SR.* depends on the denotation of the store-ref-name SR. When SR denotes a set of locations of objects, then SR.* denotes the union of the data groups of all visible instance fields of SR’s (static) type. On the other hand, if SR names a class or interface, then SR.* denotes the union of the data groups of all visible static fields of the named class or interface.
Similarly, when $SR$ denotes a set of locations containing arrays, then $SR[*]$ denotes the union of all data groups of all elements in all the arrays denoted by $SR$. Also, when $SR$ denotes a set of locations containing arrays, then $SR[L..H]$ denotes the union of all data groups of all elements in the arrays denoted by $SR$ whose indexes are between $L$ and $H$ inclusive. In the case where $SR$ denotes a set of locations containing arrays, then $SR[J]$ denotes the union of all data groups of those arrays at the index denoted by the spec-expression $J$. 
Chapter 13: Statements and Annotation Statements

13 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 15 [Model Programs], page 122, for the syntax of model-prog-statement, which is only allowed in model programs. [[[ Does this include local class declarations?]]]

\[
\text{compound-statement ::= \{ statement [ statement ] . . . \}}
\]
\[
\text{statement ::= compound-statement}
\]
\[
\text{| local-declaration ;}
\]
\[
\text{| ident : statement}
\]
\[
\text{| expression ;}
\]
\[
\text{| if ( expression )}
\]
\[
\text{statement [ else statement ]}
\]
\[
\text{| possibly-annotated-loop}
\]
\[
\text{| break [ ident ] ;}
\]
\[
\text{| continue [ ident ] ;}
\]
\[
\text{| return [ expression ] ;}
\]
\[
\text{| switch-statement}
\]
\[
\text{| try-block}
\]
\[
\text{| throw expression ;}
\]
\[
\text{| synchronized ( expression ) statement}
\]
\[
\text{| ;}
\]
\[
\text{| jml-annotation-statement}
\]
\[
\text{| assert-statement}
\]
\[
\text{| jml-annotation-statement}
\]
\[
\text{| model-prog-statement // only allowed in model programs}
\]

\[
\text{switch-statement ::= switch ( expression ) \{}
\]
\[
\text{[ switch-body ] . . . \}}
\]
\[
\text{switch-body ::= switch-label-seq [ statement ] . . .}
\]
\[
\text{switch-label-seq ::= switch-label [ switch-label ] . . .}
\]
\[
\text{switch-label ::= case expression : | default :}
\]
\[
\text{try-block ::= try compound-statement}
\]
\[
\text{[ handler ] . . .}
\]
\[
\text{[ finally compound-statement ]}
\]
\[
\text{handler ::= catch ( param-declaration ) compound-statement}
\]

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.

13.1 Local Declaration Statements

The following is the syntax of local declaration statements. See Section 7.1.2 [Field and Variable Declarations], page 49, for the syntax of variable-decls.

\[
\text{local-declaration ::= local-modifiers variable-decls}
\]
13.1.1 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 133, for the grammar of ownership-modifier.

\[\text{local-modifiers ::= [ local-modifier ] . . .} \]
\[\text{local-modifier ::= ghost | final uninitialized | non_null | nullable} \]
\[\text{ ownership-modifier // when the Universe type system is on} \]

The JML modifiers are discussed to some extent below. See Section 7.1.2.1 [JML Modifiers for Fields], page 49, 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. Furthermore, since the initializer is in an annotation (and thus not executed when runtime assertion checks are turned off), the initializer of a ghost variable must be a pure expression (i.e., it must be side effect free).
- 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.

```java
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();
    }
}
```

13.2 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])
         | for (modifiers type-spec ident : expression)
            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`.

```java
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];
        }
```
Chapter 13: Statements and Annotation Statements

//@ assert i < 0 && -1 <= i && i <= a.length;
//@ hence_by (i < 0 && -1 <= i) ==> i == -1;
//@ assert i == -1 && i <= a.length;
//@ assert 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.)

13.2.1 Loop Invariants

A loop can specify one or more loop invariants, using the following syntax.

\[ \text{loop-invariant ::= maintaining-keyword predicate ;} \]

\[ \text{maintaining-keyword ::= maintaining | maintaining_redundantly} \]

\[ | \text{loop_invariant | loop_invariant_redundantly} \]

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 12.2 [Specification Expressions], page 90). (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.

### 13.2.2 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; }
    //@ assert 0 <= vf;
    S
    //@ 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; }
    //@ assert 0 <= vf;
    S1
```
if (C) {
    S2
    //@ assert E < vf;
    continue;
}
S3
//@ assert E < vf;
}

13.3 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 12.2 [Specification Expressions], page 90, 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.

13.4 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 13.3 [Assert Statements], page 113, for the syntax of the assert-redundantly-statement. See Chapter 15 [Model Programs], page 122, for the syntax of additional statements that can only be used in model programs.

jml-annotation-statement ::= assert-redundantly-statement
13.4.1 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.

\[
\text{assume-statement} ::= \text{assume-keyword } \text{predicate} \\
\quad [ \ : \ \text{expression} ] ; \\
\text{assume-keyword} ::= \text{assume} | \text{assume_redundantly}
\]

In runtime assertion checking, assumptions are checked in the same way that assert statements are checked (see Section 13.3 [Assert Statements], page 113).

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.

13.4.2 Set Statements

The syntax of a set statement is as follows. See Section 12.3 [Expressions], page 90, for the syntax of assignment-expr.

\[
\text{set-statement} ::= \text{set} \ \text{assignment-expr} ;
\]

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. Note that:

- the target of the set statement must be a ghost variable or a ghost field, and
- the right-hand-side of the assignment-expr must be a pure expression (i.e., it must not have side effects).

Examples:

```java
//@ set i = 0;
//@ set collection.elementType = \type(int);
```

The reason that right hand side of the assignment-expr must be pure is because set-statements are not part of the normal Java code, but only occur in annotations. Hence they must not affect normal Java code execution, but only have side effects on the ghost field or ghost variable being assigned. This restriction is a conservative way to guarantee that property.

13.4.3 Refining Statements

The syntax of a refining statement is as follows. See Section 15.6 [Specification Statements], page 125, for the syntax of spec-statement and generic-spec-statement-case. See Chapter 13 [Statements and Annotation Statements], page 108, 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 15.6 [Specification Statements], page 125), which includes the grammar for a heavyweight specification case, or a generic-spec-statement-case (see Section 15.6 [Specification Statements], page 125), 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 15 [Model Programs], page 122). 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 63, for more about what it means to satisfy such a specification.

When $\texttt{old()}$ or $\texttt{pre()}$ are used in the specification part of a refining statement, they have the same meaning as in a specification statement (see Section 15.6 [Specification Statements], page 125).

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 15 [Model Programs], page 122, for more examples.

13.4.4 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:

```java
if (true) {
    ...
} else {
    //@ unreachable;
}
```

### 13.4.5 Debug Statements

The syntax of the `debug` statement is as follows. See Section 12.3 [Expressions], page 90, 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 60).

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 13.4.2 [Set Statements], page 114), which would execute even if debug statements are not being executed.

### 13.4.6 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 13.3 [Assert Statements], page 113) or between two assume statements (see Section 13.4.1 [Assume Statements], page 114).

[[[Needs example.]]]
14 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 $X_{\text{redundantly}}$, such as $\text{requires}_{\text{redundantly}}$, $\text{ensures}_{\text{redundantly}}$, 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.

$$\text{redundant-spec ::= implications } [ \text{examples } ] | \text{examples}$$

The two subsections below explain these features. The description of clauses of the form $X_{\text{redundantly}}$ is contained in the first section.

14.1 Redundant Implications and Redundantly Clauses

A redundant implication is a way of stating a claim about a specification. By itself it does not constrain an implication, but can be thought of as 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 $X_{\text{redundantly}}$. 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 63, for the syntax of spec-case-seq.)

$$\text{implications ::= implies_that spec-case-seq}$$

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:

$$V \text{ behavior } // \text{ non-redundant}$$

- 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:

$$V \text{ behavior } // \text{ redundant}$$

- 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:

- $\text{old}(\text{RedPre}) \implies \text{Pre}$,
- $(\text{old}(\text{RedPre}) \land \text{NormPost}) \implies \text{RedNormPost}$, and
- $(\text{old}(\text{RedPre}) \land \text{ExPost}) \implies \text{RedExPost}$.

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 $X_{\text{redundantly}}$ 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.

\begin{verbatim}
requires Pre;
requires_redundantly RedPre;
\end{verbatim}

These state the claim that $\text{Pre} \implies \text{RedPre}$. That is, in all pre-states, whenever $\text{Pre}$ is true, then $\text{RedPre}$ 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

\begin{verbatim}
ensures Q1;
ensures Q2;
ensures_redundantly RedQ1;
ensures_redundantly RedQ2;
\end{verbatim}

is equivalent to

\begin{verbatim}
ensures Q1 \&\& Q2;
ensures_redundantly RedQ1 \&\& RedQ2;
\end{verbatim}

In this example, the claim stated is that:

$$(Q1 \&\& Q2) \implies (\text{RedQ1} \&\& \text{RedQ2}).$$

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.]]]

### 14.2 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 examples. Each example is much
like a spec-case (see Section 9.2 [Organization of Method Specifications], page 63), 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-case
                           [ exceptional-spec-case ] . . .
normal-example-body ::= normal-spec-case
                        [ normal-spec-case ] . . .
```

As in method spec-cases (see Section 9.2 [Organization of Method Specifications],
page 63) 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
element keywords. As with spec-cases, 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 examples are the same as those for omitted
clauses in lightweight spec-cases. Similarly, heavyweight examples have the same defaults as
heavyweight spec-cases. (See Section 9.6.1 [Semantics of flat behavior specification cases],
page 68, 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 \texttt{\old{}}),
  - the precondition of the expanded meaning of the specified behaviors (also wrapped by \texttt{\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 \texttt{public\ normal\ behavior} specification case “and if there are no preconditions and assignable clauses, then the example’s postcondition should be equivalent to the conjunction of the example’s precondition and the postcondition of the \texttt{public\ normal\ behavior} specification.”

[[[[Needs concrete examples :-)]]]]
15 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].

15.1 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 13.4.3 [Refining Statements], page 114), 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.

```java
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(); ) {
            //...}
    }
```
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 13.4.3 [Refining Statements], page 114).

### 15.2 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.]]]

### 15.3 Details of Model Programs

The following gives the syntax of model programs. See Chapter 13 [Statements and Annotation Statements], page 108, 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-statements` are not flagged as errors within the `jml-compound-statement` and `jml-statements`.

```
model-program ::= [ privacy ] [ code ] model_program
jml-compound-statement ::= compound-statement
jml-statement ::= statement
model-prog-statement ::= nondeterministic-choice
| nondeterministic-if
| spec-statement
| invariant
```

### 15.4 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.

### 15.5 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.

15.6 Specification Statements

The grammar for specification statements appears below. It is unusual, compared to spec-
ification 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-case-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 | accessible-clause
  | captures-clause | callable-clause
  | when-clause | working-space-clause | duration-clause
  | ensures-clause | signals-only-clause | signals-clause
  | measured-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 13.4.3 [Refining Statements], page 114).

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.

15.6.1 Continues Clause

continues-clause ::= continues-keyword [ target-label ]
[ pred-or-not ] ;
continues-keyword ::= continues | continues_redundantly

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.

A continues-clause should only be used in a generic-spec-statement-case (with the keyword behavior) or an abrupt-spec-case (with the keyword abrupt_behavior), as in other kinds of specification cases the default pred-or-not is false. (See Section 9.6.1 [Semantics of flat behavior specification cases], page 68, for the defaults for a lightweight and heavyweight specification cases.)

15.6.2 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.

A breaks-clause should only be used in a generic-spec-statement-case (with the keyword behavior) or an abrupt-spec-case (with the keyword abrupt_behavior), as in other kinds of specification cases the default pred-or-not is false. (See Section 9.6.1 [Semantics of flat behavior specification cases], page 68, for the defaults for a lightweight and heavyweight specification cases.)

15.6.3 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 \texttt{result} to name the computed return value.

A returns-clause should only be used in a generic-spec-statement-case (with the keyword behavior) or an abrupt-spec-case (with the keyword abrupt_behavior), as in other kinds of specification cases the default pred-or-not is false. (See Section 9.6.1 [Semantics of flat behavior specification cases], page 68, for the defaults for a lightweight and heavyweight specification cases.)
16 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 67) or as a model program (see Chapter 15 [Model Programs], page 122) 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 83, for the syntax and semantics of the accessible-clause. See Section 9.9.11 [Callable Clauses], page 84, for the syntax and semantics of the callable-clause.

16.1 Method of Specifying for Subclasses

[[[This should be a synopsis of Clyde Ruby’s dissertation, with an example.]]]

16.2 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 67, 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.)
17 Separate Files for Specifications

This chapter explains how to use separate files to hold JML specifications.

The following gives more details about this feature of JML.

17.1 File Name Suffixes

The JML tools recognize three filename suffixes: `.java`, `.class`, and `.jml`. See Section 17.2 [Using Separate Files], page 129, for guidelines on how to use these suffixes.

17.2 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 a `.jml` suffix. 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.jml`. This technique also works if you are specifying code for which no sources are available (a class library in binary form).

Note that the `.jml` file should contain all the (non-inherited) specifications for the types in the compilation unit; there should be no specifications at all in the `.java` file. Thus, in our example, there should be no JML specifications at all in the `LibraryType.java` file, and all the specifications should be found in the `LibraryType.jml` file.

Files with a `.jml` suffix do not have to have implementations, as they only hold specifications. In particular, in such a file one can and must specify non-model methods without giving a method body.

Another reason for writing specifications in different files is to prevent the specifications from “cluttering up” the code (i.e., 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. The same technique of using a file with the same base name but with a `.jml` suffix works in such cases, as the specifications in the `.jml` file are added to the code in the `.java` file.

17.3 Type Checking Separate Files

There are some restrictions on what can appear in a separate specification file (i.e., in the `.jml` file). Since the Java compilers only see the `.java` files, executable code (that is not just for use in specifications) can 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, 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 pair of matching declarations.

- 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 specify a non-model method as a model method. Use a pure spec_public or spec_protected method if you want to use a non-model method in a (public or protected) specification. Also, there may be no nesting of model declarations: model classes and model methods may not contain model declarations.

- Some of the JML method modifiers do not always have to match in all declarations of the same method found in separate files. One may add pure, non_null, nullable, spec_public, or spec_protected to any of the declarations for a method in any file. Also, it is, of course, not permitted to add spec_protected to a method that has been declared public or spec_public in the other declaration. One can add non_null or nullable to any formal parameter in a separate ('.jml') file.

- The specification of a method declaration in a separate file is the only specification of that method. Therefore, unless that method is overriding a method in a supertype, it should not start with the JML keyword also. In such a case, the also is a clue to the reader that the specification is adding to a possibly inherited specification, either from the superclass or from an implemented interface. Therefore, it is an error if the specification of a non-overriding method begins with also, regardless of whether it is in a separate file. (See Appendix B [Incompatible Changes], page 143, for how this is different than earlier versions of JML.)

- 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; thus, if there is a '.jml' file, then the body of the model method must appear (if it appears at all) in the '.jml' file. Thus each method of each class can have at most one method body.

- When the same field is declared in more than one file, then the signature of each such declaration must be identical in all the files. (Two field declarations in a given type 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 pair of matching declarations.

- 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 restriction also holds for ghost fields.

- One may add non_null, nullable, spec_public, or spec_protected to any of the declarations for a field in a separate file. However, it is of course not permitted to add spec_protected to a field that has been declared public in another declaration.

- Initializers are not allowed for field declarations in separate specification files. A non-
model and non-ghost field can have an initializer expression, but this initializer 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 a separate specification file, but they may have at most one initializer expression. Thus, if there is a separate specification file, the initializer for a `ghost` field must appear there.

Model fields cannot have an initializer expression because there is no storage associated with such fields. If you want to specify initial values for a model field in a separate file, then use the `initially` clause in that file.

