Preliminary Design of JML:

A Behavioral Interface Specification Language for Java

by Gary T. Leavens, Albert L. Baker, and Clyde Ruby

TR #98-06p
June 1998, revised July, November 1998,
January, April, June, July, August, December 1999,
February, May, July, December 2000
February, April, May, August 2001

Keywords: Behavioral interface specification, Java, JML, Eiffel, Larch, model-based specification, assertion, precondition, postcondition, frame.

2000 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, tools, theory, Larch, Eiffel, JML, ESC/Java; D.2.4 [*Software Engineering*] Software/Program Verification — assertion checkers, class invariants, formal methods, programming by contract; D.2.7 [*Software Engineering*] Distribution and Maintenance — documentation; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, invariants, logics of programs, pre- and post-conditions, specification techniques.

Department of Computer Science, Iowa State University
226 Atanasoff Hall
Ames, Iowa 50011-1040, USA
jml@cs.iastate.edu

1 Introduction

Abstract

JML is a behavioral interface specification language tailored to Java. It also allows assertions to be intermixed with Java code, as an aid to verification and debugging. JML is designed to be used by working software engineers; to do this it follows Eiffel in using Java expressions in assertions. JML combines this idea from Eiffel with the model-based approach to specifications, typified by VDM and Larch, which results in greater expressiveness. Other expressiveness advantages over Eiffel include quantifiers, specification-only variables, and frame conditions.

This paper discusses the goals of JML, the overall approach, and describes the basic features of the language through examples. It is intended for readers who have some familiarity with both Java and behavioral specification using pre- and postconditions.

JML stands for “Java Modeling Language.” JML is a *behavioral interface specification language* (BISL) [Wing87] designed to specify Java [Arnold-Gosling98] [Gosling-Joy-Steele96] modules. Java *modules* are classes and interfaces.

The main goal of our research on JML is to better understand how to make BISLs (and BISL tools) that are practical and effective for production software environments. In order to understand this goal, and the more detailed discussion of our goals for JML, it helps to define more precisely what a behavioral interface specification is. After doing this, we return to describing the goals of JML, and then give a brief overview of the tool support for JML and an outline of the rest of the paper.

1.1 Behavioral Interface Specification

As a BISL, JML describes two important aspects of a Java module:

- its *interface*, which consists of the names and static information found in Java declarations, and
- its *behavior*, which tells how the module acts when used.

BISLs are inherently language-specific [Wing87], because they describe interface details for clients written in a specific programming language, For example, a BISL tailored to C++, such as Larch/C++ [Leavens97c], describes how to use a module in a C++ program. A Larch/C++ specification cannot be implemented correctly in Java, and a JML specification cannot be correctly implemented in C++ (except for methods that are specified as native code).

JML specifications can either be written in separate files or as annotations in Java code files. To a Java compiler such annotations are comments that are ignored [Luckham-vonHenke85] [Luckham-etal87] [Rosenblum95] [Tan94] [Tan95]. This allows JML specifications, such as the specification below, to be embedded in Java code files. Consider the following simple example of a behavioral interface specification in JML, written as annotations in a Java code file, ‘`IntMathOps.java`’.

```
public class IntMathOps {                                   // 1
                                                            // 2
  /*@ public normal_behavior                                // 3
    @   requires y >= 0;                                    // 4
    @   ensures \result * \result <= y                      // 5
    @         && y < (Math.abs(\result) + 1)                // 6
    @                * (Math.abs(\result) + 1);             // 7
    @*/                                                     // 8
  public static int isqrt(int y)                            // 9
  {                                                         //10
    return (int) Math.sqrt(y);                              //11
  }                                                         //12
}                                                           //13
```

The specification above describes a Java class, `IntMathOps` that contains one static method (function member) named `isqrt`. The single-line comments to the far right (which start with `//`) give the line numbers in this specification; they are ignored by both Java and JML. Comments with an immediately following at-sign, `//@`, or, as on lines 3–8, C-style comments starting with `/*@`, are *annotations*. Annotations are treated as comments by a Java compiler, but JML reads the text of an annotation. The text of an annotation is either the remainder of a line following `//@` or the characters between the annotation markers `/*@` and `@*/`. In the second form, at-signs (`@`) at the beginning of lines are ignored; they can be used to help the reader see the extent of an annotation.

In the above specification, interface information is declared in lines 1 and 9. Line 1 declares a class named `IntMathOps`, and line 9 declares a method named `isqrt`. Note that all of Java's declaration syntax is allowed in JML, including, on lines 1 and 9, that the names declared are `public`, that the method is `static` (line 9), that its return type is `int` (line 9), and that it takes one `int` argument.

Such interface declarations must be found in a Java module that correctly implements this specification. This is automatically the case in the file '`IntMathOps.java`' shown above, since that file also contains the implementation. In fact, when Java annotations are embedded in '`.java`' files, the interface specification is the actual Java source code.

To be correct, an implementation must have both the specified interface and the specified behavior. In the above specification, the behavioral information is specified in the annotation text on lines 3–8.[1] The keywords `public normal_behavior` are used to say that the specification is intended for clients (hence "public"), and that when the precondition is satisfied a call must return normally, without throwing an exception (hence "normal"). In such a public specification, only names with public visibility may be used.[2] On line

[1] In JML method specifications must be placed either before the method's header, as shown above, or between the method's header its body. In this document, we always place the specification before the method header. This convention is followed by many Java tools, in particular by Javadoc; It has the advantage of working in all cases, even when the method has no body.

[2] In a protected specification, both public and protected identifiers can be used. In a specification with default (i.e., no) visibility specified, which corresponds to Java's default visibility, public and protected identifiers can be used, as well as identifiers from the same package with default visibility. A private specification can use any identifiers that are available. The privacy level of a method specification cannot allow more access than the method being specified. Thus a public method may have a private specification, but a private method may not have a public specification.

4 is a precondition, which follows the keyword `requires`.[3] On lines 5–7 is a postcondition, which follows the keyword `ensures`.[4] The precondition says what must be true about the arguments (and other parts of the state); if the precondition is true, then the method must terminate normally in a state that satisfies the postcondition. This is a contract between the caller of the method and the implementor [Hoare69] [Jones90] [Jonkers91] [Guttag-Horning93] [Meyer92a] [Meyer97] [Morgan94]. The caller is obligated to make the precondition true, and gets the benefit of having the postcondition then be satisfied. The implementor gets the benefit of being able to assume the precondition, and is obligated to make the postcondition true in that case.

In general, pre- and postconditions in JML are written using an extended form of Java expressions. In this case, the only extension visible is the keyword `\result`, which is used in the postcondition to denote the value returned by the method. The type of `\result` is the return type of the method; for example, the type of `\result` in `isqrt` is `int`. The postcondition says that the result is an integer approximation to the square root of `y`. Note that the behavioral specification does not give an algorithm for finding the square root.

Method specifications may also be written in Java's documentation comments. The following is an example. The part that JML sees is enclosed within the HTML "tags" `<jml>` and `</jml>`.[5] The use of the surrounding tags `<pre>` and `</pre>` tells javadoc to ignore what JML sees, and to leave the formatting of it alone.

```
public class IntMathOps4 {

  /** Integer square root function.
   * @param int y
   * @return an integer approximating
   *         the positive square root of y
   * <pre><jml>
   *   public normal_behavior
   *     requires y >= 0;
   *     ensures \result >= 0
   *           && \result * \result <= y
   *           && y < (\result + 1) * (\result + 1);
   * </jml></pre>
   **/
  public static int isqrt(int y)
  {
     return (int) Math.sqrt(y);
  }
}
```

Because we expect most of our users to write specifications in Java code files, most of our examples will be given as annotations in '`.java`' files as in the specifications above. However, it is possible to use JML to write documentation in separate, non-Java, '`.jml-refined`' files, such as the file '`IntMathOps2.jml-refined`' below. Since these files are not Java code files,

[3] The keyword `pre` can also be used as a synonym for `requires`.

[4] The keyword `post` can also be used as a synonym for `ensures`.

[5] Since HTML tags are not case sensitive, in this one place JML is also not case sensitive. That is, the syntax also permits the tags `<JML>`, `</JML>`, `<esc>`, `</esc>`, `<ESC>`, and `</ESC>`.

JML allows the user to omit both the annotation markers and the code for concrete methods in a class. The specification below shows how this is done, using a semicolon (`;`), as in a Java abstract method declaration.

```
public class IntMathOps2 {

  public normal_behavior
     requires y >= 0;
     ensures \result * \result <= y
           && y < (Math.abs(\result) + 1)
                  * (Math.abs(\result) + 1);
  public static int isqrt(int y);

}
```

In a file with a suffix of '`.jml-refined`' or '`.jml`', the entire file is considered to be in an annotation. Hence, while annotation markers can be used, they need not be, as illustrated above, and all of the JML keywords are reserved in the entire file. (JML also works with files with the suffixes '`.spec`' and '`.spec-refined`'. Such file's use Java's syntax, and must use annotation markers just as in a '`.java`' file. However, since '`.spec`' and '`.spec-refined`' files are not Java files, in such a file one may also omit the code for concrete methods.)

The above specification would be implemented in the file '`IntMathOps2.java`', which is shown below. This file contains a `refine` clause, which tells the reader of the '`.java`' file what is being refined and the file in which to find its specification.

```
//@ refine IntMathOps2 <- "IntMathOps2.jml-refined";
public class IntMathOps2 {

  public static int isqrt(int y)
  {
     return (int) Math.sqrt(y);
  }
}
```

To summarize, a behavioral interface specification describes both the interface details of a module, and its behavior. The interface details are written in the syntax of the programming language; thus JML uses the Java declaration syntax. The behavioral specification uses pre- and postconditions.

1.2 Lightweight Specifications

Although we find it best to illustrate JML's features in this paper using specifications that are detailed and complete, one can use JML to write specifications at any desired level of detail. In particular, one can use JML to write "lightweight" specifications (as in ESC/Java). The syntax of JML allows one to write specifications that consist of individual clauses, so that one can say just what is desired. More precisely, a *lightweight* specification is one that does not use a behavior keyword (like `normal_behavior`).

For example, one might wish to specify just that `isqrt` should be called only on positive arguments, but not want to be bothered with saying anything formal about the result. This could be done as shown below. Notice that the only specification given below is a single `requires` clause. Since the specification of `isqrt` has no behavior keyword, it is a lightweight specification. What is the access restriction, or privacy level, of such a lightweight specification? The syntax for lightweight specifications does not have a place to specify the privacy level, so JML assumes that such a lightweight specification has the same level of visibility as the method itself. (Thus, the specification below is implicitly `public`.) What about the omitted parts of the specification, such as the ensures clause? JML assumes nothing about these. In the example below when the precondition is met, an implementation might either signal an exception or terminate normally, so this specification technically allows exceptions to be thrown. However, the gain in brevity often outweighs the need for this level of precision.

```
public class IntMathOps3 {

  //@ requires y >= 0;
  public static int isqrt(int y)
  {
    return (int) Math.sqrt(y);
  }
}
```

JML has a semantics that allows most clauses to be sensibly omitted from a specification. When the `requires` clause is omitted, for example, it means that no requirements are placed on the caller. When the `ensures` clause is omitted, it means that nothing is promised about the state resulting from a method call. See Appendix A [Specification Case Defaults], page 51, for the default meanings of various other clauses.

1.3 Goals

As mentioned above, the main goal of our research is to better understand how to develop BISLs (and BISL tools) that are practical and effective. We are concerned with both technical requirements and with other factors such as training and documentation, although in the rest of this paper we will only be concerned with technical requirements for the BISL itself. The practicality and effectiveness of JML will be judged by how well it can document reusable class libraries, frameworks, and Application Programming Interfaces (APIs).

We believe that to meet the overall goal of practical and effective behavioral interface specification, JML must meet the following subsidiary goals.

- JML must be able to document the interfaces and behavior of existing software, regardless of the analysis and design methods used to create it.

  If JML were limited to only handling certain Java features, certain kinds of software, or software designed according to certain analysis and design methods, then some APIs would not be amenable to documentation using JML. This would mean that some existing software could not be documented using JML. Since the effort put into writing such documentation will have a proportionally larger payoff for software that is more widely reused, it is important to be able to document existing software components.
- The notation used in JML should be readily understandable by Java programmers, including those with only standard mathematical training.

A preliminary study by Finney [Finney96] indicates that graphic mathematical notations, such as those found in Z [Hayes93] [Spivey92] [Woodcock-Davies96] may make such specifications hard to read, even for programmers trained in the notation. This accords with our experience in teaching formal specification notations to programmers. Hence, our strategy for meeting this goal has been to shun most special-purpose mathematical notations in favor of Java's own expression syntax.

- The language must be capable of being given a rigorous, formal semantics, and must also be amenable to tool support.

  This goal also helps ensure that the specification language does not suffer from logical problems, which would make it less useful for static analysis, prototyping, and testing tools.

We also have in mind a long range goal of a specification compiler, that would produce prototypes from specifications that happen to be constructive [Wahls-Leavens-Baker00].

