JML Reference Manual

DRAFT, $Revision: 1.108 $
$Date: 2004/12/29 21:16:19 $

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Joseph Kiniry

Version Information:
@(#) $Id: jmlrefman.texinfo,v 1.108 2004/12/29 21:16:19 leavens Exp $

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-Cheon04] and "JML: A notation for detailed design" [Leavens-Baker-Ruby99], and continuing with the paper "Preliminary Design of JML" [Leavens-Baker-Ruby04]. These are all available from the JML web site '`http://www.jmlspecs.org/`', where further readings and examples may also be found.

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

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

1.1 Behavioral Interface Specifications

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

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

The *behavior* of a method or type describes the state transformations it can perform. The behavior of a method is specified by describing: the set of states in which calling the method is defined, the set of locations the method is allowed to assign to (and hence change), and the relations between the state when the method was called and when it either returns normally or throws an exception, as well as the states 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-etal03], 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 an collection of abstract instance variables, in much the same way as other specification languages, such as Z [Hayes93] [Spivey92] or Fresco [Wills92b].

1.2 A First Example

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

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

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

The behavior of this class is specified in the JML annotations found in the special comments that have an at-sign (`@`) as their first character following the usual comment beginning. Such lines look like comments to Java, but are interpreted by JML and its tools. For example, line 5 starts with an annotation comment marker of the form `//@`, and this annotation continues until the `//` towards the end of the line, which starts a comment within the annotation which even JML ignores. The other form of such annotations can be seen on lines 7 through 14 line 17, and on lines 15 and 18. These annotations start with the characters `/*@` and end with either `@*/` or `*/`; within such annotations, at-signs (`@`) at the beginnings of lines are ignored by JML. Note that there can be no space between the start of comment marker, either `//` or `/*` and the first at-sign; thus `// @` starts a comment, not an annotation. (See Chapter 4 [Lexical Conventions], page 17, for more details about annotations.)

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

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

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

Such declarations of fields in annotations should be marked as specification-only fields, using the JML modifier `model`.[1] A model field should be thought of as an abstraction of a set of concrete fields used in the implementation of this type and its subtypes. (See Section 8.4 [Represents Clauses], page 47, 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-etal03].) That is, we imagine that objects that are instances of the type `IntHeap` have such a field, whose value is determined by the concrete fields that are known to Java in the actual object. Of course at runtime, objects of type `IntHeap` have no such field, the model fields are purely imaginary. Model fields are thus a convenient fiction that is useful for describing the behavior of an ADT. One does not have to worry about their cost (in space or time), and should only be concerned with how they clarify the behavior of an ADT.

The other annotation used on line 5 is `non_null`. This just says that in any publicly-visible state, the value of `elements` must not be `null`. It is thus a simple kind of invariant (see Section 8.2 [Invariants], page 39).

In the above specification of `IntHeap`, the specification of each method precedes its interface declaration. This follows the usual convention of Java tools, such as JavaDoc, which put such descriptive information in front of the method. In JML, it is also possible to put the specification just before the semicolon (`;`) following the method's interface information (see Chapter 9 [Method Specifications], page 49), but we will usually not to do that in this document.

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

The keyword `normal_behavior` tells JML several things. First, it says that the specification is a heavyweight method specification, as opposed to a lightweight method specification like that given on line 17. A *heavyweight* specification uses one of JML's behavior keywords, like `normal_behavior`, which tells JML that the method specification is intended to be complete. By contrast, a *lightweight* specification does not use one of JML's behavior keywords, and tells JML that the specification is incomplete in the sense of the only contains only some of what the specifier had in mind.[2] Second, the keyword `normal_behavior` tells JML that when the precondition of this method is met, then the method must return normally, without throwing an exception. In other words, it says that the exceptional postcondition is `false`, which prohibits the method from throwing an exception when the precondition holds. (Third, it says that the divergence condition defaults to `false`. See Chapter 9 [Method Specifications], page 49, for more details.)

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

[1] This is the usual way to declare a specification-only field; it is also possible to use the `ghost` modifier (see Section 2.2 [Model and Ghost], page 10).

[2] Lightweight specifications come from ESC/Java.

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

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

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

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

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

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

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

```
    //@  \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 49, 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 9.9.8 [Assignable Clauses], page 64). Thus line 9 cannot be omitted without changing the meaning of the specification.

1.3 What is JML Good For?

JML is a formal specification language tailored to Java. Its basic use is thus the formal specification of the behavior of Java program modules. As it is a behavioral interface specification language, JML specifies how to use such Java program modules from *within* a Java program; hence JML is *not* designed for specifying the behavior of an entire program. So the question "what is JML good for?" really boils down to the following question: what good is formal specification for Java program modules?

The two main benefits in using JML are:

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

Although we would like tools that would help with reasoning about concurrent aspects of Java programs, the current version of JML focuses on the sequential behavior of Java code. While there is work in progress on extending JML to support concurrency, the current version of JML does not have features that help specify how Java threads interact with each other. JML does not, for example, allow the specification of elaborate temporal properties, such as coordinated access to shared variables or the absence of deadlock. Indeed, we assume, in the rest of this manual, that there is only one thread of execution in a Java program annotated with JML, and 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] [Poll-Jacobs00].
- 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].

Recording the private interface of a class may be helpful in program development or maintenance. Usually one would expect that the public interface of a class would be specified, and then separate, more refined specifications would be given for use by derived classes and for detailed implementation (and friend classes). (See Chapter 17 [Refinement], page 93, for how to record each level in JML.)

The reader may also wish to consult the "Preliminary Design of JML" [Leavens-Baker-Ruby04] for a discussion of the goals that are behind JML's design. Apart from the improved precision in the specifications and documentation of code, the main advantage of using a formal specification language, as opposed to informal natural language, is the possibility of tool support. One specific goal that has emerged over time is that JML should be able to unify several different tool-building efforts in the area of formal methods.

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

Of course, there are more exciting forms of tool support than just parsing and typechecking. In particular JML is designed to support static analysis (as in ESC/Java [Leino-etal00]), formal verification (as in the LOOP tool [Huisman01] [Jacobs-etal98]), recording of dynamically obtained invariants (as in Daikon [Ernst-etal01]), runtime assertion checking (as in JML's runtime assertion checker, jmlc [Cheon-Leavens02b] [Cheon03]), unit testing [Cheon-Leavens02], and documentation (as in JML's jmldoc tool). The paper by Burdy et al. [Burdy-etal03] is a recent survey of tools for JML. The utility of these tools is the ultimate answer to the question of what JML is good for.

1.4 Status and Plans for JML

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

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

1.5 Historical Precedents

JML combines ideas from Eiffel [Meyer92a] [Meyer92b] [Meyer97] with ideas from model-based specification languages such as VDM [Jones90] and the Larch family [Guttag-Horning93] [LeavensLarchFAQ] [Wing87] [Wing90a]. It also adds some ideas from the refinement calculus [Back88] [Back-vonWright89a] [Back-vonWright98] [Morgan-Vickers94] [Morgan94] (see Chapter 17 [Refinement], page 93). In this section we describe the advantages and disadvantages of these approaches. Readers unfamiliar with these historical precedents may want to skip this section.

Formal, model-based languages such as those typified by the Larch family build on ideas found originally in Hoare's work. Hoare used pre- and postconditions to describe the semantics of computer programs in his famous article [Hoare69]. Later Hoare adapted these axiomatic techniques to the specification and correctness proofs of abstract data types [Hoare72a]. To specify an ADT, Hoare described a mathematical set of abstract values for the type, and then specified pre- and postconditions for each of the operations of the type in terms of how the abstract values of objects were affected. For example, one might specify a class `IntHeap` using abstract values of the form `empty` and `add(i,h)`, where `i` is an `int` and `h` is an `IntHeap`. These notations form a mathematical vocabulary used in the rest of the specification.

There are two advantages to writing specifications with abstract values instead of directly using Java variables and data structures. The first is that by using abstract values, the specification does not have to be changed when the particular data structure used in the program is changed. This permits different implementations of the same specification to use different data structures. Therefore the specification forms a contract between the rest of the program in the implementation, which ensures that the rest of the program is also independent of the particular data structures used [Liskov-Guttag86] [Meyer97] [Meyer92a] [Parnas72]. Second, it allows the specification to be written even when there are no implementation data structures, as is the case for `IntHeap`.

This idea of model-oriented specification has been followed in VDM [Jones90], VDM-SL [Fitzgerald-Larsen98] [ISO96], Z [Hayes93] [Spivey92], and the Larch family [Guttag-Horning93]. In the Larch approach, the essential elaboration of Hoare's original idea is that the abstract values also come with a set of operations. The operations on abstract values are used to precisely describe the set of abstract values and to make it possible to abbreviate interface specifications (pre- and postconditions for methods). In Z one builds abstract values using tuples, sets, relations, functions, sequences, and bags; these all come with pre-defined operations that can be used in assertions. In VDM one has a similar collection of mathematical tools to describe abstract values, and another set of pre-defined operations. In the Larch approach, there are some pre-defined kinds of abstract values (found in Guttag and Horning's LSL Handbook, Appendix A of [Guttag-Horning93]), but these are expected to be extended as needed. (The advantage of being able to extend the mathematical vocabulary is similar to one advantage of object-oriented programming: one can use a vocabulary that is close to the way one thinks about a problem.)

However, there is a problem with using mathematical notations for describing abstract values and their operations. The problem is that such mathematical notations are an extra burden on a programmer who is learning to use a specification language. The solution to this problem is the essential insight that JML takes from the Eiffel language [Meyer92a] [Meyer92b] [Meyer97]. Eiffel is a programming language with built-in specification constructs. It features pre- and postconditions, although it has no direct support for frame axioms. Programmers like Eiffel because they can easily read the assertions, which are written in Eiffel's own expression syntax. However, Eiffel does not provide support for specification-only variables, and it does not provide much explicit support for describing abstract values. Because of this, it is difficult to write specifications that are as mathematically complete in Eiffel as one can write in a language like VDM or Larch/C`++`.

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

1.6 Acknowledgments

The work of Leavens and Ruby was supported in part by a grant from Rockwell International Corporation and by NSF grant CCR-9503168. Work on JML by Leavens, and Ruby was also supported in part by NSF grant CCR-9803843. Work on JML by Cheon, Clifton, Leavens, Ruby, and others has been supported in part by NSF grants CCR-0097907, CCR-0113181, CCF-0428078, and CCF-0429567. 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, Peter Müller, Arnd Poetzsch-Heffter, and Joachim van den Berg, for many discussions about the semantics of JML specifications. Thanks for Raymie Stata for spear-heading 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 David Cok and Todd Millstein for their work on the prototype JML checker used to check the specifications in this document. Thanks to David and Joe Kiniry for various corrections to this document.

See the "Preliminary Design of JML" [Leavens-Baker-Ruby04] 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 Interfacees

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

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

2.2 Model and Ghost

In JML one can declare various names with the modifier `model`; for example one can declare model fields, methods, and even types. One can also declare some fields as `ghost` fields. JML also has a `model import` directive (see Chapter 5 [Compilation Units], page 25).

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

The most common and useful model declarations are model fields. A model field should be thought of as the abstraction of one or more non-model (i.e., Java or *concrete*) fields [Cheon-etal03]. (By contrast, some authors refer to what JML calls model fields as “abstract fields” [Leino98].) The value of a model field is determined by a the concrete fields it abstracts from; in JML this relationship is specified by a `represents` clause (see Section 8.4 [Represents Clauses], page 47). (Thus the values of the model fields in a 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 67).

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

A `ghost` field is similar to a model field, in that it is also only present for purposes of specification and thus cannot be used outside of annotations. However, unlike a model field, a ghost field does not have a value determined by a represents clause; instead its value is directly determined by its initialization or by a *set-statement* (see Chapter 13 [Statements and Annotation Statements], page 81).

Although these model and ghost names are used only for specifications, JML uses the same namespace for such names as for normal Java names. Thus, one cannot declare a field to be both a model (or ghost) field and a normal Java field in the same class (or in a refinement, see Chapter 17 [Refinement], page 93). Similarly, a method is either a model method or not. In part, this is done because JML has no syntactic distinction between Java and JML field access or method calls. This decision makes it an error for someone to use the same name as a model or ghost feature in an implementation. In such a case if the Java code is considered to be the goal, one can either change the name of the JML feature, or declare the JML feature to be `spec_public` instead of a (hidden) model or ghost feature. See Section 2.4 [Privacy Modifiers and Visibility], page 11, for more about `spec_public`.

2.3 Lightweight and Heavyweight Specifications

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

Users distinguish these between such cases of method specifications by using different syntaxes. See Section 9.2 [Organization of Method Specifications], page 49, for details, but in essence in a method specification case that uses one of the behavior keywords (such as `normal_behavior`, `exceptional_behavior`, or `behavior`) is heavyweight, while a one that does not use such a keyword is lightweight.

2.4 Privacy Modifiers and Visibility

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

Within annotations, JML imposes some extra rules in addition to the usual Java visibility rules [Leavens-Baker-Ruby04]. These rules depend not just on the declaration of the name but also on the visibility level of the context that is referring to the name in question. For purposes of this section, the *annotation context* of a reference to a name is the smallest grammatical unit with an attached (or implicit) visibility. For example, this annotation context can be a method specification case, an invariant, a history constraint, or a field declaration. The visibility level of such an annotation context can be `public`, `protected`, `private`, or default (package) visibility.

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

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

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

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

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

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

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

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

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

```
  }
```

Note that in a lightweight method specification, the privacy level is assumed to be the same privacy level as the method itself. That is, for example, a protected method with a lightweight method specification is considered to be a protected annotation context for purposes of checking proper visibility usage [Leavens-Baker-Ruby04] [Mueller02]. See Section 2.3 [Lightweight and Heavyweight Specifications], page 11, for more about the differences between lightweight and heavyweight specification cases.

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

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

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

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

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

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

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

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

2.5 Instance vs. Static

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

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

In JML declarations follow the normal Java rules for determining whether they are instance or static features of a type. However, within annotations it is possible to explicitly label features as `instance` (see Chapter 6 [Type Definitions], page 27 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 43, for more on this topic.

2.6 Locations and Aliasing

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

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

In a given program state, s, an 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 that do not encounter exceptions (i.e., those that terminate normally) have the values that are defined in the Java Langauge Specification [Gosling-etal00].

The only difference from the Java meaning for an expression executed within an annotation in JML can occur if the execution of the expression would throw an exception in Java. Following the Preliminary Design of JML [Leavens-Baker-Ruby04]: “JML deals with exceptions by having the evaluation of predicates substitute an arbitrary expressible value of the normal result type when an exception is thrown during evaluation. When the expression’s result type is a reference type, an implementation would have to return `null` if an exception is thrown while executing such a predicate. This corresponds to a mathematical model in which partial functions are mathematically modeled by underspecified total functions [Gries-Schneider95]. However, tools sometimes only approximate this semantics. In tools, instead of fully catching exceptions for all subexpressions, many tools only catch exceptions for the smallest boolean-valued subexpression that may throw an exception (and for entire expressions used in JML’s *measured-clause* and *variant-function*).”

The reason for following this semantics is that JML seeks to validate all the rules of ordinary (two-valued) logic [Jones95e] [Gries-Schneider95]. For example, in JML, the expression `x/y >= 3 && y != 0` means the same thing as `y != 0 && x/y >= 3`. If the value of `y` is 0, then both of these expressions evaluate to false, although the value of the subexpression `x/y` is determined arbitrarily.

Since tools for JML are allowed to subsitiute any value of the appropriate type when a subexpression would throw an exception (e.g., this is done in the runtime assertion checker

[1] By an identifier, we technically mean an *ident* in the Java grammar. See Section 4.6 [Tokens], page 20, for details.

[Cheon-Leavens02b] [Cheon03]) users should try to protect subexpressions, so that they will not throw exceptions. In the expression `y != 0 && x/y >= 3`, the subexpression `y != 0` is said to *protect* the subexpression `x/y >= 3` [Leavens-Wing97a]. Protection can also apply across clauses in a JML specification; for example, a precondition such as `y != 0` could be used to protect the expression `x/y >= 3` found in a postcondition. To summarize, it is sensible to write specifications in JML as if one were always trying to prevent exceptions from being thrown in assertions. This idea should be familiar to Java programmers, who constantly have to think about such issues when writing Java code. Hence, as a matter of style, `y != 0 && x/y >= 3` is preferred to `x/y >= 3 && y != 0`.

3 Syntax Notation

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

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

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

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

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

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

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

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

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

A complete summary of the JML grammar appears in an appendix (see Appendix A [Grammar Summary], page 101). 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 [Concept Index], page 140). Every nonterminal and terminal symbol in the grammar is found in the index, and each definition and use is noted. [[[Or rather, it will be...]]]

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 17) may intervene between the characters of a token. (However, outside this chapter, the opposite of this convention is in force.)

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

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

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

4.1 White Space

Blanks, horizontal and vertical tabs, carriage returns, formfeeds, and newlines, collectively called *white space*, are ignored except as they serve to separate tokens. Newlines and carriage returns are special in that they cannot appear in some contexts where other whitespace can appear, and are also used to end Java-style comments (see Section 4.3 [Comments], page 18).

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

4.2 Lexical Pragmas

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

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

See Section 4.6 [Tokens], page 20 for the syntax of *letter*.

4.3 Comments

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

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

4.4 Annotation Markers

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

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

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

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

Annotations look like comments to Java, and are thus ignored by it, but they are significant to JML. One way that this can be achieved is by having JML drop (ie., ignore) the character sequences that are *annotation-markers*: `//@`, `//+@`, `/*@`, `/*+@`, and `@+*/`, `@*/`. The at-sign (`@`) in `@*/` is optional, and more than one at-sign may appear in it and the other annotation markers. However, JML will recognize *jml-keywords* only within annotations.

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

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

4.5 Documentation Comments

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

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

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

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

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

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

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

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

4.6 Tokens

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

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

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

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

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

Similarly, when an option to parse the syntax for the Universe type system [Mueller-Poetzsch-Heffter01a] 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-keyword*s eventually.)

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

The details are given below.

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