- 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`. Such specifications may only appear in one file, and thus if there is a separate specification (`.jml`) file, it must appear in that file.

### 17.4 Default Constructors and Separate Files

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` (where the same constructor is generated as in the Java language). Consider, for example, the refinement sequence defined by the following two files: `RefineDemo.jml` and `RefineDemo.java`.

```java
// ---- file RefineDemo.jml -----------
package org.jmlspecs.samples.jmlrefman;

public class RefineDemo {
    //@ public model int x;

    protected int x_; // @ in x;

    //@ protected represents x = x_;
}

// ---- file RefineDemo.java ---------
package org.jmlspecs.samples.jmlrefman;

public class RefineDemo {
    protected int x_; // @
    public RefineDemo() { x_ = 0; }
}
```

In the protected specification declared in `RefineDemo.jml`, no constructor is specified.

The reason that JML does not generate a default constructor for such separate specification files is that it might conflict with the constructor explicitly declared in the `.java` file. To see why, consider what would happen if JML were to generate a default constructor for `RefineDemo` for the `RefineDemo.jml` file. If JML did that, then this default constructor would possibly have a different visibility from any constructor written explicitly
in the ‘RefineDemo.java’ file. To avoid such conflicts, JML does not generate a default constructor unless the file suffix is ‘.java’.

(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 65, for more details.)
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.

\[
\text{ownership-modifiers ::= ownership-modifier | ownership-modifier | ownership-modifier} \\
(\text{reserved-ownership-modifier // with –universesx parse or –universesx full}) \\
\text{reserved-ownership-modifier ::= rep | peer | readonly}
\]

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 nongeneric 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.

This owner-as-modifier property prevents the problem of representation exposure, in which a reference to X can be used to modify it, without calling one of the methods of X’s owner [Noble-Vitek-Potter98]. From the perspective of X’s owner, X is part of the owner’s representation (and thus a field holding X would be declared with the rep keyword), and passing out a mutable reference to X exposes that representation to the rest of the program. It is difficult to maintain invariants, for example, when the representation of an object can be directly modified from outside an object [Mueller02] [Mueller-Poetzsch-Heffter-Leavens06].

18.2 Rep and Peer

The rep and peer annotations are type modifiers (see Section 7.1.2.2 [Type-Specs], page 50) 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 136, 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 46, 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 \( E \ \text{instanceof} \ T \), 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 139, 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.

```java
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.

```java
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.

```java
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

The types \texttt{bigint} and \texttt{real} are designed to support arbitrary precision arithmetic for integers and floating point numbers. Both types act as primitive value types in JML, with the usual infix arithmetic and logical operations [Chalin04].

However, note that for purposes of Java reflection these types are not actually implemented as primitives. So, since JML equates \texttt{TYPE} and \texttt{java.lang.Class}, the expression \texttt{\_type(\_TYPE).isPrimitive()} will return false.

19.1 \texttt{bigint}

The type \texttt{bigint} models arbitrary precision integers [Chalin04]. This type is considered by JML to act like a primitive value type, and supports all of the infix arithmetic and logical operators, like \texttt{int} or \texttt{long}. However, note that arithmetic does not wrap around, this for all values \( i \) of type \texttt{bigint}, \( i < i+1 \).

Note also that \texttt{==} means value equality for \texttt{bigint} values, not object identity (even though these values are necessarily represented by objects in a runtime assertion checker). Hence, for example, \( i+1 == i+2-1 \) will be true. Similarly \texttt{!=} means value inequality, and does not compare object identities.

[[[Needs more discussion and examples.]]]

19.2 \texttt{real}

The type \texttt{real} models arbitrary precision floating point numbers [Chalin04]. This type is considered by JML to act like a primitive value type, and supports all of the infix arithmetic and logical operators, like \texttt{float} or \texttt{double}. However, note that arithmetic does not have precision limitations.

Note also that \texttt{==} means value equality for \texttt{real} values, not object identity (even though these values are necessarily represented by objects in a runtime assertion checker). Hence, for example, \( i+1 == i+2-1 \) will be true. Similarly \texttt{!=} means value inequality, and does not compare object identities.

[[[Needs more discussion and examples. Is there also a NaN for this type? Are we supposing that it can represent all reals?]]]
Appendix A Deprecated and Replaced Syntax

The subsections below briefly describe the deprecated and replaced features of JML. A feature is deprecated if it is supported in the current release, but slated to be removed from a subsequent release. Such features should not be used.

A feature that was formerly deprecated is replaced if it has been removed from JML in favor of some other feature or features. While we do not describe all replaced syntax in this appendix, we do mention a few of the more interesting or important features that were replaced, especially those discussed in earlier papers on JML.

A.1 Deprecated Syntax

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

A.1.1 Deprecated Annotation Markers

The following lexical syntax for annotation markers is deprecated.

\[
\text{annotation-marker ::=} \\
//+@ \text{[ } @ \text{ ] } \ldots \\
| /**+@ \text{[ } @ \text{ ] } \ldots \\
| //=+@ \text{[ } @ \text{ ] } \ldots \\
| /*-@ \text{[ } @ \text{ ] } \ldots \\
\]

A.1.2 Deprecated Represents Clause Syntax

The following syntax for a functional represents-clause is deprecated.

\[
\text{represents-clause ::= } \text{represents-keyword } \text{store-ref-expression } \leftarrow \text{spec-expression} ;
\]

Instead of using the \( \leftarrow \), one should use = in such a represents-clause. See Section 8.4 [Represents Clauses], page 60, for the supported syntax.

A.1.3 Deprecated Monitors For Clause Syntax

The following syntax for the monitors-for-clause is deprecated.

\[
\text{monitors-for-clause ::= } \text{monitors_for } \text{ident} \\
\leftarrow \text{spec-expression-list} ;
\]

Instead of using the \( \leftarrow \), one should use = in such a monitors-for-clause. See Section 8.9 [Monitors For Clause], page 62, for the supported syntax.

A.1.4 Deprecated File Name Suffixes

The set of file name suffixes supported by JML tools is being simplified. In the future, especially in new tools the suffixes The suffixes \'.refines-java\', \'.refines-spec\', \'.refines-jml\', \'.spec\', \'.java-refined\', \'.spec-refined\', and \'.jml-refined\' are no longer supported. Instead, one should write specifications into files with the suffixes \'.java\' and \'.jml\'. See Section 17.1 [File Name Suffixes], page 129, for details on the use of file names with JML tools.
A.1.5 Deprecated Refine Prefix

The following syntax involving the *refine-prefix* is deprecated.

\[
\text{compilation-unit ::= [ package-declaration ]}
\text{refine-prefix}
\text{[ import-declaration ] . . .}
\text{[ top-level-declaration ] . . .}
\]

\[
\text{refine-prefix ::= refine-keyword string-literal ;}
\text{refine-keyword ::= refine | refines}
\]

Instead of using the *refine-prefix* in a compilation unit, modern JML tools just use a *jml* file that contains any specifications not in the *java* file. See Chapter 17 [Separate Files for Specifications], page 129, for details.

A.2 Replaced Syntax

The + style of JML annotations, that is, JML annotations beginning with //+@ or /*+@, is being replaced by the annotation-key feature described in See Section 4.4 [Annotation Markers], page 27.

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 B Incompatible Changes

In older versions of JML and older tools, method specifications that were placed in separate files (see Chapter 17 [Separate Files for Specifications], page 129) had to start with the JML keyword also. However, with the present version of JML, method specifications in separate files only start with also if the method being specified is an overriding method, as is normal in the rest of JML.
Appendix C 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.

C.1 Lexical Conventions

\[
\begin{align*}
\text{microsyntax} & ::= \text{lexeme} \ [ \text{lexeme} ] \ldots \\
\text{lexeme} & ::= \text{white-space} \mid \text{lexical-pragm} \mid \text{comment} \\
& \mid \text{annotation-marker} \mid \text{doc-comment} \mid \text{token} \\
\text{token} & ::= \text{ident} \mid \text{keyword} \mid \text{special-symbol} \\
& \mid \text{java-literal} \mid \text{informal-description} \\
\text{white-space} & ::= \text{non-nl-white-space} \mid \text{end-of-line} \\
\text{non-nl-white-space} & ::= \text{a blank, tab, or formfeed character} \\
\text{end-of-line} & ::= \text{newline} \mid \text{carriage-return} \\
& \mid \text{carriage-return newline} \\
\text{newline} & ::= \text{a newline character} \\
\text{carriage-return} & ::= \text{a carriage return character} \\
\text{lexical-pragm} & ::= \text{nowarn-pragm} \\
\text{nowarn-pragm} & ::= \text{nowarn} \ [ \text{spaces} ] \ [ \text{nowarn-label-list} ] ; \\
\text{spaces} & ::= \text{non-nl-white-space} \ [ \text{non-nl-white-space} ] \ldots \\
\text{nowarn-label-list} & ::= \text{nowarn-label} \ [ \text{spaces} ] \ldots \\
\text{nowarn-label} & ::= \text{letter} \ [ \text{letter} ] \ldots \\
\text{comment} & ::= \text{C-style-comment} \mid \text{C++-style-comment} \\
\text{C-style-comment} & ::= /\* \mid \text{C-style-body} \mid \text{C-style-end} \\
\text{C-style-body} & ::= \text{non-at-plus-minus-star} \ [ \text{non-stars-slash} ] \ldots \\
& \mid \text{non-letter} \ [ \text{non-stars-slash} ] \ldots \\
& \mid \text{stars-non-slash} \ [ \text{non-stars-slash} ] \ldots \\
\text{non-letter} & ::= \text{any character except } _, \text{ a through z, or A through Z} \\
\text{non-stars-slash} & ::= \text{non-star} \\
& \mid \text{stars-non-slash} \\
\text{stars-non-slash} & ::= \text{*} \ [ \text{*} ] \ldots \text{non-star}\text{-slash} \\
\text{non-at-plus-minus-star} & ::= \text{any character except } @, +, -, \text{ or } * \\
\text{non-star} & ::= \text{any character except } * \\
\text{non-slash} & ::= \text{any character except } / \\
\text{non-star-slash} & ::= \text{any character except } * \text{ or } / \\
\text{C-style-end} & ::= \ [ \text{*} ] \ldots */ \\
\text{C++-style-comment} & ::= // \mid \text{[ + ]} \text{end-of-line} \\
& \mid // \text{non-at-plus-minus-end-of-line} \ [ \text{non-end-of-line} ] \ldots \text{end-of-line} \\
& \mid //+ \text{non-letter-end-of-line} \ [ \text{non-end-of-line} ] \ldots \text{end-of-line} \\
& \mid //\text{-} \text{non-letter-end-of-line} \ [ \text{non-end-of-line} ] \ldots \text{end-of-line}
\end{align*}
\]
non-letter-end-of-line ::= any character except _, a through z, A through Z, a newline, or a carriage return
non-end-of-line ::= any character except a newline or carriage return
non-at-plus-minus-end-of-line ::= any character except @, +, -, newline, or carriage return
non-at-end-of-line ::= any character except @, newline, or carriage return
annotation-marker ::= 
  | /* [ annotation-key ] . . . @ [ ignored-at-in-annotation ] . . . 
  | [ ignored-at-in-annotation ] . . . @*/
  | [ ignored-at-in-annotation ] . . . */
annotation-key ::= positive-key | negative-key
positive-key ::= + ident
negative-key ::= - ident
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 ] . . .
jml-specs ::= jml-tag [ methodSpecification ] 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 ::= 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
Appendix C: Grammar Summary

<table>
<thead>
<tr>
<th>final</th>
<th>finally</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>for</td>
<td>goto</td>
<td>if</td>
</tr>
<tr>
<td>implements</td>
<td>import</td>
<td>instanceof</td>
</tr>
<tr>
<td>int</td>
<td>interface</td>
<td>long</td>
</tr>
<tr>
<td>native</td>
<td>new</td>
<td>null</td>
</tr>
<tr>
<td>package</td>
<td>private</td>
<td>protected</td>
</tr>
<tr>
<td>public</td>
<td>return</td>
<td>short</td>
</tr>
<tr>
<td>static</td>
<td>strictfp</td>
<td>super</td>
</tr>
<tr>
<td>switch</td>
<td>synchronized</td>
<td>this</td>
</tr>
<tr>
<td>throw</td>
<td>throws</td>
<td>transient</td>
</tr>
<tr>
<td>true</td>
<td>try</td>
<td>void</td>
</tr>
<tr>
<td>volatile</td>
<td>while</td>
<td></td>
</tr>
</tbody>
</table>

```
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
  \nonnull_elements | \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 ::= \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
ensures | ensures_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
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
when | when_redundantly
working_space | working_space_redundantly
writable
jml-universe-keyword :: peer | readonly | rep
special-symbol ::= java-special-symbol | jml-special-symbol
java-special-symbol ::= java-separator | java-operator
Appendix C: Grammar Summary

java-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 ::= non-zero-digit [ digits ] [ integer-type-suffix ]
digits ::= digit [ digit ] . . .
digit ::= 0 | non-zero-digit

non-zero-digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

integer-type-suffix ::= 1 | 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 ]
| . 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
Appendix C: Grammar Summary

C.2 Compilation Units

```
compilation-unit ::= [ package-declaration ]
   [ import-declaration ] . . .
   [ top-level-declaration ] . . .
top-level-declaration ::= type-declaration
package-declaration ::= [ java-annotations ] package name ;
name ::= ident [ . ident ] . . .
import-declaration ::= [ model ] import [ static ] name-star ;
name-star ::= ident [ . ident ] . . . [ . * ]
```

C.3 Type Declarations

```
type-declaration ::= class-declaration
   | interface-declaration
   | ;
class-declaration ::= [ doc-comment ] modifiers class ident
   [ class-extends-clause ] [ implements-clause ]
class-block
class-extends-clause ::= [ extends name ]
implements-clause ::= implements name-list
name-list ::= name [ , name ] . . .
interface-extends ::= extends name-list
modifiers ::= [ modifier ] . . .
modifier ::= public | protected | private
   | abstract | static |
```
| final | synchronized
| transient | volatile
| native | strictfp
| const   // reserved but not used in Java
| java-annotation
| 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

java-annotations ::= java-annotation [ java-annotation ] . . .

java-annotation ::= @ name ( [ element-value-pairs ] . . . )
| @ name
| @ name ( element-values )
element-value-pairs ::= element-value [ , element-value ]
element-value-pair ::= ident = element-value

element-value ::= conditional-expr
| annotation
| element-value-array-initializer
element-value-array-initializer ::= ‘{’ element-values ‘}’
element-values ::= element-value [ , element-value ] . . . [ , ]

C.4 Class and Interface Member Declarations

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

member-decl ::= method-decl
| variable-definition
| class-declaration
| interface-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 ] ...
formals ::= ( [ param-declaration-list ] )
param-declaration-list ::= param-declaration
param-declaration ::= [ param-modifier ] ... type-spec ident [ dims ]
param-modifier ::= final | non_null | nullable
variable-definition ::= [ doc-comment ] ... modifiers variable-decls
variable-decls ::= [ field ] type-spec variable-declarators ;
variable-declarators ::= 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

C.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 . *
  | 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 = spec-expression ;
  | represents-keyword store-ref-expression such_that predicate ;
represents-keyword ::= represents | represents_redundantly
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