Our partners at Compaq SRC and the University of Nijmegen have other goals in mind. At Compaq SRC, the goal is to make static analysis tools for Java programs that can help detect bugs. At the University of Nijmegen, the goal is to be able to do full program verification on Java programs.

As a general strategy for achieving these goals, we have tried to blend the Eiffel [Meyer92a] [Meyer92b] [Meyer97], Larch [Wing87] [Wing90a] [Guttag-Horning93] [LeavensLarchFAQ], and refinement calculus [Back88] [Back-vonWright98] [Morgan-Vickers94] [Morgan94] approaches to specification. From Eiffel we have taken the idea that assertions can be written in a language that is based on Java expressions. We also adapt the "`old`" notation from Eiffel, which appears in JML as `\old`, instead of the Larch-style annotation of names with state functions. However, Eiffel specifications, as written by Meyer, are typically not as detailed as model-based specifications written, for example, in Larch BISLs or in VDM-SL [Fitzgerald-Larsen98] [ISO96] [Jones90]. Hence, we have combined these approaches, by using syntactic ideas from Eiffel and semantic ideas from model-based specification languages.

JML also has some other differences from Eiffel (and its cousins Sather and Sather-K). The most important is the concept of specification-only declarations. These declarations allow more abstract and exact specifications of behavior than is typically done in Eiffel; they allow one to write specifications that are similar to the spirit of VDM or Larch BISLs. A major difference is that we have extended the syntax of Java expressions with quantifiers and other constructs that are needed for logical expressiveness, but which are not always executable. Finally, we ban side-effects and other problematic features of code in assertions.

On the other hand, our experience with Larch/C++ has taught us to adapt the model-based approach in two ways, with the aim of making it more practical and easy to learn. The first adaptation is again the use of specification-only model variables. An object will thus have (in general) several such *model fields*, which are used only for the purpose of describing, abstractly, the values of objects. This simplifies the use of JML, as compared with most Larch BISLs, since specifiers (and their readers) hardly ever need to know about algebraic-style specification. It also makes designing a model for a Java class or interface similar, in some respects, to designing an implementation data structure in Java. We hope that this similarity will make the specification language easier to understand. (This kind of model also has some technical advantages that will be described below.)

The second adaptation is hiding the details of mathematical modeling behind a facade of Java classes. In the Larch approach to behavioral interface specification [Wing87], the mathematical notation used in assertions is presented directly to the specifier. This allows the same mathematical notation to be used in many different specification languages. However, it also means that the user of such a specification language has to learn a notation for assertions that is different than their programming language's notation for expressions. In JML we use a compromise approach, hiding these details behind Java classes. These classes are pure, in the sense that they reflect the underlying mathematics, and hence do not use side-effects (at least not in any observable way). Besides insulating the user of JML from the details of the mathematical notation, this compromise approach also insulates the design of JML from the details of the mathematical logic used for theorem proving.

We have generally taken features wholesale from the refinement calculus [Back88] [Back-vonWright98] [Morgan-Vickers94] [Morgan94]. Our adaptation of it consists in blending it with the idea of interface specification and adding features for object-oriented programming. We are using the adaptation of the refinement calculus by Büchi and Weck [Buechi-Weck00], which helps in specifying callbacks. However, since the refinement calculus is mostly needed for advanced specifications, in the remainder of this paper we do not discuss the JML features related to refinement, such as model programs.

1.4 Tool Support

Our partners at Compaq SRC are building a tool, ESC/Java, that does static analysis for Java programs [Leino-etal00]. ESC/Java uses a subset of the JML specification syntax, to help detect bugs in Java code. At the University of Nijmegen the LOOP tool [Huisman01] [Jacobs-etal98] is being adapted to use JML as its input language. This tool would generate verification conditions that could be checked using a theorem prover such as PVS or Isabelle/HOL. At the Massachusetts Institute of Technology (MIT), the Daikon invariant detector project [Ernst-etal01] is using a subset of JML to record invariants detected by runs of a program. Recent work uses ESC/Java to validate the invariants that are found.

The JML release from Iowa State has tool support for: static checking of specifications, run-time assertion checking, and for generation of HTML pages. (See the '`README.txt`' file in the Iowa State JML release for the current status of these tools.) In the rest of the section we concentrate on the tool support found in the JML release from Iowa State.

1.4.1 Type Checking Specifications

Details on how to run the JML checker can be found in its manual page. Here we only indicate the most basic uses of the checker. Running the checker with filenames as arguments will perform type checking on all the specifications contained in the given files. For example, one could check the specifications in the file '`UnboundedStack.java`' by executing the following command.

```
jml UnboundedStack.java
```

To check all the relevant files in a directory and its subdirectories, one can pass the directory name to the checker, as in the following command.

```
jml .
```

In the above example, the checker will check all files with the following suffixes: '`.refines-java`', '`.refines-spec`', '`.refines-jml`', '`.java`', '`.spec`', and '`.jml`'; these suffixes are considered to be "active". There are also three "passive" suffixes: '`.java-refined`', '`.spec-refined`', and '`.jml-refined`'. File with passive suffixes can be used in refinements (see Section 1.1 [Behavioral Interface Specification], page 1) but the checker does not start checking from these files unless they appear explicitly on the command line. Among files in a directory with the same prefix, but with different active suffixes, the one whose suffix appears first in the list of active suffixes above will be the one the checker starts checking. The JML release includes a script, `jmlstartfiles`, that displays a list of the files from which checking will start in the current directory.

1.4.2 Generating HTML Documentation

To use the checker to generate HTML documentation that can be browsed on the web, one uses the `--html` option.

```
jml --html .
```

1.4.3 Run Time Assertion Checking

Using the checker to do run-time checking of preconditions (the only kind of assertion checking that is currently supported) is more complicated. First one would execute the following command to generate a version of the code containing run-time precondition checks.

```
jml --assertc myProgram.java UnboundedStack.java
```

The code that is generated to do run-time precondition checking is stored in another directory. This prevents the checker from destroying your input files. The parallel directory where the generated code is stored is defined by the value of the environment variable `JMLOUTDIR`, which defaults to '`c:\jmlout`' on Windows and '`$HOME/jmlout`' on Unix systems. (These defaults may be changed by the system administrator. You can personalize the defaults by setting the environment variable `JMLOUTDIR` as you wish.)

After doing this, change to the appropriate directory under `JMLOUTDIR`, the one that corresponds to the package. For instance, if `myProgram` lives in the package `stacktest`, then on Windows do something like the following.

```
c:
cd jmlout\stacktest
```

On Unix this can be done with a command like

```
cd $HOME/jmlout/stacktest
```

Once you are in the right directory, you can use the following commands to compile and then execute the generated code.

```
jassertc myProgram.java UnboundedStack.java
jassert myProgram
```

The script `jassertc` calls `javac` on the generated code in the directory named by `JMLOUTDIR`, and `jassert` runs it.

1.5 Outline

In the next sections we describe more about JML and its semantics. See Chapter 2 [Class and Interface Specifications], page 10, for examples that show how Java classes and interfaces are specified; this section also briefly describes the semantics of subtyping and refinement. See Chapter 3 [Extensions to Java Expressions], page 43, for a description of the expressions that can be used in specifications. See Chapter 4 [Conclusions], page 50, for conclusions from our preliminary design effort. See Appendix B [Syntax], page 52, for details on the syntax of JML.

2 Class and Interface Specifications

In this section we give some examples of JML class specifications that illustrate the basic features of JML.

2.1 Abstract Models

A simple example of an abstract class specification is the ever-popular `UnboundedStack` type, which is presented below. It would appear in a file named '`UnboundedStack.java`'.

```
package edu.iastate.cs.jml.samples.stacks;

//@ model import edu.iastate.cs.jml.models.*;

public abstract class UnboundedStack {

  /*@ public model JMLObjectSequence theStack
    @       initially theStack != null && theStack.isEmpty();
    @*/

  //@ public invariant theStack != null;

  /*@ public normal_behavior
    @   requires !theStack.isEmpty();
    @   assignable theStack;
    @   ensures theStack.equals(\old(theStack.trailer()));
    @*/
  public abstract void pop( );

  /*@ public normal_behavior
    @   assignable theStack;
    @   ensures theStack.equals(\old(theStack.insertFront(x)));
    @*/
  public abstract void push(Object x);

  /*@ public normal_behavior
    @   requires !theStack.isEmpty();
    @   ensures \result == theStack.first();
    @*/
  public abstract Object top( );
}
```

The above specification contains the declaration of a model field, an invariant, and some method specifications. These are described below.

2.1.1 Model Fields

In the fourth non-blank line of '`UnboundedStack.java`', a model data field, `theStack`, is declared. Since it is declared using the JML modifier `model`, such a field does not have to be implemented; however, for purposes of the specification we treat it exactly as any other

Java field (i.e., as a variable location). That is, we imagine that each instance of the class `UnboundedStack` has such a field.

The type of the model field `theStack` is a pure type, `JMLObjectSequence`, which is a sequence of objects. It is provided by JML in the package `edu.iastate.cs.jml.models`, which is imported in the second non-blank line of the figure. Note that this `import` declaration does not have to appear in the implementation, since it is modified by the keyword `model`. In general, any declaration form in Java can have this modifier, with the same meaning: that the declaration in question is only used for specification purposes, and does not have to appear in an implementation.

At the end of the model field's declaration above is an `initially` clause. (Such clauses are adapted from RESOLVE [Ogden-etal94] and the refinement calculus [Back88] [Back-vonWright98] [Morgan-Vickers94] [Morgan94].) An `initially` clause attached to a field declaration permits the field to have an abstract initialization. Knowing something about the initial value of the field permits data type induction [Hoare72a] [Wing83] for abstract classes and interfaces. The `initially` clause must appear to be true of the field's starting value. That is, all reachable objects of the type `UnboundedStack` must appear to have been created as empty stacks and subsequently modified using the type's methods.

2.1.2 Invariants

Following the model field declaration is an invariant. An invariant does not have to hold during the execution of an object's methods, but it must hold, for each reachable object in each *publicly visible state*; i.e., for each state outside of a public method's execution, and at the beginning and end of each such execution. The figure's invariant just says that the value of `theStack` should never be `null`.

2.1.3 Method Specifications

Following the invariant are the specifications of the methods `pop`, `push`, and `top`. We describe the new aspects of these specifications below.

2.1.3.1 The Assignable Clause

The use of the `assignable`[1] clauses in the behavioral specifications of `pop` and `push` is interesting (and another difference from Eiffel). These clauses give frame conditions [Borgida-Mylopoulos-Reiter95]. In JML, the frame condition given by a method's assignable clause only permits the method to assign to a location, *loc*, if:

- *loc* is mentioned in the method's `assignable` clause,
- a location mentioned in the clause depends on *loc* (see Section 2.2 [Dependencies], page 14),

[1] For historical reasons, one can also use the keyword `modifiable` as a synonym for `assignable`. Also, for compatibility with (older versions of ESC/Java [Leino-etal00]), in JML, one can also use the keyword `modifies` as a synonym for `assignable`. In the literature, the most common keyword for such a clause is `modifies`, and what JML calls the "assignable clause" is usually referred to as a "modifies clause". However, in JML, "assignable" most closely corresponds to the technical meaning, so we use that throughout this document. Users of JML may write whichever they prefer.

- *loc* was not allocated when the method started execution, or
- *loc* is local to the method (i.e., a local variable, including the method's formal parameters).

For example, `push`'s specification says that it may only assign to `theStack` (and locations on which it depends). This allows `push` to assign to `theStack` (and its dependees), or to call some other method that makes such an assignment. Furthermore, `push` may assign to the formal parameter `x` itself, even though that location is not listed in the `assignable` clause, since `x` is local to the method. However, `push` may not assign to fields not mentioned in the `assignable` clause; in particular it may not assign to fields of its formal parameter `x`,[2] or call a method that makes such an assignment.

JML can statically check the body of a method's implementation to determine whether its `assignable` clause is satisfied. Each assignment statement in the implementation can be checked to see if what is being assigned to is a location that some `assignable` clause permits. It is an error to assign to any other allocated, non-local location. Furthermore, JML will flag as an error a call to a method that would assign to locations that are not permitted by the calling method's `assignable` clause. It can do this using the `assignable` clause of the called method.

In JML, a location is *modified* by a method when it is allocated in both the pre-state of the method, reachable in the post-state, and has a value that is different in these two states. The *pre-state* of a method call is the state just after the method is called and parameters have been evaluated and passed, but before execution of the method's body. The *post-state* of a method call is the state just before the method returns or throws an exception; in JML we imagine that `\result` and information about exception results is recorded in the post-state.

Since modification only involves objects allocated in the pre-state, allocation of an object, using Java's `new` operator, does not itself cause any modification. Furthermore, since the fields of new objects are locations that were not allocated when the method started execution, they may be assigned to freely.

The reason assignments to local variables are permitted by the assignable clause is that a JML specification takes the client's (i.e., the caller's) point of view. From the client's point of view, the local variables in a method are newly-allocated, and thus assignments to such variables are invisible to the client. Hence, in JML, it is an error to list formal parameters, or other local variables, in the `assignable` clause. Furthermore, when formal parameters are used in a postcondition, JML interprets these as meaning the value initially given to the formal in the pre-state, since assignments to the formals within the method do not matter to the client.