```
        | \typeof | \type | \warn_op
        | \warn | \working_space
        | jml-universe-pkeyword
jml-universe-pkeyword ::= \peer | \readonly | \rep
jml-keyword ::= abrupt_behavior
        | accessible | accessible_redundantly
        | also | assert_redundantly
        | assignable | assignable_redundantly
        | assume | assume_redundantly | axiom
        | behavior | breaks | breaks_redundantly
        | callable | callable_redundantly
        | captures | captures_redundantly
        | choose | choose_if
        | code_bigint_math | code_contract
        | 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_example
        | exsures | exsures_redundantly
        | 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_example
        | nowarn | old | or
        | post | post_redundantly
        | pre | pre_redundantly
        | pure | readable | refine
        | represents | represents_redundantly
        | requires | requires_redundantly
        | returns | returns_redundantly
```

| `set` | `signals` | `signals_redundantly`
| `spec_bigint_math` | `spec_java_math`
| `spec_protected` | `spec_public` | `spec_safe_math`
| `static_initializer` | `uninitialized`
| `unreachable` | `weakly`
| `when` | `when_redundantly`
| `working_space` | `working_space_redundantly`
| `writable`
| *jml-universe-keyword*
jml-universe-keyword ::= `peer` | `readonly` | `rep`

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

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

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

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

integer-literal ::= *decimal-integer-literal*
| *hex-integer-literal* | *octal-integer-literal*
decimal-integer-literal ::= *decimal-numeral* [*integer-type-suffix*]
decimal-numeral ::= `0` | *non-zero-digit* [*digits*]
digits ::= *digit* [*digit*] . . .
digit ::= `0` | *non-zero-digit*
non-zero-digit ::= `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
integer-type-suffix ::= `l` | `L`
hex-integer-literal ::= *hex-numeral* [*integer-type-suffix*]
hex-numeral ::= `0x` *hex-digit* [*hex-digit*] . . .
| `0X` *hex-digit* [*hex-digit*] . . .
hex-digit ::= *digit* | `a` | `b` | `c` | `d` | `e` | `f`
| `A` | `B` | `C` | `D` | `E` | `F`
octal-integer-literal ::= *octal-numeral* [*integer-type-suffix*]
octal-numeral ::= `0` *octal-digit* [*octal-digit*] . . .

octal-digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

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

boolean-literal ::= `true` | `false`

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

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

null-literal ::= `null`

An *informal-description* looks like `(* some text *)`. It is used in predicates (see Section 11.1 [Predicates], page 68) and store-ref expressions (see Section 11.3 [Store Refs], page 79) as an escape from formality.

The exact syntax is given below.

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

5 Compilation Units

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

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

The *compilation-unit* rule is the start rule for the JML grammar. (In this syntactic rule and in all other rules in the rest of the body of this manual, *white-space* may appear between any two tokens. See Chapter 4 [Lexical Conventions], page 17, for details.)

See Chapter 6 [Type Definitions], page 27, for the syntax and semantics of *type-definition*s. See Section 18.1 [Augmenting Method Declarations], page 99, for the syntax and semantics of *multijava-top-level-declaration*. See Chapter 17 [Refinement], page 93, for a discussion of the *refine-prefix* and its uses.

Some JML tools may support various optional extensions to JML. This manual partially describes two such extensions: MultiJava [Clifton-etal00] and the Universe type [Mueller-Poetzsch-Heffter01a]. Comments in the grammar indicate optional productions; these are only used by tools that select an option to parse the syntax in question. Tools for JML do not have to support these extensions to JML, and may themselves support other JML extensions. In general, JML tools will support a (hopefully well-documented) variant of the language described in this manual.

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

As in Java, JML can be implemented using files to store compilation units. When this is done there must also be a correspondence between the name of any public type defined in a compilation unit and the file name. This is done exactly as in Java, although JML allows additional file name suffixes. See Section 17.1 [File Name Suffixes], page 93, for details on the file name suffixes allowed in JML.

The specification of the compilation unit consists of the specifications of the *top-level-definition*s it contains, placed in the declared package (if any). The interface part of this specification is determined as in Java [Gosling-etal00] (or as in the Java extension used). The specifications of each *type-definition* are computed by starting from an environment that contains the declared package (if any), each top-level definition in the compilation unit (to allow for mutual recursion), and the imports [Gosling-etal00]. In JML, not only is the package `java.lang` implicitly imported, but also there is an implicit model import of `org.jmlspecs.lang`. (See Section 5.2 [Import Definitions], page 26, for the meaning of a model import.)

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

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

5.1 Package Definitions

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

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

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

5.2 Import Definitions

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

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

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

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

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

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

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

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

6 Type Definitions

The following is the syntax of type definitions.

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

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

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

6.1 Class and Interface Definitions

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

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

Documentation comments for classes and interfaces may not contain JML specification information. See Section 4.5 [Documentation Comments], page 19, for the syntax of documentation comments.

See Chapter 7 [Class and Interface Member Declarations], page 33, for the syntax and semantics of *fields*, which form the guts of classes and interfaces.

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

6.1.1 Subtyping for Type Definitions

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

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

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

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

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

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

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

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

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

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

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

See the report, “Desugaring JML Method Specifications” [Raghavan-Leavens00] for more about the details of specification inheritance in JML.

6.1.2 Modifiers for Type Definitions

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

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

See Section 6.2 [Modifiers], page 29, for the syntax and semantics of modifiers in general.

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

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

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

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

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

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

6.2 Modifiers

The following is the syntax of modifiers.

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

The *jml-modifiers* are only recognized as keywords in annotation comments. See Chapter 4 [Lexical Conventions], page 17, for more details.

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

Note that although the *modifiers* grammar non-terminal is used in many places throughout the grammar, not all modifiers can be used with every grammar construct. See the discussion regarding each grammar construct, which is summarized in Appendix B [Modifier Summary], page 120.

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

6.2.1 Suggested Modifier Ordering

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

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

6.2.2 Spec Public

The `spec_public` modifier allows one to declare a feature as public for specifiction purposes. It can only be used when the feature has a more restrictive visibility in Java.

6.2.3 Spec Protected

The `spec_protected` modifier allows one to declare a feature as protected for specifiction 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).

6.2.4 Pure

In general terms, a *pure* feature is one that has no side effects when executed. In essence `pure` only applies to methods and constructors. The use of `pure` for a type definition is shorthand for applying that modifier to all constructors and instance methods in the type (see Section 6.1.2 [Modifiers for Type Definitions], page 28).

See Section 7.1.1.3 [Pure Methods and Constructors], page 35, for the exact semantics of pure methods and constructors.

6.2.5 Model

The `model` modifier introduces a specification-only feature. For fields it also has a special meaning, which is that the field can be represented by concrete fields. See Section 2.2 [Model and Ghost], page 10.

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

6.2.6 Ghost

The `ghost` modifier introduces a specification-only field that is maintained by special set statements. See Section 2.2 [Model and Ghost], page 10.

6.2.7 Instance

The `instance` modifier says that a field is not static. See Section 2.5 [Instance vs. Static], page 13.

6.2.8 Helper

The `helper` modifier may be used on a private method or constructor to say that its specification is not augmented by invariants and history constraints that would otherwise be relevant. Normally, an invariant applies to all methods in a class or interface. However, an exception is made for methods and constructors declared with the `helper` modifier. See Section 8.2 [Invariants], page 39.

6.2.9 Monitored

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

6.2.10 Uninitialized

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

6.2.11 Math Modifiers

[[[Need explanation of these.]]]

7 Class and Interface Member Declarations

The nonterminal *field* describes all the members of classes and interfaces (see Section 6.1 [Class and Interface Definitions], page 27).

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

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 39, for the syntax and semantics of *jml-declaration*, and, more generally, how to use JML to specify the behavior of types.

7.1 Java Member Declarations

The following gives the syntax of Java member declarations.

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

See Section 6.1 [Class and Interface Definitions], page 27, for details of *class-definition* and *interface-definition*. We discuss method and variable declarations below.

7.1.1 Method and Constructor Declarations

The following is the syntax of a method declaration.

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

Notice that the specification of a method (see Chapter 9 [Method Specifications], page 49) may appear either before or after the *method-head*.

[[[JML tools allow methods in Java files to omit method bodies.]]]

[[[DRC - isn't the method-or-constructor-keyword optional ?]]]

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.

7.1.1.1 Formal Parameters

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

See Section 7.1.2.2 [Type-Specs], page 37, for more about the nonterminals *type-spec* and *dims*. See Section 18.2 [MultiMethods], page 99, for details of *multijava-param-declaration*.

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

The `non_null` modifier on a parameter is inherited in the same way as the equivalent precondition would be, so it need not be declared on every declaration of the same method in a subtype or refinement. The `non_null` modifier may be added to a method in a refinement file (see Chapter 17 [Refinement], page 93), and thus does not have to appear in any particular file in a refinement sequence. It can be added to a method override in a subtype, but that will generally make the method non-implementable, as the method must also satisfy an inherited specification without the corresponding precondition.

7.1.1.2 Model Methods and Constructors

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

As usual in JML (see Section 2.2 [Model and Ghost], page 10), a model method or constructor is a specification-only feature. A model method or constructor may have either a body or a specification, or both. The specification may be used in various verification tools, while the body allows it to be executed during runtime assertion checking. Model methods may also be abstract, and both model methods and constructors may be final.

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

It is usual in JML to declare model methods and constructors as `pure`. However, it is possible to have a model method or constructor that is not pure; such methods are useful in model programs (see Chapter 15 [Model Programs], page 87).

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-Ruby04].

We say a method is *pure* if it is either specified with the modifier `pure` or is a non-static 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.

[[[Why are static methods of a pure class allowed to be non-pure ??? - DRC]]]

A *pure method* that is not a constructor implicitly has a specification that does not allow any side-effects. That is, its specification refines (i.e., is stronger than) the following[1]:

```
behavior
   assignable \nothing;
```

A *pure constructor* implicitly has a specification that only allows it to assign to the non-static fields of the class in which it appears (including those inherited from its superclasses and model instance fields from the interfaces that implements). [[[What about ghost instance fields from interfaces? - DRCok]]]

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, a pure method or constructor implementation is prohibited from calling methods or constructors that are not pure.

A pure method or constructor must also be provably terminating.[2] [[[OK - but some rationale on this restriction would be useful - off-hand it does not seem necessary. - DRCok]]] Recursion is permitted, both in the implementation of pure methods and the data structures they manipulate, and in the specifications of pure methods. When recursion is used in a specification, the proof of well-formedness for the specification involves the use of JML's `measured_by` clause.

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

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.

In JML the modifiers `model` and `pure` are orthogonal. (Recall something declared with the modifier `model` does not have to be implemented and is used purely for specification purposes.) Therefore, one can have a model method that is not pure (these might be useful in JML's model programs) and a pure method that is not a model method. Nevertheless,

[1] For this reason, if one is writing a pure method, it is not necessary to otherwise specify an assignable clause (see Section 9.9.8 [Assignable Clauses], page 64), although doing so may improve the specification's clarity.

[2] This is already implicit in the specification given above for pure methods, since the default `diverges` clause is `false`.

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.

7.1.1.4 Helper Methods and Constructors

The `helper` modifier may only be used on a private method or constructor. [[[This restriction needs to be clarified - ESC/Java limits helper to non-overridable methods.]]] Such a helper method or constructor has a specification that is not augmented by invariants and history constraints that would otherwise apply to it. It can thus be thought of as not really a method or constructor, but merely an abbreviation device. [[[Except that its own specifications still apply. - DRCok]]] See Section 8.2 [Invariants], page 39, for more details.

7.1.2 Field and Variable Declarations

The following is the syntax of field and variable declarations.

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

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

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

In a non-Java file, such as a file with suffix '`.refines-java`' (see Chapter 17 [Refinement], page 93), 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 67, for more about *jml-data-group-clauses*. See Section 11.2 [Specification Expressions], page 76, 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 81), 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 47). See Section 2.2 [Model and Ghost], page 10, 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 39, 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 13.

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`.

> *type-spec* ::= *type* [*dims*] `|` `\TYPE` [*dims*]
> *type* ::= *reference-type* `|` *built-in-type*
> *reference-type* ::= *name*
> *dims* ::= ‘`[`’ ‘`]`’ [‘`[`’ ‘`]`’] . . .

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

7.2 Class Initializer Declarations

The following is the syntax of class initializers.

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

The first form above is the form of Java class instance and static intializers. The initializer is static, and thus run when the class is loaded, if it is labeled `static`. The effect of the intializer can be specified by a JML method specification (see Chapter 9 [Method Specifications], page 49), 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 81).

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 [Refinement], page 93). 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 49).

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

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

8 Type Specifications

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

Overall the mechanisms used in JML to specify ADTs can be described as follows. First, the interface of a type is described using the Java syntax for such a type's declaration (see Chapter 7 [Class and Interface Member Declarations], page 33); this includes any required fields and methods, along with their types and visibilities, etc. Second, the behavior of a type is described by declaring model and ghost fields to be the client (or subtype) visible abstractions of the concrete state of the objects of that type, by writing method specifications using those fields, and by writing various *jml-declaration*s to further refine the logical model defined by these fields. These *jml-declaration*s can also be used to record various design and implementation decisions.

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

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

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 Exampless

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

8.2 Invariants

The syntax of an invariant declaration is as follows.

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

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

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

public abstract class Invariant {

    boolean[] b;
```

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

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

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

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

Note that visible states for an object *o* do not include states at the beginning and end of invocations of constructors and methods declared with the `helper` modifier.

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

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

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

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

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

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

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

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

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

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

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

To accomplish this ideal, it seems that some mechanism for object ownership and alias control [Noble-Vitek-Potter98] [Mueller-Poetzsch-Heffter00] [Mueller-Poetzsch-Heffter00a] [Mueller-Poetzsch-Heffter01a] [Mueller02] 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-Leavens02].

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 36.

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 43.
- 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 44.
- 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 43.
- A class inherits all visible invariants specified in its superclasses and superinterfaces; see Section 8.2.4 [Invariants and Inheritance], page 44.
- 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 49.
- When considering an individual method body, remember that invariants should not just hold in the beginning and the end of it, but also at any program point halfway where another (non-`helper`) method or constructor is invoked. After all, these program points are also visible states, and, as stated above, invariants should hold at all visible states.
- A method invocation on an object should not just preserve the instance invariants of that object and the static invariants of the class, but it should preserve the invariants of all other (reachable) objects as well [Poetzsch-Heffter97].

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

8.2.1 Static vs. instance invariants

As discussed above (see Section 8.2 [Invariants], page 39), invariants can be declared `static` or `instance`. Just like a static method, a static invariant cannot refer to the current object `this` and thus cannot refer to instance fields of `this` or non-static methods of the type.

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

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

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

The table below summarizes this:

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

A word of warning about terminology. In standard Java terminology static members are also called class members. However, static invariants should *never* be called class invariants! This would conflict with the standard use of the term "class invariant" in the literature, where "class invariant" always means instance invariant.

8.2.2 Invariants and Exceptions

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

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

The requirement that invariants hold after abrupt termination of a method or constructor may seen excessively strong. However, it is the only sound option in the long run. After 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 11, for the details and an example using invariants.

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

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

8.2.4 Invariants and Inheritance

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

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

8.3 Constraints

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

The syntax of history constraints in JML is as follows.

history-constraint ::= *constraint-keyword predicate*
        [`for` *constrained-list*] `;`
constraint-keyword ::= `constraint` | `constraint_redundantly`
constrained-list ::= *method-name-list* | `\everything`
method-name-list ::= *method-name* [`,` *method-name*] . . .
method-name ::= *method-ref* [`(` [*param-disambig-list*] `)`]
method-ref ::= *method-ref-start* [`.` *method-ref-rest*] . . .
    | `new` *reference-type*
method-ref-start ::= `super` | `this` | *ident* | `\other`
method-ref-rest ::= `this` | *ident* | `\other`
param-disambig-list ::= *param-disambig* [`,` *param-disambig*] . . .
param-disambig ::= *type-spec* [*ident* [*dims*]]

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

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

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

public abstract class Constraint {

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

    boolean[] b;

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

    boolean[] c;

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

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

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

A constraint declaration may optionally explicitly list one or more methods. It is the listed methods that must *respect* the constraint. If no methods are listed, then all non-helper methods of the class (and any subclasses) must respect the constraint. A method respects a history constraint iff the pre-state and the post-state of a non-static method invocation are in the relation specified by the history constraint (modulo static initialization). [[[What does static initialization have to do with it? - DRC]]] 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 46.
- Constraints can be declared with the access modifiers `public`, `protected`, and `private`; see Section 8.3.2 [Access Modifiers for Constraints], page 47.
- 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 47.
- Although some aspects of constraints are discussed in isolation here, the full explanation of their semantics can only be given considered together with that of method specifications. After all, a method only has to respect constraints when one of the preconditions (ie. `requires` clauses) specified for that method holds. So constraints are an integral part of the explanation of method specifications in Chapter 9 [Method Specifications], page 49.
- When considering an individual method body, remember that constraints not only have to hold between the pre-state and the post-state, but between all visible state that arise during execution of the method. So, given that any program points in the method where (non-`helper`) methods or constructors are invoked are also visible states, constraints should also hold between the pre-state and any such program points, between these program points themselves, and between any such program points and the post-state.
- A method invocation on an object `o` should not just respect the constraints of `o`, but should respect the constraints of all other (reachable) objects as well.

These aspects of constraints are discussed in more detail below.

8.3.1 Static vs. instance constraints

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

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

Instance constraints must be respected by all instance methods.

The table below summarizes this:

	static initialization	non-helper static method	non-helper constructor	non-helper instance method
static invariant	(irrelevant)	respect	respect	respect
instance invariant	(irrelevant)	(irrelevant)	(irrelevant)	respect

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 44.

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

The first form of `represents` clauses is called a *functional abstraction* and the second form is called a *relational abstraction*.

- 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 33, for details of model fields).
- In a functional abstraction form, the type of right-hand side of a `represents clause` must be assignment-compatible to the type of left-hand side.

A `represents` clause can be declared as `static` (See Chapter 6 [Type Definitions], page 27, for `static` declarations). In a `static represents` clause, only static elements can be referenced both in the left-hand side and the right-hand side. In addition, the following restriction is enforced.

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

Unless explicitly declared as `static`, a `represents` clause is non-`static` (for exceptions see see Chapter 6 [Type Definitions], page 27). 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 in) 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 responsibilty to make sure such definitions are well-defined. But such recursive represents clauses can be useful when dealing with recursive datatypes [Mueller-Poetzsch-Heffter-Leavens02].

8.5 Initially Clauses

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

8.6 Axioms

[[[grammar production needed]]] [[[description needed]]]

8.7 Readable If Clauses

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

8.8 Writable If Clauses

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

8.9 Monitors For Clause

The *monitors-for-clause* is adapted from ESC/Java. It specifies an object and a set of objects, one of which must be locked for the first object to be accessed.

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

9 Method Specifications

Although the use of pre- and postconditions for specification of the behavior of methods is very standard, JML offers some features that are not so standard. A good example 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 is the use of privacy modifiers to specify for different readers, and the use of redundancy [Tan94] [Leavens-Baker99]. [[[Are we using this document as a reference manual or to motivate the design of JML as well? In any case the last sentence above needs improvement- David]]]

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 87.

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 86.

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 these components specifies a property that must be satisfied by the implementation of the method or constructor in question. A *method-specification* must satisfy all the specified properties together. (So, speaking loosely, its meaning is the "conjunction" of the semantics of these individual components.)