C.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
  spec-header
  [ generic-spec-body ]
  | [ spec-var-decls ]
generic-spec-body ::= simple-spec-body
  | { | generic-spec-case-seq | }
generic-spec-case-seq ::= generic-spec-case
  | { | generic-spec-case-seq | }
spec-header ::= requires-clause [ requires-clause ] . . .
spec-case-seq ::= spec-case [ also spec-case ] . . .
simple-spec-body ::= simple-spec-body-clause
  | simple-spec-body-clause
simple-spec-body-clause ::= diverges-clause
  | assignable-clause | accessible-clause
  | captures-clause | callable-clause
  | when-clause | working-space-clause
  | duration-clause | ensures-clause
  | signals-only-clause | signals-clause
  | measured-clause
heavweight-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-keyword ::= normal_behavior | normal_behaviour
| exceptional-behavior-keyword ::= exceptional_behavior | exceptional_behaviour
| spec-var-decls ::= forall-var-decls [ old-var-decls ]
| old-var-decls ::= forall-var-declarator [ forall-var-declarator ] . . .
| forall-var-declarator ::= forall [ bound-var-modifiers ] type-spec 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 ::= 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
| ensures | ensures_redundantly
| signals-only-clause ::= signals-only-keyword reference-type [ , reference-type ] . . . ;
| 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
| modifies | modifies_redundantly
| modifies | modifies_redundantly
| callable-clause ::= callable-keyword callable-methods-list ;
| callable-keyword ::= callable | callable_redundantly
| callable-methods-list ::= method-name-list | store-ref-keyword
Appendix C: Grammar Summary

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

C.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 ] . . .
maps-spec-array-dim ::= '[' spec-array-ref-expr ']'

C.8 Specification Inheritance

C.9 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
appendix c: grammar summary

[ ? conditional-expr : conditional-expr ]
equivalence-expr ::= implies-expr
    [ equivalence-op implies-expr ] ... 
equivalence-op ::= <=> | <=!=>
implies-expr ::= logical-or-expr
    [ ==> implies-non-backward-expr ]
    | logical-expr <= logical-expr
    [ <=> logical-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 ::= primary-expr [ primary-suffix ] ... [ ++ ]
    | primary-expr [ primary-suffix ] ... [ -- ]
    | built-in-type [ ‘[’ ‘]’ ] ... . class
primary-suffix ::= . ident
    | . this
    | . class
    | . new-expr
Appendix C: Grammar Summary

| . 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 |
| elmentype-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 ::= \nonnull ( 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-expression ::= \invARIANT ( 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
Appendix C: Grammar Summary

| 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

C.10 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 ] . . .
loop-stmt ::= while ( expression ) statement
| do statement while ( expression ) ;
| for ( [ for-init ]; [ expression ]; [ expression-list ] )
  statement
| for ( modifiers type-spec ident : expression )
  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

C.11 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
C.12 Model Programs

\[
\text{model-program} := [ \text{privacy} ] [ \text{code} ] \text{model_program} \\
\quad \text{jml-compound-statement} \\
\text{jml-statement} := \text{statement} \\
\text{model-prog-statement} := \text{nondeterministic-choice} \\
\quad \text{nondeterministic-if} \\
\quad \text{spec-statement} \\
\quad \text{invariant} \\
\text{nondeterministic-choice} := \text{choose alternative-statements} \\
\text{alternative-statements} := \text{jml-compound-statement} \\
\quad \text{or jml-compound-statement} \ldots \\
\text{nondeterministic-if} := \text{choose_if guarded-statements} \\
\quad \text{else jml-compound-statement} \\
\text{guarded-statements} := \text{guarded-statement} \\
\quad \text{or guarded-statement} \ldots \\
\text{guarded-statement} := \{ \\
\quad \text{assume-statement} \\
\quad \text{jml-statement [ jml-statement] \ldots } \} \\
\text{spec-statement} := [ \text{privacy} ] \text{behavior-keyword} \\
\quad \text{generic-spec-statement-case} \\
\quad [ \text{privacy} ] \text{exceptional-behavior-keyword} \\
\text{exceptional-spec-case} \\
\quad [ \text{privacy} ] \text{normal-behavior-keyword} \\
\text{normal-spec-case} \\
\quad [ \text{privacy} ] \text{abrupt-behavior-keyword}
\]
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-case-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 | accessible-clause
   | captures-clause | callable-clause
   | when-clause | working-space-clause | duration-clause
   | ensures-clause | signals-only-clause | signals-clause
   | measured-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

C.13 Specification for Subtypes

C.14 Separate Files for Specifications

C.15 Universe Type System

ownership-modifiers ::= ownership-modifier [ ownership-modifier ]
ownership-modifier ::= \rep | \peer | \readonly
   | reserved-ownership-modifier // with --universes parse or --universes full
reserved-ownership-modifier ::= rep | peer | readonly
C.16 Safe Math Extensions

annotation-marker ::= 
  | /++@ [ @ ] . . .
  | /-@ @ . . .
  | /*-@ [ @ ] . . .
represents-clause ::= represents-keyword store-ref-expression <- spec-expression ;
monitors-for-clause ::= monitors_for ident
compilation-unit ::= [ package-declaration ]
   refine-prefix
   [ import-declaration ] . . .
   [ top-level-declaration ] . . .
refine-prefix ::= refine-keyword string-literal ;
refine-keyword ::= refine | refines
## Appendix D Modifier Summary

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

<table>
<thead>
<tr>
<th>Grammatical construct</th>
<th>Java modifiers</th>
<th>JML modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modifiers</td>
<td>public protected</td>
<td>spec_public spec_protected</td>
</tr>
<tr>
<td></td>
<td>private abstract</td>
<td>protected model</td>
</tr>
<tr>
<td></td>
<td>static final</td>
<td>ghost pure instance</td>
</tr>
<tr>
<td></td>
<td>synchronized</td>
<td>helper non_null</td>
</tr>
<tr>
<td></td>
<td>transient volatile</td>
<td>nullable nullable_by_default</td>
</tr>
<tr>
<td></td>
<td>native strictfp</td>
<td>by_default monitored</td>
</tr>
<tr>
<td></td>
<td></td>
<td>uninitialized</td>
</tr>
<tr>
<td>Class declaration</td>
<td>public final</td>
<td>pure model</td>
</tr>
<tr>
<td></td>
<td>abstract strictfp</td>
<td>nullable_by_default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>spec_public</td>
</tr>
<tr>
<td></td>
<td></td>
<td>spec_protected</td>
</tr>
<tr>
<td>Interface declaration</td>
<td>public strictfp</td>
<td>pure model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nullable_by_default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>spec_public</td>
</tr>
<tr>
<td></td>
<td></td>
<td>spec_protected</td>
</tr>
<tr>
<td>Nested Class declaration</td>
<td>public protected</td>
<td>spec_public spec_protected</td>
</tr>
<tr>
<td></td>
<td>private static final</td>
<td>protected model</td>
</tr>
<tr>
<td></td>
<td>abstract strictfp</td>
<td>pure</td>
</tr>
<tr>
<td>Nested interface declaration</td>
<td>public protected</td>
<td>spec_public spec_protected</td>
</tr>
<tr>
<td></td>
<td>private static</td>
<td>protected model</td>
</tr>
<tr>
<td></td>
<td>strictfp</td>
<td>pure</td>
</tr>
<tr>
<td>Local Class (and local model class) declaration</td>
<td>final abstract</td>
<td>pure model</td>
</tr>
<tr>
<td></td>
<td>strictfp</td>
<td></td>
</tr>
<tr>
<td>Type specification (e.g. invariant)</td>
<td>public protected</td>
<td>instance</td>
</tr>
<tr>
<td></td>
<td>private static</td>
<td></td>
</tr>
<tr>
<td>Field declaration</td>
<td>public protected</td>
<td>spec_public spec_</td>
</tr>
<tr>
<td></td>
<td>private final</td>
<td>protected non_null</td>
</tr>
<tr>
<td></td>
<td>volatile transient</td>
<td>nullable instance</td>
</tr>
<tr>
<td></td>
<td>static</td>
<td>monitored</td>
</tr>
<tr>
<td>Ghost Field declaration</td>
<td>public protected</td>
<td>non_null nullable</td>
</tr>
<tr>
<td></td>
<td>private static final</td>
<td>instance monitored</td>
</tr>
<tr>
<td>Model Field declaration</td>
<td>public protected</td>
<td>non_null nullable</td>
</tr>
<tr>
<td></td>
<td>private static</td>
<td>instance</td>
</tr>
<tr>
<td>Method declaration in a class</td>
<td>public protected</td>
<td>spec_public spec_</td>
</tr>
<tr>
<td></td>
<td>private abstract</td>
<td>protected pure</td>
</tr>
<tr>
<td></td>
<td>final static</td>
<td>non_null nullable</td>
</tr>
<tr>
<td></td>
<td>synchronized native</td>
<td>helper extract</td>
</tr>
<tr>
<td></td>
<td>strictfp</td>
<td></td>
</tr>
<tr>
<td>Method declaration in an interface</td>
<td>public abstract</td>
<td>spec_public spec_</td>
</tr>
<tr>
<td></td>
<td></td>
<td>protected pure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>non_null nullable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>helper</td>
</tr>
<tr>
<td>Constructor declaration</td>
<td>public protected</td>
<td>spec_public spec_</td>
</tr>
<tr>
<td></td>
<td>private</td>
<td>protected helper pure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>extract</td>
</tr>
<tr>
<td>Model method (in a class or interface)</td>
<td>public protected</td>
<td>pure non_null</td>
</tr>
<tr>
<td></td>
<td>private abstract</td>
<td>nullable helper</td>
</tr>
<tr>
<td></td>
<td>static final</td>
<td>extract</td>
</tr>
<tr>
<td></td>
<td>synchronized</td>
<td></td>
</tr>
<tr>
<td></td>
<td>strictfp</td>
<td></td>
</tr>
<tr>
<td>Model constructor</td>
<td>public protected</td>
<td>pure helper extract</td>
</tr>
<tr>
<td></td>
<td>private</td>
<td></td>
</tr>
<tr>
<td>Java initialization block</td>
<td>static</td>
<td>-</td>
</tr>
<tr>
<td>JML initializer and static_initializer annotation</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Appendix D: Modifier Summary

Formal parameter  

<table>
<thead>
<tr>
<th>Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>final</td>
</tr>
<tr>
<td>non_null nullable</td>
</tr>
</tbody>
</table>

Local variable and local ghost variable declaration  

<table>
<thead>
<tr>
<th>Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>final</td>
</tr>
<tr>
<td>ghost non_null nullable uninitialized</td>
</tr>
</tbody>
</table>

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 E  Type Checking Summary

[[[Hope to generate this automatically]]]
Appendix F Verification Logic Summary

[[[Hope to generate this automatically]]]
Appendix G Differences

The subsections below detail the differences between the JML Common Tools release of JML and other tools and between JML and Java itself.

G.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.

G.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.
G.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.

G.2.1 Non-null by Default

As described earlier (see Section 2.8 [Null is Not the Default], page 16), JML does not, by default, allow null to be a value in a field, formal parameter, method or a bound variable (see Section 12.4.24.5 [Modifiers for Bound Variables], page 103). 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 6.2.13 [Nullity Modifiers], page 44, for more details.
Appendix H  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

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


[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.


[Fitzgerald-Larsen98]  

[Gosling-etal00]  

[Gosling-etal05]  

[Gries-Schneider95]  

[Guttag-Horning-Wing85b]  

[Guttag-Horning93]  

[Hall90]  

[Hayes93]  

[Hesselink92]  

[Hoare69]  

[Hoare72a]  

[Huisman01]  

[ISO96]  
International Standards Organization. *Information Technology - Programming Languages, Their Environments and System Software Interfaces - Vienna Devel-

[Khurshid-Marinov-Jackson02]

[Jacobs-etal98]


[Kiczales-Lamping92]

[Kiniry-Cok04]

[Krone-Ogden-Sitaraman03]

[Lamport89]

[LeavensLarchFAQ]

[Leavens-Baker99]
Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In Jeannette M. Wing, Jim Wood-


[Leavens-Weihl90]

[Leavens-Weihl95]

[Leavens-Wing98]

[Leavens90]

[Leavens91]

[Leavens96b]

[Leavens97c]

[Leavens06b]

[Ledgard80]


Index

! .......................... 32, 91
!= ........................................ 32, 91
" .......................... 33
" .......................... 33
$ .......................... 29
$% .......................... 32, 91
%= ........................................ 32, 91
& .......................... 32, 91
&= ........................................ 32, 91
, .......................... 33
( .......................... 32, 41, 46, 57, 78, 91, 93, 94, 95, 96, 97, 98, 99,
 100, 101, 108, 109, 126
(* .......................... 34
) .......................... 32, 34, 41, 46, 57, 78, 91, 93, 94, 95, 96, 97, 98,
 99, 100, 101, 108, 109, 126
* .......................... 27, 28, 29, 32, 34, 36, 88, 91, 106
*/ ........................................ 34
*/= ........................................ 32, 91
+ .......................... 27, 32, 33, 91
+= ........................................ 32, 91
+= ........................................ 32, 91

; .......................... 25, 32, 37, 41, 45, 46, 49, 57, 79, 83, 87, 90, 91,
 93, 101, 106
- .......................... 27, 32, 33, 91
-= ........................................ 32, 91
-> ........................................ 32, 126
-@list suffix ................................ 25
-@seq suffix ................................ 25

. .......................... 32, 33, 36, 57, 87, 88, 91, 104, 106
. .......................... 32, 106
.. .......................... 25
/.java' .................................. 129
'.java-refined' ................................ 141
'.jml' .................................. 129
'.jml-refined' ................................ 141
'.refines-java' ................................ 141
'.refines-jml' ................................ 141
'.refines-spec' ................................ 141
'.spec' .................................. 141
'.spec-refined' ................................ 141

/ .......................... 27, 32, 91
/* .................................. 27
/** .................................. 28
/**@ .................................. 27, 141
/**@IDENT@ ................................ 27
/*@ .................................. 32, 91
/*@ .................................. 3, 27
// .................................. 27
// .................................. 3
// .................................. 3
// .................................. 27
// .................................. 141
//@ .................................. 3, 27
//= .................................. 32, 91

;

;

; .......................... 4, 26, 32, 36, 37, 45, 49, 52, 57, 60, 61, 62, 75,
 76, 77, 78, 79, 81, 82, 83, 84, 85, 87, 88, 101,
108, 109, 111, 112, 113, 114, 115, 116, 126, 141, 142

; in quantifiers ........................................ 101

<
<................................. 32, 91
<#................................. 32, 106
<=#................................. 32, 106
<................................. 32, 141
</esc>.................................. 29
</ESC.................................. 29
</jml>.................................. 29
</JML>.................................. 29
<=................................. 32, 91, 105
<<................................. 32, 91
<<=................................. 32, 91
<=................................. 32, 91
<==!=>.............................. 32, 91, 105
<==.................................. 16, 32, 91, 105
<=.................................. 32, 91, 105
<esc>............................... 29
<ESC>................................. 29
<jml>.................................. 29
</jml>.................................. 29

=
=.. .................................. 32, 41, 49, 60, 91, 101
=. used.................................. 62
===................................. 32, 91
===>................................. 16, 32, 91, 105

>
>................................. 32, 91
=>................................. 32, 91
>>................................. 32, 91
>>>................................. 32, 91
>>>=................................. 32, 91
>>>=................................. 32, 91

?
?................................. 32, 91

@param................................... 29
@deprecated.............................. 29
@author................................. 29
@version............................... 29
@see....................................... 29
@serial................................. 29
@serialdata............................. 29
@serialfield............................ 29
@since................................. 29
@throws................................. 29
@serial................................. 29

[ ....................................... 32, 50, 88, 91, 106
] ....................................... 25