JML's interpretation of the assignable clause does not permit either temporary side effects or benevolent side effects. A method with a *temporary side effect* assigns a location, does some work, and then assigns the original value back to that location. In JML, a method may not have temporary side effects on locations that it is not permitted to modify [Ruby-Leavens00]. A method has a *benevolent side effect* if it assigns to a location in a way that is not observable by clients. In JML, a method may not have benevolent side effects on locations that it is not permitted to modify [Leino95] [Leino95a].

[2] Assuming that `x` is not the same object as `this`!

Because JML's assignable clauses give permission to assign to locations, it is safe for clients to assume that only the listed locations (and their dependees) may have their values modified. Because locations listed in the `assignable` clause are the only ones that can be modified, we often speak of what locations a method can "modify," instead of the more precise "can assign to."

What does the assignable clause say about the modification of locations? In particular, although the "locations" for model fields and model variables cannot be directly assigned to in JML, their value is determined by the concrete fields and variables that they (ultimately) depend on. Thus, model fields and variables can be modified by assignments to their concrete dependees. So a method's assignable clause only permits the method to modify a location if:

- the location is mentioned in the method's `assignable` clause,
- a location mentioned in the clause depends on it (see Section 2.2 [Dependencies], page 14),
- it was not allocated when the method started execution, or
- it is local to the method.

When the `assignable` clause is omitted, as it is in the specification of `top`, this means that no locations can be modified, because none may be assigned to.

2.1.3.2 Old Values

When a method can modify some locations, they may have different values in the pre-state and post-state of a call. Often the post-condition must refer to the values held in both of these states. JML uses a notation similar to Eiffel's to refer to the pre-state value of a variable. In JML the syntax is `\old(`*E*`)`, where *E* is an expression. (Unlike Eiffel, we use parentheses following `\old` to delimit the expression to be evaluated in the pre-state explicitly.)

The meaning of `\old(`*E*`)` is as if *E* were evaluated in the pre-state and that value is used in place of `\old(`*E*`)` in the assertion. It follows that, an expression like `\old(myVar).theStack` may not mean what is desired, since only the old value of `myVar` is saved; access to the field `theStack` is done in the post-state. If it is the field, `theStack`, not the variable, `myVar`, that is changing, then probably what is desired is `\old(myVar.theStack)`. To avoid such problems, it is good practice to have the expression *E* in `\old(`*E*`)` be such that its type is either the type of a primitive value, such as an `int`, or a pure type, such as `JMLObjectSequence`.

As another example, in `pop`'s postcondition the expression `\old(theStack.trailer())` has type `JMLObjectSequence`, which is a pure type. The value of `theStack.trailer()` is computed in the pre-state of the method.

Note also that, since `JMLObjectSequence` is a reference type, one is required to use `equals` instead of `==` to compare them for equality of values. (Using `==` would be a mistake, since it would only compare them for object identity, which in combination with `new` would always yield false.)

2.1.3.3 Correct Implementation

The specification of `push` does not have a `requires` clause. This means that the method imposes no obligations on the caller. (The meaning of an omitted `requires` clause is that the method's precondition is `true`, which is satisfied by all states, and hence imposes no obligations on the caller.) This seems to imply that the implementation must provide a literally unbounded stack, which is surely impossible. We avoid this problem, by following Poetzsch-Heffter [Poetzsch-Heffter97] in releasing implementations from their obligations to fulfill the postcondition when Java runs out of storage. In general, a method specified with `normal_behavior` has a correct implementation if, whenever it is called in a state that satisfies its precondition, either

- the method terminates normally in a state that satisfies its postcondition, having assigned to only the locations permitted by its `assignable` clause, or
- Java signals an error, by throwing an exception that inherits from `java.lang.Error`.

We discuss the specification of methods with exceptions in the next subsection.

2.2 Dependencies

In this subsection we present two example specifications. The two example specifications, `BoundedThing` and `BoundedStackInterface`, are used to describe how model (and concrete) fields can be related to one another, and how dependencies among them affect the meaning of the `assignable` clause. Along the way we also demonstrate how to specify methods that can throw exceptions and other features of JML.

2.2.1 Specification of BoundedThing

The specification in the file '`BoundedThing.java`', shown below, is an interface specification with a simple abstract model. In this case, there are two model fields `MAX_SIZE` and `size`.

```
package edu.iastate.cs.jml.samples.stacks;

public interface BoundedThing {

  //@ public model static int MAX_SIZE;
  //@ public model instance int size;

  //@ public invariant MAX_SIZE > 0 && 0 <= size && size <= MAX_SIZE;

  //@ public constraint MAX_SIZE == \old(MAX_SIZE);

  /*@  public normal_behavior
         ensures \result == MAX_SIZE;
    @*/
  public int getSizeLimit();

  /*@  public normal_behavior
         ensures \result <==> size == 0;
```

```
      @*/
    public boolean isEmpty();

    /*@  public normal_behavior
           ensures \result <==> size == MAX_SIZE;
      @*/
    public boolean isFull();

    /*@ also
          public behavior
            ensures \result instanceof BoundedThing
                 && size == ((BoundedThing)\result).size;
            signals (CloneNotSupportedException) true;
      @*/
    public Object clone () throws CloneNotSupportedException;
  }
```

After discussing the model fields, we describe the other parts of the specification below.

2.2.1.1 Model Fields in Interfaces

In the specification above, the variable `MAX_SIZE` is a static model field, which is treated as a class variable.

The variable `size` is treated as a normal model field, i.e., as an instance variable, because of the use of the keyword `instance`. This keyword tells the reader that the variable being declared is not static, but has a copy in each instance of a class that implements this interface.[3]

In specifications of interfaces that extend or classes that implement this interface, these model fields are inherited. Thus, for example, every object that has a type that is a subtype of the `BoundedThing` interface is thought of, abstractly, as having a field `size`, of type `int`. Similarly, every class that implements `BoundedThing` is thought of as having a static model field `MAX_SIZE`.

2.2.1.2 Invariant and History Constraint

Two pieces of class-level specification come after the abstract model in the above specification.

The first is an `invariant` clause. The figure's invariant says that in every publicly visible state, the `MAX_SIZE` variable has to be positive, and that every reachable object that is a `BoundedThing` must have a size field that has a value less than or equal to `MAX_SIZE`.

Following the invariant is a history constraint [Liskov-Wing94]. A history constraint is used to say how values can change between earlier and later publicly-visible states, such as a method's pre-state and its post-state. This prohibits subtypes from making certain state changes, even if they implement more methods than are specified in a given class. The

[3] By default, fields declared in Java interfaces are static. Java does not allow non-static fields to be declared in interfaces. However, JML allows non-static model or ghost fields in interfaces when one uses `instance`. The reason for this extension is that such fields are essential for defining the abstract values and behavior of the objects being specified.

history constraint in the specification above says that the value of `MAX_SIZE` cannot change, since in every pre-state and post-state, its value in the post-state, written `MAX_SIZE`, must equal its value in the pre-state, written `\old(MAX_SIZE)`.

2.2.1.3 Details of the Method Specifications

Following the history constraint are the interfaces and specifications for four public methods. Notice that, if desired, the at-signs (`@`) may be omitted from the left sides of intermediate lines, as we do in this specification.

The use of `==` in the method specifications is okay, since in each case, the things being compared are primitive values, not references. The notation `<==>` can be read "if and only if". It has the same meaning for Boolean values as `==`, but has a lower precedence. Therefore, the expression "`\result <==> size == 0`" in the postcondition of the `isEmpty` method means the same thing as "`\result == (size == 0)`".

2.2.1.4 Adding to Method Specifications

The specification of the last method of `BoundedThing`, `clone`, is interesting. Note that it begins with the keyword `also`. This form is intended to tell the reader that the specification given is in addition to any specification that might have been given in the superclass `Object`, where `clone` is declared as a protected method. A form like this must be used whenever a specification is given for a method that overrides a method in a superclass, or that implements a method from an implemented interface.

2.2.1.5 Specifying Exceptional Behavior

The specification of `clone` also uses `behavior` instead of `normal_behavior`. In a specification that starts this way, one can describe not just the case where the execution returns normally, but also executions where exceptions are thrown. In such a specification, the conditions under which exceptions can be thrown can be described by the predicate in the `signals` clauses,[4] and the conditions under which the method may return without throwing an exception are described by the `ensures` clause. In this specification, the `clone` method may always throw the exception, because it only needs to make the predicate "`true`" true to do so. When the method returns normally, it must make the given postcondition true.

In JML, a `normal_behavior` specification can be thought of as a syntactic sugar for a `behavior` specification to which the following clause is added [Raghavan-Leavens00].

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

This formalizes the idea that a method with a `normal_behavior` specification may not throw an exception when the specification's precondition is satisfied.

JML also has a specification form `exceptional_behavior`, which can be used to specify when exceptions must be thrown. A specification that uses `exceptional_behavior` can be thought of as a syntactic sugar for a `behavior` specification to which the following clause is added [Raghavan-Leavens00].

[4] The keyword "`exsures`" can also be used in place of `signals`.

```
          ensures false;
```

This formalizes the idea that a method with a `exceptional_behavior` specification may not return normally when the specification's precondition is satisfied.

Since in the specification of `clone`, we want to allow the implementation to make a choice between either returning normally or throwing an exception, and we do not wish to distinguish the preconditions under which each choice must be made, we cannot use either of the more specialized forms `normal_behavior` or `exceptional_behavior`. Thus the specification of `clone` demonstrates the somewhat unusual case when the more general form of a `behavior` specification is needed.

Finally note that in the specification of `clone`, the postcondition says that the result will be a `BoundedThing` and that its size will be the same as the model field `size`. The use of the cast in this postcondition is necessary, since the type of `\result` is `Object`. (This also adheres to our goal of using Java syntax and semantics to the extent possible.) Note also that the conjunct `\result instanceof BoundedThing` "protects" the next conjunct [Leavens-Wing97a] since if it is false the meaning of the cast does not matter.

2.2.2 Specification of BoundedStackInterface

The specification in the file '`BoundedStackInterface.java`' below gives an interface for bounded stacks that extends the interface for `BoundedThing`. In the specification below, one can refer to the static field `MAX_SIZE` from the `BoundedThing` interface, and to `size` as an inherited instance field of its objects.

```
package edu.iastate.cs.jml.samples.stacks;
//@ model import edu.iastate.cs.jml.models.*;
public interface BoundedStackInterface extends BoundedThing {
  /*@ public model instance JMLObjectSequence theStack
    @             initially theStack != null && theStack.isEmpty();
    @*/
  //@ public depends size <- theStack;
  //@ public represents size <- theStack.length();
  //@ public invariant theStack != null;
  //@ public invariant_redundantly theStack.length() <= MAX_SIZE;
  /*@ public invariant
    @         (\forall int i; 0 <= i && i < theStack.length();
    @                         theStack.itemAt(i) != null);
    @*/

  /*@   public normal_behavior
    @     requires !theStack.isEmpty();
    @     assignable size, theStack;
    @     ensures theStack.equals(\old(theStack.trailer()));
    @ also
    @   public exceptional_behavior
    @     requires theStack.isEmpty();
    @     signals (BoundedStackException);
    @*/
  public void pop( ) throws BoundedStackException;
```

```
    /*@   public normal_behavior
      @     requires theStack.length() < MAX_SIZE && x != null;
      @     assignable size, theStack;
      @     ensures theStack.equals(\old(theStack.insertFront(x)));
      @     ensures_redundantly theStack != null && top() == x
      @              && theStack.length() == \old(theStack.length()+1);
      @ also
      @   public exceptional_behavior
      @     requires theStack.length() >= MAX_SIZE || x == null;
      @     signals (BoundedStackException)
      @                  theStack.length() == MAX_SIZE;
      @     signals (NullPointerException) x == null;
      @*/
    public void push(Object x )
           throws BoundedStackException, NullPointerException;

    /*@   public normal_behavior
      @     requires !theStack.isEmpty();
      @     ensures \result == theStack.first() && \result != null;
      @ also
      @   public exceptional_behavior
      @     requires theStack.isEmpty();
      @     signals (BoundedStackException);
      @*/
    public /*@ pure @*/ Object top( ) throws BoundedStackException;
  }
```

The abstract model for `BoundedStackInterface` adds to the inherited model by declaring a model instance field named `theStack`. This field is typed as a `JMLObjectSequence`.

In the following we describe how the new model instance field, `theStack`, is related to `size` from `BoundedThing`. We also use this example to explain more JML features.

2.2.2.1 Depends and Represents Clauses

The `depends` and `represents` clauses that follow the declaration of `theStack` are an important feature in modeling with layers of model fields.[5] They also play a crucial role in relating model fields to the concrete fields of objects, which can be considered to be the final layer of detail in a design.

The `depends` clause in `BoundedStackInterface` says that `size` *depends* on `theStack`; this means that `size` might change its value when the `theStack` changes. Another way of looking at this is that, if one wants to change `size`, this can be done by changing `theStack`. We also say that `theStack` is a *dependee* of `size`.