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

spec-case ::= *lightweight-spec-case* | *heavyweight-spec-case*
| *model-program* | *code-contract-spec*

Model programs are discussed in Chapter 15 [Model Programs], page 87. The remainder of this chapter concentrates on lightweight and heavyweight behavior specification cases. [[[Comment on where code-contract-spec is discussed.]]] JML distinguishes between

- *heavyweight specification cases*, which start with one of the keywords `behavior`, `normal_behavior` or `exceptional_behavior` (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-Ruby04]. It also is possible to desugar all such specification cases into behavior specification cases [Raghavan-Leavens00].

9.3 Access Control in Specification Cases

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

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

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

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

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

JML's usual visibility rules apply to specification cases. So, for example, a public specification case may only refer to public members, a protected specification case may refer to both public and protected members, as long as the protected members are otherwise accessible according to Java's rules, etc. See Section 2.4 [Privacy Modifiers and Visibility], page 11, for more details and examples.

9.4 Lightweight Specification Cases

Syntax

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

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

As far as the syntax is concerned, the only difference between a lightweight specification cases and a *behavior-specification-case* (see Section 9.6 [Behavior Specification Cases], page 53) is that the latter has the keyword `behavior` and possibly an access control modifier.

A lightweight specification case always has the same access modifier as the method being specified, see Section 9.3 [Access Control in Specification Cases], page 50. To specify a different access control modifier, one must use a heavyweight specification.

Semantics

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

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

public abstract class Lightweight {

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

    /*@ requires P;
      @  assignable X;
      @  ensures Q;
      @  signals (Exception) R;
```

```
          @*/
        protected abstract int m() throws Exception;
    }
```

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

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

    public abstract class Heavyweight {

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

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

As this example illustrates, the default for an omitted clause in a lightweight specification is `\not_specified` for every clause [Leavens-Baker-Ruby04]. It is intended that the meaning of `\not_specified` may vary between different uses of a JML specification. For example, a static checker might treat a `requires` clause that is `\not_specified` as if it were `true`, while a verification logic would need to treat it as if it were `false`.

In JML, a completely omitted specification is taken to be a lightweight specification. [[[Yes, but see the desugaring procedure for nuances of this - the defaults are different according to whether this would be an extending-specficiation or not - DRCok]]]

9.5 Heavyweight Specification Cases

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

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

Like lightweight specification cases, normal behavior and exceptional behavior specification cases can be understood as syntactic sugar for special kinds of `behavior` specification cases [Raghavan-Leavens00].

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 keyword `behavior`, and the optional access control modifier.

behavior-spec-case ::= [*privacy*] `behavior`
generic-spec-case

Semantics

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

- *Flat* specification cases are of the form

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

  A flat specification case is just made up of a sequence of method specification clauses, ie. `require`, `ensures`, etc. clauses, and its semantics is explained directly in Section 9.6.1 [Semantics of flat behavior specification cases], page 53.
- *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-Leavens00]. This is explained in Section 9.6.5 [Semantics of nested behavior specification cases], page 55.

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 39 and Section 8.3 [Constraints], page 44. 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 specification case with precisely one `requires` clause, one `ensures` clause, one `when` clause one `assignable` clause one `accessible` clause, and at least one `diverges` clause.

A `behavior` specification case can contain any number of these clauses, but, as explained in Section 9.9 [Method Specification Clauses], page 58, any `behavior` specification case is equivalent with a `behavior` specification case of this form.

9.6.2 Non-helper methods

The semantics of a specification

```
behavior
  requires P;
  diverges D;
  assignable A;
  when W;
  ensures Q;
  signals (E1 e1) R1;
  ...
  signals (En en) Rn;
also
code_contract
  accessible C;
  callable p();
```

for a non-`helper` instance method `m` is as follows. [[[What about duration and working_space clauses? - David]]]

If the method is invoked in a pre-state where

- the precondition P holds, and
- all applicable invariants hold

then either:

- the Java virtual machine throws an error that inherits from `java.lang.Error`, or
- the execution of the method does not terminate (i.e., it loops forever or exits without returning or throwing an exception), and the predicate D holds in the pre-state, or
- the method terminates by returning or throwing an exception, and:

  during execution of the method (which includes all 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 by the assignable clause's list A, or are dependees (see Chapter 10 [Data Groups], page 67) of such locations, are assigned to by the method (including by methods or constructors called by the method), and

  in all visible states, all applicable invariants and history constraints hold, as explained in detail in Section 8.2 [Invariants], page 39 and see Section 8.3 [Constraints], page 44, and

  if the execution of the method terminates normally, then the normal postcondition Q holds, as do all applicable invariants and history constraints, and

  if the execution of the method terminates throwing an exception of type Ei, then the exceptional postcondition Ri holds, with the exception object thrown substituted for ei, as do all invariants and constraints, and in the body of the method, only the locations mentioned in the accessible clause list C are directly accessed, and

in the body of the method, only the methods mentioned in the callable clause list p are directly called.

Note that if there is more than one `signals` clause, and the types Ei are related in the subclass hierarchy, then more than one of the exceptional postconditions Ri may apply if an exception is thrown. This is explained in detail in See Section 9.9.4 [Signals Clauses], page 60.

If the formal parameters of the method are used in a (normal or exceptional) postcondition, then these always take the value the parameters had in the pre-state, and not the value they have in the post-state, as explained in See Section 9.9.5 [Parameters in Postconditions], page 61.

9.6.3 Non-helper constructors

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

any instance invariants of the object being initialized by the constructor are not assumed to hold in the precondition,

any instance constraints do not have to be established as implicit part of the postcondition of the constructor.

These two differences are also discussed in Section 8.2 [Invariants], page 39 and Section 8.3 [Constraints], page 44.

9.6.4 Helper methods and constructors

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

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

These differences are also discussed in Section 8.2 [Invariants], page 39 and Section 8.3 [Constraints], page 44.

9.6.5 Semantics of nested behavior specifications

We now explain how all behavior specification cases can be desugared into a list of one or more flat specification cases joined by the `also` keyword [Raghavan-Leavens00]. The semantics of a behavior specification case is then simply the semantics of this desugared version. The meaning of a such a list of specification cases is explained in Section 9.2 [Organization of Method Specifications], page 49, the meaning of a single simple specification case is explained in Section 9.6.1 [Semantics of flat behavior specification cases], page 53.

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

can be desugared into

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

[[[EXAMPLE]]]

9.7 Normal Behavior Specification Cases

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

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

ruling out abrupt termination, i.e., the throwing of any exception. [[What about unchecked exceptions? -JRK]]

The following gives the syntax of the body of a normal behavior specification case. [[[Specifying that a normal behavior spec case does not allow a signals clause by using the grammar seems just confusing and repetitive. Why not use a common grammar with behavior and just specify the restiction separately - aslo for exceptionaly behavior cases - DRCok]]]

normal-behavior-spec-case ::= [*privacy*] `normal_behavior`
                normal-spec-case
normal-spec-case ::= [*spec-var-decls*] [*spec-header*]
                normal-spec-body
        `|` [*spec-var-decls*]
          normal-spec-body
normal-spec-body ::= *normal-spec-clause*
                [*normal-spec-clause*] . . .
        `|` `{|` *normal-spec-case-seq* `|}`
normal-spec-clause ::= *diverges-clause*
        `|` *assignable-clause*
        `|` *when-clause* `|` *working-space-clause*
        `|` *duration-clause* `|` *ensures-clause*
normal-spec-case-seq ::= *normal-spec-case*
                [`also` *normal-spec-case*] . . .

As far as syntax is concerned, the only difference with the base behavior specification case is that normal behavior specification cases use a different behavior keyword and cannot include *signals-clause*s.

The semantics of a normal behavior specification case is the same as the `behavior` specification case obtained by adding the following *signals-clause*

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

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

9.8 Exceptional Behavior Specification Cases

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

exceptional-behavior-spec-case ::= [*privacy*] `exceptional_behavior`
        exceptional-spec-case
exceptional-spec-case ::= [*spec-var-decls*] [*spec-header*]
        exceptional-spec-body
    `|` [*spec-var-decls*]
        exceptional-spec-body
exceptional-spec-body ::= *exceptional-spec-clause*
        [*exceptional-spec-clause*] . . .
    `|` `{|` *exceptional-spec-case-seq* `|}`
exceptional-spec-clause ::= *diverges-clause*
    `|` *assignable-clause*
    `|` *when-clause* `|` *working-space-clause*
    `|` *duration-clause* `|` *signals-clause*
exceptional-spec-case-seq ::= *exceptional-spec-case*
        [`also` *exceptional-spec-case*] . . .

As far as syntax is concerned, the only difference from a standard behavior specification case is that an exceptional behavior specification case uses a different behavior keyword and cannot include an `ensures` clause.

The semantics of a exceptional behavior specification case is the same as the behavior specification case obtained by adding the following `ensures` clause.

```
ensures false;
```

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

9.8.1 Pragmatics of Exceptional Behavior Specifications Cases

Note that an exceptional behavior specification case says that an exception *must* be thrown if its precondition is met (assuming the diverges clause predicate is `false`, as is the default.) Beware of the difference between specifying that an exception *must* be thrown and specifying that an exception *may* be thrown. To specify that an exception *may* be thrown you should *not* use an exceptional behavior, but should instead use a behavior specification case [Leavens-Baker-Ruby04].

For example, the following method specification

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

public abstract class InconsistentMethodSpec {

    /** A specification that can't be satisfied. */
    /*@  public normal_behavior
     @    requires z <= 99;
     @    assignable \nothing;
     @    ensures \result > z;
     @ also
     @  public exceptional_behavior
     @    requires z < 0;
     @    assignable \nothing;
     @    signals (IllegalArgumentException) true;
     @*/
    public abstract int cantBeSatisfied(int z)
        throws IllegalArgumentException;
}
```

is *inconsistent* because the preconditions `z <= 99` and `z < 0` overlap, for example when `z` is `-1`. When both preconditions hold then the exceptional behavior case specifies that an exception *must* be thrown and the normal behavior case specifies that an exception *may 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,]]]

This specification is inconsistent (ie. it is impossible to come up with an implementation that meets this specification), because if [[[..]]] and [[[...]]] then the specification requires that two different exception are thrown, which is clearly impossible.

[[[Also need discussion and an example for the following. If one just gives a clause such as "signals (E e)" it does not require anything - sometimes writers assume that this requires that the exception thrown be of type E. Not so. - DRCok]]]

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 50, for the overall syntax that ties these clauses together.

9.9.1 Specification Variable Declarations

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

spec-var-decls ::= *forall-var-decls* [*old-var-decls*]
        | *old-var-decls*
forall-var-decls ::= *forall-var-decl* [*forall-var-decl*] . . .
forall-var-decl ::= `forall` *quantified-var-decl* `;`

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

9.9.2 Requires Clauses

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

requires-clause ::= *requires-keyword pred-or-not* `;`
requires-keyword ::= `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 11, for more details on visibility in JML.

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

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

means the same thing as:

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

[[[How do we deal with `\not_specified` for this conjunction semantics?]]] [[[Also explain the meaning if P or Q throws an exception. Is the above equivalence true if Q is defined only if P is true (i.e. making use of the short-circuit nature of &&)?]]]

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`.

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 59, for the syntax of *pred-or-not*.

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

A *predicate* in an `ensures` clause can refer to any visible fields, the parameters of the method, `\result` if the method is non-void, and may contain expressions of the form `\old(E)`. See Section 2.4 [Privacy Modifiers and Visibility], page 11, for more details on visibility in JML.

Informally,

```
ensures Q;
```

means

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

In an ensures clause, `\result` stands for the result that is returned by the method. The postcondition Q may contain expressions of the form `\old(e)`. Such expressions are evaluated in the pre-state, and not in the post-state, and allow Q to express a relation between the pre- and the post-state. If parameters of the method occur in the postcondition Q, these are always evaluated in the pre-state, not the post-state. In other words, if a method parameter x occurs in Q, it is treated as `\old(`x`)`. For a detailed explanation of this see Section 9.9.5 [Parameters in Postconditions], page 61.

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

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

means the same as

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

[[[Also explain the meaning if P or Q throws an exception. Is the above equivalence true if Q is defined only if P is true (i.e. making use of the short-circuit nature of &&)?]]]

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

9.9.4 Signals Clauses

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

The syntax is as follows See Section 9.9.2 [Requires Clauses], page 59, for the syntax of *pred-or-not*.

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

In a *signals-clause* of the form

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

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

Informally,

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

means

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

A signals clause of the form

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

is equivalent to the signals clause

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

Several signals clauses can be given in a single lightweight, behavior or exceptional behavior specification case. Multiple signals clauses in a specification case mean the same as a single signals clause whose exceptional postcondition predicate is the *conjunction* of the exceptional postcondition predicates in the given signals clauses. This should be understood to take place after the desugaring given above, which makes all the signals clauses refer to exceptions of type `java.lang.Exception`. Also, the names in the given signals clauses have to be standardized [Raghavan-Leavens00]. So for example,

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

means the same as

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

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

[[[EXAMPLE]]]

Beware that a `signals` clause specifies when a certain exception *may* be thrown, not when a certain exception *must* be thrown. To say that an exception must be thrown in some situation, one has to exclude that situation from other signals clauses and from ensures clause (and any diverges clauses).

[[[EXAMPLE?]]]

9.9.5 Parameters in Postconditions

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

This leads us to the following two rules:

- The parameters of a method or constructor can never be listed in the its assignable clause.
- 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 ensures or signals clause, then these always have their value in the pre-state of the method (or constructor), not the

post-state. In other words, there is an implicit `\old()` placed around any occurrence of a formal parameter in a postcondition.

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

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

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

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

public abstract class ImplicitOld {

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

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

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

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

9.9.6 Diverges Clauses

The diverges clause is a seldom-used feature of JML. It says when a method may loop forever or otherwise not return to its caller, by either throwing an exception or returning normally. The syntax is as follows See Section 9.9.2 [Requires Clauses], page 59, for the syntax of *pred-or-not*.

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

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

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

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

public abstract class Diverges {

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

}
```

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

9.9.7 When Clauses

The `when` clause allows concurrency aspects of a method or constructor to be specified [Lerner91] [Sivaprasad95]. A caller of a method will be delayed until the condition given in the `when` clause holds. (Note that support for concurrency in JML is in its infancy.)

The syntax is as follows See Section 9.9.2 [Requires Clauses], page 59, 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`.

[[[Need an example of a when clause and how it is used.]]]

9.9.8 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 their dependees) can be assigned to during the execution of the method. However, locations that are local to the method (or methods it calls) and locations that are created during the method's execution are not subject to this restriction.

The syntax is as follows. See Section 11.3 [Store Refs], page 79, for the syntax of *store-ref-list*.

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

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

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

```
assignable \nothing;
```

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

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

[[[Flesh out other restrictions]]]

9.9.9 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 a adapted from the work of Krone, Ogden, and Sitaraman on RESOLVE [Krone-Ogden-Sitaraman03].

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

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

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

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

```
working_space \not_specified;
```

See Section 11.1.8 [Backslash working space], page 71, for information about the `\working_space` expression that can be used to describe the working space needed by a method call. See Section 11.1.7 [Backslash space], page 71, for information about the `\space` expression that can be used to describe the heap space occupied by an object.

9.9.10 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 a adapted from the work of Krone, Ogden, and Sitaraman on RESOLVE [Krone-Ogden-Sitaraman03].

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

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

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

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

```
        duration \not_specified;
```

See Section 11.1.6 [Backslash duration], page 71, for information about the `\duration` expression that can be used in the duration clause to specify the duration of other methods.

10 Frame Conditions and Data Groups

[[[discussion needed]]]

10.1 Data Groups

[[[needs discussion - general usage, default data groups, use in modifies statements]]]

The following is the syntax for `in` and `maps` data group clauses.

jml-data-group-clause ::= *in-group-clause* `|` *maps-into-clause*
in-group-clause ::= *in-keyword group-list* `;`
in-keyword ::= `in` `|` `in_redundantly`
group-list ::= *group-name* [`,` *group-name*] . . .
group-name ::= [*group-name-prefix*] *ident*
group-name-prefix ::= `super .` `|` `this .`
maps-into-clause ::= *maps-keyword member-field-ref* `\into` *group-list* `;`
maps-keyword ::= `maps` `|` `maps_redundantly`
member-field-ref ::= *ident* `.` *maps-member-ref-expr*
        `|` *maps-array-ref-expr* [`.` *maps-member-ref-expr*]
maps-member-ref-expr ::= *ident* `|` `*`
maps-array-ref-expr ::= *ident maps-spec-array-dim*
                [*maps-spec-array-dim*] . . .
maps-spec-array-dim ::= ‘`[`’ *spec-array-ref-expr* ‘`]`’

See [Leino98]. [[[Need def of spec-array-ref-expr]]]

10.2 Static Data Group Inclusions

[[[`in`, `in_redundantly`, needs discussion]]]

10.3 Dynamic Data Group Mappings

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).

[[[`maps`, `maps_redundantly`, `\into` needs discussion]]]

11 Predicates and Specification Expressions

11.1 Predicates

The following gives the syntax of predicates and specification expressions. Within a *spec-expression*, one cannot use any of the operators (such as `++`, `--`, and the assignment operators) that would necessarily cause side effects. See Section 11.2 [Specification Expressions], page 76, for the syntax of expressions.