\ ....................................... 32, 50, 88, 91, 106

\ ....................................... 25

\ ....................................... 32, 50, 88, 91, 106

\ ....................................... 25

\ ....................................... 32, 50, 88, 91, 106

\ ....................................... 25

\ ....................................... 32, 50, 88, 91, 106

\ ....................................... 25

\ ....................................... 32, 50, 88, 91, 106

\ ....................................... 25

\ ....................................... 32, 50, 88, 91, 106

\ ....................................... 25

\ ....................................... 32, 50, 88, 91, 106
Index

\invariant_for .................................. 30, 101
\is_initialized ................................. 30, 100
\java_math ........................................ 30
\lbneg ............................................ 30, 101
\lbpso ............................................. 30, 101
\lockset .......................................... 30, 100
\max ................................................ 5, 30, 100, 101, 102
\min ................................................ 30, 101, 102
\n .................................................. 33
\nonnull_elements ................................ 30, 99
\not_assigned .................................... 30, 94
\not_modified .................................... 30, 95
\not_specified ................................ 30, 66, 76, 84, 85, 106
\not_specified, for requires clauses .......... 76
\not_specified, meaning of .................... 66
\nothing .......................................... 5, 30, 79, 83, 84, 106
\nowarn ........................................... 30
\novarn_op ....................................... 30
\num_of .......................................... 30, 101, 103
\old .............................................. 9, 30, 93
\old, in duration-clause ....................... 86
\old, in working-space-clause ................. 85
\only_accessed .................................. 30, 95
\only_assigned .................................. 30, 96
\only_called ..................................... 30, 96
\only_captured ................................ 30, 97
\peer ............................................. 30, 133, 135
\pre ................................................ 30, 93, 94
\product ......................................... 30, 101, 102
\r .................................................. 33
\reach ............................................. 30, 97
\readonly ....................................... 30, 133, 135
\real ............................................. 30, 140
\rep .............................................. 30, 133, 134
\result .......................................... 5, 30, 93
\result, in duration-clause ................... 86
\result, in working-space-clause .............. 85
\safe_math ...................................... 30
\same ............................................. 30, 76
\same, semantics of ............................. 76
\same, used in a requires clause .............. 76
\space ............................................. 30, 98
\such_that ....................................... 30, 60
\sum ............................................. 30, 101, 102
\t .................................................. 33
\type ............................................ 30, 100
\TYPE ............................................ 30, 50
\typeof .......................................... 30, 99
\u .................................................. 33
\warn ............................................. 30
\warn_op ......................................... 30
\working_space .................................. 30, 98

| .................................................. 32, 91, 104
|= .................................................. 32, 91
|} .................................................. 32, 65, 125

A .................................................. 33
A .................................................. 33
a-z .................................................. 29
A-Z .................................................. 29
abrupt-behavior-keyword, defined ............... 125
abrupt-behavior-keyword, used ................... 125
abrupt-spec-case, defined ....................... 125

Index 184
abort-case, used .................................. 125
abort_behavior .................................. 30, 125
abstract ......................................... 30
abstract algorithm ................................ 122
abstract data type ................................ 2, 8
abstract field ..................................... 11
abstract fields .................................... 2
abstract value ..................................... 8
abstract value, of an ADT ......................... 2
access control rules ............................... 12
access control, for specification cases ........... 64
access control, in JML ............................ 12
access control, in lightweight specifications .... 13
access path ....................................... 15
accessible ........................................... 30, 68, 70, 83
accessible clause ................................ 83
accessible clause, omitted ......................... 84
accessible-clause, defined ......................... 83
accessible-clause, used ............................ 65, 125
accessible-keyword, defined ....................... 83
accessible-keyword, used .......................... 83
accessible_redundantly ............................ 30, 83
acknowledgments .................................. 9
addition, quantified see ∃ ........................... 102
additive-op, defined ................................ 91
additive-op, used .................................. 91
additive-op, defined ................................ 91
additive-op, used .................................. 91
ADT .................................................. 2
alias control ....................................... 54
alias control, universe type system for .......... 133
aliased location .................................... 15
aliases ............................................. 15
also .................................................. 30, 63, 65, 120, 125
also, former use in separate files changed ...... 143
also, in separate files .............................. 130
alternative-statements, defined ....................... 124
alternative-statements, used ......................... 124
and-op, defined .................................... 91
and-op, used ....................................... 91
annotation ......................................... 27
annotation comments ................................ 3
annotation context .................................. 12
annotation keys, syntax ............................. 27
annotation markers, syntax .......................... 27
annotation, Java ................................... 41
annotation, JML ................................... 27
annotation-key ..................................... 28
annotation-key, defined ............................. 27
annotation-key, used ................................ 27
annotation-marker, defined ......................... 27
annotation-marker, defined, deprecated ............ 141
annotation-marker, used ............................ 26
annotations and tools ................................ 28
annotations vs. comments ........................... 28
annotations, and documentation comments ....... 28
annotations, splitting across lines .................. 28
arbitrary precision arithmetic types ................. 140
Arnold .............................................. 1
array types, default ownership modifiers for .... 137
array types, ownership modifiers for ............... 136
array, element type expression ...................... 99
array, specifying elements are non-null ............ 99
array-decl, defined .................................. 91
array-decl, used ..................................... 91
array-initializer, defined ......................... 49, 91
array-initializer, used .............................. 49, 91
assert ............................................... 30, 113
assert, in JML vs. Java............................... 113
assert-redundantly-statement, defined ............... 113
assert-redundantly-statement, used ................. 113
assert-statement, defined ......................... 113
assert-statement, in JML vs. Java .................... 113
assert-statement, used .............................. 108
assert_redundantly .................................. 30, 113
assertion, expressions for use in .................... 90
assertions, and exceptions .......................... 15
assertions, validity of .............................. 15
assignable ........................................... 4, 5, 30, 68, 70, 83
assignable clause ................................... 4, 83
assignable clause, omitted ......................... 83
assignable clauses, and information hiding ........ 87
assignable clauses, and model fields ............... 83
assignable, in comparing specifications .......... 119
assignable-clause, defined ......................... 83
assignable-clause, used ............................. 125
assignable-keyword, defined ......................... 83
assignable-keyword, used .......................... 83
assignable_redundantly ............................ 30, 83
assignable-clause, used ............................. 65
assignment-op, defined ............................. 91
assignment-op, used ................................ 91
assume .............................................. 30, 114
assume-keyword, defined ............................. 114
assume-keyword, used ................................ 114
assume-statement, defined ......................... 114
assume-statement, used .............................. 113, 124
assume_redundantly ................................ 30, 114
assuming, an invariant ............................. 53
augmented pre-state .................................. 69
axiom .............................................. 30, 61
axiom, frame ....................................... 83
axiom-clause, defined ............................... 61
axiom-clause, used .................................. 52

B
b .................................................... 33
B .................................................... 33
Back ............................................... 8, 122
backslash ........................................... 33
backspace .......................................... 33
Bakus .............................................. 25
Baker ............................................... 1, 7, 12, 13, 15, 46, 118, 120
Index

Bandera ........................................ 7
behavior ...................................... 2
behavior ....................................... 12, 30, 67, 68
behavior specification cases, syntax and semantics of .................................... 67
behavior, British spelling of .................. 67
behavior, sequential ............................ 6
behavior-keyword ................................ 67
behavior-keyword, used ......................... 67
behavior-keyword, used ....................... 125
behavior-spec-case ............................. 67
behavior-spec-case, defined ................... 67
behavioral interface specification ............ 1
behavior .......................................... 30, 67
benefits of JML .................................. 6
big integer type ................................ 140
blank .............................................. 26
BNF notation .................................... 25
body of a quantifier ............................. 102
body, in quantifier .............................. 101
body, of method, in separate files ............ 130
body, of quantifier .............................. 5
body, of refining statement .................... 115
boolean .......................................... 30, 91
boolean-literal, defined ....................... 33
boolean-literal, used ........................... 33
Borgida ......................................... 2
bound variable, in quantifier ................... 101
bound variables, modifiers for ................ 103
bound-var-modifiers, defined .................. 103
bound-var-modifiers, used .................... 75, 101, 104
Boyland .......................................... 70
break ............................................. 30, 108
break, loops containing ....................... 111
breaks ............................................ 30, 126
breaks-clause, defined ......................... 126
breaks-clause, used ............................ 125
breaks-keyword, defined ....................... 126
breaks-keyword, used ........................... 126
breaks_redundantly ............................ 30, 126
British, spelling of behavior ................... 67
Büchi ............................................. 122
Büchi ............................................. 122
built-in-type, defined .......................... 91
built-in-type, used ............................ 50, 91
Burdy ............................................. 1, 6, 7
byte .............................................. 30, 91

C

c .................................................. 33
C .................................................. 33
C++-style-comment, defined .................. 27
C++-style-comment, used ..................... 27
C-Style comment ................................ 27
C-style-body, defined ......................... 27
C-style-body, used ............................. 27
C-style-comment, defined .................... 27
C-style-comment, used ........................ 27
C-style-end, defined ......................... 27
C-style-end, used .............................. 27
call, post-state of ............................. 69
call, pre-state of ............................. 69
callable ......................................... 30, 68, 70, 84
callable clause .................................. 84
callable clause, omitted ....................... 84
callable-clause, defined ....................... 84
callable-clause, used .......................... 65, 125
callable-keyword, defined .................... 84
callable-keyword, used ....................... 84
callable-methods-list, defined ............... 84
callable-methods-list, used ................... 84
callable_redundantly .......................... 30, 84
captured ........................................ 97
captures ......................................... 30, 68, 70, 84
captures clause .................................. 84
captures clause, omitted ....................... 85
captures-clause, defined ....................... 84
captures-clause, used .......................... 65, 125
captures-keyword, defined .................... 84
captures-keyword, used ....................... 84
captures_redundantly .......................... 30, 84
carriage return ................................ 26, 27, 33
carriage-return, defined ....................... 26
carriage-return, used .......................... 26
case ............................................. 30, 108
cast expressions, default ownership modifiers for types in ......................... 137
casts, and ownership types ................... 139
catch ............................................. 30, 108
Chalin .......................................... 15, 43, 140
changes, incompatible ......................... 143
char .............................................. 30, 91
character-literal, defined ..................... 33
character-literal, used ......................... 33
Cheon .......................................... 1, 2, 3, 7, 11
choose .......................................... 30, 124
choose_if ....................................... 30, 124
claim, procedure ............................. 119
claims, about a specification ................ 118
class ............................................. 30, 37, 91
class declaration .............................. 37
class declarations ............................. 37
class initialization predicate ................. 100
class invariant, see instance invariant ........ 56
class, inheritance ............................. 37
class, modifiers for declarations of ......... 38
class-block, defined ......................... 37
class-block, used ............................. 37, 91
class-declaration, defined .................... 37
class-declaration, used ....................... 37, 45
class-extends-clause, defined ................. 37
class-extends-clause, used .................... 37
class-initializer-decl, defined ............... 50
class-initializer-decl, used ................... 45
Clifton .......................................... 7
Index

<table>
<thead>
<tr>
<th>Code</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>30, 67, 72, 73, 124</td>
</tr>
<tr>
<td>code contract</td>
<td>127</td>
</tr>
<tr>
<td>code, modifier, semantics of</td>
<td>127</td>
</tr>
<tr>
<td>code</td>
<td>127</td>
</tr>
<tr>
<td>code_bigint_math</td>
<td>30, 38, 39, 43</td>
</tr>
<tr>
<td>code_java_math</td>
<td>30, 38, 39, 43</td>
</tr>
<tr>
<td>code_safe_math</td>
<td>30, 38, 39, 43</td>
</tr>
<tr>
<td>Cohen</td>
<td>102, 103</td>
</tr>
<tr>
<td>comment</td>
<td>168</td>
</tr>
<tr>
<td>comment, defined</td>
<td>27</td>
</tr>
<tr>
<td>comment, syntax of</td>
<td>27</td>
</tr>
<tr>
<td>comment, used</td>
<td>26</td>
</tr>
<tr>
<td>comments vs. annotations</td>
<td>28</td>
</tr>
<tr>
<td>comments, annotations in</td>
<td>3</td>
</tr>
<tr>
<td>commit</td>
<td>70</td>
</tr>
<tr>
<td>commit point</td>
<td>70</td>
</tr>
<tr>
<td>compilation unit</td>
<td>35</td>
</tr>
<tr>
<td>compilation unit, and public types</td>
<td>35</td>
</tr>
<tr>
<td>compilation unit, file name for</td>
<td>35</td>
</tr>
<tr>
<td>compilation unit, mutual recursion in</td>
<td>35</td>
</tr>
<tr>
<td>compilation unit, satisfaction of</td>
<td>35</td>
</tr>
<tr>
<td>compilation-unit, defined</td>
<td>35</td>
</tr>
<tr>
<td>completely omitted specification</td>
<td>66</td>
</tr>
<tr>
<td>completeness, of method specifications</td>
<td>5</td>
</tr>
<tr>
<td>completeness, of specification</td>
<td>4</td>
</tr>
<tr>
<td>compound-statement, defined</td>
<td>108</td>
</tr>
<tr>
<td>compound-statement, used</td>
<td>45, 50, 108, 124</td>
</tr>
<tr>
<td>concepts, fundamental</td>
<td>11</td>
</tr>
<tr>
<td>concrete field</td>
<td>11</td>
</tr>
<tr>
<td>concurrency, lack of support in JML</td>
<td>6</td>
</tr>
<tr>
<td>conditional-expr, defined</td>
<td>91</td>
</tr>
<tr>
<td>conditional-expr, used</td>
<td>91</td>
</tr>
<tr>
<td>const</td>
<td>30, 39</td>
</tr>
<tr>
<td>constant, defined</td>
<td>91</td>
</tr>
<tr>
<td>constant, used</td>
<td>91</td>
</tr>
<tr>
<td>constrained-list, defined</td>
<td>57</td>
</tr>
<tr>
<td>constrained-list, used</td>
<td>57</td>
</tr>
<tr>
<td>constraint</td>
<td>30, 57</td>
</tr>
<tr>
<td>Constraint</td>
<td>58</td>
</tr>
<tr>
<td>constraint, instance vs. static</td>
<td>59</td>
</tr>
<tr>
<td>constraint, static vs. instance</td>
<td>59</td>
</tr>
<tr>
<td>constraint-keyword, defined</td>
<td>57</td>
</tr>
<tr>
<td>constraint-keyword, used</td>
<td>57</td>
</tr>
<tr>
<td>constraint_redundantly</td>
<td>30, 57</td>
</tr>
<tr>
<td>constraints, vs. helper</td>
<td>42</td>
</tr>
<tr>
<td>constructor</td>
<td>30, 45</td>
</tr>
<tr>
<td>constructor specification</td>
<td>63</td>
</tr>
<tr>
<td>constructor specifications, and \fresh</td>
<td>97</td>
</tr>
<tr>
<td>constructor, and invariants</td>
<td>53</td>
</tr>
<tr>
<td>constructor, default, specification of</td>
<td>66</td>
</tr>
<tr>
<td>constructor, helper</td>
<td>48</td>
</tr>
<tr>
<td>constructor, model</td>
<td>46</td>
</tr>
<tr>
<td>constructor, pure</td>
<td>47</td>
</tr>
<tr>
<td>context, ownership</td>
<td>134</td>
</tr>
<tr>
<td>continue</td>
<td>30, 108, 112</td>
</tr>
<tr>
<td>continues</td>
<td>30, 126</td>
</tr>
<tr>
<td>continues-clause, defined</td>
<td>126</td>
</tr>
<tr>
<td>continues-clause, used</td>
<td>125</td>
</tr>
<tr>
<td>continues-keyword, defined</td>
<td>126</td>
</tr>
<tr>
<td>continues-keyword, used</td>
<td>126</td>
</tr>
<tr>
<td>continues_redundantly</td>
<td>30, 126</td>
</tr>
<tr>
<td>Corbett</td>
<td>7</td>
</tr>
<tr>
<td>current ownership context</td>
<td>135</td>
</tr>
<tr>
<td>cycle, virtual machine</td>
<td>98</td>
</tr>
</tbody>
</table>