A model field can be declared to depend on another model field or on a *concrete* field (i.e., a field that is not a model field). However, concrete fields cannot be declared to depend on other fields.

[5] Of course, one could specify `BoundedStack` without separating out the interface `BoundedThing`, and in that case, these layers would be unnecessary. We have made this separation partly to demonstrate more advanced features of JML, and partly to make the parts of the example smaller.

The `depends` clause is important in "loosening up" the `assignable` clause, for example to permit the fields of an object that implement the abstract model to be changed [Leino95] [Leino95a]. This "loosening up" also applies to model fields that have dependencies declared. For example, since `size` depends on `theStack`, whenever `size` is mentioned in a `assignable` clause, then `theStack` is implicitly allowed to be modified.[6] Thus it is only for rhetorical purposes that we mention both `size` and `theStack` in the assignable clauses of `pop` and `push`. Note, however, that just mentioning `theStack` would not permit `size` to be modified, because `theStack` does not depend on `size`.

The `represents` clause in `BoundedStackInterface` says how the value of `size` is related to the value of `theStack`. It says that the value of `size` is `theStack.length()`.

A `represents` clause gives additional facts that can be used in reasoning about the specification. It serves the same purpose as an abstraction function in various proof methods for abstract data types (such as [Hoare72a]).

One can only use a represents clause to state facts about a field and its dependees. To state relationships among concrete data fields or on fields that are not related by a dependency, one should use an invariant.

2.2.2.2 Redundant Specification

The second `invariant` clause that follows the `represents` clause in the specification of `BoundedStackInterface` above is our first example of checkable redundancy in a specification [Leavens-Baker99] [Tan94] [Tan95]. This concept is signaled in JML by the use of the suffix `_redundantly` on a keyword (as in `ensures_redundantly`). It says both that the stated property is specified to hold and that this property is believed to follow from the other properties of the specification. In this case the redundant invariant follows from the given invariant, the invariant inherited from the specification of `BoundedThing`, and the fact stated in the `represents` clause. Even though this invariant is redundant, it is sometimes helpful to state such properties, to bring them to the attention of the readers of the specification.

Checking that such claimed redundancies really do follow from other information is also a good way to make sure that what is being specified is really what is intended. Such checks could be done manually, during reviews, or with the aid of a theorem prover.

2.2.2.3 Multiple Specification Cases

Following the redundant invariant of `BoundedStackInterface` are the specifications of the `pop`, `push`, and `top` methods. These are interesting for several new features that they present. Each of these has both a normal and exceptional behavior specified. The meaning of such multiple *specification cases* is that, when the precondition of one of them is satisfied, the rest of that specification case must also be obeyed.

[6] Note that the permission to assign a field goes from the more abstract field to the one it depends on (which in this case is also abstract). Müller points out that this direction is necessary for information hiding, because concrete fields are often hidden (e.g., they may be `private`), and as such cannot appear in public specifications, so the public specification has to mention the more abstract field, which give assignment rights to their dependees [Mueller01].

A specification with several specification cases is shorthand for one in which the separate specifications are combined [Dhara-Leavens96] [Leavens97c] [Wing83] [Wills94]. The desugaring can be thought of as proceeding in two steps (see [Raghavan-Leavens00] for more details). First, the `public` `normal_behavior` and `public` `exceptional_behavior` cases are converted into `public` `behavior` specifications as explained above. This would produce a specification for `pop` as shown below. The use of `implies_that` introduces a redundant specification that can be used, as is done here, to point out consequences of the specification to the reader. In this case the specification in question is the one mentioned in the `refine` clause. Note that in the second specification case of this figure, the predicate "`true`" has been added for the signals clause; since this is often the predicate desired in a signals clause, JML allows it to be omitted.

```
//@ refine BoundedStackInterface <- "BoundedStackInterface.java";
//@ model import edu.iastate.cs.jml.models.*;
public interface BoundedStackInterface extends BoundedThing {
  /*@ also
    @ implies_that
    @   public behavior
    @     requires !theStack.isEmpty();
    @     assignable size, theStack;
    @     ensures theStack.equals(\old(theStack.trailer()));
    @     signals (java.lang.Exception) false;
    @  also
    @   public behavior
    @     requires theStack.isEmpty();
    @     ensures false;
    @     signals (BoundedStackException) true;
    @*/
  public void pop( ) throws BoundedStackException;
}
```

The second step of the desugaring is shown below. As can be seen from this example, `public` `behavior` specifications that are joined together using `also` have a precondition that is the disjunction of the preconditions of the combined specification cases. The `assignable` clause for the expanded specification is the union of all the assignable clauses for the cases, with each modification governed by the corresponding precondition (which follows the keyword `if`). That is, variables are only allowed to be modified if the modification was permitted in the corresponding case, as determined by its precondition. The ensures clauses of the second desugaring step correspond to the ensures clauses for each specification case; they say that whenever the precondition for that specification case held in the pre-state, its postcondition must also hold. As can be seen in the specification below, in logic this is written using an implication between `\old` wrapped around the case's precondition and its postcondition. Having multiple ensures clauses is equivalent to writing a single ensures clause that has as its postcondition the conjunction of the given postconditions. Similarly, the signals clauses in the desugaring correspond those in the given specification cases; as for the ensures clauses, each has a predicate that says that signaling that exception can only happen when the predicate in that case's precondition holds.

```
/*@ refine BoundedStackInterface
  @                 <- "BoundedStackInterface.java-refined";
  @*/
//@ model import edu.iastate.cs.jml.models.*;
public interface BoundedStackInterface extends BoundedThing {
  /*@ also
    @ implies_that
    @   public behavior
    @     requires !theStack.isEmpty() || theStack.isEmpty();
    @     assignable size if !theStack.isEmpty(),
    @                 theStack if !theStack.isEmpty();
    @     ensures \old(!theStack.isEmpty())
    @               ==> theStack.equals(\old(theStack.trailer()));
    @     ensures \old(theStack.isEmpty()) ==> false;
    @     signals (java.lang.Exception)
    @              \old(!theStack.isEmpty()) ==> false;
    @     signals (BoundedStackException)
    @              \old(theStack.isEmpty()) ==> true;
    @*/
  public void pop( ) throws BoundedStackException;
}
```

In the file ‘`BoundedStackInterface.refines-java`’ above, the precondition of `pop` reduces to `true`. However, the precondition shown is the general form of the expansion. Similar remarks apply to other predicates.

2.2.2.4 Pitfalls in Specifying Exceptions

A particularly interesting example of multiple specification cases occurs in the specification of the `BoundedStackInterface`’s `push` method. Like the other methods, this example has two specification cases; one of these is a `normal_behavior` and one is an `exceptional_behavior`. However, the exceptional behavior of `push` is interesting because it specifies more than one exception that may be thrown. The requires clause of the exceptional behavior says that an exception must be thrown when either the stack cannot grow larger, or when the argument `x` is null. The first signals clause says that, if a `BoundedStackException` is thrown, then the stack cannot grow larger, and the second signals clause says that, if a `NullPointerException` is thrown, then `x` must be null. The specification is written in this way because it may be that *both* conditions occur; the specification allows the implementation to choose (even nondeterministically) which exception is thrown in that case.

Specifiers should be wary of such situations, where two exceptional conditions may both be true simultaneously, because it is impossible in Java to throw more than one exception from a method call. Thus, for example, if the specification of `push` had been written as follows, it would not be implementable.[7] The problem is that both exceptional preconditions may be true, and in that case an implementation cannot throw both exceptions.

```
  /*@   public normal_behavior
    @     requires theStack.length() < MAX_SIZE && x != null;
    @     assignable size, theStack;
    @     ensures theStack.equals(\old(theStack.insertFront(x)));
    @     ensures_redundantly theStack != null && top() == x
```

[7] Thanks to Erik Poll for pointing this out.

```
    @               && theStack.length() == \old(theStack.length()+1);
    @ also
    @   public exceptional_behavior
    @     requires theStack.length() >= MAX_SIZE;
    @     signals (BoundedStackException);
    @ also                                  // this is wrong!
    @   public exceptional_behavior
    @     requires x == null;
    @     signals (NullPointerException);
    @*/
  public void push(Object x )
         throws BoundedStackException, NullPointerException;
```

One could fix the example above by writing one of the requires clauses in the two exceptional behaviors to exclude the other, although this would make the specification deterministic about which exception would be thrown when both exceptional conditions occur. In general, it seems best to avoid this pitfall by writing several exceptional cases together in a single exceptional behavior clause, as was done originally for `push` (see Section 2.2.2.3 [Multiple Specification Cases], page 19) or to simply use a single behavior clause.

2.2.2.5 Redundant Ensures Clauses

Finally, there is more redundancy in the specifications of `push` in the original specification of `BoundedStackInterface` above, which has a redundant `ensures` clause in its normal behavior. For an `ensures_redundantly` clause, what one checks is that the conjunction of the precondition, the meaning of the `assignable` clause, and the (non-redundant) postcondition together imply the redundant postcondition. It is interesting to note that, for `push`, the specifications for stacks written in Eiffel (see page 339 of [Meyer97]) expresses just what we specify in `push`'s redundant postcondition. This conveys strictly less information than the non-redundant postcondition for `push`'s normal behavior, since it says little about the elements of the stack.[8]

2.3 Making New Pure Types

JML comes with a suite of pure types, implemented as Java classes that can be used for defining abstract models. These are found in the package `edu.iastate.cs.jml.models`, which includes both collection and non-collection types (such as `JMLInteger`) and a few helper classes (such as exceptions and enumerators).

The pure collection types can hold either objects or values, and this distinction determines the notion of equality used on their elements and whether cloning is done on the elements. The object containers use `==` and do not clone. Simple collection types include the set types, `JMLObjectSet` and `JMLValueSet`, and sequence types `JMLObjectSequence` and `JMLValueSequence`. The binary relation and map types can independently have objects in their domain or range. The binary relation types are named `JMLObjectToObjectRelation`, `JMLObjectToValueRelation`, and so on. The four map types are similarly named according to the scheme `JML...To...Map`.

[8] Meyer's second specification and implementation of stacks (see page 349 of [Meyer97]) is no better in this respect, although, of course, the implementation does keep track of the elements properly.

Users can also create their own pure types if desired. Since these types are to be treated as purely immutable values in specifications, they must be declared with the modifier `pure` and pass certain conservative checks that make sure there is no possibility of observable side-effects from using such objects.

A *pure interface* must have a specification such that:

- all the methods in each interface it extends are pure (these may be either in pure interfaces or the methods may be explicitly specified as pure),
- all the methods it specifies must be pure in the sense described below.

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

A *pure method* (not a constructor) must have a specification that does not allow any side-effects. That is, it must have a specification that refines (i.e., is stronger than) the following:

```
behavior
   assignable \nothing;
```

A *pure constructor* must have a specification such that: it assigns to only 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).

Implementations of pure methods and constructors will be checked to see that they meet these conditions. In particular, a pure method or constructor implementation is prohibited from calling methods or constructors that are not pure. It must also be provably terminating.

A pure method or constructor can be declared in any class. JML will specify the intuitively pure methods and constructors in the standard Java libraries as pure.[9]

A *pure class* must have a specification such that:

- it only extends other pure classes,
- all the methods in each interface it extends are pure,
- all its methods and constructors must be specified to be pure in the sense described above, and
- all its data fields must be of some primitive value type or a pure type.

Recursion is permitted, both in pure methods and in data members of pure classes. However, a pure method must be proved to terminate when its preconditions is met. When recursion is used in a specification, the proof involves the use of a `measured_by` clause in the specification.

Model classes should also be pure, since there is no way to use non-pure operations in an assertion. However, the modifiers `model` and `pure` are orthogonal, and thus usually one will want to list both of them when declaring a model class. In particular, one may specify a pure class that is not a model class; such a class would have to be implemented.

We give some examples of pure interfaces, abstract classes, and classes below.

[9] Such a specification of the Java libraries is, however, not yet available. Until such a specification is available, JML may need to relax purity checking for methods with no specification.

2.3.1 Money

The following example begins a specification of money that would be suitable for use in abstract models. Our specification is rather artificially broken up into pieces to allow each piece to have a specification that fits on a page. This organization is not necessarily something we would recommend, but it does give us a chance to illustrate more features of JML.

Consider first the interface `Money` specified below. The abstract model here is a single field of the primitive Java type `long`, which holds a number of pennies. Note that the declaration of this field, `pennies`, again uses the JML keyword `instance`.

```
package edu.iastate.cs.jml.docs.prelimdesign;

import edu.iastate.cs.jml.models.JMLType;

public /*@ pure @*/ interface Money extends JMLType
{
  //@ public model instance long pennies;

  //@ public constraint pennies == \old(pennies);

  /*@     public normal_behavior
    @       ensures \result == pennies / 100;
    @ for_example
    @     public normal_example
    @       requires pennies == 703;
    @       ensures \result == 7;
    @   also
    @     public normal_example
    @       requires pennies == 799;
    @       ensures \result == 7;
    @   also
    @     public normal_example
    @       requires pennies == -503;
    @       ensures \result == -5;
    @*/
  public long dollars();

  /*@   public normal_behavior
    @     ensures \result == pennies % 100;
    @ for_example
    @     requires pennies == 703;
    @     ensures \result == 3;
    @   also
    @     requires pennies == -503;
    @     ensures \result == -3;
    @*/
  public long cents();

  /*@ also
```

```
      @   public normal_behavior
      @     ensures \result <==> o2 instanceof Money
      @                          && pennies == ((Money)o2).pennies;
      @*/
    public boolean equals(Object o2);

    /*@ also
      @   public normal_behavior
      @     ensures \result instanceof Money
      @       && ((Money)\result).pennies == pennies;
      @*/
    public Object clone();
  }
```

This interface has a history constraint, which says that the number of pennies in an object cannot change.[10]

The following explain more aspects of JML related to the above specification.