```
predicate ::= spec-expression
spec-expression-list ::= spec-expression
                         [ , spec-expression ] . . .
spec-expression ::= expression

jml-primary ::= \result
        | \old ( spec-expression )
        | \not_assigned ( store-ref-list )
        | \not_modified ( store-ref-list )
        | \fresh ( spec-expression-list )
        | \reach ( spec-expression )
        | \duration ( expression )
        | \space ( spec-expression )
        | \max ( spec-expression )
        | \working_space ( expression )
        | informal-description
        | \nonnullelements ( spec-expression )
        | \typeof ( spec-expression )
        | \elemtype ( spec-expression )
        | \type ( type )
        | \lockset
        | \is_initialized ( reference-type )
        | \invariant_for ( spec-expression )
        | ( \lblneg ident spec-expression )
        | ( \lblpos ident spec-expression )
        | spec-quantified-expr

set-comprehension ::= { type-spec
          quantified-var-declarator '|'
          set-comprehension-pred }
set-comprehension-pred ::=
          postfix-expr . has ( ident ) &&
          predicate

spec-quantified-expr ::= ( quantifier quantified-var-decls ;
                          [ [ predicate ] ; ]
                          spec-expression )
```

> *quantifier* ::= `\forall` | `\exists`
> | `\max` | `\min`
> | `\num_of` | `\product` | `\sum`
> *quantified-var-decls* ::= *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*

[[[SUggest that we not allow [dims] after an ident - only asd part of the typespec. - DRCok]]] [[[Shouldn't set-comphrehension be a jml-primary?]]]

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-Ruby04].

11.1.1 `\result`

The JML keyword `\result` can only be used in `ensures` 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. The keyword `\result` can only be used in an *ensures-clause*; it cannot be used, for example, in preconditions or in signals clauses. [[[Also can be used in duration and workingspace specifications – DRCok]]]

11.1.2 `\old`

The JML keyword `\old` can be used in both normal and exceptional postconditions (ie. in `ensures` and `signals` clauses), and in history constraints. An expression of the form `\old(`*Expr*`)` refers to the value that the expression *Expr* had in the pre-state of a method. [[[Also can be used in duration and workingspace specifications – DRCok]]] [[[Would be nice to use \old in assert, assume, set statements – DRCok]]]

The type of `\old(`*Expr*`)` is simply the type of *Expr*.

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

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

The problem with the above example is that there is no easy way to evaluate `\old(i < y)` in the pre-state. [[[Why isn't it just equivalent to the expression below, instead of being illegal ? - DRCok]]]

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

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

11.1.3 `\not_assigned`

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 named by the argument were not assigned to during the execution of the method being specified (or all methods to which a history constraint applies). For example, `\not_assigned(xval,yval)` says that the locations in the data groups named by `xval` and `yval` were not assigned during the method's execution.

A predicate such as `\not_assigned(x.f)` refers to the entire data group named by `x.f` not just to the location `x.f` itself. This allows one to specify absence of even temporary side-effects in various cases of a method. See Section 11.1.4 [Backslash not_modified], page 70, for ways to specify that just the value of a given field was not changed, which allows temporary side effects.

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

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

11.1.4 `\not_modified`

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

A predicate such as `\not_modified(x.f)` refers to the location named by `x.f`, not to the entire data group of `x.f`. This allows one to specify benevolent side-effects, as one can name `x.f` (or a data group in which it participates) in an assignable clause, but use `\not_modified(x.f)` in the postcondition. See Section 11.1.3 [Backslash not_assigned], page 70, for ways to specify that no assignments were made to any location in a data group, disallowing temporary side effects.

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

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

11.1.5 `\fresh`

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

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

11.1.6 `\duration`

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

The argument expression passed to `\duration` must be a method call or explicit constructor invocation expression; the type of a `\duration` expression is `long`. [[[Why not make this simply an arbitrary expression? - DRCok]]]

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

11.1.7 `\space`

`\space`, which describes the amount of heap space, in bytes, allocated to the object refered to by its argument [Krone-Ogden-Sitaraman03]; e.g., `\space(myStack)` is number of bytes in the heap used by `myStack`, not including the objects it contains. The type of the *spec-expression* that is the argument must be a reference type, and the result type of a `\space` expression is `long`. [[[The argument is evaluated, no? - DRCok]]]

11.1.8 `\working_space`

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

11.1.9 `\reach`

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

11.1.10 `\nonnullelements`

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

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

11.1.11 Subtype operator

The relational operator `<:` compares two reference types and returns true when the type on the left is a subtype of the type on the right [Leino-Nelson-Saxe00]. Although the notation might suggest otherwise, this operator is also reflexive; a type will compare as `<:` with itself. In an expression of the form *E1* `<:` *E2*, both *E1* and *E2* must have type `\TYPE`; since in JML `\TYPE` is the same as `java.lang.Class` the expression *E1* `<:` *E2* means the same thing as the expression *E2*`.isAssignableFrom(`*E1*`)`. As a result, primitive types are not subtypes of `java.lang.Object`, nor of each other, though they are of themselves; so, for example, `Integer.TYPE <: Integer.TYPE` is true.

11.1.12 `\typeof`

The operator `\typeof` returns the most-specific dynamic type of an expression's value [Leino-Nelson-Saxe00]. The meaning of `\typeof(`*E*`)` is unspecified if *E* is null. If *E* has a static type that is a reference type, then `\typeof(`*E*`)` means the same thing as

E`.getClass()`. For example, if `c` is a variable of static type `Collection` that holds an object of class `HashSet`, then `\typeof(c)` is `HashSet.class`, which is the same thing as `\type(HashSet)`. If *E* has a static type that is not a reference type, then `\typeof(`*E*`)` means the instance of `java.lang.Class` that represents its static type. For example, `\typeof(true)` is `Boolean.TYPE`, which is the same as `\type(boolean)`. Thus an expression of the form `\typeof(`*E*`)` has type `\TYPE`, which JML considers to be the same as `java.lang.Class`.

11.1.13 `\elemtype`

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

[[[Should the result be null or \typeof(null) ??? SHould \typeof(null) be null ?]]]

11.1.14 `\type`

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

For example, in

```
\typeof(myObj) <: \type(PlusAccount)
```

the use of `\type(PlusAccount)` is used to introduce the type `PlusAccount` into this expression context.

11.1.15 `\lockset`

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

11.1.16 `\max`

The `\max` operator returns the "largest" (as defined by `<`) of a set of lock objects, given a lock set as an argument. (This is an adaptation from ESC/Java [Leino-etal00] [Leino-Nelson-Saxe00] for dealing with threads.)

11.1.17 `\is_initialized`

The `\is_initialized` operator returns true just when its *reference-type* argument is a class that has finished its static initialization. It is of type `boolean`.

[[[The grammar production reference-type is not defined. DO we write T t; ... \isInitialized(t) or \isInitialized(T) or \isInitialized(T.class)? - DRCok]]]

11.1.18 `\invariant_for`

The `\invariant_for` operator returns true just when its argument satisfies the invariant of its static type; for example, `\invariant_for((MyClass)o)` is true when `o` satisfies the invariant of `MyClass`. The entire `\invariant_for` expression is of type `boolean`.

11.1.19 `\lblneg` and `\lblpos`

Parenthesized expressions that start with `\lblneg` and `\lblpos` can be used to attach labels to expressions [Leino-Nelson-Saxe00]; these labels might be printed in various messages by support tools, for example, to identify an assertion that failed. Such an expression has a *label* and a *body*; for example, in

```
(\lblneg indexInBounds 0 <= index && index < length)
```

the label is `indexInBounds` and the body is the expression `0 <= index && index < length`. The value of a labeled expression is the value of its body, hence its type is the type of its body. The idea is that if this expression is used in an assertion and its value is `false` (e.g., when doing run-time checking of assertions), then a warning will be printed that includes the label `indexInBounds`. The form using `\lblpos` has a similar syntax, but should be used for warnings when the value of the enclosed expression is `true`.

11.1.20 Universal and Existential Quantifiers

The quantifiers `\forall` and `\exists`, are universal and existential quantifiers (respectively). For example,

```
(\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])
```

says that `a` is sorted at indexes between 0 and 9. The quantifiers range over all potential values of the variables declared which satisfy the *range* predicate, given between the semicolons (`;`). If the range predicate is omitted, it defaults to `true`. Since a quantifier quantifies over all potential values of the variables, when the variables declared are reference types, they may be null, or may refer to objects not constructed by the program; one should use a range predicate to eliminate such cases if they are not desired. The type of a universal and existential quantifier is `boolean`. [[[May the bound formal identifier in a quantifier expression be any identifier, or must it not redeclare a local variable? - DRCok]]]

11.1.21 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 range predicate must be of type `boolean`. The expression in the body must be a built-in numeric type, such as `int` or `double`; the type of the quantified expression as a whole is the type of its body. The *body* of a quantified expression is the last top-level expression it contains; it is the expression following the range predicate, if there is one. As with the universal and existential quantifiers, if the range predicate is omitted, it defaults to `true`. For example, the following equations are all true (see chapter 3 of [Cohen90]):

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
(\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
(\max int i; 0 <= i && i < 5; i) == 4
(\min int i; 0 <= i && i < 5; i-1) == -1
```

For computing the value of a sum or product, Java's arithmetic is used. The meaning thus depends on the type of the expression. For example, in Java, floating point numbers use the IEEE 754 standard, and thus when an overflow occurs, the appropriate positive or negative infinity is returned. However, Java integers wrap on overflow. Consider the following examples.

```
(\product float f; 1.0e30f < f && f < 1.0e38f; f)
  == Float.POSITIVE_INFINITY

(\sum int i; i == Integer.MAX_VALUE || i == 1; i)
  == Integer.MAX_VALUE + 1
  == Integer.MIN_VALUE
```

When the range predicate is not satisfiable, the sum is 0 and the product is 1; for example:

```
(\sum int i; false; i) == 0
(\product double d; false; d*d) == 1.0
```

When the range predicate is not satisfiable for `\max` the result is the smallest number with the type of the expression in the body; for floating point numbers, negative infinity is used. Similarly, when the range predicate is not satisfiable for `\min`, the result is the largest number with the type of the expression in the body. [[[What about if the body type is \bigint ? - DRCok]]]

11.1.22 Numerical Quantifier

The numerical quantifier, `\num_of`, returns the number of values for its variables for which the range and the expression in its body are true. Both the range predicate and the body must have type `boolean`, and the entire quantified expression has type `long`. The meaning of this quantifier is defined by the following equation (see p. 57 of [Cohen90]).

```
(\num_of T x; R(x); P(x)) == (\sum T x; R(x) && P(x); 1L)
```

11.1.23 Set Comprehension

The set comprehension notation can be used to succinctly define sets. For example, the following is the `JMLObjectSet` that is the subset of non-`null` `Integer` objects found in the set `myIntSet` whose values are between 0 and 10, inclusive.

```
new JMLObjectSet {Integer i | myIntSet.has(i) &&
    i != null && 0 <= i.getInteger() && i.getInteger() <= 10 }
```

The syntax of JML limits set comprehensions so that following the vertical bar (`|`) is always an invocation of the has method of some set on the variable declared. (This restriction is used to avoid Russell's paradox [Whitehead-Russell25].) In practice, one either starts from some relevant set at hand, or one can start from the sets containing the objects of primitive types found in `org.jmlspecs.models.JMLObjectSet` and (in the same Java package) `JMLValueSet`. The type of such an expression is the type named following new, which must be `JMLObjectSet` or `JMLValueSet`.

[[[May the bound variable hide other variables that are in scope? – DRCok]]]

11.1.24 <==> and <=!=>

[[[discussion needed]]]

11.1.25 ==> and <==

[[[discussion needed]]]

11.1.26 informal predicates

[[[discussion needed]]]

11.1.27 primitives for safe arithmetic

[[[discussion needed]]]

11.1.28 lockset membership

[[[discussion needed]]]

11.2 Specification Expressions

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

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

```
highest  new () \forall \exists \max \min
           \num_of \product \sum informal-description
           [] . and method calls
         unary + and - ~ ! (typecast)
         * / %
```

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

The following is the syntax of Java expressions, with JML additions. The additions are the operators `==>`, `<==`, `<==>`, `<=!=>`, and `<:`, and the syntax found under the nonterminals *jml-primary*, *set-comprehension*, and *spec-quantified-expr* (see Chapter 11 [Predicates and Specification Expressions], page 68). The JML additions to the Java syntax can only be used in assertions and other annotations. Furthermore, within assertions, one cannot use any of the operators (such as `++`, `--`, and the assignment operators) that would necessarily cause side effects.

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

```
          | shift-expr <= shift-expr
          | shift-expr >= shift-expr
          | shift-expr <: shift-expr
          | shift-expr [ instanceof type-spec ]
shift-expr ::= additive-expr [ shift-op additive-expr ] . . .
shift-op ::= << | >> | >>>
additive-expr ::= mult-expr [ additive-op mult-expr ] . . .
additive-op ::= + | -
mult-expr ::= unary-expr [ mult-op unary-expr ] . . .
mult-op ::= * | / | %
unary-expr ::= ( type-spec ) unary-expr
          | ++ unary-expr
          | -- unary-expr
          | + unary-expr
          | - unary-expr
          | unary-expr-not-plus-minus
unary-expr-not-plus-minus ::= ~ unary-expr
          | ! unary-expr
          | ( builtinType ) unary-expr
          | ( reference-type ) unary-expr-not-plus-minus
          | postfix-expr
postfix-expr ::= primary-expr [ primary-suffix ] . . . [ ++ ]
          | primary-expr [ primary-suffix ] . . . [ -- ]
          | builtinType [ '[' ']' ] . . . . 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
          | informal-description
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*

11.3 Store Refs

[[[Needs discussion, where used, what do they represent?]]]

The syntax related to the *store-ref* production is used in several places.

store-ref-list ::= *store-ref* [`,` *store-ref*] . . .
store-ref ::= *store-ref-expression*
        `|` *informal-description*
        `|` *store-ref-keyword*
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` `|` `\private_data`

12 JML primitive types

[[[Needs grammar]]]

12.1 \TYPE

The type `\TYPE` stands for the type of all types. It is equivalent to the type `java.lang.Class`.

12.2 \real

[[[needs discussion]]]

12.3 \bigint

[[[needs discussion]]]

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 87, for the syntax of *model-prog-statement*, which is only allowed in model programs. [[[Does this include local class declarations?]]]

```
compound-statement ::= { statement [ statement ] ... }
statement ::= compound-statement
        | local-declaration ;
        | ident : statement
        | expression ;
        | if ( expression )
          statement [ else statement ]
        | [ loop-invariant ] ...
          [ variant-function ] ...
          [ ident ] : loop-stmt
        | break [ ident ] ;
        | continue [ ident ] ;
        | return [ expression ] ;
        | switch-statement
        | try-block
        | throw expression ;
        | synchronized ( expression ) statement
        | ;
        | assert-statement
        | hence-by-statement
        | assert-redundantly-statement
        | assume-statement
        | set-statement
        | unreachable-statement
        | debug-statement
        | model-prog-statement // only allowed in model programs
loop-stmt ::= while ( expression ) statement
        | do statement while ( expression ) ;
        | for ( [ for-init ] ; [ expression ] ; [ expression-list ] )
            statement
for-init ::= local-declaration | expression-list
local-declaration ::= local-modifiers variable-decls
local-modifiers ::= [ local-modifier ] ...
local-modifier ::= ghost | final | non_null
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*
> *assert-statement* ::= `assert` *expression* [`:` *expression*] `;`

Note that Java (as of J2SDK 1.4) also has its own `assert` statement. JML distinguishes between assert statements that occur inside and outside annotations. Inside an annotation, such a statement is a JML assert statement, and the first *expression* thus can't have side effects. (The second *expression* is a `String` that, as in Java, is printed if the assertion fails.) Outside an annotation, an assert statement is a Java assert statement, and so the first *expression* can have side effects (potentially, although it shouldn't).

The following gives the syntax of JML annotations that can be used on statements. See Chapter 15 [Model Programs], page 87, for the syntax of statements that can only be used in model programs.

> *hence-by-statement* ::= *hence-by-keyword* *predicate* `;`
> *hence-by-keyword* ::= `hence_by` | `hence_by_redundantly`
> *assert-redundantly-statement* ::= `assert_redundantly` *predicate*
>   [`:` *expression*] `;`
> *assume-statement* ::= *assume-keyword* *predicate*
>   [`:` *expression*] `;`
> *assume-keyword* ::= `assume` | `assume_redundantly`
> *set-statement* ::= `set` *assignment-expr* `;`
> *unreachable-statement* ::= `unreachable ;`
> *debug-statement* ::= `debug` *expression* `;`
> *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`

[[[Why does the assume statement have an extra expression - what is it used for? - DRCok]]]

13.1 assume statement

[[[needs discussion]]]

13.2 assert statement

[[[needs discussion]]]

13.3 Local ghost declaration

A local ghost declaration is a variable declaration with a `ghost` modifier, entirely contained in an annotation. It introduces a new variable that may be used in subsequent annotations within the remainder of the block in which the declaration appears. A ghost variable is not used in program execution as Java variables are, but is used by runtime assertion checkers or a static checker to reason about the execution of the routine body in which the ghost variable is used.

- The variable name may not be already declared as a local variable or local ghost variable or as a formal parameter of the routine in which the declaration appears.
- Each variable declared may have an initializer; the initializer is in the scope of the newly declared variable.
- The modifiers `final`, `uninitialized`, `non_null` may be used on the ghost declaration.

Examples:

```
//@ ghost int i = 0;
//@ ghost int i = 0, j, k = i+3;
//@ ghost float[] a = { 1,2,3 };
//@ ghost Object o;
//@ final ghost non_null Object o = new Object();
//@ ghost \TYPE t = \typeof(t);
```

13.4 set statement

A set statement is the equivalent of an assignment statement but is within an annotation. It is used to assign a value to a ghost variable or to a ghost field. A set statement serves to assist the static checker in reasoning about the execution of the routine body in which it appears.

- the target of the set statement must be a ghost variable or a ghost field
- the right-hand-side of the set statement must be pure (not have side effects)

Examples:

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

[[[Questions: must the rhs be pure? Should we allow an arabitrary statement, not just an assignment? such as set ++i; or set i += 5;]]]

13.5 unreachable statement

The `unreachable` statement is an annotation that asserts that the control flow of a routine will never reach that point in the program. It is equivalent to the annotation `assert false`. If control flow does reach an `unreachable` statement, a tool that checks (by reasoning or at runtime) the behavior of the routine should issue an error of some kind. The following is an example:

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

13.6 debug statement

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.out.println(x);
```

where the variable x may be a ghost variable.

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 its state can be mutated by using the debug statement, as its value is given by a represents clause (see Section 8.4 [Represents Clauses], page 47).

There is no restriction on the type of expression allowed in the debug statement.

[[We might as well generalize set to allow non-pure expressions as well; and then distinguish the two by suggesting the tools provide ways to turn debug statements on and off. -DRCok]]]

13.7 hence_by statement

[[[needs discussion]]]

13.8 loop_invariant statement

[[[needs discussion]]]

13.9 decreases statement

[[[needs discussion]]]

13.10 JML Modifiers for Local Declarations

The following JML modifiers may be applied to local declarations of either Java variables or ghost variables within routine bodies, in addition to the Java modifier `final`.

13.10.1 non_null

[[[needs discussion]]]

13.10.2 uninitialized

[[[needs discussion]]]

14 Redundancy

redundant-spec ::= *implications* [*examples*] `|` *examples*
examples ::= `for_example` *example* [`also` *example*] . . .

14.1 Redundant Implications

implications ::= `implies_that` *spec-case-seq*

14.2 Redundant Examples

The following gives the syntax of examples.

example ::= [[*privacy*] `example`]
  [*spec-var-decls*]
  [*spec-header*]
  simple-spec-body
 `|` [*privacy*] `exceptional_example`
  [*spec-var-decls*]
  spec-header
  [*exceptional-example-body*]
 `|` [*privacy*] `exceptional_example`
  [*spec-var-decls*]
  exceptional-example-body
 `|` [*privacy*] `normal_example`
  [*spec-var-decls*]
  spec-header
  [*normal-example-body*]
 `|` [*privacy*] `normal_example`
  [*spec-var-decls*]
  normal-example-body
exceptional-example-body ::= *exceptional-spec-clause*
  [*exceptional-spec-clause*] . . .
normal-example-body ::= *normal-spec-clause*
  [*normal-spec-clause*] . . .

15 Model Programs

This chapter discusses JML's model programs, which are adapted from the refinement calculus [Back88] [Back-vonWright89a] [Buechi-Weck00] [Morgan94] [Morris87].

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.

In JML adopt the semantics of the "greybox approach" to refinement calculus [Buechi-Weck00] [Buechi00]. In this semantics, calls to non-pure methods in a model program must occur in the same states in a correct implementation. That is, the notion of refinement is restricted to not permit all implementations with equivalent behavior, but to require that the implementation make certain method calls in the model program. This turns out to be very nice for describing both higher-order features and callbacks.

Consider the following example (from a survey on behavioral subtyping by Leavens and Dhara [Leavens-Dhara00]). In this example, both the methods are specified using model programs, which are explained below.