D

<table>
<thead>
<tr>
<th>D</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>33</td>
</tr>
<tr>
<td>d</td>
<td>33</td>
</tr>
<tr>
<td>datatypes</td>
<td>8</td>
</tr>
<tr>
<td>data group</td>
<td>87</td>
</tr>
<tr>
<td>debug</td>
<td>116</td>
</tr>
<tr>
<td>debug-statement, defined</td>
<td>116</td>
</tr>
<tr>
<td>debug-statement, used</td>
<td>113</td>
</tr>
<tr>
<td>decimal-integer-literal, defined</td>
<td>33</td>
</tr>
<tr>
<td>decimal-integer-literal, used</td>
<td>33</td>
</tr>
<tr>
<td>decreases</td>
<td>30, 112</td>
</tr>
<tr>
<td>decreases_redundantly</td>
<td>30, 112</td>
</tr>
<tr>
<td>decreasing</td>
<td>30, 112</td>
</tr>
<tr>
<td>decreasing-keyword, defined</td>
<td>112</td>
</tr>
<tr>
<td>decreasing-keyword, used</td>
<td>112</td>
</tr>
<tr>
<td>decreasing_redundantly</td>
<td>30, 112</td>
</tr>
<tr>
<td>default</td>
<td>30, 108</td>
</tr>
<tr>
<td>default access</td>
<td>12</td>
</tr>
<tr>
<td>default constructor, specification of</td>
<td>66</td>
</tr>
<tr>
<td>default ownership modifiers for types</td>
<td>137</td>
</tr>
<tr>
<td>default signals clause, and RuntimeExceptions</td>
<td>80</td>
</tr>
<tr>
<td>defaults, for lightweight specification cases</td>
<td>66</td>
</tr>
<tr>
<td>depends, replaced by in and maps</td>
<td>142</td>
</tr>
<tr>
<td>deprecated syntax</td>
<td>141</td>
</tr>
<tr>
<td>description, defined</td>
<td>29</td>
</tr>
<tr>
<td>description, used</td>
<td>29</td>
</tr>
<tr>
<td>design, documentation of</td>
<td>7</td>
</tr>
<tr>
<td>destructor, and invariants</td>
<td>53</td>
</tr>
<tr>
<td>deterministic, pure method</td>
<td>48</td>
</tr>
<tr>
<td>Dhara</td>
<td>38, 89, 122, 128</td>
</tr>
<tr>
<td>Dietl</td>
<td>30, 35, 133, 134, 138</td>
</tr>
<tr>
<td>digit</td>
<td>33</td>
</tr>
<tr>
<td>digit, defined</td>
<td>29, 33</td>
</tr>
<tr>
<td>digit, used</td>
<td>29, 33</td>
</tr>
<tr>
<td>digits, defined</td>
<td>33</td>
</tr>
<tr>
<td>digits, used</td>
<td>33</td>
</tr>
<tr>
<td>dim-exps, defined</td>
<td>91</td>
</tr>
<tr>
<td>dim-exps, used</td>
<td>91</td>
</tr>
<tr>
<td>dims, defined</td>
<td>50</td>
</tr>
<tr>
<td>dims, used</td>
<td>45, 46, 49, 50, 57, 91, 101</td>
</tr>
<tr>
<td>Directory</td>
<td>122</td>
</tr>
<tr>
<td>diverges</td>
<td>30, 68, 69, 81</td>
</tr>
<tr>
<td>Diverges</td>
<td>82</td>
</tr>
<tr>
<td>diverges clause</td>
<td>81</td>
</tr>
<tr>
<td>diverges clause, omitted</td>
<td>81</td>
</tr>
<tr>
<td>diverges-clause, defined</td>
<td>81</td>
</tr>
<tr>
<td>diverges-clause, used</td>
<td>65, 125</td>
</tr>
<tr>
<td>diverges-keyword, defined</td>
<td>81</td>
</tr>
</tbody>
</table>
ensures-clause, used ........................................... 65, 125
ensures-clause, defined ........................................ 65
ensures-keyword, defined ...................................... 77
ensures-keyword, used ......................................... 77
ensures-redundantly ................................................ 30, 77
equality-expr, defined .......................................... 91
equality-expr, used ............................................. 91
equivalence-expr, defined ...................................... 91
equivalence-expr, used ......................................... 91
equivalence-op, defined ........................................ 91
equivalence-op, used ............................................ 91
Eiffel ........................................................................ 7
Errors and method semantics ..................................... 69
ESC/Java ................................................................. 1, 7, 14, 28, 100
ESC/Java, differences from JML ................................ 168
ESC/Java2 ................................................................ 14, 28
ESC/Java2, differences from JML ................................. 168
escape-sequence, defined ......................................... 33
escape-sequence, used ............................................. 33
establishing, an invariant ........................................ 53
element ................................................................. 30, 120
element, defaults for ............................................... 120
element, defined ..................................................... 120
element, lightweight ............................................... 120
element, used .......................................................... 120
elements, checking ................................................... 120
elements, defined .................................................... 120
elements, meaning ................................................... 120
elements, semantics ................................................ 120
elements, specification of ......................................... 120
elements, used ........................................................ 118
exceptional-behavior-keyword, defined ...................... 73
exceptional-behavior-keyword, used ........................... 73
exceptional-behavior-spec-case, defined ...................... 73
exceptional-behavior-spec-case, used .......................... 67
exceptional-example-body, defined ............................. 120
exceptional-example-body, used ................................. 120
exceptional-spec-case, defined .................................. 73
exceptional-spec-case, used ...................................... 73, 120, 125
exceptional_behavior ............................................... 12, 30, 73
exceptional_behaviour ............................................. 30, 73
exceptional_example ............................................... 30, 120
exceptions ........................................................... 120
exceptions, and method specification semantics ............. 70
exceptions, avoiding in assertion evaluation .................. 70
exceptions, prohibiting ............................................ 4
exceptions, specifying when they must be thrown .......... 79
exclusive-or-expr, defined ........................................ 91
exclusive-or-expr, used .......................................... 91
executeability of quantified expressions ....................... 103
experimental, features of JML ................................... 18
explicitly nullable .................................................. 44
exponent-indicator, defined ..................................... 33
<table>
<thead>
<tr>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>ident, defined .................................................. 29</td>
</tr>
<tr>
<td>ident, used .... 26, 36, 37, 41, 45, 46, 49, 57, 61, 62, 78, 87, 88, 91, 93, 101, 104, 106, 108, 109, 126</td>
</tr>
<tr>
<td>identifiers ...................................................... 29</td>
</tr>
<tr>
<td>if ................................................................. 30, 61, 62, 84, 85, 108</td>
</tr>
<tr>
<td>ignored-at-in-annotation, defined ................................ 27</td>
</tr>
<tr>
<td>ignored-at-in-annotation, used ................................... 27</td>
</tr>
<tr>
<td>immutable .......................................................... 39</td>
</tr>
<tr>
<td>immutable, vs. pure ............................................... 39</td>
</tr>
<tr>
<td>implementation of interfaces ...................................... 37</td>
</tr>
<tr>
<td>implements .......................................................... 30, 37</td>
</tr>
<tr>
<td>implements, for classes ............................................ 38</td>
</tr>
<tr>
<td>implements-clause, defined ......................................... 37</td>
</tr>
<tr>
<td>implements-clause, used ............................................ 37</td>
</tr>
<tr>
<td>implication, redundant .............................................. 118</td>
</tr>
<tr>
<td>implication, see =&gt; ................................................. 105</td>
</tr>
<tr>
<td>implications, defined ............................................... 118</td>
</tr>
<tr>
<td>implications, used .................................................. 118</td>
</tr>
<tr>
<td>implicitly nullable .................................................. 44</td>
</tr>
<tr>
<td>ImplicitOld ............................................................ 81</td>
</tr>
<tr>
<td>implies-expr, defined ............................................... 91</td>
</tr>
<tr>
<td>implies-expr, used .................................................. 91</td>
</tr>
<tr>
<td>implies-non-backward-expr, defined ................................ 91</td>
</tr>
<tr>
<td>implies-non-backward-expr, used ................................... 91</td>
</tr>
<tr>
<td>implies_that .......................................................... 30, 118</td>
</tr>
<tr>
<td>import ................................................................. 30, 36</td>
</tr>
<tr>
<td>import declaration .................................................. 36</td>
</tr>
<tr>
<td>import, model .......................................................... 36</td>
</tr>
<tr>
<td>import-declaration, defined ......................................... 36</td>
</tr>
<tr>
<td>import-declaration, used ............................................. 35</td>
</tr>
<tr>
<td>incompatible changes ................................................ 143</td>
</tr>
<tr>
<td>InconsistentMethodSpec ............................................... 73</td>
</tr>
<tr>
<td>InconsistentMethodSpec2 ............................................... 74</td>
</tr>
<tr>
<td>infinite precision numeric types ................................... 140</td>
</tr>
<tr>
<td>influences, on JML evolution ...................................... 7</td>
</tr>
<tr>
<td>informal descriptions ................................................. 34</td>
</tr>
<tr>
<td>informal-description, defined ..................................... 34</td>
</tr>
<tr>
<td>informal-description, used ........................................ 26, 92, 106</td>
</tr>
<tr>
<td>information hiding, in assignable clauses .......................... 87</td>
</tr>
<tr>
<td>inheritance ............................................................. 37</td>
</tr>
<tr>
<td>inheritance, multiple ................................................ 38</td>
</tr>
<tr>
<td>inheritance, of JML features ...................................... 38, 89</td>
</tr>
<tr>
<td>inheritance, of model methods from interfaces ...................... 38</td>
</tr>
<tr>
<td>inheritance, of specifications ...................................... 38, 89</td>
</tr>
<tr>
<td>inherits ............................................................... 37</td>
</tr>
<tr>
<td>initialization, specification that a class is ...................... 100</td>
</tr>
<tr>
<td>initializer ............................................................. 30</td>
</tr>
<tr>
<td>initializer, defined ................................................. 49, 91</td>
</tr>
<tr>
<td>initializer, separate files ......................................... 131</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hall ................................................................. 7</td>
</tr>
<tr>
<td>handbook, for LSL ................................................... 8</td>
</tr>
<tr>
<td>Handbook, for LSL .................................................... 8</td>
</tr>
<tr>
<td>handler, defined .................................................... 108</td>
</tr>
<tr>
<td>has ................................................................. 104</td>
</tr>
<tr>
<td>Hayes ................................................................. 2, 8</td>
</tr>
<tr>
<td>Heavyweight ........................................................... 66</td>
</tr>
<tr>
<td>heavyweight example .................................................. 120</td>
</tr>
<tr>
<td>heavyweight specification .......................................... 4, 12</td>
</tr>
<tr>
<td>heavyweight specification case ..................................... 67</td>
</tr>
<tr>
<td>heavyweight specification, vs. lightweight .......................... 4</td>
</tr>
<tr>
<td>heavyweight-spec-case, defined ..................................... 67</td>
</tr>
<tr>
<td>heavyweight-spec-case, used ......................................... 64</td>
</tr>
<tr>
<td>helper ................................................................. 30, 39, 42, 48, 53, 55</td>
</tr>
<tr>
<td>helper constructor, and invariants .................................. 53</td>
</tr>
<tr>
<td>helper method, and invariants ........................................ 53</td>
</tr>
<tr>
<td>helper, and invariants ................................................ 42</td>
</tr>
<tr>
<td>helper, and private ................................................... 42</td>
</tr>
<tr>
<td>hence-by-keyword, defined ........................................... 116</td>
</tr>
<tr>
<td>hence-by-keyword, used .............................................. 116</td>
</tr>
<tr>
<td>hence-by-statement, defined ......................................... 116</td>
</tr>
<tr>
<td>hence-by-statement, used ............................................. 113</td>
</tr>
<tr>
<td>hence_by ............................................................. 30, 116</td>
</tr>
<tr>
<td>hence_by_redundantly ................................................ 30, 116</td>
</tr>
<tr>
<td>hex-digit, defined ................................................... 33</td>
</tr>
<tr>
<td>hex-digit, used ........................................................ 33</td>
</tr>
<tr>
<td>hex-integer-literal, defined ......................................... 33</td>
</tr>
<tr>
<td>hex-integer-literal, used ............................................. 33</td>
</tr>
<tr>
<td>hex-numeral, defined .................................................. 33</td>
</tr>
<tr>
<td>hex-numeral, used ..................................................... 33</td>
</tr>
<tr>
<td>higher-order method specification ................................... 122</td>
</tr>
<tr>
<td>history constraint .................................................... 38, 89</td>
</tr>
<tr>
<td>history constraints, vs. helper ...................................... 42</td>
</tr>
<tr>
<td>history-constraint, defined .......................................... 57</td>
</tr>
<tr>
<td>history-constraint, used ............................................. 52</td>
</tr>
<tr>
<td>Hoare ................................................................. 8, 11</td>
</tr>
<tr>
<td>Holmes ................................................................. 1</td>
</tr>
<tr>
<td>Hornig ................................................................. 1, 7, 8</td>
</tr>
<tr>
<td>Huisman ................................................................. 7</td>
</tr>
<tr>
<td>Hussain, Faraz .......................................................... 9</td>
</tr>
<tr>
<td>Keyword</td>
</tr>
<tr>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>initializer, used</td>
</tr>
<tr>
<td>initializer, used</td>
</tr>
<tr>
<td>initializer, used</td>
</tr>
<tr>
<td>initializer-list, defined</td>
</tr>
<tr>
<td>initializer-list, used</td>
</tr>
<tr>
<td>initializers, for fields</td>
</tr>
<tr>
<td>initializers, separate files</td>
</tr>
<tr>
<td>initially</td>
</tr>
<tr>
<td>initially, clause and separate files</td>
</tr>
<tr>
<td>initially-clause, defined</td>
</tr>
<tr>
<td>initially-clause, used</td>
</tr>
<tr>
<td>instance</td>
</tr>
<tr>
<td>instance constraint</td>
</tr>
<tr>
<td>instance features</td>
</tr>
<tr>
<td>instance invariant</td>
</tr>
<tr>
<td>instance vs. final, in interfaces</td>
</tr>
<tr>
<td>instance vs. static</td>
</tr>
<tr>
<td>instanceof</td>
</tr>
<tr>
<td>instanceof, and ownership types</td>
</tr>
<tr>
<td>instanceof, default ownership modifiers for</td>
</tr>
<tr>
<td>in</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>integer-literal, defined</td>
</tr>
<tr>
<td>integer-literal, used</td>
</tr>
<tr>
<td>integer-type-suffix, defined</td>
</tr>
<tr>
<td>integer-type-suffix, used</td>
</tr>
<tr>
<td>interface</td>
</tr>
<tr>
<td>interface declaration</td>
</tr>
<tr>
<td>interface declarations</td>
</tr>
<tr>
<td>interface declarations, used</td>
</tr>
<tr>
<td>interface specification</td>
</tr>
<tr>
<td>interface, field</td>
</tr>
<tr>
<td>interface, method</td>
</tr>
<tr>
<td>interface, modifiers for declarations of</td>
</tr>
<tr>
<td>interface, type</td>
</tr>
<tr>
<td>interface-declaration, defined</td>
</tr>
<tr>
<td>interface-declaration, used</td>
</tr>
<tr>
<td>interface-extends, defined</td>
</tr>
<tr>
<td>interface-extends, used</td>
</tr>
<tr>
<td>interfaces, and default modifier for fields</td>
</tr>
<tr>
<td>interfaces, and ghost fields</td>
</tr>
<tr>
<td>interfaces, and model fields</td>
</tr>
<tr>
<td>IntHeap</td>
</tr>
<tr>
<td>invariant</td>
</tr>
<tr>
<td>invariant</td>
</tr>
<tr>
<td>Invariant</td>
</tr>
<tr>
<td>Invariant, and helper constructors</td>
</tr>
<tr>
<td>invariant, and helper methods</td>
</tr>
<tr>
<td>invariant, assuming</td>
</tr>
<tr>
<td>invariant, defined</td>
</tr>
<tr>
<td>invariant, enforcement</td>
</tr>
<tr>
<td>invariant, establishing</td>
</tr>
<tr>
<td>invariant, for an object</td>
</tr>
<tr>
<td>invariant, instance</td>
</tr>
<tr>
<td>invariant, instance vs. static</td>
</tr>
<tr>
<td>invariant, preserving</td>
</tr>
<tr>
<td>invariant, reasoning about</td>
</tr>
<tr>
<td>invariant, static</td>
</tr>
<tr>
<td>invariant, static vs. instance</td>
</tr>
<tr>
<td>invariant</td>
</tr>
<tr>
<td>invariant-for-expression, defined</td>
</tr>
<tr>
<td>invariant-for-expression, used</td>
</tr>
<tr>
<td>invariant-keyword, defined</td>
</tr>
<tr>
<td>invariant-keyword, used</td>
</tr>
<tr>
<td>invariant_redundantly</td>
</tr>
<tr>
<td>isAssignedFrom, method of java.lang.Class</td>
</tr>
<tr>
<td>ISO</td>
</tr>
</tbody>
</table>