2.3.1.1 Redundant Examples

The interesting aspect of `Money`'s method specifications is another kind of redundancy. This new form of redundancy is examples, which follow the keyword "`for_example`".

Individual examples are given by `normal_example` clauses (adapted from our previous work on Larch/C++ [Leavens96b] [Leavens-Baker99]). Any number of these[11] can be given in a specification. In the specification of `Money` above there are three normal examples given for `dollars` and two in the specification of `cents`.

The specification in each example should be such that:

- the example's precondition implies the precondition of the expanded meaning of the specified behaviors,
- the example's assignable clause specifies a subset of the locations that are assignable according to the expanded meaning of the specified behaviors, and
- the conjunction of the example's precondition (wrapped by `\old()`), the precondition of the expanded meaning of the specified behaviors (also wrapped by `\old()`), the assignable clause of the expanded meaning of the specified behaviors, and the postcondition of the expanded meaning of the specified behaviors should be equivalent to the example's postcondition.

Requiring equivalence to the example's postcondition means that it can serve as a test oracle for the inputs described by the example's precondition. If there is only one specified `public normal_behavior` clause and if there are no preconditions and assignable clauses, then the example's postcondition should the equivalent to the conjunction of the example's

[10] There is no use of `initially` in this interface, so data type induction cannot assume any particular starting value. But this is desirable, since if a particular starting value was specified, then by the history constraint, all objects would have that value.

[11] One may also give `exceptional_example` clauses, which are analogous to `exceptional_behavior` specifications, and `example` clauses, which are analogous to`behavior` specifications. There is also a lightweight form,that is similar to the `example` form, except that the introductory keywords "`public example`" are omitted.

precondition and the postcondition of the `public` `normal_behavior` specification. Typically, examples are concrete, and serve to make various rhetorical points about the use of the specification to the reader. (Exercise: check all the examples given!)

2.3.1.2 JMLType and Informal Predicates

The interface `Money` is specified to extend the interface `JMLType`. This interface is given below. Classes that implement this interface must have `equals` and `clone` methods with the specified behavior. The methods specified override methods in the class `Object`, and so they use the form of specification that begins with the keyword “`also`”.

```
package edu.iastate.cs.jml.models;

/** Objects with a clone and equals method.
    JMLObjectType and JMLValueType are refinements
    for object and value containers (respectively).
    @see JMLObjectType
    @see JMLValueType
 **/
public interface JMLType extends Cloneable, java.io.Serializable {

  /*@ also
    @   public normal_behavior
    @     ensures \result instanceof JMLType
    @           && ((JMLType)\result).equals(this);
    @*/
  public /*@ pure @*/ Object clone();

  /*@ also
    @   public normal_behavior
    @     ensures \result <==>
    @             (* ob2 is not distinguishable from this,
    @                except by using mutation or == *);
    @ implies_that
    @   public normal_behavior
    @   {|
    @      requires ob2 instanceof JMLType;
    @      ensures ob2.equals(this) == \result;
    @    also
    @      requires ob2 == this;
    @      ensures \result <==> true;
    @   |}
    @*/
  public /*@ pure @*/ boolean equals(Object ob2);
}
```

The specification of `JMLType` is noteworthy in its use of informal predicates [Leavens96b]. In JML these start with an open parenthesis and an asterisk (‘`(*`’) and continue until a matching asterisk and closing parenthesis (‘`*)`’). In the public specification of `equals`, the `normal_behavior`’s `ensures` clause uses an informal predicate as an escape from formality.

The use of informal predicates avoids the delicate issues of saying formally what observable aliasing means, and what equality of values means in general.[12]

In the `implies_that` section of the specification of the `equals` method is a nested case analysis, between `{|` and `|}`. The meaning of this is that each pre- and postcondition pair has to be obeyed. The first of these nested pairs is essentially saying that `equals` has to be symmetric. The second of these is saying that it has to be reflexive.

2.3.2 MoneyComparable and MoneyOps

The type `Money` lacks some useful operations. The extensions below provide specifications of comparison operations and arithmetic, respectively.

The specification in file '`MoneyComparable.java`' is interesting because each of the specified preconditions protects the postcondition from undefinedness in the postcondition [Leavens-Wing97a]. For example, if the argument `m2` in the `greaterThan` method were `null`, then the expression `m2.pennies` would not be defined.

```
package edu.iastate.cs.jml.docs.prelimdesign;

public /*@ pure @*/ interface MoneyComparable extends Money
{
  /*@ public normal_behavior
    @   requires m2 != null;
    @   ensures \result <==> pennies > m2.pennies;
    @*/
  public boolean greaterThan(Money m2);

  /*@ public normal_behavior
    @   requires m2 != null;
    @   ensures \result <==> pennies >= m2.pennies;
    @*/
  public boolean greaterThanOrEqualTo(Money m2);

  /*@ public normal_behavior
    @   requires m2 != null;
    @   ensures \result <==> pennies < m2.pennies;
    @*/
  public boolean lessThan(Money m2);

  /*@ public normal_behavior
    @   requires m2 != null;
    @   ensures \result <==> pennies <= m2.pennies;
    @*/
  public boolean lessThanOrEqualTo(Money m2);
}
```

[12] *Observable aliasing* is a sharing relation between objects that can be detected by a program. Such a program, might, for example modify one object and read a changed value from the shared object. Formalizing this in general is beyond the scope of this paper, and probably beyond what JML can describe.

The interface specified in the file '`MoneyOps.java`' below extends the interface specified above. `MoneyOps` is interesting for the use of its pure model methods: `inRange`, `can_add`, and `can_scaleBy`. These methods cannot be invoked by Java programs; that is, they would not appear in the Java implementation. When, for example `inRange` is called in a predicate it is equivalent to using some correct implementation of its specification. The specification of `inRange` also makes use of a local specification variable declaration, which follows the keyword "`old`". Such declarations allow one to abbreviate long expressions, or, to make rhetorical points by naming constants, as is done with `epsilon`. These `old` declarations are treated as locations that are initialized to the pre-state value of the given expression. Model methods can be normal (instance) methods as well as static (class) methods.

```
package edu.iastate.cs.jml.docs.prelimdesign;

public /*@ pure @*/ interface MoneyOps extends MoneyComparable
{
  /*@  public normal_behavior
    @     old model double epsilon = 1.0;
    @     ensures \result <==> Long.MIN_VALUE + epsilon < d
    @                          && d < Long.MAX_VALUE - epsilon;
    @ public model boolean inRange(double d);
    @
    @  public normal_behavior
    @     requires m2!= null;
    @     ensures \result <==> inRange((double) pennies + m2.pennies);
    @ public model boolean can_add(Money m2);
    @
    @  public normal_behavior
    @     ensures \result <==> inRange(factor * pennies);
    @ public model boolean can_scaleBy(double factor);
    @*/

  /*@   public normal_behavior
    @     requires m2 != null && can_add(m2);
    @     ensures \result != null
    @           && \result.pennies == this.pennies + m2.pennies;
    @ for_example
    @   public normal_example
    @     requires this.pennies == 300 && m2.pennies == 400;
    @     ensures \result != null && \result.pennies == 700;
    @*/
  public MoneyOps plus(Money m2);

  /*@   public normal_behavior
    @     requires m2 != null
    @            && inRange((double) pennies - m2.pennies);
    @     ensures \result != null
    @           && \result.pennies == this.pennies - m2.pennies;
    @ for_example
    @   public normal_example
```

```
      @     requires this.pennies == 400 && m2.pennies == 300;
      @     ensures  \result != null && \result.pennies == 100;
      @*/
    public MoneyOps minus(Money m2);

    /*@   public normal_behavior
      @     requires can_scaleBy(factor);
      @     ensures \result != null
      @           && \result.pennies == (long)(factor * pennies);
      @ for_example
      @   public normal_example
      @     requires pennies == 400 && factor == 1.01;
      @     ensures \result != null && \result.pennies == 404;
      @*/
    public MoneyOps scaleBy(double factor);
  }
```

Note also that JML uses the Java semantics for mixed-type expressions. For example in the ensures clause of the above specification of `plus`, `m2.pennies` is implicitly coerced to a double-precision floating point number, as it would be in Java.

2.3.3 MoneyAC

The key to proofs that an implementation of a class or interface specification is correct lies in the use of `depends` and `represents` clauses [Hoare72a] [Leino95].

Consider, for example, the abstract class specified in the file '`MoneyAC.java`' below. This class is abstract and has no constructors. The class declares a concrete field `numCents`, which is related to the model instance field `pennies` by the `represents` clause. This allows relatively trivial proofs of the correctness of the `dollars` and `cents` methods, and is key to the proofs of the other methods.

```
  package edu.iastate.cs.jml.docs.prelimdesign;

  public /*@ pure @*/ abstract class MoneyAC implements Money {

    protected long numCents;
    //@ protected depends pennies <- numCents;
    //@ protected represents pennies <- numCents;

    //@ protected constraint_redundantly numCents == \old(numCents);

    public long dollars()
    {
      return numCents / 100;
    }

    public long cents()
    {
      return numCents % 100;
    }
    public boolean equals(Object o2)
    {
      try {
        Money m2 = (Money)o2;
        return numCents == (100 * m2.dollars() + m2.cents());
      } catch (ClassCastException e) {
        return false;
      }
    }

    public Object clone()
    {
      return this;
    }
  }
```

2.3.4 MoneyComparableAC

The straightforward implementation of the pure abstract subclass `MoneyComparableAC` is given below. Besides extending the class `MoneyAC`, it implements the interface `MoneyComparable`. Note that the model and concrete fields are both inherited by this class.

```
package edu.iastate.cs.jml.docs.prelimdesign;

public /*@ pure @*/ abstract class MoneyComparableAC
    extends MoneyAC implements MoneyComparable
{
  protected static long totalCents(Money m2)
  {
    long res = 100 * m2.dollars() + m2.cents();
    //@ assert res == m2.pennies;
    return res;
  }

  public boolean greaterThan(Money m2)
  {
    return numCents > totalCents(m2);
  }

  public boolean greaterThanOrEqualTo(Money m2)
  {
    return numCents >= totalCents(m2);
  }

  public boolean lessThan(Money m2)
  {
    return numCents < totalCents(m2);
  }
```

```
    public boolean lessThanOrEqualTo(Money m2)
    {
      return numCents <= totalCents(m2);
    }
  }
```

An interesting feature of the class `MoneyComparableAC` is the protected static method named `totalCents`. For this method, we give its code with an embedded assertion, written following the keyword `assert`.[13]

Note that the model method, `inRange` is not implemented, and does not need to be implemented to make this class correctly implement the interface `MoneyComparable`.

2.3.5 USMoney

Finally, a concrete class implementation is given in the file '`USMoney.java`' shown below. The class `USMoney` implements the interface `MoneyOps`. Note that specifications as well as code are given for the constructors.

```
package edu.iastate.cs.jml.docs.prelimdesign;

public /*@ pure @*/ class USMoney
                extends MoneyComparableAC implements MoneyOps
{
  /*@   public normal_behavior
    @     assignable pennies;
    @     ensures pennies == cs;
    @ implies_that
    @   protected normal_behavior
    @     assignable numCents;
    @     ensures numCents == cs;
    @*/
  public USMoney(long cs)
  {
    numCents = cs;
  }

  /*@ public normal_behavior
    @   assignable pennies;
    @   ensures pennies == (long)(100.0 * amt);
    @   ensures_redundantly (* pennies holds amt dollars *);  @*/
  public USMoney(double amt)
```

[13] As of JDK 1.4, `assert` is also a reserved word in Java. One can thus write assert statements either in standard Java or in JML annotations. If one writes an assert statement as a JML annotation, all of the JML extensions to the Java expression syntax see Section 3.1 [Extensions to Java Expressions for Predicates], page 43 for the predicate can be used, but no side-effects are allowed in this predicate. Such a JML *assert-statement* may also refer to model and ghost variables. In a Java assert statement, i.e., in an *assert-statement* that is not in an annotation, one cannot use JML's extensions for assertions, because such assertions must compile with a Java compiler.

```
    {
      numCents = (long)(100.0 * amt);
    }

    public MoneyOps plus(Money m2)
    {
      //@ assume m2 != null;
      return new USMoney(numCents + totalCents(m2));
    }

    public MoneyOps minus(Money m2)
    {
      //@ assume m2 != null;
      return new USMoney(numCents - totalCents(m2));
    }

    public MoneyOps scaleBy(double factor)
    {
      return new USMoney(numCents * factor / 100.0);
    }
  }
```

The constructors each mention the fields that they initialize in their `assignable` clause. This is because the constructor's job is to initialize these fields. One can think of a `new` expression in Java as executing in two steps: allocating an object, and then calling the constructor. Thus the specification of a constructor needs to mention the fields that it can initialize in the `assignable` clause.