```
package org.jmlspecs.samples.dirobserver;

//@ model import org.jmlspecs.models.JMLString;
//@ model import org.jmlspecs.models.JMLObjectSetEnumerator;

/** Directories that can be both read and written. */
public interface Directory extends RODirectory {

  /** Add a mapping from the given string to the given file to this directory.
   */
  /*@ public model_program {
    @   normal_behavior
    @     requires !in_notifier && n != null && n != "" && f != null;
    @     assignable entries;
    @     ensures entries != null
    @        && entries.equals(\old(entries.extend(new JMLString(n), f)));
    @   for (JMLObjectSetEnumerator e = listeners.elements();
    @        e.hasMoreElements(); ) {
    @     set in_notifier = true;
    @     ((DirObserver)e.nextElement()).addNotification(this, n);
    @     set in_notifier = false;
    @   }
    @ }
    @*/
  public void addEntry(String n, File f);
```

```
    /** Remove the entry with the given name from this directory. */
    /*@ public model_program {
      @   normal_behavior
      @     requires !in_notifier && n != null && n != "";
      @     assignable entries;
      @     ensures entries != null
      @        && entries.equals
      @               (\old(entries.removeDomainElement(new JMLString(n))));
      @   for (JMLObjectSetEnumerator e = listeners.elements();
      @        e.hasMoreElements(); ) {
      @     set in_notifier = true;
      @     ((DirObserver)e.nextElement()).removeNotification(this, n);
      @     set in_notifier = false;
      @   }
      @ }
      @*/
    public void removeEntry(String n);
  }
```

Both model programs in the above example are formed from a specification statement, which begins with the keyword `normal_behavior` in these examples, and a for-loop. The key event in the for loop bodies is a method call to a non-pure 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 behavior 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 behavior statements 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].

15.2 Details of Model Programs

The following gives the syntax of model programs. See Chapter 13 [Statements and Annotation Statements], page 81, for the parts of the syntax of statements that are unchanged from Java. The *jml-compound-statement* and *jml-statement* syntax is the same as the *compound-statement* and *statement* syntax, except that *model-prog-statement*s are not flagged as errors within the *jml-compound-statement* and *jml-statement*s.

model-program ::= [*privacy*] `model_program`
jml-compound-statement
jml-compound-statement ::= *compound-statement*

jml-statement ::= *statement*
model-prog-statement ::= *nondeterministic-choice*
  | *nondeterministic-if*
  | *spec-statement*
  | *invariant*
nondeterministic-choice ::= `choose` *alternative-statements*
alternative-statements ::= *jml-compound-statement*
    [`or` *jml-compound-statement*] ...
nondeterministic-if ::= `choose_if` *guarded-statements*
    [`else` *jml-compound-statement*]
guarded-statements ::= *guarded-statement*
    [`or` *guarded-statement*] ...
guarded-statement ::= `{`
    assume-statement
    jml-statement [*jml-statement*] ... `}`

The grammar for specification statements appears below. It is unusual, compared to specification statements in refinement calculus, in that it allows one to specify statements that can signal exceptions, or terminate abruptly. The reasons for this are based on verification logics for Java [Huisman01] [Poll-Jacobs00], 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.

spec-statement ::= [*privacy*] `behavior`
      generic-spec-statement-case
  | [*privacy*] `exceptional_behavior`
   exceptional-spec-case
  | [*privacy*] `normal_behavior`
   normal-spec-case
  | [*privacy*] `abrupt_behavior`
   abrupt-spec-case
generic-spec-statement-case ::= [*spec-var-decls*]
        generic-spec-statement-body
  | [*spec-var-decls*]
   spec-header
   [*generic-spec-statement-body*]
generic-spec-statement-body ::= *simple-spec-statement-body*
  | `{|` *generic-spec-statement-case-seq* `|}`
generic-spec-statement-body-seq ::= *generic-spec-statement-case*
    [`also` *generic-spec-statement-case*] ...
simple-spec-statement-body ::= *simple-spec-statement-clause*
        [*simple-spec-statement-clause*] ...
simple-spec-statement-clause ::= *diverges-clause*
  | *assignable-clause*
  | *when-clause* | *working-space-clause* | *duration-clause*
  | *ensures-clause* | *signals-clause*
  | *continues-clause* | *breaks-clause* | *returns-clause*

abrupt-spec-case ::= [*spec-var-decls*]
spec-header
[*abrupt-spec-body*]
`|` [*spec-var-decls*]
abrupt-spec-body
abrupt-spec-body ::= *abrupt-spec-clause*
[*abrupt-spec-clause*] . . .
`|` `{|` *abrupt-spec-case-seq* `|}`
abrupt-spec-clause ::= *diverges-clause*
`|` *assignable-clause*
`|` *when-clause* `|` *working-space-clause* `|` *duration-clause*
`|` *continues-clause* `|` *breaks-clause* `|` *returns-clause*
abrupt-spec-case-seq ::= *abrupt-spec-case*
[`also` *abrupt-spec-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`

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], 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 is intended to be created automatically, by a tool (which does not, as of this writing exist). It has the following syntax.

code-contract-spec ::= `code_contract` *code-contract-clause*
        [*code-contract-clause*] . . .

code-contract-clause ::= *accessible-clause*
    | *callable-clause*
    | *measured-clause*
    | *captures-clause*

See Section 16.2.1 [Accessible Clauses], page 91, for the syntax and semantics of the *accessible-clause*. See Section 16.2.2 [Callable Clauses], page 92, for the syntax and semantics of the *callable-clause*. See Section 16.2.3 [Measured By Clauses], page 92, for the syntax and semantics of the *measured-clause*. See Section 16.2.4 [Captures Clauses], page 92, for the syntax and semantics of the *captures-clause*.

16.1 Method of Specifying for Subclasses

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

16.2 Code Contract Clauses

This section discusses the syntax and semantics of the various clauses in the code contract.

16.2.1 Accessible Clauses

The accessible clause says what locations a method may directly access. It has the following syntax.

accessible-clause ::= *accessible-keyword object-store-ref-list* `;`
accessible-keyword ::= `accessible` | `accessible_redundantly`
object-store-ref-list ::= *object-store-ref* [*object-store-ref* ,] . . .
object-store-ref ::= *store-ref* | *other-ref*
other-ref ::= `\other` [*store-ref-name-suffix*] . . .

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

[[[Need some discussion of the meaning of an accessible clause.]]]

16.2.2 Callable Clauses

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

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

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`.

[[[Need some discussion of the meaning of an callable clause.]]]

16.2.3 Measured By Clauses

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

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

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

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

```
measured_by \not_specified;
```

16.2.4 Captures Clauses

The captures clause can appear in a *code-contract-spec* (see Chapter 16 [Specification for Subtypes], page 91). It has the following syntax.

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

A captures clause says that 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.

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

[[[Need some discussion of the meaning of an captures clause.]]]

17 Refinement

This chapter explains JML's notion of refinement files, which uses the following syntax.

refine-prefix ::= `refine` *string-literal* `;`

The *refine-prefix* in a compilation unit says that the declarations in this compilation unit refine the corresponding declarations in the file named by the *string-literal*. The *string-literal* should name a file, complete with a suffix, for example, `"MyType.java-refined"`. The suffix of such a file is used by JML tools to find the file that is the base of a refinement chain, and all other files in the chain are found using the files named in the *refine-prefix* of a previous file in the chain.

One use of this feature is to allow specifications to be written first, in a separate file from the source code. For example, one might specify the class `MyType` in a file named '`MyType.java-refined`'. Then one could write the implementation of `MyType` in a file called '`MyType.java`'. The file '`MyType.java`' would include the following *refine-prefix*:

```
refine "MyType.java-refined";
```

Thus, the specification found in '`MyType.java-refined`' *is a refinement* of the implementation found in '`MyType.java`'.

Another typical use of this feature is to allow one to add specifications to source code that one does not want to modify. To do that, one would use a '`.refines-java`' (or '`.refines-spec`' or '`.refines-jml`') file with the specifications of the corresponding Java file in it. For example, if one wants to specify the type `LibraryType`, without touching the file '`LibraryType.java`' then one could write specifications in the file '`LibraryType.refines-java`', and include in that file the following *refine-prefix*.

```
refine "LibraryType.java";
```

The following gives more details about the checks and meaning of this feature of JML.

17.1 File Name Suffixes

The JML tools recognize several filename suffixes. The following are considered to be *active* suffixes: '`.refines-java`', '`.refines-spec`', '`.refines-jml`', '`.java`', '`.spec`', and '`.jml`'; There are also three *passive* suffixes: '`.java-refined`', '`.spec-refined`', and '`.jml-refined`'. Files with passive suffixes can be used in refinements but should not normally be passed explicitly to the tools directly. These filename suffixes are ordered from most acctive to least active, in the order given above.

17.2 Refinement Chains

Compilation Units that jointly give the specifications of a type form a refinement chain. It begins at a base (or most-refined) compilation unit, proceding by means of the `refine` annotation links, until a file is found that has no `refine` statement. That file is the end of the refinement chain and is the least-refined compilation unit.

For a given type in a given package, the base of the refinement chain is found as follows. Each entry of the classpath is searched in order for a directory whose name matches the package of the type and that contains a file whose name has a prefix matching the type name and a suffix that is an active suffix as defined above. The first such file found is the base of the refinement chain. If the first classpath entry to contain a candidate file contains more than one candidate file, then the file with the most active suffix is the base of the chain.

The subsequent elements of the refinement chain are given by the filenames provided in the `refine` statements. Each element of the chain is in the same package. Thus the file corresponding to the `refine` statement is the first file found by searching the classpath entries in order and that is in the directory corresponding to the package of the type and has the filename given in the `refine` statement.

To help ensure that the base is correctly selected, the file with the most active suffix must be the base of a refinement sequence, otherwise the JML typechecker issues an error message. Also, the prefix of the base file must be the same as the public type declared in that compilation unit or an error message is issued. However, it is not necessary that the file being refined have the same prefix as the file at the base of the refinement chain (except that the .java file, if it is in the refinement sequence, must have a name given by the Java rules for naming compilation units). Furthermore, a file with the same prefix as the base file may not be in a different refinement sequence. For example, '`SomeName.java-refined`' can be refined by '`MyType.java`' as long as there is no refinement sequence with '`SomeName`' as the prefix of the base of another refinement.

The JML tools deal with all files in a refinement chain whenever one of them is selected for processing by the tool. This allows all of the specifications that apply to be consistently dealt with at all times. For example, supose that there are files named '`Foo.refines-java`' and '`Foo.java`', then if a tool selects the '`Foo.java`', e.g., with the command:

```
$ jmlc *.java
```

then it will see both the '`Foo.refines-java`' and the '`Foo.java`' file.

17.3 Type Checking Refinements

There are some restrictions on what can appear in the different files involved in a particular refinement. Since the Java compilers only see the '`.java`' files, executable code (that is not just for use in specifications) should only be placed in the '`.java`' files. In particular the following restrictions are enforced by JML.

- When the same method is declared in more than one file in a refinement sequence, most parts of the method declaration must be identical in all the files. (Two method declarations are considered to be declaring the *same method* if they have the same signature, i.e., same name and static formal parameter types.) However, in addition to the signature of such a method, the return type, the names of the formal parameters, the declared exceptions the method may throw, and the non-JML modifiers `public`, `protected`, `private`, `static`, and `final`, must all match exactly in each such declaration in a refinement chain.
- The `model` modifier must appear in all declarations of a given method or it must appear in none of them. It is not permitted to implement a model method with a non-model method or to refine a non-model method with a model method. Use a `spec_public` or `spec_protected` method if you want to use that method in a specification.

- Some of the JML method modifiers do not always have to match in all declarations of the same method in a refinement chain. One may add `pure`, `non_null`, `spec_public`, or `spec_protected` to any of the declarations for a method in any file. However, if `pure` is added to a method specification, then all subsequent declarations of that method in a refinement sequence must also be declared `pure`. Also, it is, of course, not permitted to add `spec_protected` to a method that has been declared `public` or `spec_public` in other declarations. One can add `non_null` to any formal parameter in any file, although good style suggests that all of these annotations appear on one declaration of that method.
- The specification of a refining method declaration must start with the JML keyword `also`; if it does not an error message is issued. A *refining method declaration* is a declaration that overrides a superclass method or refines the specification of the same method in a refinement chain. In JML, method specifications are inherited by subclasses and in refinement chains. The `also` keyword indicates that the current specification is refining the specification inherited either from the superclass or from the previous declaration of the method in a refinement sequence. Therefore, it is an error if the specification of a non-refining method begins with `also` (unless it overrides an inherited method).
- If a non-model method has a body, then the body can only appear in a '`.java`' file; an error message is issued if the body of a non-model method appears in a file with any other suffix. Furthermore, the body of a model method may only appear in one file of a refinement sequence. This means that each method of each class can have at most one method body.
- When the same field is declared in more than one file in a refinement sequence, then the signature of each such declaration must be identical in all the files. (Two field declarations are considered to be declaring the *same field* if they have the same name.) The signature of such a field, including its type, the non-JML modifiers `public`, `protected`, `private`, `static`, and `final`, must all match exactly in each such declaration.
- All declarations of a given field must either use the modifier `model` or not. It is not permitted to implement a model field with a non-model field or vice versa. Use a `spec_public` or `spec_protected` field if you want to use the same name. The same comment holds for `ghost` fields.
- Some of the JML field modifiers do not always have to match in all declarations of the same field in a refinement chain. One may add `non_null`, `spec_public`, or `spec_protected` to any of of the declarations for a field in any file. However, it is of course not permitted to add `spec_protected` to a field that has been declared public in other declarations.
- Initializers are not allowed in all field declarations. A non-model field can have an initializer expression but it can only appear in a '`.java`' file because this is where a compiler expects to find it.

  Fields declared using the `ghost` modifier can have an initializer expression in any file, but they may have at most one initializer expression in all the files.

  Model fields cannot have an initializer expression because there is no storage associated with such fields. Use the `initially` clause to specify the initial state of model fields (although the initial state is usually determined from the represents clause).

- Any number of *jml-var-assertion*'s can be declared for any field declaration and these are all conjoined. For example, if a variable `int count` is declared and there are two `initially` clauses, in the same or different files, then these initially clause predicates are conjoined; that is, both must be satisfied initially.
- An initializer block or a static initializer block (with code) may only appear in a '`.java`' file. One can write annotations to specify the effects of such initializers in JML annotations in other files, using the keywords `initializer` and `static_initializer`.

JML uses specification inheritance to impose the specifications of supertypes on their subtypes [Dhara-Leavens96] to support the concept of behavioral subtyping [America87] [Leavens90] [Leavens91] [Leavens-Weihl90] [Leavens-Weihl95] [Liskov-Wing94]. JML also supports a notion of weak behavioral subtyping [Dhara-Leavens94b] [Dhara97].

17.4 Refinement Viewpoints

In refinements, specification inheritance allows the specifier to separate the public, protected, and private specifications into different files. Public specifications give the public behavior and are meant for clients of the class. Protected specifications are meant for programmers creating subclasses and give the protected behavior of the type; they give the behavior of protected methods and fields that are not visible to clients. Similarly, private specifications are meant for implementors of the class and provide the behavior related to private methods and fields of the class; implementors must satisfy the combined public, protected, and private specifications of a method.

[[[Needs work]]]

17.4.1 default constructor refinement

In Java, a default constructor is automatically generated for a class when no constructors are declared in a class. However, in JML, a default constructor is not generated for a class unless the file suffix is '`.java`' (the same constructor is generated as in the Java language). Consider, for example, the following refinement sequence.

```
// ------ file MyClass.jml-refined --------
public class MyClass {
   //@ public model int x;

   /*@ public normal_behavior
     @    ensures x == 0; @*/
   public MyClass();
}

// --------- file MyClass.jml -------------
//@ refine "MyClass.jml-refined";
public class MyClass {
   protected int x_;
   //@           in x;
   //@ protected represents x <- x_;
}

// ---------- file MyClass.java -----------
//@ refine "MyClass.jml";
public class MyClass {
   protected int x_;
   public MyClass() { x_ = 0; }
}

// ---------------------------------------
```

In the protected specification declared in 'MyClass.jml', no constructor is defined. If JML were to generate a default constructor for this class declaration, then the public constructor defined earlier in the refinement chain, in 'MyClass.jml-refined', could have a visibility modifier that conflicts with the one automatically generated for the protected specification (the visibility modifier of an automatically generated default constructor depends on other factors including the visibility of the class). Recall that the signature, including the visibility modifier, must match for every method and constructor declared in a refinement chain. To avoid such conflicts, JML does not generate a default constructor unless the file suffix is '.java' (as part of the standard compilation process).

A similar problem can occur when the only constructor is protected or private as in the following example.