**J**

Jackson .................................................. 127
Jacobs .................................................. 7, 9
Java .................................................... 1
Java annotation ........................................ 41
'java' filename suffix ................................ 129
Java modifiers .......................................... 40
Java reserved words .................................... 30
Java virtual machine error, and method semantics 69
Java vs. JML-only names, resolving conflicts ...... 11
java-annotation, defined .............................. 41
java-annotation, used ................................ 39, 41
java-annotations, defined ............................. 41
java-annotations, used ................................ 36
java-literal, defined ................................. 33
java-literal, used ...................................... 26, 91
java-operator, defined .............................. 32
java-operator, used ................................... 32
java-reserved-word, defined ......................... 30
java-reserved-word, used .............................. 30
java-separator, defined .............................. 32
java-separator, used .................................. 32
java-special-symbol, defined ......................... 32
java-special-symbol, used ............................ 32
java-universe-reserved, defined ..................... 30
java.lang.Class, and \\TYPE ......................... 50
javadoc .................................................. 29
jml-annotation-statement, defined ................ 113
jml-annotation-statement, used ...................... 108
jml-compound-statement, defined .................... 124
jml-compound-statement, used ......................... 124
| jml-data-group-clause, defined | 87 |
| jml-data-group-clause, used | 49 |
| jml-declaration, defined | 52 |
| jml-declaration, used | 45 |
| jml-keyword, defined | 30 |
| jml-keyword, used | 30 |
| jml-modifier, defined | 39 |
| jml-modifier, used | 39 |
| JML-only vs. Java names, resolving conflicts | 11 |
| jml-predicate-keyword, defined | 30 |
| jml-predicate-keyword, used | 30 |
| jml-primary, defined | 92 |
| jml-primary, used | 91 |
| jml-special-symbol, defined | 32 |
| jml-special-symbol, used | 32 |
| jml-specs, defined | 29 |
| jml-specs, used | 29 |
| jml-statement, defined | 124 |
| jml-statement, used | 124 |
| jml-tag, defined | 29 |
| jml-tag, used | 29 |
| jml-universe-keyword, defined | 30 |
| jml-universe-keyword, used | 30 |
| jml-universe-pkeyword, defined | 30 |
| jml-universe-pkeyword, used | 30 |
| jnle | 7 |
| jnle, warnings for non-executable assertions | 103 |
| jnldoc | 7 |
| Jones | 8 |
| Joy | 32, 33 |

**K**

| key, negative | 28 |
| key, positive | 28 |
| keys, for conditional annotation, syntax | 27 |
| keyword, defined | 30 |
| keyword, used | 26 |
| keywords | 29 |
| Khurshid | 127 |
| Kiczales | 127 |
| Kiniry | 168 |

**L**

| 1 | 33 |
| L | 33 |
| label expression (negative) | 101 |
| label expression (positive) | 101 |
| Lamping | 127 |
| Lamport | 1 |
| language level 0 features | 18 |
| language level 1 features | 21 |
| language level 2 features | 22 |
| language level 3 features | 24 |
| language level C features | 24 |
| language level X features | 24 |
| language levels | 17 |
| language levels, and learning JML | 17 |
| language levels, and tools | 18 |
| Larch | 1, 8 |
| Larch Shared Language (LSL) | 1 |
| Larch style specification language | 1 |
| Larch/C++ | 8 |
| Larsen | 8 |
| learning JML, and language levels | 17 |
| Leavens | 1, 4, 7, 8, 12, 13, 15, 16, 38, 42, 46, 54, 89, 118, 120, 122, 127, 128, 134 |

**K**

| Ledgard | 25 |
| Leino | 1, 7, 9, 11, 14, 28, 43, 50, 100, 168 |
| letter, defined | 29 |
| letter, used | 26, 29 |
| letter-or-digit, defined | 29 |
| letter-or-digit, used | 29 |
| level 0, JML features | 17, 18 |
| level 1, JML features | 17, 21 |
| level 2, JML features | 17, 22 |
| level 3, JML features | 17, 24 |
| level C, JML features | 17, 24 |
| level X, JML features | 17, 24 |
| levels, of language support | 17 |
| lexeme, defined | 26 |
| lexeme, used | 26 |
| lexical conventions | 26 |
| lexical-pragmas, defined | 26 |
| lexical-pragmas, used | 26 |
| Lightweight | 65 |
| lightweight example | 120 |
| lightweight specification | 12 |
| lightweight specification case | 65 |
| lightweight specification, example of | 5 |
| lightweight specification, vs. heavyweight | 4 |
| lightweight specifications and access control | 13 |
| lightweight-spec-case, defined | 65 |
| lightweight-spec-case, used | 64 |
| Liskov | 5, 8 |
| list vs. sequence, in grammar | 25 |
| literals | 33 |
| local-declaration, defined | 108 |
| local-declaration, used | 108, 109 |
| local-modifier, defined | 109 |
| local-modifier, used | 109 |
| local-modifiers, defined | 109 |
| local-modifiers, used | 108 |
| location | 5, 15, 87 |
| locking order | 106 |
| locks held by a thread | 100 |
| lockset-expression, defined | 100 |
| lockset-expression, used | 92 |
| logic, three-valued | 15 |
| logic, two-valued | 15 |
Index

logical implication, see $implies$ .................................. 105
logical rules, valid in JML .......................................... 15
logical-and-expr, defined ........................................... 91
logical-and-expr, used ................................................ 91
logical-or-expr, defined ............................................. 91
logical-or-expr, used .................................................. 91
long ................................................................. 30, 91
LOOP ................................................................. 7
loop, exiting via break ............................................... 111
loop-invariant, defined ............................................... 111
loop-invariant, used .................................................. 109
loop-stmt, defined .................................................... 109
loopInvariant ......................................................... 30, 111
loopInvariant_redundantly ........................................ 30, 111
LSL ................................................................. 1
LSL Handbook ......................................................... 8

M
maintaining ........................................................... 30, 111
maintaining-keyword, defined ..................................... 111
maintaining-keyword, used ......................................... 111
maintaining_redundantly ............................................ 30, 111
maps ................................................................. 30, 88
maps-array-ref-expr, defined ...................................... 88
maps-array-ref-expr, used .......................................... 88
maps-into-clause, defined .......................................... 88
maps-into-clause, used ............................................. 87, 88
maps-keyword, defined .............................................. 88
maps-keyword, used ................................................ 88
maps-member-ref-expr, defined .................................. 88
maps-member-ref-expr, used ....................................... 88
maps-spec-array-dim, defined .................................... 88
maps-spec-array-dim, used ......................................... 88
maps_redundantly .................................................... 30, 88
Marinov ............................................................... 127
matching, of implementations to model programs .............. 123
max of a set of lock objects ....................................... 100
max-expression, defined ........................................... 100
max-expression, used .............................................. 92
maximum, see $\max$ ................................................ 102
meaning of expressions in JML .................................... 15
measured by clause .................................................. 84
measured-by-keyword, defined .................................... 84
measured-by-keyword, used ........................................ 84
measured-clause, defined .......................................... 84
measured-clause, used .............................................. 65, 125
measured_by ......................................................... 30, 68, 84
measured_by_redundantly ......................................... 30, 84
member-decl, defined .............................................. 45
member-decl, used .................................................. 45
member-field-ref, defined ......................................... 88
member-field-ref, used ............................................ 88
method .............................................................. 30, 45
method body, in separate files ................................... 130
method call, space used by ........................................ 98
method calls, and invariants ....................................... 53
method calls, and ownership typing rules ....................... 138
method declaration, in separate files ......................... 130
method refinement .................................................. 129
method specification .............................................. 63
method specification semantics, and exceptions ............... 70
method specification, omitted .................................... 66
method, behavior of ................................................ 2
method, helper ..................................................... 48
method, model ....................................................... 46
method, pure ....................................................... 46
method-body, defined ............................................. 45
method-body, used ................................................ 45
method-decl, defined .............................................. 45
method-decl, used ................................................ 45
method-head, defined ............................................. 45
method-head, used ................................................ 45
method-name, defined ............................................. 57
method-name, used ................................................ 57
method-name-list, defined ........................................ 57
method-name-list, used ........................................... 57, 84, 96
method-or-constructor-keyword, defined ....................... 45
method-or-constructor-keyword, used ......................... 45
method-ref, defined .............................................. 57
method-ref, used .................................................. 57
method-ref-rest, defined ......................................... 57
method-ref-rest, used ............................................. 57
method-ref-start, defined ........................................ 57
method-ref-start, used ............................................. 57
method-specification, defined ................................... 63
method-specification, in documentation comments ............ 29
method-specification, used ....................................... 29, 45, 50
methodology, and JML ............................................. 6
Meyer ................................................................. 1, 5, 8
microsyntax ......................................................... 26
microsyntax, defined .............................................. 26
minimum, see $\min$ ................................................ 102
model ............................................................... 3, 11, 30, 36, 38, 39, 41, 46, 49
model and final .................................................... 42
model and pure, constructors ..................................... 46
model and pure, methods .......................................... 46
model and static, in interfaces ................................... 42
model classes, vs. pure classes .................................. 48
model constructor .................................................. 46
model features ...................................................... 11
model features, and namespace issues ......................... 11
model field ........................................................ 3, 11
model fields ......................................................... 4
model fields, from $\text{spec\_protected}$ ......................... 14
model fields, from $\text{spec\_public}$ ............................ 14
model fields, in interfaces ........................................ 42
model fields, of an ADT .......................................... 3
model import ....................................................... 11
model import declaration ......................................... 36
model import, vs. import .......................................... 36
model method ....................................................... 11, 46
model method, in separate files ................................ 130
<table>
<thead>
<tr>
<th>Index</th>
<th>194</th>
</tr>
</thead>
<tbody>
<tr>
<td>model methods, vs. pure methods</td>
<td>48</td>
</tr>
<tr>
<td>model program, ideas behind</td>
<td>122</td>
</tr>
<tr>
<td>model program, matching of</td>
<td>123</td>
</tr>
<tr>
<td>model program, via extract</td>
<td>45</td>
</tr>
<tr>
<td>model type</td>
<td>11</td>
</tr>
<tr>
<td>model type, vs. pure type used for modeling</td>
<td>39</td>
</tr>
<tr>
<td>model vs. ghost</td>
<td>41</td>
</tr>
<tr>
<td>model, in separate files</td>
<td>130</td>
</tr>
<tr>
<td>model, meaning of</td>
<td>11</td>
</tr>
<tr>
<td>model, modifier in separate files</td>
<td>130</td>
</tr>
<tr>
<td>model, modifier in separate files</td>
<td>130</td>
</tr>
<tr>
<td>model, type declaration modifier</td>
<td>39</td>
</tr>
<tr>
<td>model-oriented specification</td>
<td>1</td>
</tr>
<tr>
<td>model-prog-statement, defined</td>
<td>124</td>
</tr>
<tr>
<td>model-prog-statement, used</td>
<td>108</td>
</tr>
<tr>
<td>model-program, defined</td>
<td>124</td>
</tr>
<tr>
<td>model-program, used</td>
<td>124</td>
</tr>
<tr>
<td>model_program</td>
<td>30, 124</td>
</tr>
<tr>
<td>modifiable</td>
<td>30, 83</td>
</tr>
<tr>
<td>modifiable clause</td>
<td>83</td>
</tr>
<tr>
<td>modifiable clause, omitted</td>
<td>83</td>
</tr>
<tr>
<td>modifiable_redundantly</td>
<td>30, 83</td>
</tr>
<tr>
<td>modifier ordering, suggested</td>
<td>40</td>
</tr>
<tr>
<td>modifier, defined</td>
<td>39</td>
</tr>
<tr>
<td>modifier, general description of</td>
<td>39</td>
</tr>
<tr>
<td>modifier, pure</td>
<td>41</td>
</tr>
<tr>
<td>modifier, used</td>
<td>39</td>
</tr>
<tr>
<td>modifiers for bound variables</td>
<td>103</td>
</tr>
<tr>
<td>modifiers, defined</td>
<td>39</td>
</tr>
<tr>
<td>modifiers, for classes</td>
<td>38</td>
</tr>
<tr>
<td>modifiers, for interfaces</td>
<td>38</td>
</tr>
<tr>
<td>modifiers, for type declarations</td>
<td>38</td>
</tr>
<tr>
<td>modifiers, Java</td>
<td>40</td>
</tr>
<tr>
<td>modifiers, summary of</td>
<td>163</td>
</tr>
<tr>
<td>modifiers, used</td>
<td>37, 45, 49, 52, 100</td>
</tr>
<tr>
<td>modifies</td>
<td>30, 83</td>
</tr>
<tr>
<td>modifies clause</td>
<td>83</td>
</tr>
<tr>
<td>modifies clause, omitted</td>
<td>83</td>
</tr>
<tr>
<td>modifies_redundantly</td>
<td>30, 83</td>
</tr>
<tr>
<td>monitored</td>
<td>30, 39, 43, 50</td>
</tr>
<tr>
<td>monitors-for-clause, defined</td>
<td>62, 141</td>
</tr>
<tr>
<td>monitors-for-clause, used</td>
<td>52</td>
</tr>
<tr>
<td>monitors_for</td>
<td>30, 62, 141</td>
</tr>
<tr>
<td>Morgan</td>
<td>8, 122</td>
</tr>
<tr>
<td>Morris</td>
<td>122</td>
</tr>
<tr>
<td>Müller</td>
<td>13, 54, 134</td>
</tr>
<tr>
<td>Müller</td>
<td>9, 30, 35, 42, 133, 134, 138</td>
</tr>
<tr>
<td>mult-expr, defined</td>
<td>91</td>
</tr>
<tr>
<td>mult-expr, used</td>
<td>91</td>
</tr>
<tr>
<td>mult-op, defined</td>
<td>91</td>
</tr>
<tr>
<td>mult-op, used</td>
<td>91</td>
</tr>
<tr>
<td>multiline comment, see C-Style comment</td>
<td>27</td>
</tr>
<tr>
<td>multiple inheritance</td>
<td>38</td>
</tr>
<tr>
<td>multiplication, quantified, see \product</td>
<td>102</td>
</tr>
<tr>
<td>name clash, between Java and JML-only names, resolving</td>
<td>11</td>
</tr>
<tr>
<td>name, defined</td>
<td>36</td>
</tr>
<tr>
<td>name, used</td>
<td>36, 37, 41, 45, 50</td>
</tr>
<tr>
<td>name-list, defined</td>
<td>37</td>
</tr>
<tr>
<td>name-list, used</td>
<td>37</td>
</tr>
<tr>
<td>name-star, defined</td>
<td>36</td>
</tr>
<tr>
<td>name-star, used</td>
<td>36</td>
</tr>
<tr>
<td>namespace, for ghost fields</td>
<td>11</td>
</tr>
<tr>
<td>namespace, for model features</td>
<td>11</td>
</tr>
<tr>
<td>native</td>
<td>30, 39</td>
</tr>
<tr>
<td>Naumann</td>
<td>122</td>
</tr>
<tr>
<td>Naur</td>
<td>25</td>
</tr>
<tr>
<td>negative key</td>
<td>28</td>
</tr>
<tr>
<td>negative-key, defined</td>
<td>27</td>
</tr>
<tr>
<td>negative-key, used</td>
<td>27</td>
</tr>
<tr>
<td>Nelson</td>
<td>1, 14, 28, 43, 50, 168</td>
</tr>
<tr>
<td>new</td>
<td>30, 57, 91, 138</td>
</tr>
<tr>
<td>new-expr, defined</td>
<td>91</td>
</tr>
<tr>
<td>new-expr, used</td>
<td>91</td>
</tr>
<tr>
<td>new-expr, with set comprehension suffix</td>
<td>104</td>
</tr>
<tr>
<td>new-suffix, defined</td>
<td>91</td>
</tr>
<tr>
<td>new-suffix, used</td>
<td>91</td>
</tr>
<tr>
<td>newline</td>
<td>26, 27, 33</td>
</tr>
<tr>
<td>newline, defined</td>
<td>26</td>
</tr>
<tr>
<td>newline, used</td>
<td>26</td>
</tr>
<tr>
<td>Noble</td>
<td>54, 134</td>
</tr>
<tr>
<td>non-at-end-of-line, defined</td>
<td>27</td>
</tr>
<tr>
<td>non-at-end-of-line, used</td>
<td>29</td>
</tr>
<tr>
<td>non-at-plus-minus-end-of-line, defined</td>
<td>27</td>
</tr>
<tr>
<td>non-at-plus-minus-end-of-line, used</td>
<td>27</td>
</tr>
<tr>
<td>non-at-plus-minus-star, defined</td>
<td>27</td>
</tr>
<tr>
<td>non-at-plus-minus-star, used</td>
<td>27</td>
</tr>
<tr>
<td>non-end-of-line, defined</td>
<td>27</td>
</tr>
<tr>
<td>non-end-of-line, used</td>
<td>27, 29</td>
</tr>
<tr>
<td>non-helper methods, semantics of specifications for</td>
<td>68</td>
</tr>
<tr>
<td>non-letter, defined</td>
<td>27</td>
</tr>
<tr>
<td>non-letter, used</td>
<td>27</td>
</tr>
<tr>
<td>non-nl-white-space, defined</td>
<td>26</td>
</tr>
<tr>
<td>non-nl-white-space, used</td>
<td>26, 29</td>
</tr>
<tr>
<td>non-null elements, of an array</td>
<td>99</td>
</tr>
<tr>
<td>non-slash, defined</td>
<td>27</td>
</tr>
<tr>
<td>non-slash, used</td>
<td>27</td>
</tr>
<tr>
<td>non-star, defined</td>
<td>27</td>
</tr>
<tr>
<td>non-star, used</td>
<td>27, 29</td>
</tr>
<tr>
<td>non-star-close, defined</td>
<td>34</td>
</tr>
<tr>
<td>non-star-close, used</td>
<td>34</td>
</tr>
<tr>
<td>non-star-slash, defined</td>
<td>27</td>
</tr>
<tr>
<td>non-star-slash, used</td>
<td>27</td>
</tr>
<tr>
<td>non-stars-close, defined</td>
<td>34</td>
</tr>
<tr>
<td>non-stars-close, used</td>
<td>34</td>
</tr>
<tr>
<td>non-stars-slash, defined</td>
<td>27</td>
</tr>
<tr>
<td>non-stars-slash, used</td>
<td>27</td>
</tr>
<tr>
<td>non-zero-digit, defined</td>
<td>33</td>
</tr>
<tr>
<td>non-zero-digit, used</td>
<td>33</td>
</tr>
<tr>
<td>non_null</td>
<td>4, 16, 30, 39, 44, 46, 49, 66, 109, 169</td>
</tr>
</tbody>
</table>
## Index