The first constructor's specification also illustrates that redundancy can also be used in an `assignable` clause. A redundant `assignable` clause follows if the meaning of the set of locations named is a subset of the ones given in the non-redundant clause for the same specification case. In this example the redundant assignable clause follows from the given assignable clause and the meaning of the `depends` clause inherited from the superclass `MoneyAC`.

The second constructor above is noteworthy in that there is a redundant ensures clauses that uses an informal predicate [Leavens96b]. In this instance, the informal predicate is used as a comment (which could also be used). Recall that informal predicates allow an escape from formality when one does not wish to give part of a specification in formal detail.

The `plus` and `minus` methods use `assume` statements; these are like assertions, but are intended to impose obligations on the callers [Back-Mikhajlova-vonWright98]. The main distinction between a `assume` statement and a `requires` clause is that the former is a statement and can be used within code. These may also be treated differently by different tools. For example, ESC/Java [Leino-etal00] will require callers to satisfy the requires clause of a method, but will not enforce the precondition if it is stated as an assumption.

2.4 Use of Pure Classes

Since `USMoney` is a pure class, it can be used to make models of other classes. An example is the abstract class specified in the file '`Account.jml`' below. The first model field in this class has the type USMoney, which was specified above. (Further explanation follows the specification below.)

```
package edu.iastate.cs.jml.docs.prelimdesign;

public class Account {

  //@ public model MoneyOps credit;
  //@ public model String owner;

  /*@ public invariant owner != null && credit != null
    @             && credit.greaterThanOrEqualTo(new USMoney(0));
    @*/
  //@ public constraint owner.equals(\old(owner));

  /*@  public normal_behavior
    @    requires own != null && amt != null
    @          && (new USMoney(1)).lessThanOrEqualTo(amt);
    @    assignable credit, owner;
    @    ensures credit.equals(amt) && owner.equals(own);
    @*/
  public Account(MoneyOps amt, String own);

  /*@  public normal_behavior
    @    ensures \result.equals(credit);
    @*/
  public pure MoneyOps balance();

  /*@  public normal_behavior
    @    requires 0.0 <= rate && rate <= 1.0
    @          && credit.can_scaleBy(1.0 + rate);
    @    assignable credit;
    @    ensures credit.equals(\old(credit.scaleBy(1.0 + rate)));
    @ for_example
    @  public normal_example
    @    requires rate == 0.05 && (new USMoney(4000)).equals(credit);
    @    ensures credit.equals(new USMoney(4200));
    @*/
  public void payInterest(double rate);

  /*@  public normal_behavior
    @    requires amt != null
    @           && amt.greaterThanOrEqualTo(new USMoney(0))
    @           && credit.can_add(amt);
    @    assignable credit;
    @    ensures credit.equals(\old(credit.plus(amt)));
    @ for_example
    @  public normal_example
    @    requires credit.equals(new USMoney(40000))
    @           && amt.equals(new USMoney(1));
```

```
      @    ensures credit.equals(new USMoney(40001));
      @*/
    public void deposit(MoneyOps amt);

    /*@  public normal_behavior
      @   requires amt != null && (new USMoney(0)).lessThanOrEqualTo(amt)
      @          && amt.lessThanOrEqualTo(credit);
      @    assignable credit;
      @    ensures credit.equals(\old(credit.minus(amt)));
      @ for_example
      @  public normal_example
      @    requires credit.equals(new USMoney(40001))
      @           && amt.equals(new USMoney(40000));
      @    ensures credit.equals(new USMoney(1));
      @*/
    public void withdraw(MoneyOps amt);
  }
```

The specification of `Account` makes good use of examples. It also demonstrates the various ways of protecting predicates used in the specification from undefinedness [Leavens-Wing97a]. The principal concern here, as is often the case when using reference types in a model, is to protect against the model fields being `null`. As in Java, fields and variables of reference types can be `null`. In the specification of `Account`, the invariant states that these fields should not be null. Since implementations of public methods must preserve the invariants, one can think of the invariant as conjoined to the precondition and postcondition of each public method, and the postcondition of each public constructor. Hence, for example, method pre- and postconditions do not have to state that the fields are not null. However, often other parts of the specification must be written to allow the invariant to be preserved, or established by a constructor. For example, in the specification of `Account`'s constructor, this is done by requiring `amt` and `own` are not null, since, if they could be null, then the invariant could not be established.

2.5 Composition for Container Classes

The following specifications lead to the specification of a class `Digraph` (directed graph). This gives a more interesting example of how more complex models can be composed in JML from other classes. In this example we use model classes and the pure containers provided in the package `edu.iastate.cs.jml.models`.

2.5.1 NodeType

The file '`NodeType.java`' contains the specification of an abstract class `NodeType`. `NodeType` is an abstract class, as opposed to a model class, because it will require an implementation and does appear in the interface of the model class `Digraph`. However, we also declare this abstract class as `pure`, since we will also use `NodeType` in the specification of other classes. (And we do so appropriately, since all the methods for class `NodeType` are side-effect-free.) In the abstract class specification for `NodeType` we simply provide a model field `iD`, which would represent a unique identifier for nodes.

```
package edu.iastate.cs.jml.samples.Digraph;

import edu.iastate.cs.jml.models.*;

public /*@ pure @*/ abstract class NodeType implements JMLType {

  //@ public model int iD;

  /*@ public normal_behavior
    @   {|
    @      requires o instanceof NodeType;
    @      ensures \result == (iD == ((NodeType)o).iD);
    @    also
    @      requires !(o instanceof NodeType);
    @      ensures \result == false;
    @   |}
    @*/
  public abstract boolean equals(Object o);

  /*@ public normal_behavior
    @   ensures \result instanceof NodeType
    @        && ((NodeType)\result).equals(this);
    @*/
  public abstract Object clone();

} // end of class NodeType
```

The use of `also` in the specification of `NodeType`'s `equals` method is interesting. It separates two cases of the normal behavior for that method. This is equivalent to using two `public normal_behavior` clauses, one for each case. That is, when the argument is an instance of `NodeType`, the method must return true just when `this` and `o` have the same `iD` field. And when `o` is not an instance of `NodeType`, the `equals` method must return false. Compare this with the specification of the `equals` method for the class `ArcType` below (see Section 2.5.2 [ArcType], page 35).

2.5.2 ArcType

`ArcType` is specified as a pure model class in the file '`ArcType.jml`' shown below. It is a model class because it does not appear in the interface to `Digraph`, and so does not need to be implemented. We declare `ArcType` to be a pure class so that its methods can be used in assertions. The two model fields for `ArcType`, `from` and `to`, are both of type `NodeType`. We specify the `equals` method so that two references to objects of type `ArcType` are equal if and only if they have equal values in the `from` and `to` model fields. Thus, `equals` is specified using `NodeType.equals`. We also specify that `ArcType` has a public `clone` method, fulfilling the obligations of a type that implements `JMLType`. `ArcType` must implement `JMLType` so that its objects can be placed in a `JMLValueSet`. We use such a set for one of the model fields of `DiGraph`.

```
package edu.iastate.cs.jml.samples.Digraph;
```

```
import edu.iastate.cs.jml.models.JMLType;

public pure model class ArcType implements JMLType {

    public model NodeType from;
    public model NodeType to;
    public invariant from != null && to != null;

    public normal_behavior
      requires from != null && to != null;
      assignable this.from, this.to;
      ensures this.from.equals(from) && this.to.equals(to);
    public ArcType(NodeType from, NodeType to);

    also
      public normal_behavior
        requires o instanceof ArcType;
        ensures \result <==> ((ArcType)o).from.equals(from)
             && ((ArcType)o).to.equals(to);
    also
      public normal_behavior
        requires !(o instanceof ArcType);
        ensures \result == false;
    public boolean equals(Object o);

    also
      public normal_behavior
        ensures \result instanceof ArcType
             && ((ArcType)\result).equals(this);
    public Object clone();
}
```

2.5.3 Digraph

Finally, the specification of the class `Digraph` is given in the file '`Digraph.jml`' shown below. This specification demonstrates how to use container classes, like `JMLValueSet`, combined with appropriate invariants to specify models that are compositions of other classes. Both the model fields `nodes` and `arcs` are of type `JMLValueSet`. However, the first invariant clause restricts `nodes` so that every object in `nodes` is, in fact, of type `NodeType`. Similarly, the next invariant clause we restrict `arcs` to be a set of `ArcType` objects. In both cases, since the type is `JMLValueSet`, membership is determined by the `equals` method for the type of the elements (rather than reference equality).

```
package edu.iastate.cs.jml.samples.Digraph;
//@ model import edu.iastate.cs.jml.models.*;
public class Digraph {

 //@ public model JMLValueSet nodes;
 //@ public model JMLValueSet arcs;
```

```
/*@ public invariant nodes != null
  @   && (\forall JMLType n; nodes.has(n); n instanceof NodeType);
  @ public invariant arcs != null
  @   && (\forall JMLType a; arcs.has(a); a instanceof ArcType);
  @ public invariant (\forall ArcType a; arcs.has(a);
  @                       nodes.has(a.from) && nodes.has(a.to));
  @*/

/*@  public normal_behavior
  @   assignable nodes, arcs;
  @   ensures nodes.isEmpty() && arcs.isEmpty();
  @*/
public Digraph();

/*@  public normal_behavior
  @   requires n != null;
  @   assignable nodes;
  @   ensures nodes.equals(\old(nodes.insert(n)));
  @*/
public void addNode(NodeType n);

/*@  public normal_behavior
  @   requires unconnected(n);
  @   assignable nodes;
  @   ensures nodes.equals(\old(nodes.remove(n)));
  @*/
public void removeNode(NodeType n);

/*@  public normal_behavior
  @   requires inFrom != null && inTo != null
  @         && nodes.has(inFrom) && nodes.has(inTo);
  @   assignable arcs;
  @   ensures arcs.equals(
  @              \old(arcs.insert(new ArcType(inFrom, inTo))));
  @*/
public void addArc(NodeType inFrom, NodeType inTo);

/*@  public normal_behavior
  @   ensures \result == nodes.has(n);
  @*/
public pure boolean isNode(NodeType n);

/*@   public normal_behavior
  @    ensures \result == arcs.has(new ArcType(inFrom, inTo));
  @
  @*/
public pure boolean isArc(NodeType inFrom, NodeType inTo);
```

```
    /*@   public normal_behavior
      @  requires nodes.has(start) && nodes.has(end);
      @  ensures \result == reachSet(new JMLValueSet(start)).has(end);
      @*/
    public pure boolean isAPath(NodeType start, NodeType end);

    /*@  public normal_behavior
      @   ensures \result <==>
      @             !(\exists ArcType a; arcs.has(a);
      @                    a.from.equals(n) || a.to.equals(n));
      @ public pure model boolean unconnected(NodeType n);
      @*/

    /*@  public normal_behavior
      @   requires nodeSet != null
      @    && (\forall Object o; nodeSet.has(o);
      @         o instanceof NodeType && nodes.has(o));
      @   measured_by nodes.size() - nodeSet.size();
      @   {|
      @      requires nodeSet.equals(OneMoreStep(nodeSet));
      @      ensures \result != null && \result.equals(nodeSet);
      @    also
      @      requires !nodeSet.equals(OneMoreStep(nodeSet));
      @      ensures \result != null
      @              && \result.equals(reachSet(OneMoreStep(nodeSet)));
      @   |}
      @ public pure model JMLValueSet reachSet(JMLValueSet nodeSet);
      @*/

    /*@  public normal_behavior
      @   requires nodeSet != null
      @    && (\forall Object o; nodeSet.has(o);
      @         o instanceof NodeType && nodes.has(o));
      @  ensures \result != null
      @    && \result.equals(nodeSet.union(
      @        new JMLValueSet { NodeType n | nodes.has(n)
      @          && (\exists ArcType a; a != null && arcs.has(a);
      @                   nodeSet.has(a.from) && n.equals(a.to))}));
      @ public pure model JMLValueSet OneMoreStep(JMLValueSet nodeSet);
      @*/
   }  // end of class Digraph
```

An interesting use of pure model methods appears at the end of the specification of `Digraph` in the pure model method `reachSet`. This method constructively defines the set of all nodes that are reachable from the nodes in the argument `nodeSet`. This specification uses a nested case analysis, between `{|` and `|}`. The meaning of this is again that each pre- and postcondition pair has to be obeyed, but by using nesting, one can avoid duplication of the requires clause that is found at the beginning of the specification. The `measured_by` clause is needed because this specification is recursive; the measure given allows one to describe a termination argument, and thus ensure that the specification is well-defined. This clause defines an integer-valued measure that must always be at least zero; furthermore, the measure for a call and recursive uses in the specification must strictly decrease [Owre-etal95]. The recursion in the specification builds up the entire set of reachable nodes by, for each recursive reference, adding the nodes that can be reached directly (via a single arc) from the nodes in `nodeSet`.