```
// ------ file MyClass2.jml-refined --------
public class MyClass2 {
   //@ public model int x;
   //@ public initially x == 0;
}

// --------- file MyClass2.jml -------------
//@ refine "MyClass2.jml-refined";
public class MyClass2 {
   protected int x_;
   //@           in x;

   //@ protected represents x <- x_;

   /*@ protected normal_behavior
     @    ensures x == 0; @*/
   protected MyClass2();
}

// ---------- file MyClass2.java -----------
```

```
//@ refine "MyClass2.jml";
public class MyClass2 {
   protected int x_;
   protected MyClass2() { x_ = 0; }
}

// ----------------------------------------
```

In this example, no constructor is defined for the public specification in '`MyClass2.jml-refined`'. If a default constructor were generated for this class declaration, then the `protected` constructor defined later in the refinement chain, in '`MyClass2.jml`', would have a visibility modifier that conflicts with the one automatically generated and JML would emit an error. Thus JML only generates the default constructor for the executable declaration of a class in the '`.java`' file and only when required by the Java language.

18 MultiJava Extensions to JML

This section describes extensions to JML to support the MultiJava [Clifton-etal00] language. All of these extensions are optional and are only used when an option (or special tool) is used to parse this syntax.

The sections below explain the extensions that MultiJava makes to JML.

18.1 Augmenting Method Declarations

MultiJava has a feature, called "open classes" [Clifton-etal00] or "augmenting methods" that allows methods to be added to an existing class. It has the following syntax, which, in JML, permits method specifications.

multijava-top-level-declaration ::= *multijava-top-level-method*
multijava-top-level-method ::= [*method-specification*]
        modifiers [`method`]
        [*type-spec*] *extending-method-head method-body*
extending-method-head ::= *name* `.` `ident` *formals* [*dims*]
            [*throws-clause*]

This syntax adds a method to the class named by the *name* in the *extending-method-head*.

The method must satisfy the given *method-specification*, if there is one.

18.2 MultiMethods

The other feature in MultiJava is multiple dispatch, which is used to define multimethods. Multiple dispatch is defined using the following syntax.

multijava-param-declaration ::= [*param-modifier*] . . .
        type-spec specializer ident [*dims*]
specializer ::= `@` *type-spec*
    `|` `@@` *value-specializer*
value-specializer ::= *expression*

See the MultiJava paper [Clifton-etal00] for how the use of a *specializer* affects the meaning of method calls.

19 Universe Type System Extensions to JML

Appendix A Grammar Summary

The following is a summary of the context-free grammar for JML. See Chapter 3 [Syntax Notation], page 16, 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 17) may intervene between the characters of a token.

A.1 Lexical Conventions

microsyntax ::= *lexeme* [*lexeme*] . . .
lexeme ::= *white-space* | *lexical-pragma* | *comment*
        | *annotation-marker* | *doc-comment* | *token*
token ::= *ident* | *keyword* | *special-symbol*
        | *java-literal* | *informal-description*
white-space ::= *non-nl-white-space* | *end-of-line*
non-nl-white-space ::= a blank, tab, or formfeed character
end-of-line ::= *newline* | *carriage-return*
        | *carriage-return newline*
newline ::= a newline character
carriage-return ::= a carriage return character
lexical-pragma ::= *nowarn-pragma*
nowarn-pragma ::= `nowarn` [*spaces*] [*nowarn-label-list*] `;`
spaces ::= *non-nl-white-space* [*non-nl-white-space*] . . .
nowarn-label-list ::= *nowarn-label* [*spaces*]
            [`,` [*spaces*] *nowarn-label* [*spaces*]] . . .
nowarn-label ::= *letter* [*letter*] . . .
comment ::= *C-style-comment* | *C++-style-comment*
C-style-comment ::= `/*` [*C-style-body*] *C-style-end*
C-style-body ::= *non-at-plus-star* [*non-stars-slash*] . . .
        | `+` *non-at* [*non-stars-slash*] . . .
        | *stars-non-slash* [*non-stars-slash*] . . .
non-stars-slash ::= *non-star*
        | *stars-non-slash*
stars-non-slash ::= `*` [`*`] . . . *non-star-slash*
non-at-plus-star ::= any character except `@`, `+`, or `*`
non-at ::= any character except `@`
non-star ::= any character except `*`
non-slash ::= any character except `/`
non-star-slash ::= any character except `*` or `/`
C-style-end ::= [`*`] . . . `*/`
C++-style-comment ::= `//` [`+`] *end-of-line*
        | `//` *non-at-plus-end-of-line* [*non-end-of-line*] . . . *end-of-line*
        | `//+` *non-at-end-of-line* [*non-end-of-line*] . . . *end-of-line*
non-end-of-line ::= any character except a newline or carriage return
non-at-plus-end-of-line ::= any character except `@`, `+`, newline, or carriage return

non-at-end-of-line ::= any character except `@`, newline, or carriage return
annotation-marker ::= `//@` [`@`] . . . `|` `//+@` [`@`] . . .
    `|` `/*@` [`@`] . . . `|` `/*+@` [`@`] . . . `|` [`@`] . . . `@+*/` `|` [`@`] . . . `*/`
ignored-at-in-annotation ::= `@`
doc-comment ::= `/**` [`*`] . . . *doc-comment-body* `*/`
doc-comment-ignored ::= *doc-comment*
doc-comment-body ::= [*description*] . . .
    [*tagged-paragraph*] . . .
    [*jml-specs*]
description ::= *doc-non-empty-textline*
tagged-paragraph ::= *paragraph-tag* [*doc-non-nl-ws*] . . .
    [*doc-atsign*] . . . [*description*] . . .
jml-specs ::= *jml-tag* [*method-specification*] *end-jml-tag*
    [*jml-tag* [*method-specification*] *end-jml-tag*] . . .
paragraph-tag ::= `@author` `|` `@deprecated` `|` `@exception`
    `|` `@param` `|` `@return` `|` `@see`
    `|` `@serial` `|` `@serialdata` `|` `@serialfield`
    `|` `@since` `|` `@throws` `|` `@version`
    `|` `@` *letter* [*letter*] . . .
doc-atsign ::= `@`
doc-nl-ws ::= *end-of-line*
    [*doc-non-nl-ws*] . . . [`*` [`*`] . . . [*doc-non-nl-ws*] . . .]
doc-non-nl-ws ::= *non-nl-white-space*
doc-non-empty-textline ::= *non-at-end-of-line* [*non-end-of-line*] . . .
jml-tag ::= `<jml>` `|` `<JML>` `|` `<esc>` `|` `<ESC>`
end-jml-tag ::= `</jml>` `|` `</JML>` `|` `</esc>` `|` `</ESC>`
ident ::= *letter* [*letter-or-digit*] . . .
letter ::= `_`, `$`, `a` through `z`, or `A` through `Z`
digit ::= `0` `|` `1` `|` `2` `|` `3` `|` `4` `|` `5` `|` `6` `|` `7` `|` `8` `|` `9`
letter-or-digit ::= *letter* `|` *digit*
keyword ::= *java-reserved-word*
    `|` *jml-predicate-keyword* `|` *jml-keyword*
java-reserved-word ::= `abstract` `|` `assert`
    `|` `boolean` `|` `break` `|` `byte`
    `|` `case` `|` `catch` `|` `char`
    `|` `class` `|` `const` `|` `continue`
    `|` `default` `|` `do` `|` `double`
    `|` `else` `|` `extends` `|` `false`
    `|` `final` `|` `finally` `|` `float`
    `|` `for` `|` `goto` `|` `if`
    `|` `implements` `|` `import` `|` `instanceof`
    `|` `int` `|` `interface` `|` `long`
    `|` `native` `|` `new` `|` `null`
    `|` `package` `|` `private` `|` `protected`
    `|` `public` `|` `return` `|` `short`
    `|` `static` `|` `strictfp` `|` `super`
    `|` `switch` `|` `synchronized` `|` `this`

| `throw` | `throws` | `transient`
| `true` | `try` | `void`
| `volatile` | `while`
| *multijava-reserved* *// When the MultiJava option is on*
| *java-universe-reserved* *// When the Universe option is on*
multijava-reserved ::= `resend`
java-universe-reserved ::= `peer` | `pure`
| `readonly` | `rep`
jml-predicate-keyword ::= `\TYPE`
| `\bigint` | `\bigint_math` | `\duration`
| `\elemtype` | `\everything` | `\exists`
| `\fields_of` | `\forall` | `\fresh`
| `\into` | `\invariant_for` | `\is_initialized`
| `\java_math` | `\lblneg` | `\lblpos`
| `\lockset` | `\max` | `\min`
| `\nonnullelements` | `\not_assigned`
| `\not_modified` | `\not_specified`
| `\nothing` | `\nowarn` | `\nowarn_op`
| `\num_of` | `\old` | `\other`
| `\private_data` | `\product` | `\reach`
| `\real` | `\result` | `\safe_math`
| `\space` | `\such_that` | `\sum`
| `\typeof` | `\type` | `\warn_op`
| `\warn` | `\working_space`
| *jml-universe-pkeyword*
jml-universe-pkeyword ::= `\peer` | `\readonly` | `\rep`
jml-keyword ::= `abrupt_behavior`
| `accessible` | `accessible_redundantly`
| `also` | `assert_redundantly`
| `assignable` | `assignable_redundantly`
| `assume` | `assume_redundantly` | `axiom`
| `behavior` | `breaks` | `breaks_redundantly`
| `callable` | `callable_redundantly`
| `captures` | `captures_redundantly`
| `choose` | `choose_if`
| `code_bigint_math` | `code_contract`
| `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_example`
| `exsures` | `exsures_redundantly`

```
          | 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_example
          | nowarn | old | or
          | post | post_redundantly
          | pre | pre_redundantly
          | pure | readable | refine
          | represents | represents_redundantly
          | requires | requires_redundantly
          | returns | returns_redundantly
          | set | signals | signals_redundantly
          | spec_bigint_math | spec_java_math
          | spec_protected | spec_public | spec_safe_math
          | static_initializer | uninitialized
          | unreachable | weakly
          | when | when_redundantly
          | working_space | working_space_redundantly
          | writable
          | jml-universe-keyword
jml-universe-keyword ::= peer | readonly | rep
special-symbol ::= java-special-symbol | jml-special-symbol
java-special-symbol ::= java-separator | java-operator
java-separator ::= ( | ) | { | } | ‘[’ | ‘]’ | ; | , | .
          | multijava-separator   // When the MultiJava option is on
multijava-separator ::= @ | @@
java-operator ::= = | < | > | ! | ~ | ? | :
          | == | <= | >= | != | && | ‘||’ | ++ | --
          | + | - | * | / | & | ‘|’ | ^ | % | << | >> | >>>
          | += | -= | *= | /= | &= | ‘|=’ | ^= | %=
          | <<= | >>= | >>>=
jml-special-symbol ::= ==> | <== | <==> | <=!=>
          | -> | <- | <: | .. | ‘{|’ | ‘|}’
java-literal ::= integer-literal
          | floating-point-literal | boolean-literal
          | character-literal | string-literal | null-literal
```

```
integer-literal ::= decimal-integer-literal
        | hex-integer-literal | octal-integer-literal
decimal-integer-literal ::= decimal-numeral [ integer-type-suffix ]
decimal-numeral ::= 0 | non-zero-digit [ digits ]
digits ::= digit [ digit ] . . .
digit ::= 0 | non-zero-digit
non-zero-digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
integer-type-suffix ::= l | L
hex-integer-literal ::= hex-numeral  [ integer-type-suffix ]
hex-numeral ::= 0x hex-digit [ hex-digit ] . . .
        | 0X hex-digit  [ hex-digit ] . . .
hex-digit ::= digit | a | b | c | d | e | f
        | A | B | C | D | E | F
octal-integer-literal ::= octal-numeral  [ integer-type-suffix ]
octal-numeral ::= 0 octal-digit [ octal-digit ] . . .
octal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
floating-point-literal ::= digits . [ digits ]
          [ exponent-part ] [ float-type-suffix ]
        | . digits [ exponent-part ] [ float-type-suffix ]
        | digits exponent-part [ float-type-suffix ]
        | digits [ exponent-part ] float-type-suffix
exponent-part ::= exponent-indicator signed-integer
exponent-indicator ::= e | E
signed-integer ::= [ sign ] digits
sign ::= + | -
float-type-suffix ::= f | F | d | D
boolean-literal ::= true | false
character-literal ::= ' single-character ' | ' escape-sequence '
single-character ::= any character except ', \, carriage return, or newline
escape-sequence ::= \b    // backspace
         | \t             // tab
         | \n             // newline
         | \r             // carriage return
         | \'             // single quote
         | \"             // double quote
         | \\             // backslash
         | octal-escape
         | unicode-escape
octal-escape ::= \ octal-digit [ octal-digit ]
         | \ zero-to-three octal-digit octal-digit
zero-to-three ::= 0 | 1 | 2 | 3
unicode-escape ::= \u hex-digit hex-digit hex-digit hex-digit
string-literal ::= " [ string-character ] . . . "
string-character ::= escape-sequence
         | any character except ", \, carriage return, or newline
null-literal ::= null
informal-description ::= (* non-stars-close [ non-stars-close ] . . . *)
```

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

A.2 Compilation Units

compilation-unit ::= [*package-definition*]
  [*refine-prefix*]
  [*import-definition*] . . .
  [*top-level-definition*] . . .
top-level-definition ::= *type-definition*
  | *multijava-top-level-declaration* *// When parsing MultiJava*
package-definition ::= `package` *name* `;`
name ::= *ident* [`.` *ident*] . . .
import-definition ::= [`model`] `import` *name-star* `;`
name-star ::= *ident* [`.` *ident*] . . . [`.` `*`]

A.3 Type Definitions

type-definition ::= *class-definition*
  | *interface-definition*
  | `;`
class-definition ::= [*doc-comment*] *modifiers* `class` *ident*
  [*class-extends-clause*] [*implements-clause*]
  class-block
class-block ::= `{` [*field*] . . . `}`
interface-definition ::= [*doc-comment*] *modifiers* `interface` *ident*
  [*interface-extends*]
  class-block
class-extends-clause ::= [`extends` *name* [`weakly`]]
implements-clause ::= `implements` *name-weakly-list*
name-weakly-list ::= *name* [`weakly`] [`,` *name* [`weakly`]] . . .
interface-extends ::= `extends` *name-weakly-list*
modifiers ::= [*modifier*] . . .
modifier ::= `public` | `protected` | `private`
  | `abstract` | `static` |
  | `final` | `synchronized`
  | `transient` | `volatile`
  | `native` | `strictfp`
  | `const` *// reserved but not used in Java*
  | *jml-modifier*
jml-modifier ::= `spec_public` | `spec_protected`
  | `model` | `ghost` | `pure`

```
        | instance | helper
        | uninitialized
        | spec_java_math | spec_safe_math | spec_bigint_math
        | code_java_math | code_safe_math | code_bigint_math
        | non_null
```

A.4 Class and Interface Member Declarations

```
field ::= member-decl
      | jml-declaration
      | class-initializer-decl
      | ;
member-decl ::= method-decl
      | variable-definition
      | class-definition
      | interface-definition
method-decl ::= [ doc-comment ] . . .
          method-specification
          modifiers method-or-constructor-keyword
          [ type-spec ] method-head
          method-body
      | [ doc-comment ] . . .
        modifiers method-or-constructor-keyword
        [ type-spec ] method-head
        [ method-specification ]
        method-body
method-or-constructor-keyword ::= method | constructor
method-head ::= ident formals [ dims ] [ throws-clause ]
method-body ::= compound-statement | ;
throws-clause ::= throws name [ , name ] . . .
formals ::= ( [ param-declaration-list ] )
param-declaration-list ::= param-declaration
                    [ , param-declaration ] . . .
param-declaration ::= [ param-modifier ] . . . type-spec ident [ dims ]
      | multijava-param-declaration   // When MultiJava parsing is on
param-modifier ::= final | non_null
variable-definition ::= [ doc-comment ] . . . modifiers variable-decls
variable-decls ::= [ field ] type-spec variable-declarators ;
              [ jml-data-group-clause ] . . .
variable-declarators ::= variable-declarator
                    [ , variable-declarator ] . . .
variable-declarator ::= ident [ dims ] [ = initializer ]
initializer ::= expression | array-initializer
array-initializer ::= { [ initializer-list ] }
initializer-list ::= initializer [ , initializer ] . . . [ , ]
```

type-spec ::= *type* [*dims*] | `\TYPE` [*dims*]
type ::= *reference-type* | *built-in-type*
reference-type ::= *name*
dims ::= ‘`[`’ ‘`]`’ [‘`[`’ ‘`]`’] . . .
class-initializer-decl ::= [*method-specification*]
      [`static`] *compound-statement*
   | *method-specification* `static_initializer`
   | *method-specification* `initializer`

A.5 Type Specifications

jml-declaration ::= *modifiers invariant*
   | *modifiers history-constraint*
   | *modifiers represents-decl*
   | *modifiers initially-clause*
   | *modifiers monitors-for-clause*
   | *modifiers readable-if-clause*
   | *modifiers writable-if-clause*
   | `axiom` *predicate* `;`
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 ::= *method-ref-start* [`.` *method-ref-rest*] . . .
   | `new` *reference-type*
method-ref-start ::= `super` | `this` | *ident* | `\other`
method-ref-rest ::= `this` | *ident* | `\other`
param-disambig-list ::= *param-disambig* [`,` *param-disambig*] . . .
param-disambig ::= *type-spec* [*ident* [*dims*]]
represents-decl ::= *represents-keyword store-ref-expression*
      l-arrow-or-eq spec-expression `;`
   | *represents-keyword store-ref-expression* `\such_that`
    predicate `;`
represents-keyword ::= `represents` | `represents_redundantly`
l-arrow-or-eq ::= `<-` | `=`
initially-clause ::= `initially` *predicate* `;`
readable-if-clause ::= `readable` *ident* `if` *predicate* `;`
writable-if-clause ::= `writable` *ident* `if` *predicate* `;`
monitors-for-clause ::= `monitors_for` *ident*
      l-arrow-or-eq spec-expression-list `;`

A.6 Method Specifications

method-specification ::= *specification* | *extending-specification*
extending-specification ::= `also` *specification*
specification ::= *spec-case-seq* [*redundant-spec*]
  | *redundant-spec*
spec-case-seq ::= *spec-case* [`also` *spec-case*] . . .
spec-case ::= *lightweight-spec-case* | *heavyweight-spec-case*
  | *model-program* | *code-contract-spec*
privacy ::= `public` | `protected` | `private`
lightweight-spec-case ::= *generic-spec-case*
generic-spec-case ::= [*spec-var-decls*]
  spec-header
  [*generic-spec-body*]
  | [*spec-var-decls*]
  generic-spec-body
generic-spec-body ::= *simple-spec-body*
  | `{|` *generic-spec-case-seq* `|}`
generic-spec-case-seq ::= *generic-spec-case*
  [`also` *generic-spec-case*] . . .
spec-header ::= *requires-clause* [`{`varrequires-clause] . . .
simple-spec-body ::= *simple-spec-body-clause*
  [*simple-spec-body-clause*] . . .
simple-spec-body-clause ::= *diverges-clause*
  | *assignable-clause*
  | *when-clause* | *working-space-clause*
  | *duration-clause* | *ensures-clause* | *signals-clause*
heavyweight-spec-case ::= *behavior-spec-case*
  | *exceptional-behavior-spec-case*
  | *normal-behavior-spec-case*
behavior-spec-case ::= [*privacy*] `behavior`
  generic-spec-case
normal-behavior-spec-case ::= [*privacy*] `normal_behavior`
  normal-spec-case
normal-spec-case ::= [*spec-var-decls*] [*spec-header*]
  normal-spec-body
  | [*spec-var-decls*]
  normal-spec-body
normal-spec-body ::= *normal-spec-clause*
  [*normal-spec-clause*] . . .
  | `{|` *normal-spec-case-seq* `|}`
normal-spec-clause ::= *diverges-clause*
  | *assignable-clause*
  | *when-clause* | *working-space-clause*
  | *duration-clause* | *ensures-clause*
normal-spec-case-seq ::= *normal-spec-case*

[`also` *normal-spec-case*] . . .
exceptional-behavior-spec-case ::= [*privacy*] `exceptional_behavior`
exceptional-spec-case
exceptional-spec-case ::= [*spec-var-decls*] [*spec-header*]
exceptional-spec-body
| [*spec-var-decls*]
exceptional-spec-body
exceptional-spec-body ::= *exceptional-spec-clause*
[*exceptional-spec-clause*] . . .
| `{|` *exceptional-spec-case-seq* `|}`
exceptional-spec-clause ::= *diverges-clause*
| *assignable-clause*
| *when-clause* | *working-space-clause*
| *duration-clause* | *signals-clause*
exceptional-spec-case-seq ::= *exceptional-spec-case*
[`also` *exceptional-spec-case*] . . .
spec-var-decls ::= *forall-var-decls* [*old-var-decls*]
| *old-var-decls*
forall-var-decls ::= *forall-var-decl* [*forall-var-decl*] . . .
forall-var-decl ::= `forall` *quantified-var-decl* `;`
old-var-decls ::= `old-var-decl` [*old-var-decl*] . . .
old-var-decl ::= `old` *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`
| `exsures` | `exsures_redundantly`
diverges-clause ::= *diverges-keyword pred-or-not* `;`
diverges-keyword ::= `diverges` | `diverges_redundantly`
when-clause ::= *when-keyword pred-or-not* `;`
when-keyword ::= `when` | `when_redundantly`
assignable-clause ::= *assignable-keyword store-ref-list* `;`
assignable-keyword ::= `assignable` | `assignable_redundantly`
| `modifiable` | `modifiable_redundantly`
| `modifies` | `modifies_redundantly`
working-space-clause ::= *working-space-keyword* `\not_specified` `;`
| *working-space-keyword spec-expression* [`if` *predicate*] `;`
working-space-keyword ::= `working_space` | `working_space_redundantly`
duration-clause ::= *duration-keyword* `\not_specified` `;`
| *duration-keyword spec-expression* [`if` *predicate*] `;`
duration-keyword ::= `duration` | `duration_redundantly`

A.7 Frame Conditions and Data Groups