| non_null, in method declaration | 45 |
| non_null, modifier in separate file | 130 |
| non_null, parameter modifier | 46 |
| nondeterministic-choice, defined | 124 |
| nondeterministic-choice, used | 124 |
| nondeterministic-if, defined | 124 |
| nondeterministic-if, used | 124 |
| nonnull-elements-expression, defined | 99 |
| nonnull-elements-expression, used | 92 |
| nonterminal symbols, notation | 25 |
| normal postcondition | 77 |
| normal-behavior-keyword, defined | 72 |
| normal-behavior-keyword, used | 72, 125 |
| normal-behavior-spec-case, defined | 72 |
| normal-behavior-spec-case, used | 67 |
| normal-example-body, defined | 120 |
| normal-example-body, used | 120 |
| normal-spec-case, defined | 72 |
| normal-spec-case, used | 72, 120, 125 |
| normal_behavior | 4, 12, 30, 72, 123 |
| normal_behavior | 30, 72 |
| normal_example | 30, 120 |
| nowarn | 26, 30 |
| nowarn-label, defined | 26 |
| nowarn-label, used | 26 |
| nowarn-label-list, defined | 26 |
| nowarn-label-list, used | 26 |
| nowarn-pragma, defined | 26 |
| nowarn-pragma, used | 26 |
| NSF | 9 |
| null | 30, 33, 91, 104 |
| null-literal, defined | 33 |
| null-literal, used | 33 |
| nullable | 16, 30, 39, 44, 104, 109, 160 |
| nullable, explicitly | 44 |
| nullable, implicitly | 44 |
| nullable, modifier in separate file | 130 |
| nullable_by_default | 30, 39, 44, 160 |
| numeric types, arbitrary precision | 140 |
| numerical quantifier, see `\num_of` | 103 |
| octal-numeral, defined | 33 |
| octal-numeral, used | 33 |
| old | 30, 68, 69, 75 |
| old-expression | 93 |
| old-expression, defined | 93 |
| old-expression, used | 92 |
| old-var-declarator, defined | 75 |
| old-var-declarator, used | 75 |
| old-var-decls, defined | 75 |
| old-var-decls, used | 75 |
| omitted specification, meaning of | 66 |
| only-accessed-expression, defined | 95 |
| only-accessed-expression, used | 96 |
| only-called-expression, defined | 96 |
| only-called-expression, used | 97 |
| operation | 8 |
| operator precedence | 90 |
| operator, of LSL | 8 |
| operators, added to JML | 105 |
| optional elements in syntax | 25 |
| overriding method, meaning of omitted specification for | 66 |
| overriding methods, and `pure` | 39 |
| owner | 134 |
| owner-as-modifier property | 134 |
| ownership | 54 |
| ownership context | 134 |
| ownership context, root | 134 |
| ownership modifiers for array types | 136 |
| ownership modifiers for types, defaults | 137 |
| ownership modifiers and type checking | 138 |
| ownership types, and subtyping | 138 |
| ownership types, and subtyping | 138 |
| ownership-modifier, defined | 133 |
| ownership-modifier, used | 109, 133 |
| ownership-modifiers, defined | 133 |
| ownership-modifiers, used | 50 |