2.6 Subtyping

Following Dhara and Leavens [Dhara-Leavens96] [Leavens97c], a subtype inherits the specifications of its supertype's public and protected members (fields and methods), as well as its public and protected invariants and history constraints.[14] This ensures that a subclass specifies a behavioral subtype of its supertypes. This inheritance can be thought of textually, by copying the public and protected specifications of the methods of a class's ancestors and all interfaces that a class implements into the class's specification and combining the specifications using `also`.[15] (This is the reason for the use of `also` at the beginning of specifications in overriding methods.) By the semantics of method combination using `also`, these behaviors must all be satisfied by the method, in addition to any explicitly specified behaviors.

For example, consider the class `PlusAccount`, specified in file '`PlusAccount.jml`' shown below. It is specified as a subclass of `Account` (see Section 2.4 [Use of Pure Classes], page 32). Thus it inherits the fields of `Account`, and `Account`'s public invariants, history constraints, and method specifications. (The specification of `Account` given above does not have any `protected` specification information.) Because it inherits the fields of its superclass, inherited method specifications of behavior are still meaningful when copied to the subclass. The trick is to always add new model fields to the subclass and relate them to the existing ones.

Note that in the represents clause below, instead of a left-facing arrow, `<-`, the connective "`\such_that`" is used to introduce a relationship predicate. This form of the represents clause allows one to specify abstraction relations, instead of abstraction functions.

```
package edu.iastate.cs.jml.docs.prelimdesign;

public class PlusAccount extends Account {
  //@ public model MoneyOps savings, checking;

  //@ public depends credit <- savings, checking;
  /*@ public represents credit \such_that
    @                     credit.equals(savings.plus(checking));
    @*/
  //@ public invariant savings != null && checking != null;
  /*@ public invariant_redundantly savings.plus(checking)
    @                        .greaterThanOrEqualTo(new USMoney(0));
```

[14] A subtype also inherits default privacy (package-protected) method specifications, invariants, and history constraints if it is in the same package as its supertype.

[15] However, textual copying shouldn't be taken literally; if a subclass declares a field that hides the fields of its superclass, renaming must be done to prevent name capture.

```
    @*/

  /*@  public normal_behavior
    @    requires sav != null && chk != null && own != null
    @          && (new USMoney(1)).lessThanOrEqualTo(sav)
    @          && (new USMoney(1)).lessThanOrEqualTo(chk);
    @    assignable credit, owner;
    @    assignable_redundantly savings, checking;
    @    ensures savings.equals(sav) && checking.equals(chk)
    @             && owner.equals(own);
    @    ensures_redundantly credit.equals(sav.plus(chk));
    @*/
  public PlusAccount(MoneyOps sav, MoneyOps chk, String own);


  /*@ also
    @  public normal_behavior
    @    requires 0.0 <= rate && rate <= 1.0
    @          && credit.can_scaleBy(1.0 + rate);
    @    assignable credit, savings, checking;
    @    ensures checking.equals(\old(checking.scaleBy(1.0 + rate)));
    @ for_example
    @  public normal_example
    @    requires rate == 0.05 && checking.equals(new USMoney(2000));
    @    ensures checking.equals(new USMoney(2100));
    @*/
  public void payInterest(double rate);

  /*@ also
    @  public normal_behavior
    @    requires amt != null
    @            && (new USMoney(0)).lessThanOrEqualTo(amt)
    @            && amt.lessThanOrEqualTo(savings);
    @    assignable credit, savings;
    @    ensures savings.equals(\old(savings.minus(amt)))
    @            && \not_modified(checking);
    @ also
    @  public normal_behavior
    @    requires amt != null
    @            && (new USMoney(0)).lessThanOrEqualTo(amt)
    @            && amt.lessThanOrEqualTo(credit)
    @            && amt.greaterThan(savings);
    @    assignable credit, savings, checking;
    @    ensures savings.equals(new USMoney(0))
    @           && checking.equals(
    @                    \old(checking.minus(amt.minus(savings))));
    @ for_example
    @   public normal_example
    @    requires savings.equals(new USMoney(40001))
    @             && amt.equals(new USMoney(40000));
    @    ensures savings.equals(new USMoney(1))
    @            && \not_modified(checking);
    @  also
    @   public normal_example
    @    requires savings.equals(new USMoney(30001))
    @           && checking.equals(new USMoney(10000))
    @           && amt.equals(new USMoney(40000));
    @    ensures savings.equals(new USMoney(0))
    @           && checking.equals(new USMoney(1));
    @*/
  public void withdraw(MoneyOps amt);

  /*@ also
    @  public normal_behavior
    @    requires amt != null
    @           && amt.greaterThanOrEqualTo(new USMoney(0))
    @           && credit.can_add(amt);
    @    assignable credit, savings;
    @    ensures savings.equals(\old(savings.plus(amt)))
    @            && \not_modified(checking);
    @ for_example
    @  public normal_example
    @    requires savings.equals(new USMoney(20000))
    @           && amt.equals(new USMoney(1));
    @    ensures savings.equals(new USMoney(20001));
    @*/
  public void deposit(MoneyOps amt);

  /*@    public normal_behavior
    @    requires amt != null
    @           && amt.greaterThanOrEqualTo(new USMoney(0))
    @           && credit.can_add(amt);
    @    assignable credit, checking;
    @    ensures checking.equals(\old(checking.plus(amt)))
    @          && \not_modified(savings);
    @ for_example
    @  public normal_example
    @    requires checking.equals(new USMoney(20000))
    @           && amt.equals(new USMoney(1));
    @    ensures checking.equals(new USMoney(20001));
    @*/
  public void depositToChecking(MoneyOps amt);

  /*@  public normal_behavior
    @    requires amt != null;
    @    {|
    @      requires (new USMoney(0)).lessThanOrEqualTo(amt)
    @             && amt.lessThanOrEqualTo(checking);
    @      assignable credit, checking;
    @      ensures checking.equals(\old(checking.minus(amt)))
    @             && \not_modified(savings);
    @     also
    @      requires (new USMoney(0)).lessThanOrEqualTo(amt)
    @             && amt.lessThanOrEqualTo(credit)
    @             && amt.greaterThan(checking);
    @      assignable credit, checking, savings;
    @      ensures checking.equals(new USMoney(0))
    @          && savings.equals(
    @                   \old(savings.minus(amt.minus(checking))));
    @    |}
    @ for_example
    @  public normal_example
    @    requires checking.equals(new USMoney(40001))
    @           && amt.equals(new USMoney(40000));
    @    ensures checking.equals(new USMoney(1))
    @             && \not_modified(savings);
    @ also
    @  public normal_example
    @    requires savings.equals(new USMoney(30001))
    @           && checking.equals(new USMoney(10000))
    @           && amt.equals(new USMoney(40000));
    @    ensures checking.equals(new USMoney(0))
    @          && savings.equals(new USMoney(1));
    @*/
  public void payCheck(MoneyOps amt);
}
```

3 Extensions to Java Expressions

JML makes extensions to the Java expression syntax for two uses. The main set of extensions are used in predicates. But there are also some extensions used in *store-ref*s, which are themselves used in the `assignable`, `accessible`, `depends`, `represents` clauses.

3.1 Extensions to Java Expressions for Predicates

The expressions that can be used as predicates in JML are an extension to the side-effect free Java expressions. Since predicates are required to be side-effect free, the following Java operators are *not* allowed within predicates:

- assignment (`=`), and the various assignment operators (such as `+=`, `-=`, etc.)
- all forms of increment and decrement operators (`++` and `--`),
- calls to methods that are not pure, and
- any use of operator `new` that would call a constructor that is not pure.

Furthermore, within method specification that are not model programs, one cannot use `super` to call a pure superclass method, because it is confusing in combination with JML's specification inheritance.[1]

We allow the allocation of storage (e.g., using operator `new` and pure constructors) in predicates, because such storage can never be referred to after the evaluation of the predicate, and because such pure constructors have no side-effects other than initializing the new objects so created.

JML adds the following new syntax to the Java expression syntax, for use in predicates (see Section B.1.10 [Predicate and Specification Expression Syntax], page 60 for syntactic details such as precedence):

- Informal descriptions, which look like

  ```
  (* some text describing a Boolean-valued predicate *)
  ```

  and have type `boolean`. Their meaning is entirely determined by the reader, and tools will mostly treat them the same as `true`.
- `==>` for logical implication; for example, the formula `raining ==> getsWet` is true if either `raining` is false or `getsWet` is true. The notation `<==` is used for reverse implication. Finally, the notation `<==>` is used for logical equivalence, and `<=!=>` for logical inequivalence. Note that, for expressions of type `boolean`, `<==>` means the same thing as `==`, and `<=!=>` means the same thing as `!=`; however, `<==>` and `<=!=>` have a much lower precedence, and are also associative and symmetric.
- `\forall` and `\exists`, which 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

[1] Suppose A is the superclass of B, and B is the superclass of C. Suppose B's specification used `super` to call a method of A. The problem is that when this specification is inherited by C, if we imagine copying B's specification to C, then this use of *super* no longer refers to A, but to B.

semicolons (`;`). If the range predicate is omitted, it defaults to `true`. Thus, 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.

- `\max`, `\min`, `\product`, and `\sum`, which 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 == Intger.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.

- `\num_of`, which is "numerical quantifier." It 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)
  ```

- Set comprehensions, which 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 (see Section B.1.10 [Predicate and Specification Expression Syntax], page 60) 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].) One may often start from the sets containing the objects of primitive types found in `edu.iastate.cs.jml.models.JMLModelObjectSet` and (in the same Java package) `JMLModelValueSet`.

- `\elemtype`, which returns the most-specific static type shared by all elements of its array argument.
- `\fresh`, which asserts that objects were freshly allocated; for example, `\fresh(x,y)` asserts that the objects bound to `x` and `y` were not allocated in the pre-state.
- `\invariant_for`, which is 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`.
- `\is_initialized`, which is true just when its *reference-type* argument is a class that has finished its static initialization. It is of type `boolean`.
- `\lblneg` and `\lblpos` can be used to attach labels to expressions; these labels might be printed in various messages by support tools, for example, to identify an assertion that failed. One would only write an expression such as

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

  which has value that is the same as `0 <= index && index < length`. 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`.
- `\lockset`, which is the set of locks held by the current thread. It is of type `JMLObjectSet`. (This is an adaptation from ESC/Java [Leino-etal00] for dealing with threads.)
- `\not_modified`, which asserts that the values of objects (and their dependees) are the same in the post-state as in the pre-state; for example, `\not_modified(xval,yval)` says that `xval` and `yval` have the same value in the pre- and post-states (in the sense of an `equals` method). The keyword `\not_modified` can only be used in an *ensures-clause* or a *signals-clause*; it cannot be used, for example, in preconditions.
- `\old`, which can be used to refer to values in the pre-state; e.g., `\old(myPoint.x)` is the value of the `x` field of the object `myPoint` in the pre-state. The keyword `\old` can only be used in an *ensures-clause*, a *signals-clause*, or a *history-constraint*; it cannot be used, for example, in preconditions.
- Several notations using `\reach` allow one to refer to the set of objects reachable from some particular object. The `\reach` syntax is overloaded to reduce the number of keywords. There are three cases, each of which has two alternatives depending on the static type of the first argument:

- 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 the expression *x* has static type `edu.iastate.cs.jml.models.JMLObjectSet`, then `\reach(`*x*`)` denotes the smallest `JMLObjectSet` containing the non-null objects in *x*, if any, and all objects accessible through all fields of objects in this set.
- The syntax `\reach(`*x*`, `*T*`)` denotes the smallest `JMLObjectSet` containing the object denoted by *x*, if such an object exists and has type *T*, and all objects of type *T* accessible through all fields of objects in this set. If *x*, the argument to `\reach` has the static type `edu.iastate.cs.jml.models.JMLObjectSet`, then this syntax denotes the smallest `JMLObjectSet` containing the non-null objects in *x* of type *T*, if any, and all objects accessible through all fields of objects in this set.

  Note that if *x* is a `JMLObjectSet`, it may contain objects of different types; the presence of objects of other types does not matter. Only the instances of type *T* participate, and there need not be any instances of type *T* in the set.
- The syntax `\reach(`*x*`, `*T*`, `*f*`)` denotes the smallest `JMLObjectSet` containing the object denoted by *x*, if such an object exists and has type *T*, and all objects of type *T* accessible using the field *f* on objects in this set. The type *T* must have been declared with a (non-static) field *f*. If *x* has static type `edu.iastate.cs.jml.models.JMLObjectSet`, then this denotes the smallest `JMLObjectSet` containing the objects in *x* that have type *T*, and all objects of type *T* accessible using the field *f* on objects in this set.

  More generally, in this syntax one can use instead of *f*, a *store-ref-expression*. For example, in