```
jml-data-group-clause ::= in-group-clause | maps-into-clause
in-group-clause ::= in-keyword group-list ;
in-keyword ::= in | in_redundantly
group-list ::= group-name [ , group-name ] . . .
group-name ::= [ group-name-prefix ] ident
group-name-prefix ::= super . | this .
maps-into-clause ::= maps-keyword member-field-ref \into group-list ;
maps-keyword ::= maps | maps_redundantly
member-field-ref ::= ident . maps-member-ref-expr
        | maps-array-ref-expr [ . maps-member-ref-expr ]
maps-member-ref-expr ::= ident | *
maps-array-ref-expr ::= ident maps-spec-array-dim
                        [ maps-spec-array-dim ] . . .
maps-spec-array-dim ::= '[' spec-array-ref-expr ']'
```

A.8 Predicates and Specification Expressions

```
predicate ::= spec-expression
spec-expression-list ::= spec-expression
                         [ , spec-expression ] . . .
spec-expression ::= expression
jml-primary ::= \result
        | \old ( spec-expression )
        | \not_assigned ( store-ref-list )
        | \not_modified ( store-ref-list )
        | \fresh ( spec-expression-list )
        | \reach ( spec-expression )
        | \duration ( expression )
        | \space ( spec-expression )
        | \max ( spec-expression )
        | \working_space ( expression )
        | informal-description
        | \nonnullelements ( spec-expression )
        | \typeof ( spec-expression )
        | \elemtype ( spec-expression )
        | \type ( type )
        | \lockset
        | \is_initialized ( reference-type )
        | \invariant_for ( spec-expression )
        | ( \lblneg ident spec-expression )
        | ( \lblpos ident spec-expression )
        | spec-quantified-expr
```

set-comprehension ::= `{` *type-spec*
  quantified-var-declarator ‘`|`’
  set-comprehension-pred `}`
set-comprehension-pred ::=
  postfix-expr `. has (` *ident* `) &&`
  predicate
spec-quantified-expr ::= `(` *quantifier quantified-var-decls* `;`
  [[*predicate*] `;`]
  spec-expression `)`
quantifier ::= `\forall` | `\exists`
  | `\max` | `\min`
  | `\num_of` | `\product` | `\sum`
quantified-var-decls ::= *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*
expression-list ::= *expression* [`,` *expression*] . . .
expression ::= *assignment-expr*
assignment-expr ::= *conditional-expr*
  [*assignment-op assignment-expr*]
assignment-op ::= `=` | `+=` | `-=` | `*=` | `/=` | `%=` | `>>=`
  | `>>>=` | `<<=` | `&=` | ‘`|=`’ | `^=`
conditional-expr ::= *equivalence-expr*
  [`?` *conditional-expr* `:` *conditional-expr*]
equivalence-expr ::= *implies-expr*
  [*equivalence-op implies-expr*] . . .
equivalence-op ::= `<==>` | `<=!=>`
implies-expr ::= *logical-or-expr*
  [`==>` *implies-non-backward-expr*]
  | *logical-or-expr* `<==` *logical-or-expr*
  [`<==` *logical-or-expr*] . . .
implies-non-backward-expr ::= *logical-or-expr*
  [`==>` *implies-non-backward-expr*]
logical-or-expr ::= *logical-and-expr* [‘`||`’ *logical-and-expr*] . . .
logical-and-expr ::= *inclusive-or-expr* [`&&` *inclusive-or-expr*] . . .
inclusive-or-expr ::= *exclusive-or-expr* [‘`|`’ *exclusive-or-expr*] . . .
exclusive-or-expr ::= *and-expr* [`^` *and-expr*] . . .
and-expr ::= *equality-expr* [`&` *equality-expr*] . . .
equality-expr ::= *relational-expr* [`==` *relational-expr*] . . .
  | *relational-expr* [`!=` *relational-expr*] . . .

relational-expr ::= *shift-expr* `<` *shift-expr*
  | *shift-expr* `>` *shift-expr*
  | *shift-expr* `<=` *shift-expr*
  | *shift-expr* `>=` *shift-expr*
  | *shift-expr* `<:` *shift-expr*
  | *shift-expr* [`instanceof` *type-spec*]
shift-expr ::= *additive-expr* [*shift-op additive-expr*] . . .
shift-op ::= `<<` | `>>` | `>>>`
additive-expr ::= *mult-expr* [*additive-op mult-expr*] . . .
additive-op ::= `+` | `-`
mult-expr ::= *unary-expr* [*mult-op unary-expr*] . . .
mult-op ::= `*` | `/` | `%`
unary-expr ::= `(` *type-spec* `)` *unary-expr*
  | `++` *unary-expr*
  | `--` *unary-expr*
  | `+` *unary-expr*
  | `-` *unary-expr*
  | *unary-expr-not-plus-minus*
unary-expr-not-plus-minus ::= `~` *unary-expr*
  | `!` *unary-expr*
  | `(` *builtinType* `)` *unary-expr*
  | `(` *reference-type* `)` *unary-expr-not-plus-minus*
  | *postfix-expr*
postfix-expr ::= *primary-expr* [*primary-suffix*] . . . [`++`]
  | *primary-expr* [*primary-suffix*] . . . [`--`]
  | *builtinType* [‘`[`’ ‘`]`’] . . . `. 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*
  | *informal-description*
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
store-ref-list ::= store-ref [ , store-ref ] ...
store-ref ::= store-ref-expression
        | informal-description
        | store-ref-keyword
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 | \private_data
```

A.9 JML primitive types

A.10 Statements and Annotation Statements

```
compound-statement ::= { statement [ statement ] ... }
statement ::= compound-statement
        | local-declaration ;
        | ident : statement
        | expression ;
        | if ( expression )
          statement [ else statement ]
        | [ loop-invariant ] ...
          [ variant-function ] ...
          [ ident ] : loop-stmt
        | break [ ident ] ;
        | continue [ ident ] ;
        | return [ expression ] ;
        | switch-statement
        | try-block
        | throw expression ;
        | synchronized ( expression ) statement
        | ;
        | assert-statement
        | hence-by-statement
        | assert-redundantly-statement
```

| *assume-statement*
| *set-statement*
| *unreachable-statement*
| *debug-statement*
| *model-prog-statement // only allowed in model programs*
loop-stmt ::= `while` `(` *expression* `)` *statement*
| `do` *statement* `while` `(` *expression* `)` `;`
| `for` `(` [*for-init*] `;` [*expression*] `;` [*expression-list*] `)`
statement
for-init ::= *local-declaration* | *expression-list*
local-declaration ::= *local-modifiers variable-decls*
local-modifiers ::= [*local-modifier*] . . .
local-modifier ::= `ghost` | `final` | `non_null`
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*
assert-statement ::= `assert` *expression* [`:` *expression*] `;`
hence-by-statement ::= *hence-by-keyword predicate* `;`
hence-by-keyword ::= `hence_by` | `hence_by_redundantly`
assert-redundantly-statement ::= `assert_redundantly` *predicate*
[`:` *expression*] `;`
assume-statement ::= *assume-keyword predicate*
[`:` *expression*] `;`
assume-keyword ::= `assume` | `assume_redundantly`
set-statement ::= `set` *assignment-expr* `;`
unreachable-statement ::= `unreachable` `;`
debug-statement ::= `debug` *expression* `;`
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`

A.11 Redundancy

redundant-spec ::= *implications* [*examples*] | *examples*
examples ::= `for_example` *example* [`also` *example*] . . .
implications ::= `implies_that` *spec-case-seq*

example ::= [[*privacy*] `example`]
    [*spec-var-decls*]
    [*spec-header*]
    simple-spec-body
  | [*privacy*] `exceptional_example`
    [*spec-var-decls*]
    spec-header
    [*exceptional-example-body*]
  | [*privacy*] `exceptional_example`
    [*spec-var-decls*]
    exceptional-example-body
  | [*privacy*] `normal_example`
    [*spec-var-decls*]
    spec-header
    [*normal-example-body*]
  | [*privacy*] `normal_example`
    [*spec-var-decls*]
    normal-example-body
exceptional-example-body ::= *exceptional-spec-clause*
    [*exceptional-spec-clause*] . . .
normal-example-body ::= *normal-spec-clause*
    [*normal-spec-clause*] . . .

A.12 Model Programs

model-program ::= [*privacy*] `model_program`
    jml-compound-statement
jml-compound-statement ::= *compound-statement*
jml-statement ::= *statement*
model-prog-statement ::= *nondeterministic-choice*
  | *nondeterministic-if*
  | *spec-statement*
  | *invariant*
nondeterministic-choice ::= `choose` *alternative-statements*
alternative-statements ::= *jml-compound-statement*
    [`or` *jml-compound-statement*] . . .
nondeterministic-if ::= `choose_if` *guarded-statements*
    [`else` *jml-compound-statement*]
guarded-statements ::= *guarded-statement*
    [`or` *guarded-statement*] . . .
guarded-statement ::= `{`
    assume-statement
    jml-statement [*jml-statement*] . . . `}`
spec-statement ::= [*privacy*] `behavior`
    generic-spec-statement-case

| [*privacy*] `exceptional_behavior`
exceptional-spec-case
| [*privacy*] `normal_behavior`
normal-spec-case
| [*privacy*] `abrupt_behavior`
abrupt-spec-case
generic-spec-statement-case ::= [*spec-var-decls*]
generic-spec-statement-body
| [*spec-var-decls*]
spec-header
[*generic-spec-statement-body*]
generic-spec-statement-body ::= *simple-spec-statement-body*
| `{|` *generic-spec-statement-case-seq* `|}`
generic-spec-statement-body-seq ::= *generic-spec-statement-case*
[`also` *generic-spec-statement-case*] . . .
simple-spec-statement-body ::= *simple-spec-statement-clause*
[*simple-spec-statement-clause*] . . .
simple-spec-statement-clause ::= *diverges-clause*
| *assignable-clause*
| *when-clause* | *working-space-clause* | *duration-clause*
| *ensures-clause* | *signals-clause*
| *continues-clause* | *breaks-clause* | *returns-clause*
abrupt-spec-case ::= [*spec-var-decls*]
spec-header
[*abrupt-spec-body*]
| [*spec-var-decls*]
abrupt-spec-body
abrupt-spec-body ::= *abrupt-spec-clause*
[*abrupt-spec-clause*] . . .
| `{|` *abrupt-spec-case-seq* `|}`
abrupt-spec-clause ::= *diverges-clause*
| *assignable-clause*
| *when-clause* | *working-space-clause* | *duration-clause*
| *continues-clause* | *breaks-clause* | *returns-clause*
abrupt-spec-case-seq ::= *abrupt-spec-case*
[`also` *abrupt-spec-case*] . . .
continues-clause ::= *continues-keyword* [*target-label*]
[*pred-or-not*] `;`
continues-keyword ::= `continues` | `continues_redundantly`
target-label ::= `->` `(` *ident* `)`
breaks-clause ::= *breaks-keyword* [*target-label*]
[*pred-or-not*] `;`
breaks-keyword ::= `breaks` | `breaks_redundantly`
returns-clause ::= *returns-keyword* [*pred-or-not*] `;`
returns-keyword ::= `returns` | `returns_redundantly`

A.13 Specification for Subtypes

code-contract-spec ::= `code_contract` *code-contract-clause*
  [*code-contract-clause*] . . .
code-contract-clause ::= *accessible-clause*
  | *callable-clause*
  | *measured-clause*
  | *captures-clause*
accessible-clause ::= *accessible-keyword object-store-ref-list* `;`
accessible-keyword ::= `accessible` | `accessible_redundantly`
object-store-ref-list ::= *object-store-ref* [*object-store-ref* ,] . . .
object-store-ref ::= *store-ref* | *other-ref*
other-ref ::= `\other` [*store-ref-name-suffix*] . . .
callable-clause ::= *callable-keyword callable-methods-list* `;`
callable-keyword ::= `callable` | `callable_redundantly`
callable-methods-list ::= *method-name-list* | *store-ref-keyword*
measured-clause ::= *measured-by-keyword* `\not_specified` `;`
  | *measured-by-keyword spec-expression* [`if` *predicate*] `;`
measured-by-keyword ::= `measured_by` | `measured_by_redundantly`
captures-clause ::= *captures-keyword store-ref-list* `;`
captures-keyword ::= `captures` | `captures_redundantly`

A.14 Refinement

refine-prefix ::= `refine` *string-literal* `;`

A.15 MultiJava Extensions to JML

multijava-top-level-declaration ::= *multijava-top-level-method*
multijava-top-level-method ::= [*method-specification*]
  modifiers [`method`]
  [*type-spec*] *extending-method-head method-body*
extending-method-head ::= *name* `.` `ident` *formals* [*dims*]
  [*throws-clause*]
multijava-param-declaration ::= [*param-modifier*] . . .
  type-spec specializer ident [*dims*]
specializer ::= `@` *type-spec*
  | `@@` *value-specializer*
value-specializer ::= *expression*

A.16 Universe Type System Extensions to JML

conditional-store-ref-list ::= *conditional-store-ref*
    [`,` *conditional-store-ref*] . . .
conditional-store-ref ::= *store-ref* [`if` *predicate*]
    `|` *other-ref* [`if` *predicate*]

Appendix B Modifier Summary

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

Grammatical construct	Java modifiers	JML modifiers
All modifiers	`public`	`spec_public`
	`protected`	`spec_protected`
	`private`	`model` `ghost`
	`abstract`	`pure` `instance`
	`static` `final`	`helper`
	`synchronized`	`non_null`
	`transient`	
	`volatile`	
	`native`	
	`strictfp`	
Class declaration	`public` `final`	`pure` `model`
	`abstract`	
	`strictfp`	
Interface declaration	`public`	`pure` `model`
	`strictfp`	
Nested Class declaration	`public`	`spec_public`
	`protected`	`spec_protected`
	`private`	`model` `pure`
	`static` `final`	
	`abstract`	
	`strictfp`	
Nested interface declaration	`public`	`spec_public`
	`protected`	`spec_protected`
	`private`	`model` `pure`
	`static`	
	`strictfp`	
Local Class (and local model class) declaration	`final`	`pure` `model`
	`abstract`	
	`strictfp`	
Type specification (e.g. invariant)	`public`	-
	`protected`	
	`private`	
	`static`	

Field declaration	`public` `protected` `private` `final` `volatile` `transient` `static`	`spec_public` `spec_protected` `non_null` `instance` `monitored`
Ghost Field declaration	`public` `protected` `private` `static` `final`	`non_null` `instance` `monitored`
Model Field declaration	`public` `protected` `private` `static`	`non_null` `instance`
Method declaration	`public` `protected` `private` `abstract` `final` `static` `synchronized` `native` `strictfp`	`spec_public` `spec_protected` `pure` `non_null` `helper`
Constructor declaration	`public` `protected` `private`	`spec_public` `spec_protected` `helper` `pure`
Model method	`public` `protected` `private` `abstract` `static` `final` `synchronized` `strictfp`	`pure` `non_null` `helper`
Model constructor	`public` `protected` `private`	`pure` `helper`
Java initialization block	`static`	-

JML initializer and static_initializer annotation	-	-
Formal parameter	`final`	`non_null`
Local variable and local ghost variable declaration	`final`	`ghost non_null uninitialized`

Note that within interfaces, fields are implicitly public, static and final [Gosling-etal00]. In an interface, ghost and model fields are implicitly public and static, though they may be declared as `instance` fields, which makes them not static.

Appendix C Type Checking Summary

[[[Hope to generate this automatically]]]

Appendix D Verification Logic Summary

[[[Hope to generate this automatically]]]

Appendix E Differences

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

E.1 Differences Between JML and Other Tools

ESC/Java [Leino-Nelson-Saxe00] and JML share a common syntax; this is even more true of ESC/Java2 and JML. The initial efforts to merge syntaxes were due to the efforts of Raymie Stata. After a long process, the syntax of ESC/Java and JML were both changed and JML was nearly a superset of ESC/Java when work on ESC/Java stopped with ESC/Java 1.2.4. Following the open-source release of ESC/Java, Kiniry and Cok began work on ESC/Java2, which is now very compatible with JML's syntax [Kiniry-Cok04]. Users can thus use both tools with little or no changes to their files.

Similarly the Daikon tool [Ernst-etal01] also uses a variant of JML's syntax, as do several other tools [Burdy-etal03]. While efforts are ongoing to avoid differences, some differences are unavoidable, as research is ongoing (and people have other things to do).

We discuss the differences between the JML language described in this manual and the variants used in these other tools below.

E.1.1 Differences Between JML and ESC/Java2

This section discusses the current state of affairs of ESC/Java2 compatability 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 wher `helper` annotations are permitted
- JML does not support model classes (at least in runtime assertion checking).
- ESC/Java2 reads but ignores model programs.

The following differences between ESC/Java2 and JML are designed to remain differences. While the plan is for ESC/Java2 to parse all of JML's syntax, there are times when one needs to write annotations for one of these tool that are not understood by the other. Thus these differences are intended to allow users of both tools to write such annotations.

- JML supports annotation forms `//+@` and `/*+@` ... `@+*/`, so that annotations that JML understands but ESC/Java doesn't can be written.
- ESC/Java2 supports annotation forms `//-@` and `/*-@` ... `@-*/`, so that annotations that ESC/Java2 understands but JML doesn't can be written.

E.2 Differences Between JML and Java

Appendix F 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.

F.1 Deprecated Syntax

The following syntax is deprecated.