## P

<p>| package declaration, satisfaction of | 36 |
| package declarations | 36 |
| package visibility | 12 |
| package-declaration, defined | 36 |
| package-declaration, used | 35 |
| paragraph-tag, defined | 29 |
| paragraph-tag, used | 29 |
| param-declaration, defined | 46 |
| param-declaration, used | 46, 108 |
| param-declaration-list, defined | 46 |
| param-declaration-list, used | 46 |
| param-disambig, defined | 57 |
| param-disambig, used | 57 |
| param-disambig-list, defined | 57 |
| param-disambig-list, used | 57 |
| param-modifier, defined | 46 |
| param-modifier, used | 46 |</p>
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parnas</td>
<td>8</td>
</tr>
<tr>
<td>parsing</td>
<td>7</td>
</tr>
<tr>
<td>partial correctness</td>
<td>81</td>
</tr>
<tr>
<td>peer</td>
<td>30, 133, 134, 135</td>
</tr>
<tr>
<td>plans, for JML</td>
<td>7</td>
</tr>
<tr>
<td>Poetzsche-Heffter</td>
<td>9, 54, 56, 133, 134</td>
</tr>
<tr>
<td>Poll</td>
<td>7</td>
</tr>
<tr>
<td>portability, and language levels</td>
<td>18</td>
</tr>
<tr>
<td>positive key</td>
<td>28</td>
</tr>
<tr>
<td>positive-key, defined</td>
<td>27</td>
</tr>
<tr>
<td>positive-key, used</td>
<td>27</td>
</tr>
<tr>
<td>possibly-annotated-loop, defined</td>
<td>109</td>
</tr>
<tr>
<td>possibly-annotated-loop, used</td>
<td>108</td>
</tr>
<tr>
<td>post</td>
<td>30, 77</td>
</tr>
<tr>
<td>post-state</td>
<td>69</td>
</tr>
<tr>
<td>post_redundantly</td>
<td>30, 77</td>
</tr>
<tr>
<td>postcondition, redundant</td>
<td>1, 5, 8</td>
</tr>
<tr>
<td>postcondition, exceptional</td>
<td>2, 77, 79</td>
</tr>
<tr>
<td>postcondition, normal</td>
<td>2, 77</td>
</tr>
<tr>
<td>postcondition, via non-null</td>
<td>45</td>
</tr>
<tr>
<td>postfix-expr, defined</td>
<td>91</td>
</tr>
<tr>
<td>postfix-expr, used</td>
<td>91, 104</td>
</tr>
<tr>
<td>Potter</td>
<td>54, 134</td>
</tr>
<tr>
<td>pre</td>
<td>30, 76</td>
</tr>
<tr>
<td>pre-state</td>
<td>69</td>
</tr>
<tr>
<td>pre_redundantly</td>
<td>30, 76</td>
</tr>
<tr>
<td>precedence, table of</td>
<td>90</td>
</tr>
<tr>
<td>precondition</td>
<td>1, 2, 5, 8, 76</td>
</tr>
<tr>
<td>precondition, protective</td>
<td>16</td>
</tr>
<tr>
<td>pred-or-not, defined</td>
<td>76</td>
</tr>
<tr>
<td>pred-or-not, used</td>
<td>76, 77, 78, 81, 126</td>
</tr>
<tr>
<td>predicate</td>
<td>90</td>
</tr>
<tr>
<td>predicate, defined</td>
<td>90</td>
</tr>
<tr>
<td>predicate, used</td>
<td>52, 57, 60, 61, 62, 84, 85, 101, 104, 111, 113, 114, 116</td>
</tr>
<tr>
<td>predicates, and exceptions</td>
<td>15</td>
</tr>
<tr>
<td>preserving, an invariant</td>
<td>53</td>
</tr>
<tr>
<td>primary-expr, defined</td>
<td>91</td>
</tr>
<tr>
<td>primary-expr, used</td>
<td>91</td>
</tr>
<tr>
<td>primary-suffix, defined</td>
<td>91</td>
</tr>
<tr>
<td>primary-suffix, used</td>
<td>91</td>
</tr>
<tr>
<td>primitive value type</td>
<td>11</td>
</tr>
<tr>
<td>privacy</td>
<td>67</td>
</tr>
<tr>
<td>privacy modifiers</td>
<td>12</td>
</tr>
<tr>
<td>privacy, defined</td>
<td>64</td>
</tr>
<tr>
<td>privacy, used</td>
<td>67, 72, 73, 120, 124, 125</td>
</tr>
<tr>
<td>PrivacyDemoIllegal</td>
<td>14</td>
</tr>
<tr>
<td>PrivacyDemoLegalAndIllegal</td>
<td>13</td>
</tr>
<tr>
<td>private</td>
<td>7, 12, 30, 39, 64</td>
</tr>
<tr>
<td>private, and helper</td>
<td>42</td>
</tr>
<tr>
<td>private, modifier in separate file</td>
<td>129, 130</td>
</tr>
<tr>
<td>procedure claims</td>
<td>119</td>
</tr>
<tr>
<td>product, see <code>product</code></td>
<td>102</td>
</tr>
<tr>
<td>programming method, and JML</td>
<td>6</td>
</tr>
<tr>
<td>protected</td>
<td>12, 30, 39, 64</td>
</tr>
<tr>
<td>protected, modifier in separate file</td>
<td>129, 130</td>
</tr>
<tr>
<td>protective specifications</td>
<td>16</td>
</tr>
<tr>
<td>public</td>
<td>4, 7, 12, 30, 39, 64</td>
</tr>
<tr>
<td>public specification</td>
<td>4</td>
</tr>
<tr>
<td>public type, in a compilation unit</td>
<td>35</td>
</tr>
<tr>
<td>public, modifier in separate file</td>
<td>129, 130</td>
</tr>
<tr>
<td>pure</td>
<td>5, 30, 38, 39, 41, 46, 66</td>
</tr>
<tr>
<td>pure and model, constructors</td>
<td>46</td>
</tr>
<tr>
<td>pure and model, methods</td>
<td>46</td>
</tr>
<tr>
<td>pure and void methods</td>
<td>48</td>
</tr>
<tr>
<td>pure classes, vs. model classes</td>
<td>48</td>
</tr>
<tr>
<td>pure constructor</td>
<td>47</td>
</tr>
<tr>
<td>pure interface</td>
<td>48</td>
</tr>
<tr>
<td>pure method</td>
<td>46</td>
</tr>
<tr>
<td>pure methods, default ownership modifiers for parameter types of</td>
<td>137</td>
</tr>
<tr>
<td>pure methods, vs. model methods</td>
<td>48</td>
</tr>
<tr>
<td>pure type used for modeling, vs. model type</td>
<td>39</td>
</tr>
<tr>
<td>pure, and overriding methods</td>
<td>39</td>
</tr>
<tr>
<td>pure, implicit verification condition for termination</td>
<td>47</td>
</tr>
<tr>
<td>pure, modifier in separate file</td>
<td>130</td>
</tr>
<tr>
<td>pure, type declaration modifier</td>
<td>39</td>
</tr>
<tr>
<td>pure, vs. immutable objects</td>
<td>39</td>
</tr>
<tr>
<td>purity, and determinism</td>
<td>48</td>
</tr>
<tr>
<td>purpose, of this reference manual</td>
<td>1</td>
</tr>
<tr>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>quantified addition, see <code>\sum</code></td>
<td>102</td>
</tr>
<tr>
<td>quantified maximum, see <code>\max</code></td>
<td>102</td>
</tr>
<tr>
<td>quantified minimum, see <code>\min</code></td>
<td>102</td>
</tr>
<tr>
<td>quantified multiplication, see <code>\product</code></td>
<td>102</td>
</tr>
<tr>
<td>quantified-var-declarator, defined</td>
<td>101</td>
</tr>
<tr>
<td>quantified-var-declarator, used</td>
<td>75, 101, 104</td>
</tr>
<tr>
<td>quantified-var-decls, defined</td>
<td>101</td>
</tr>
<tr>
<td>quantified-var-decls, used</td>
<td>101</td>
</tr>
<tr>
<td>quantifier</td>
<td>5</td>
</tr>
<tr>
<td>quantifier body</td>
<td>5</td>
</tr>
<tr>
<td>quantifier, body</td>
<td>102</td>
</tr>
<tr>
<td>quantifier, body of</td>
<td>101</td>
</tr>
<tr>
<td>quantifier, defined</td>
<td>101</td>
</tr>
<tr>
<td>quantifier, executability of</td>
<td>103</td>
</tr>
<tr>
<td>quantifier, generalized</td>
<td>102</td>
</tr>
<tr>
<td>quantifier, range predicate in</td>
<td>101</td>
</tr>
<tr>
<td>quantifier, used</td>
<td>101</td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Raghavan</td>
<td>89</td>
</tr>
<tr>
<td>range predicate</td>
<td>5</td>
</tr>
<tr>
<td>range predicate, and executability of quantifiers</td>
<td>103</td>
</tr>
<tr>
<td>range predicate, in quantifier</td>
<td>101</td>
</tr>
<tr>
<td>range predicate, not satisfiable</td>
<td>102</td>
</tr>
<tr>
<td>Ravelo, Jesus</td>
<td>9</td>
</tr>
<tr>
<td>reach-expression, defined</td>
<td>97</td>
</tr>
<tr>
<td>reach-expression, used</td>
<td>92</td>
</tr>
<tr>
<td>reachable objects</td>
<td>97</td>
</tr>
<tr>
<td>readable</td>
<td>30, 61</td>
</tr>
<tr>
<td>readable-if-clause, defined</td>
<td>61</td>
</tr>
</tbody>
</table>
signals-only-keyword, defined .......................... 79
signals-only-keyword, used ................................ 79
signals_only ............................................. 30, 68, 70, 74, 79
signals_only, default for ................................ 79
signals_only, in comparing specifications ............... 119
signals_only_redundantly ................................ 30, 79
signals_redundantly ...................................... 30, 78
SignalsClause ............................................. 74
signed-integer, defined .................................. 33
signed-integer, used ....................................... 33
simple-spec-body, defined ................................ 65
simple-spec-body, used ..................................... 65, 120
simple-spec-body-clause, defined ......................... 65
simple-spec-body-clause, used ............................ 65
simple-spec-statement-body, defined ..................... 125
simple-spec-statement-body, used ......................... 125
simple-spec-statement-clause, defined ................... 125
simple-spec-statement-clause, used ....................... 125
single line comment, see C++Style comment ............ 27
single quote .............................................. 33
single-character, defined ................................ 33
single-character, used .................................... 33
space ....................................................... 26
space, specification of .................................... 98
space, taken up by an object ................................ 98
space-expression, defined ................................ 98
space-expression, used .................................... 92
spaces, defined .......................................... 26
spaces, used ............................................. 26
spec-array-initializer, defined ............................ 101
spec-array-initializer, used ................................ 101
spec-array-ref-expr, defined ............................. 106
spec-array-ref-expr, used ................................ 88, 106
spec-case, defined ........................................ 64
spec-case, used .......................................... 63
spec-case-seq, defined ................................... 63
spec-case-seq, used ....................................... 63, 118
spec-expression, defined ................................ 90
spec-expression, used ..................................... 60, 84, 85, 90, 93, 97, 98, 99, 100, 101, 106, 112, 141
spec-expression-list, defined ............................. 90
spec-expression-list, used ................................ 62, 97, 141
spec-header, defined ...................................... 65
spec-header, used ......................................... 65, 120, 125
spec-initializer, defined ................................ 101
spec-initializer, used ..................................... 101
spec-quantified-expr, defined ............................. 101
spec-quantified-expr, used ............................... 101
spec-statement, defined .................................. 125
spec-statement, used ..................................... 114, 124
spec-var-decls, defined ................................ 75
spec-var-decls, used ..................................... 65, 120, 125
spec-variable-declarator, defined ....................... 101
spec-variable-declarator, used ......................... 101
spec-variable-declarators, defined ...................... 101
spec-variable-declarators, used ......................... 75
spec_bigint_math ......................................... 30, 38, 39, 43
spec_java_math ............................................ 30, 38, 39, 43
spec_protected .......................................... 2, 14, 30, 39, 41
spec_protected, as a model field shorthand .............. 14
spec_protected, modifier in separate file ............... 130
spec_public .............................................. 2, 14, 30, 39, 41
spec_public, as a model field shorthand ................. 14
spec_public, modifier in separate file .................. 130
spec_safe_math ........................................... 30, 38, 39, 43
special symbols .......................................... 32
special-symbol, defined .................................. 32
special-symbol, used ..................................... 26
specification for subtypes ................................ 127
specification statement ................................... 123
specification, completely omitted ...................... 66
specification, completeness of ............................ 4
specification, defined ................................... 63
callspecification, heavyweight .......................... 12
specification, in refining statement ..................... 115
specification, lightweight ................................ 12
specification, of interface behavior ..................... 1
specification, used ....................................... 63
specification-only type .................................. 39
specifications for non-helper methods, semantics of ... 68
specifications-only-keyword, used ....................... 79
specifying examples ...................................... 120
Spivey ..................................................... 2, 8
stars-non-close, defined ................................ 34
stars-non-close, used .................................... 34
stars-non-slash, defined ................................ 27
stars-non-slash, used ..................................... 27
start rule, in JML grammar ................................ 35
Stata ....................................................... 9, 168
state, post-state of a call ................................ 69
state, pre-state of a call ................................ 69
state, visible ............................................ 53
statement, defined ...................................... 108
statement, refining ....................................... 114
statement, used ......................................... 108, 109, 114, 124
static ..................................................... 15, 30, 39, 50, 56, 59
static constraint ......................................... 59
static features ........................................... 15
static invariant ......................................... 53, 56
static, modifier in separate file ......................... 129, 130
static_initializer ....................................... 30
static_initializer, separate files ....................... 131
static_initializer, used ................................ 50
status, of JML ............................................ 7
Steele ..................................................... 32, 33
Steyaert ............................................... 127
store-ref, defined ....................................... 106
store-ref, used .......................................... 83, 106
store-ref-expression, defined ............................ 106
store-ref-expression, used ............................... 60, 106, 141
store-ref-keyword, defined .............................. 106
store-ref-keyword, used ................................ 84, 106
store-ref-list, defined .................................. 106
store-ref-list, used ..................................... 83, 84, 94, 95, 96, 97
store-ref-name, defined ................................ 106
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>store-ref-name, used</td>
<td>106</td>
</tr>
<tr>
<td>store-ref-name-suffix, defined</td>
<td>106</td>
</tr>
<tr>
<td>store-ref-name-suffix, used</td>
<td>106</td>
</tr>
<tr>
<td>strictfp</td>
<td>30, 39</td>
</tr>
<tr>
<td>string-literal, defined</td>
<td>33</td>
</tr>
<tr>
<td>string-literal, used</td>
<td>33, 142</td>
</tr>
<tr>
<td>strong validity</td>
<td>15</td>
</tr>
<tr>
<td>subclass</td>
<td>37</td>
</tr>
<tr>
<td>subclassing_contract, replaced by</td>
<td>142</td>
</tr>
<tr>
<td>code_contract</td>
<td>37</td>
</tr>
<tr>
<td>subtype</td>
<td>105</td>
</tr>
<tr>
<td>subtype, for an interface</td>
<td>38</td>
</tr>
<tr>
<td>subtype, of an interface</td>
<td>38</td>
</tr>
<tr>
<td>subtypes, specification for</td>
<td>127</td>
</tr>
<tr>
<td>subtyping</td>
<td>37</td>
</tr>
<tr>
<td>subtyping, for arrays, with ownership</td>
<td>138</td>
</tr>
<tr>
<td>subtyping, for ownership types</td>
<td>138</td>
</tr>
<tr>
<td>suffixes, of filenames</td>
<td>129</td>
</tr>
<tr>
<td>SumArrayLoop</td>
<td>110</td>
</tr>
<tr>
<td>summation, see \sum</td>
<td>102</td>
</tr>
<tr>
<td>super</td>
<td>30, 57, 87, 91, 106</td>
</tr>
<tr>
<td>superclass</td>
<td>37</td>
</tr>
<tr>
<td>supertypes, specification of</td>
<td>127</td>
</tr>
<tr>
<td>switch</td>
<td>30, 108</td>
</tr>
<tr>
<td>switch-body, defined</td>
<td>108</td>
</tr>
<tr>
<td>switch-body, used</td>
<td>108</td>
</tr>
<tr>
<td>switch-label, defined</td>
<td>108</td>
</tr>
<tr>
<td>switch-label, used</td>
<td>108</td>
</tr>
<tr>
<td>switch-label-seq, defined</td>
<td>108</td>
</tr>
<tr>
<td>switch-label-seq, used</td>
<td>108</td>
</tr>
<tr>
<td>switch-statement, defined</td>
<td>108</td>
</tr>
<tr>
<td>switch-statement, used</td>
<td>108</td>
</tr>
<tr>
<td>synchronized</td>
<td>30, 39, 108</td>
</tr>
<tr>
<td>syntax notations</td>
<td>25</td>
</tr>
<tr>
<td>syntax options</td>
<td>30</td>
</tr>
<tr>
<td>syntax, deprecated</td>
<td>141</td>
</tr>
<tr>
<td>syntax, replaced</td>
<td>141</td>
</tr>
<tr>
<td>time, specification of</td>
<td>98</td>
</tr>
<tr>
<td>time, virtual machine cycle</td>
<td>98</td>
</tr>
<tr>
<td>token, defined</td>
<td>26</td>
</tr>
<tr>
<td>token, used</td>
<td>26</td>
</tr>
<tr>
<td>tokens</td>
<td>29</td>
</tr>
<tr>
<td>tool support</td>
<td>7</td>
</tr>
<tr>
<td>tools and annotations</td>
<td>28</td>
</tr>
<tr>
<td>tools, advice for builders</td>
<td>18</td>
</tr>
<tr>
<td>top-level-declaration, defined</td>
<td>35</td>
</tr>
<tr>
<td>top-level-declaration, used</td>
<td>35</td>
</tr>
<tr>
<td>total correctness</td>
<td>81</td>
</tr>
<tr>
<td>trait</td>
<td>8</td>
</tr>
<tr>
<td>trait function</td>
<td>8</td>
</tr>
<tr>
<td>transient</td>
<td>30, 39</td>
</tr>
<tr>
<td>true</td>
<td>30, 33, 91</td>
</tr>
<tr>
<td>try</td>
<td>30, 108</td>
</tr>
<tr>
<td>try-block, defined</td>
<td>108</td>
</tr>
<tr>
<td>try-block, used</td>
<td>108</td>
</tr>
<tr>
<td>two-valued logic</td>
<td>15</td>
</tr>
<tr>
<td>type</td>
<td>11</td>
</tr>
<tr>
<td>type checking</td>
<td>7</td>
</tr>
<tr>
<td>type checking, with ownership types</td>
<td>138</td>
</tr>
<tr>
<td>type declarations</td>
<td>37</td>
</tr>
<tr>
<td>type specs, for declarations</td>
<td>50</td>
</tr>
<tr>
<td>type system, Universe</td>
<td>133</td>
</tr>
<tr>
<td>type, abstract</td>
<td>8</td>
</tr>
<tr>
<td>type, defined</td>
<td>50</td>
</tr>
<tr>
<td>type, modifiers for declarations of</td>
<td>38</td>
</tr>
<tr>
<td>type, specifying in a declaration</td>
<td>50</td>
</tr>
<tr>
<td>type, used</td>
<td>50, 91, 100</td>
</tr>
<tr>
<td>type-declaration, defined</td>
<td>37</td>
</tr>
<tr>
<td>type-declaration, used</td>
<td>35</td>
</tr>
<tr>
<td>type-expression, defined</td>
<td>100</td>
</tr>
<tr>
<td>type-expression, used</td>
<td>92</td>
</tr>
<tr>
<td>type-spec, defined</td>
<td>50</td>
</tr>
<tr>
<td>type-spec, used</td>
<td>45, 46, 49, 57, 75, 91, 101, 104, 109</td>
</tr>
<tr>
<td>typeof expression</td>
<td>99</td>
</tr>
<tr>
<td>typeof-expression, defined</td>
<td>99</td>
</tr>
<tr>
<td>typeof-expression, used</td>
<td>92</td>
</tr>
<tr>
<td>types, comparing</td>
<td>105</td>
</tr>
<tr>
<td>types, marking in expressions</td>
<td>100</td>
</tr>
<tr>
<td>Universe</td>
<td>35</td>
</tr>
<tr>
<td>Universe keywords, where recognized</td>
<td>30</td>
</tr>
<tr>
<td>universe type system</td>
<td>133</td>
</tr>
<tr>
<td>Universe type system</td>
<td>133</td>
</tr>
</tbody>
</table>

**T**

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>tab</td>
<td>26, 33</td>
</tr>
<tr>
<td>table of precedence</td>
<td>90</td>
</tr>
<tr>
<td>tagged-paragraph, defined</td>
<td>29</td>
</tr>
<tr>
<td>tagged-paragraph, used</td>
<td>29</td>
</tr>
<tr>
<td>Tan</td>
<td>119</td>
</tr>
<tr>
<td>target-label, used</td>
<td>126</td>
</tr>
<tr>
<td>terminal symbols, notation</td>
<td>25</td>
</tr>
<tr>
<td>termination, of pure methods</td>
<td>47</td>
</tr>
<tr>
<td>terminology, for invariants</td>
<td>56</td>
</tr>
<tr>
<td>this</td>
<td>30, 56, 57, 59, 87, 91, 106, 135</td>
</tr>
<tr>
<td>this, and rep</td>
<td>134</td>
</tr>
<tr>
<td>thread, specifying locks held by</td>
<td>100</td>
</tr>
<tr>
<td>threads, specification of</td>
<td>6</td>
</tr>
<tr>
<td>throw</td>
<td>30, 108</td>
</tr>
<tr>
<td>throws</td>
<td>30, 45, 80</td>
</tr>
<tr>
<td>throws-clause, defined</td>
<td>45</td>
</tr>
<tr>
<td>throws-clause, used</td>
<td>45</td>
</tr>
</tbody>
</table>

**U**

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary-expr, defined</td>
<td>91</td>
</tr>
<tr>
<td>unary-expr, used</td>
<td>91</td>
</tr>
<tr>
<td>unary-expr-not-plus-minus, defined</td>
<td>91</td>
</tr>
<tr>
<td>unary-expr-not-plus-minus, used</td>
<td>91</td>
</tr>
<tr>
<td>undefinedness, in expression evaluation</td>
<td>15</td>
</tr>
<tr>
<td>underspecified total functions</td>
<td>15</td>
</tr>
<tr>
<td>unicode-escape, defined</td>
<td>33</td>
</tr>
<tr>
<td>unicode-escape, used</td>
<td>33</td>
</tr>
<tr>
<td>uninitialized</td>
<td>30, 39, 43</td>
</tr>
<tr>
<td>Universe</td>
<td>35</td>
</tr>
<tr>
<td>Universe keywords, where recognized</td>
<td>30</td>
</tr>
<tr>
<td>universe type system</td>
<td>133</td>
</tr>
<tr>
<td>Universe type system</td>
<td>133</td>
</tr>
</tbody>
</table>
Index

<table>
<thead>
<tr>
<th>Vocabulary</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universe type system syntax</td>
<td>30</td>
</tr>
<tr>
<td>Universe type system, basic concepts</td>
<td>134</td>
</tr>
<tr>
<td>Universe type system, options for</td>
<td>133</td>
</tr>
<tr>
<td>unreachable</td>
<td>30, 115</td>
</tr>
<tr>
<td>unreachable-statement, defined</td>
<td>115</td>
</tr>
<tr>
<td>unreachable-statement, used</td>
<td>113</td>
</tr>
<tr>
<td>usefulness, of JML</td>
<td>6</td>
</tr>
<tr>
<td>uses, of JML</td>
<td>6</td>
</tr>
<tr>
<td>utility, of JML</td>
<td>6</td>
</tr>
</tbody>
</table>

V
validity, of assertions | 15 |
validity, strong | 15 |
variable-declarator, defined | 49 |
variable-declarator, used | 49 |
variable-declarators, defined | 49 |
variable-declarators, used | 49 |
variable-decls, defined | 49 |
variable-decls, used | 49, 108 |
variable-definition, defined | 49 |
variable-definition, used | 45 |
variant-function, defined | 112 |
variant-function, used | 109 |
VDM | 8 |
VDM-SL | 8 |
vertical tab | 26 |
Vickers | 8 |
virtual machine cycle time | 98 |
visibility | 7, 12 |
visibility, in JML | 12 |
visibility, in lightweight specifications | 13 |
visibility, in method specifications | 64 |
visible state | 53 |
visible state, for a type | 53 |
Vitek | 54, 134 |
vocabulary | 1 |
void | 30, 91 |
void and pure methods | 48 |
volatile | 30, 39 |
von Wright | 8, 122 |

W
Watt | 26 |
web site, for JML | 1 |
Weck | 122 |
when | 30, 68, 70, 82 |
when clause, omitted | 82 |
when-clause, defined | 82 |
when-clause, used | 65, 125 |
when-keyword, defined | 82 |
when-keyword, used | 82 |
when_redundantly | 30, 82 |
while | 30, 109 |
white space | 26 |
white-space | 35 |
white-space, defined | 26 |
white-space, used | 26 |
Wills | 2 |
Wing | 1, 8, 16 |
working space, specification of | 98 |
working-space-clause, defined | 85 |
working-space-clause, used | 65, 125 |
working-space-expression, defined | 98 |
working-space-expression, used | 92 |
working-space-keyword, defined | 85 |
working-space-keyword, used | 85 |
working_space | 30, 68, 71, 85 |
working_space_redundantly | 30, 85 |
writable | 30, 62 |
writable-if-clause, defined | 62 |
writable-if-clause, used | 52 |

Z
z | 2, 8 |
zero-to-three, defined | 33 |
zero-to-three, used | 33 |