The *conditional-store-ref-list* syntax was formerly used in assignable clauses (see Section 9.9.8 [Assignable Clauses], page 64). Instead of using conditional store refs, one should now use a `\not_assigned` expression in the appropriate case of the postcondition to say when various data groups are not allowed to be assigned (see Section 11.1.3 [Backslash not_assigned], page 70).

```
conditional-store-ref-list ::= conditional-store-ref
            [ , conditional-store-ref ] . . .
conditional-store-ref ::= store-ref [ if predicate ]
        | other-ref [ if predicate ]
```

F.2 Replaced Syntax

As a note for readers of older papers, the keyword `subclassing_contract` has been replaced with `code_contract`.

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

Appendix G What's Missing

What is missing from this reference manual?

The following constructs are not discussed at all:

- `\other`
- `\private_data`
- `\such_that`
- `abrupt_behavior`
- `breaks` and `breaks_redundantly`
- `callable` and `callable_redundantly`
- `choose` and `choose_if`
- `continues` and `continues_redundantly`
- `in` and `in_redundantly`
- `maps` and `maps_redundantly`
- `example` and `exceptional_example`
- `forall`
- `implies_that`
- `hence_by` and `hence_by_redundantly`
- `measured_by` and `measured_by_redundantly`
- `model_program`
- `old` – In fact, it is unclear if `old` is a separate keyword from `\old`.
- `represents` and `represents_redundantly`
- `returns` and `returns_redundantly`
- `weakly` xxx
- The fact that `if` is a keyword is not discussed at all.

Other stuff not to forget - DRCok

- \not_specified
- \private_data
- \nothing
- \everything
- nowarn annotation
- methods and constructors without bodies in java files
- methods and constructors with bodies in specification files
- methods and constructors in annotation expressions - purity - modifies clauses - various checking
- anonymous and block-level classes
- field, method, constructor keywords
- exceptions in annotation expressions

Bibliography

[America87]
Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In Jean Bezivin and others (eds.), *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France.* Lecture Notes in Computer Science, Vol. 276 (Springer-Verlag, NY), pages 234-242.

[Arnold-Gosling-Holmes00]
Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language Third Edition.* The Java Series. Addison-Wesley, Reading, MA, 2000.

[ANSI95] *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language Java.* CBEMA, 1250 Eye Street NW, Suite 200, Washington DC 20005, April 28, 1995. (Obtained by anonymous ftp to research.att.com, directory dist/c++std/WP.)

[Back88] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593-624, August 1988.

[Back-vonWright89a]
R. J. R. Back and J. von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J. W. de Bakker, et al, (eds.), *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*, Mook, The Netherlands, May/June 1989, pages 42-66. Volume 430 of *Lecture Notes Computer Science*, Spring-Verlag, 1989.

[Back-vonWright98]
Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.

[Borgida-etal95]
Alex Borgida, John Mylopoulos, and Raymond Reiter. On the Frame Problem in Procedure Specifications. *IEEE Transactions on Software Engineering*, 21(10):785-798, October 1995.

[Buechi-Weck00]
Martin Büchi and Wolfgang Weck. The Greybox Approach: When Blackbox Specifications Hide Too Much. Technical Report 297, Turku Centre for Computer Science, August 1999.
'`http://www.tucs.abo.fi/publications/techreports/TR297.html`'.

[Buechi00] Martin Büchi. Safe Language Mechanisms for Modularization and Concurrency. Ph.D. Thesis, Turku Center for Computer Science, May 2000. TUCS Dissertations No. 28.

[Burdy-etal03]
Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. Dept. of Computer Science, University of Nijmegen, TR NIII-R0309, 2003.
'`ftp://ftp.cs.iastate.edu/pub/leavens/JML/jml-white-paper.pdf`'.

[Cheon-Leavens02]
Yoonsik Cheon and Gary T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *ECOOP 2002 – Object-Oriented Programming, 16th European Conference, Malaga, Spain*, pages 231–255. Springer-Verlag, June 2002. Also Department of Computer Science, Iowa State University, TR #01-12a, November 2001, revised March 2002 which is available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR01-12/TR.pdf`'.

[Cheon-Leavens02b]
Yoonsik Cheon and Gary T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun (eds.), *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA*, pages 322–328. CSREA Press, June 2002. Also Department of Computer Science, Iowa State University, TR #02-05, March 2002 which is available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR02-05/TR.pdf`'.

[Cheon-etal03]
Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model Variables: Cleanly Supporting Abstraction in Design By Contract. Department of Computer Science, Iowa State University, TR 03-10, March 2003.
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR03-10/TR.pdf`'.

[Cheon03] Yoonsik Cheon. A Runtime Assertion Checker for the Java Modeling Language. Department of Computer Science, Iowa State University, TR 03-09, April, 2003.
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR03-09/TR.pdf`'.

[Clifton-etal00]
Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota* (ACM SIGPLAN Notices, 35(10):130-145, Oct., 2000).

[Cohen90] Edward Cohen. *Programming in the 1990s: An Introduction to the Calculation of Programs*. Springer-Verlag, New York, N.Y., 1990.

[Corbett-etal00]
James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In S. Brookes and M. Main and A. Melton and M. Mislove (eds.), *Proceedings of the 22nd International Conference on Software Engineering*, pp. 439-448, ACM Press, 2000.

[Dhara-Leavens94b]
Krishna Kishore Dhara and Gary T. Leavens. Weak Behavioral Subtyping for Types with Mutable Objects. In S. Brookes and M. Main and A. Melton and M. Mislove (eds.), *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*,

‘`http://www.elsevier.nl/locate/entcs/volume1.html`’. Volume 1 of *Electronic Notes in Computer Science*, Elsevier, 1995.

[Dhara-Leavens96]
Krishna Kishore Dhara and Gary T. Leavens. Forcing Behavioral Subtyping Through Specification Inheritance. In *Proceedings 18th International Conference on Software Engineering*, Berlin, Germany, pages 258-267. IEEE 1996. An extended version is Department of Computer Science, Iowa State University, TR #95-20b, December 1995, which is available from the URL ‘`ftp://ftp.cs.iastate.edu/pub/techreports/TR95-20/TR.ps.Z`’.

[Dhara97] Krishna Kishore Dhara. Behavioral Subtyping in Object-Oriented Languages. Ph.D. Thesis, Department of Computer Science, Iowa State University. Technical Report TR #97-09, May 1997. Available from the URL ‘`ftp://ftp.cs.iastate.edu/pub/techreports/TR97-09/TR.ps.gz`’.

[Dijkstra76]
Edsger W. Dijkstra. *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, N.J., 1976).

[Edwards-etal94]
Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, Murali Sitaraman, and Bruce W. Weide. Part II: Specifying Components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):29-39 (Oct. 1994).

[Ernst-etal01]
Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1-25 (Feb. 2001).

[Fitzgerald-Larsen98]
John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development.* Cambridge University Press, Cambridge, UK, 1998.

[Gosling-etal00]
James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, Boston, MA, 2000.

[Gries-Schneider95]
David Gries and Fred B. Schneider. Avoiding the Undefined by Underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer-Verlag, New York, N.Y., 1995.

[Guttag-Horning-Wing85b]
John V. Guttag and James J. Horning and Jeannette M. Wing. The Larch Family of Specification Languages. *IEEE Software*, 2(5):24-36 (Sept. 1985).

[Guttag-Horning93]
John V. Guttag and James J. Horning with S.J. Garland, K.D. Jones, A. Modet and J.M. Wing. *Larch: Languages and Tools for Formal Specification* (Springer-

Verlag, NY, 1993). (The ISBN numbers are 0-387-94006-5 and 3-540-94006-5.) The traits in Appendix A (the "Handbook") are found on-line at the following URL
'`http://www.research.digital.com/SRC/larch/`'.

[Hall90] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11-19 (Sept. 1990).

[Hayes93] I. Hayes (ed.), *Specification Case Studies*, second edition (Prentice-Hall, Englewood Cliffs, N.J., 1990).

[Hesselink92]
Wim H. Hesselink. *Programs, Recursion, and Unbounded Choice* (Cambridge University Press, Cambridge, UK, 1992).

[Hoare69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12(10):576-583 (Oct. 1969).

[Hoare72a]
C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271-281 (1972).

[Huisman01]
Marieke Huisman. Reasoning about JAVA programs in higher order logic with PVS and Isabelle. IPA dissertation series, 2001-03. Ph.D. dissertation, University of Nijmegen, 2001.

[ISO96] International Standards Organization. *Information Technology - Programming Languages, Their Environments and System Software Interfaces - Vienna Development Method - Specification Language - Part 1: Base language*. International Standard ISO/IEC 13817-1, December, 1996.

[Jacobs-etal98]
Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java Classes (Preliminary Report) In *OOPSLA '98 Proceedings* (ACM SIGPLAN Notices, 33(10):329-490, Oct., 1998).

[Jones90] Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[Jones95e] C.B. Jones, Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65-67, 1995.

[Kiczales-Lamping92]
Gregor Kiczales and John Lamping. Issues in the Design and Documentation of Class Libraries. In Andreas Paepcke (ed.), *OOPSLA '92 Proceedings* (ACM SIGPLAN Notices, 27(10):435-451, Oct., 1992).

[Kiniry-Cok04]
Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting

tally system. In Marieke Huisman (ed.), *CASSIS 2004 - Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille, France, 2004, Proceedings*, volume ???? of *Lecture Notes in Computer Science*, pages ?–?. Springer-Verlag, 2004.

[Krone-Ogden-Sitaraman03]
Joan Krone, William F. Ogden, Murali Sitaraman. Modular Verification of Performance Constraints. Technical Report RSRG-03-04, Department of Computer Science, Clemson University, May, 2003. Available from '`http://www.cs.clemson.edu/~resolve/reports/RSRG-03-04.pdf`'

[Lamport89]
Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. *CACM*, 32(1):32-45 (Jan. 1989).

[LeavensLarchFAQ]
Gary T. Leavens. Larch frequently asked questions. Version 1.110. Available in '`http://www.cs.iastate.edu/~leavens/larch-faq.html`', May 2000.

[Leavens-Baker99]
Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999.

[Leavens-Baker-Ruby99]
Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175-188.

[Leavens-Baker-Ruby04]
Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java Iowa State University, Department of Computer Science, TR #98-06y, June 2004, which is available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.pdf`'.

[Leavens-Cheon04]
Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. March, 2004, which is available from the URL
'`ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`'.

[Leavens-Dhara00]
Gary T. Leavens and Krishna Kishore Dhara. Concepts of Behavioral Subtyping and a Sketch of Their Extension to Component-Based Systems. In Gary T. Leavens and Murali Sitaraman (eds.), *Foundations of Component-Based Systems*, Cambridge University Press, 2000, pp. 113-135.
'`http://www.cs.iastate.edu/~leavens/FoCBS-book/06-leavens-dhara.pdf`'

[Leavens-Weihl90]
Gary T. Leavens and William E. Weihl. Reasoning about Object-oriented Programs that use Subtypes (extended abstract). In N. Meyrowitz (ed.), *OOPSLA ECOOP '90 Proceedings* (ACM SIGPLAN Notices, 25(10):212-223, Oct., 1990).

[Leavens-Weihl95]
Gary T. Leavens and William E. Weihl. Specification and Verification of Object-Oriented Programs Using Supertype Abstraction. *Acta Informatica*, 32(8):705-778 (Nov. 1995).

[Leavens-Wing97a]
Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 520–534. Springer-Verlag, New York, N.Y., 1997.

[Leavens90]
Gary T. Leavens. Modular Verification of Object-Oriented Programs with Subtypes. Department of Computer Science, Iowa State University (Ames, Iowa, 50011), TR 90-09, July 1990. Available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR90-09/TR.ps.Z`'.

[Leavens91]
Gary T. Leavens. Modular Specification and Verification of Object-Oriented Programs. *IEEE Software*, 8(4):72-80 (July 1991).

[Leavens96b]
Gary T. Leavens. An Overview of Larch/C++: Behavioral Specifications for C++ Modules. In Haim Kilov and William Harvey (editors), *Specification of Behavioral Semantics in Object-Oriented Information Modeling* (Kluwer Academic Publishers, 1996), Chapter 8, pages 121-142. An extended version is Department of Computer Science, Iowa State University, TR #96-01c, July 1996, which is available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR96-01/TR.ps.Z`'.

[Leavens97c]
Gary T. Leavens. *Larch/C++ Reference Manual*. Version 5.14. Available in '`ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz`' or on the World Wide Web at the URL
'`http://www.cs.iastate.edu/~leavens/larchc++.html`', October 1997.

[Ledgard80]
Henry. F. Ledgard. A Human Engineered Variant of BNF. *ACM SIGPLAN Notices*, 15(10):57-62 (October 1980).

[Leino-Nelson-Saxe00]
K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java User's Manual. SRC Technical Note 2000-02, October, 2000.

[Leino-etal00]
K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking. Web page at ‘`http://research.compaq.com/SRC/esc/Esc.html`’.

[Leino95] K. Rustan M. Leino. Towards Reliable Modular Programs. PhD thesis, California Institute of Technology, January 1995. Available from the URL ‘`ftp://ftp.cs.caltech.edu/tr/cs-tr-95-03.ps.Z`’.

[Leino95b] K. Rustan M. Leino. A myth in the modular specification of programs. KRML 63, November 1995. Obtained from the author (rustan@pa.dec.com).

[Leino98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. *OOPSLA '98 Conference Proceedings*, ACM SIGPLAN Notices, Vol 33, Num 10, October 1998, pp. 144-153.

[Lerner91] Richard Allen Lerner. Specifying Objects of Concurrent Systems. School of Computer Science, Carnegie Mellon University, CMU-CS-91-131, May 1991. Available from the URL ‘`ftp://ftp.cs.cmu.edu/afs/cs.cmu.edu/project/larch/ftp/thesis.ps.Z`’.

[Liskov-Guttag86]
Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development* (MIT Press, Cambridge, Mass., 1986).

[Liskov-Wing93b]
Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. In Andreas Paepcke, editor, *OOPSLA '93 Proceedings*, volume 28, number 10 of *ACM SIGPLAN Notices*, pages 16-28. ACM Press, October 1993.

[Liskov-Wing94]
Barbara Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM TOPLAS*, 16(6):1811-1841 (Nov. 1994).

[Meyer92a]
Bertrand Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, October 1992.

[Meyer92b]
Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[Meyer97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[Morgan-Vickers94]
Carroll Morgan and Trevor Vickers. *On the refinement calculus*. Springer-Verlag, New York, N.Y., 1994.

[Morgan94]
Carroll Morgan. *Programming from Specifications*, second edition (Prentice-Hall, 1994).

[Morris87] Joseph~M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287-306, December 1987.

[Mueller-Poetzsch-Heffter00]
Peter Müller and Arnd Poetzsch-Heffter. Modular Specification and Verification Techniques for Object-Oriented Software Components. In Gary T. Leavens and Murali Sitaraman (eds.), *Foundations of Component-Based Systems*, pages 137-159. Cambridge University Press, 2000.

[Mueller-Poetzsch-Heffter00a]
Peter Müller and Arnd Poetzsch-Heffter. A Type System for Controlling Representation Exposure in Java. In S. Drossopoulou, et al. (eds.), *Formal Techniques for Java Programs*, 2000. Technical Report 269, Fernuniversität Hagen, Available from
'`http://www.informatik.fernuni-hagen.de/pi5/publications.html`'

[Mueller-Poetzsch-Heffter01a]
Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Alias and Dependency Control. Technical Report 279, Fernuniversität Hagen, 2001. Available from
'`http://www.informatik.fernuni-hagen.de/pi5/publications.html`'

[Mueller-Poetzsch-Heffter-Leavens02]
Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular Specification of Frame Properties in JML. Technical Report TR 02-02, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, February 2002. Available from
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR02-02/TR.pdf`'

[Mueller02]
Peter Müller. Modular Specification and Verification of Object-Oriented Programs. Volume 2262 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002.

[Nelson89] Greg Nelson. A Generalization of Dijkstra's Calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517-561 (Oct. 1989).

[Noble-Vitek-Potter98]
James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In Eric Jul (ed.), *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, pages volume 1445 of *Lecture Notes in Computer Science*, pages 158-185. Springer-Verlag, New York, N.Y., 1998.

[Parnas72] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACM*, 15(12) (Dec., 1972).

[Poetzsch-Heffter97]
Arnd Poetzsch-Heffter. Specification and Verification of Object-Oriented Programs. Habilitationsschrift, Technische Universitaet Muenchen, 1997. Available from the URL
'`http://wwweickel.informatik.tu-muenchen.de/persons/poetzsch/habil.ps.gz`'.

[Poll-Jacobs00]
E. Poll and B.P.F. Jacobs. A Logic for the Java Modeling Language JML. Computing Science Institute Nijmegen, Technical Report CSI-R0018. Catholic University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, November 2000.

[Raghavan-Leavens00]
Arun D. Raghavan and Gary T. Leavens. Desugaring JML Method Specifications. Technical Report 00-03a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, April, 2000, revised July 2000. Available in '`ftp://ftp.cs.iastate.edu/pub/techreports/TR00-03/TR.ps.gz`'.

[Rosenblum95]
David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[Ruby-Leavens00]
Clyde Ruby and Gary T. Leavens. Safely Creating Correct Subclasses without Seeing Superclass Code. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, pp. 208-228. Volume 35, number 10 of *ACM SIGPLAN Notices*, October, 2000. Also technical report 00-05d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011. April 2000, revised April, June, July 2000. Available in
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR00-05/TR.ps.gz`'.

[Spivey92] J. Michael Spivey. *The Z Notation: A Reference Manual*, second edition, (Prentice-Hall, Englewood Cliffs, N.J., 1992).

[Steyaert-etal96]
Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Issues in the Design and Documentation of Class Libraries. In *OOPSLA '96 Proceedings* (ACM SIGPLAN Notices, 31(10):268-285, Oct., 1996).

[Tan94] Yang Meng Tan. Formal Specification Techniques for Promoting Software Modularity, Enhancing Documentation, and Testing Specifications. MIT Lab. for Comp. Sci., TR 619, June 1994. Also published as *Formal Specification Techniques for Engineering Modular C Programs.* International Series in Software Engineering (Kluwer Academic Publishers, Boston, 1995).

[Watt91] David A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall, International Series in Computer Science, New York, 1991.

[Wills92b] Alan Wills. Specification in Fresco. In Susan Stepney and Rosalind Barden and David Cooper (eds.), *Object Orientation in Z*, chapter 11, pages 127-135. Springer-Verlag, Workshops in Computing Series, Cambridge CB2 1LQ, UK, 1992.

[Wing83] Jeannette Marie Wing. *A Two-Tiered Approach to Specifying Programs* Technical Report TR-299, Mass. Institute of Technology, Laboratory for Computer Science, 1983.

[Wing87] Jeannette M. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1-24 (Jan. 1987).

[Wing90a] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *Computer*, 23(9):8-24 (Sept. 1990).

Example Index

Concept Index

Table of Contents