  ```
      \reach(myPointSet, ColorPoint, neighbor[3])
  ```

  if `myPointSet` is a `JMLObjectSet`, then this expression denotes the smallest set of objects of type `ColorPoint` such that the objects contained in `myPointSet` of type `ColorPoint` are in the set, and for each object `cp` of type `ColorPoint` in the set, `cp.neighbor[3]` is in the set.

• `\result`, which, in an `ensures` clause is the value or object that is being returned by a method. The keyword `\result` can only be used in an *ensures-clause*; it cannot be used, for example, in preconditions or in signals clauses.

• `\nonnullelements`, which can be used to assert that the elements of an array are all non-null. For example, `\nonnullelements(myArray)`, is equivalent to

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

• `\typeof`, which returns the most-specific static type of an expression. An expression of the form `\typeof(`*E*`)` has type `\TYPE`.[2] For example, `\typeof(true == false)` is the type `boolean`.

[2] The type `\TYPE` stands for the type of all types.

- The operator `<:` which compares two reference types and returns true when the type on the left is a subtype of the type on the right. Although the notation might suggest otherwise, this operator is also reflexive; a type will compare as `<:` with itself.
- `\type`, which can be used to mark types in expressions. An expression of the form `\type(T)` has the type `\TYPE`. For example, in

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

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

As in Java itself, most types are reference types, and hence many expressions yield references (i.e., object identities or addresses), as opposed to primitive values. This means that `==`, except when used to compare pure values of primitive types such as `boolean` or `int`, is reference equality. As in Java, to get value equality for reference types one uses the `equals` method in assertions. For example, the predicate `myString == yourString`, is only true if the objects denoted by `myString` and `yourString` are the same object (i.e., if the names are aliases); to compare their values one must write `myString.equals(yourString)`.

The reference semantics makes interpreting predicates that involve the use of `\old` interesting. We want to have the semantics suited for two purposes:

- execution of assertions for purposes of debugging and testing, as in Eiffel, and
- generation of mathematical assertions for static analysis and possible theorem proving (e.g., to verify program correctness).

The key to the semantics of `\old` is to treat it as an abbreviation for a local definition. That is, E in `\old(`E`)` can be evaluated in the pre-state, and its value bound to a locally defined name, and then the name can be used in the postcondition.

To avoid refering to the value of initialized locations, a constructor's precondition can only refer to locations in the object being constructed that are not assignable. This allows a constructor to refer to instance fields of the object being constructed if they are not made assignable by the constructor's assignable clause, for example, if they are declared with initializers. In particular, the precondition of a constructor may not mention a "blank final" instance variable that it must assign.

Since we are using Java expressions for predicates, there are some additional problems in mathematical modeling. We are excluding the possibility of side-effects by limiting the syntax of predicates, and by using type checking [Gifford-Lucassen86] [Lucassen87] [Lucassen-Gifford88] [Nielson-Nielson-Amtoft97] [Talpin-Jouvelot94] [Wright92] to make sure that only pure methods and constructors may be called in predicates.

Exceptions in expressions are particularly important, since they may arise in type casts. Logically, we originally planned to deal 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, instead of doing this for all subexpressions, we only require this to be done for the smallest boolean-valued subexpression that may throw an exception (and for entire expressions used in JML's *measured-clause* and *variant-function*).

JML will check that errors (i.e., exceptions that inherit from Error) are not explicitly thrown by pure methods. This means that they can be ignored during mathematical modeling. When executing predicates, errors will cause run-time errors.

3.2 Extensions to Java Expressions for Store-Refs

The grammatical production *store-ref* (see Section B.1.5 [Store Ref Syntax], page 54) is used to name locations in the `assignable`, `depends`, `represents` clauses. A similar production for *object-ref* is used in the `accessible` clause. A *store-ref* names a location, not an object; a location is either a field of an object, or an array element. Besides the Java syntax of names and field and array references, JML supports the following syntax for *store-ref*s. See Section B.1.4 [Behavioral Specification Syntax for Types], page 54, for more details on the syntax.

- Array ranges, of the form *A*`[`*E1* `..` *E2*`]`, denote the locations in the array *A* between the value of *E1* and the value of *E2* (inclusive). For example, the clause

  ```
  assignable myArray[3 .. 5]
  ```

  can be thought of an abbreviation for the following.

  ```
  assignable myArray[3], myArray[4], myArray[5]
  ```

- One can also name all the indexes in an array *A* by writing, *A*`[*]`, which is shorthand for *A*`[0 ..` *A*`.length-1]`.
- Several notations using `\fields_of` allow one to refer to the fields and array elements in a set of objects, or in some particular object. The `\fields_of` syntax is overloaded to reduce the number of keywords. There are three cases, each of which has two alternatives depending on the static type of the first argument:
  - The syntax `\fields_of(`*x*`)` names all the non-static fields and array elements of the object(s) referred to by *x*. If *x* has static type `edu.iastate.cs.jml.models.JMLObjectSet`, then this names all the fields and array elements in all the objects in the set *x*, otherwise it simply names all the fields and array elements of the object *x*. For example, if `p` is a `Point` object with two fields, `x` and `y` of type `BigInteger`, then `\fields_of(p)` names the fields `p.x` and `p.y`. Notice that the fields of the `BigInteger` objects are not named. As another example, if `a` is an array of type `Rocket []`, then the *store-ref* `\fields_of(a)` is equivalent to `a[*]`.
  - The syntax `\fields_of(`*x*`,` *T*`)` names all the non-static fields and array elements of *x* in objects of type *T*. If *x* has static type `edu.iastate.cs.jml.models.JMLObjectSet`, then this names all non-static fields of all instances of type *T* (or a subtype) in the set *x*, otherwise *x* must have static type *T* (or a subtype), this *store-ref* names all the non-static fields of *x* found in type *T* (or the array elements of *x*, if *T* is an array type.)

    Note that if *x* is a `JMLObjectSet`, it may contain objects of different types; the presence of objects of other types does not matter. Only the instances of type *T* participate, and there need not be any instances of type *T* in the set.
  - The syntax `\fields_of(`*x*`,` *T*`,` *f*`)` names the *f* fields of *x* in objects of type *T*. The type *T* must have been declared with a (non-static) field *f*. Note that in this case, `T` cannot be an array type. If *x* has static type

`edu.iastate.cs.jml.models.JMLObjectSet`, then this names the *f* fields in all instances of type *T* in the set *x*, otherwise *x* must have static type *T*, this *store-ref* is the same as writing *x* . *f*.

More generally, in this syntax one can use instead of *f*, a *store-ref-expression*. For example, in

```
\fields_of(myPointSet, ColorPoint, val[3].color)
```

if `myPointSet` is of type `JMLObjectSet`, then this expression refers to the locations `cp.val[3].color`, for each object `cp` of type `ColorPoint` in `myPointSet`.

Note that `\reach` is useful for constructing sets of objects for use as the first argument to `\fields_of`. For example, one might write `\fields_of(\reach(myVar))`.

4 Conclusions

One area of future work for JML is concurrency. The main feature currently in JML that supports concurrency is the `when` clause [Lerner91] [Sivaprasad95]; it says that the caller will be delayed until the condition given holds. This permits the specification of when the caller is delayed to obtain a lock, for example. While syntax for this exists in the JML parser, our exploration of this topic is still in an early stage. JML also has several primitives from ESC/Java that deal with monitors and locks.

JML is an expressive behavioral interface specification language for Java. It combines the best features of the Eiffel and Larch approaches to specification. It allows one to write specifications that are quite precise and detailed, but also allows one to write lightweight specifications. It has examples and other forms of redundancy to allow for debugging specifications and for making rhetorical points. It supports behavioral subtyping by specification inheritance.

More information on JML, including software to aid in working with JML specifications, can be obtained from '`http://www.cs.iastate.edu/~leavens/JML.html`'.

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, Baker, and Ruby was also supported in part by NSF grant CCR-9803843. Work on JML by Leavens, Ruby, and others is supported in part by NSF grant CCR-0097907.

Thanks to Bart Jacobs, Rustan Leino, Peter Müller, Erik Poll, 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. For comments on earlier drafts and discussions about JML thanks to Bart, Rustan, Peter, Eric, Joachim, Raymie, Abhay Bhorkar, Curtis Clifton, John Boyland, Martin Büchi, Yoonsik Cheon, David Cok, Gary Daugherty, Michael Ernst, Arthur Fleck, Karl Hoech, Marieke Huisman, Anand Ganapathy, Doug Lea, Yogy Namara, Sevtap Oltes, Arnd Poetzsch-Heffter, Jim Potts, Arun Raghavan, Jim Saxe, Tammy Scherbring, Tim Wahls, Wolfgang Weck, and others we may have forgotten. Thanks to Yoonsik, Curtis, Abhay, Anand, Yogy, and Arun for their work on the prototype JML checker used to check the specifications in this document. Thanks to the students in 22C:181 at the University of Iowa in Spring 2001 for suggestions and comments about JML.

Appendix A Specification Case Defaults

As noted above (see Section 1.2 [Lightweight Specifications], page 4), specifications in JML do not need to be as detailed as most of the examples given in this document. If a *spec-case* or *conjoinable-spec* (see Section B.1.6 [Behavioral Specification Syntax for Methods], page 55) does not use one of the behavior keywords (`behavior`, `normal_behavior`, or `exceptional_behavior`), or if an *example* (see Section B.1.8 [Example Syntax], page 59) does not use one of the example keywords (`example`, `normal_example`, `exceptional_example`), then it is called a *lightweight* specification or example.

When the various clauses of a *spec-case*, *conjoinable-spec*, or *example* are omitted, they have the defaults given in the table below. The table distinguishes between lightweight and non-lightweight specifications and examples. In each case the default for the lightweight form is that no assumption is made about the omitted clause. However, in a non-lightweight specification or example, the specifier is assumed to be giving a complete specification or example. Therefore, in a non-lightweight specification the meaning of an omitted clause is given a definite default. For example, the meaning of an omitted `assignable` clause is that nothing can be modified. Furthermore, in a non-lightweight specification, the meaning of an omitted diverges clause is that the method may not diverge in that case. (The `diverges` clause is almost always omitted; it can be used to say what should be true, of the pre-state, when the specification is allowed to loop forever or signal an error.)

	Default	
Omitted clause	lightweight	non-lightweight
requires	\not_specified	true
when	\not_specified	true
measured_by	\not_specified	\not_specified
assignable	\not_specified	\nothing
ensures	\not_specified	true
signals	(Exception) \not_specified	(Exception) true
diverges	\not_specified	false

A completely omitted specification is taken to be a lightweight specification. Thus one can read off the meaning of a completely omitted specification from the lightweight column of table.

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`.

Note that specification statements (see Section B.1.7 [Model Program Syntax], page 57) cannot be lightweight. In addition, a *spec-statement* can specify abrupt termination. The additional clauses possible in a *spec-statement* have the following defaults.

	Default
Omitted clause	(non-lightweight)
continues	false
breaks	false
returns	false

Appendix B Syntax

We use an extended 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 HTML they are also hyperlinked to their definitions).
- Terminal symbols are written as follows: `terminal`. In a few cases we also quote terminal symbols using ‘ and ’, 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 the nonterminal *name*, which is a non-empty list of *ident*’s separated by periods (`.`).

name ::= *ident* [`.` *ident*] ...

To remind the reader that the notation ‘...’ means zero or more repetitions, we use ‘...’ only following optional text.

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

B.1 Context-Free Syntax

B.1.1 Compilation Unit Syntax

The following is the syntax of compilation units in JML. The *compilation-unit* rule is the start rule for the JML grammar.

compilation-unit ::= [*package-definition*]
    [*refine-prefix*]
    [*import-definition*] ...
    [*type-definition*] ...
package-definition ::= `package` *name* `;`
refine-prefix ::= `refine` *ident-list* `<-` *string-literal* `;`
ident-list ::= *ident* [, *ident*] ...
import-definition ::= [`model`] `import` *name-star* `;`
name ::= *ident* [`.` *ident*] ...
name-star ::= *ident* [`.` *ident*] ... [`.` `*`]

B.1.2 Type Definition Syntax

The following is the syntax of type definitions. The order of the *modifier* productions suggests the relative order for writing the modifiers in code and specifications, with `public` before all other modifiers, and `non_null` last.

type-definition ::= [*doc-comment*] *modifiers* *class-or-interface-def*
    | `;`
class-or-interface-def ::= *class-definition* | *interface-definition*
type-spec ::= *type* [*dims*] | `\TYPE` [*dims*]

type ::= *reference-type* | *builtInType*
reference-type ::= *name*
modifiers ::= [*modifier*] . . .
modifier ::= `public` | `private` | `protected`
| `spec_public` | `spec_protected`
| `abstract` | `static` |
| `model` | `ghost` | `pure`
| `final` | `synchronized`
| `instance`
| `transient` | `volatile`
| `native` | `strictfp`
| `const` | `monitored` | `uninitialized`
| `non_null`
class-definition ::= `class` *ident* [`extends` *name* [`weakly`]]
[*implements-clause*] *class-block*
interface-definition ::= `interface` *ident* [*interface-extends*] *class-block*
interface-extends ::= `extends` *name-weakly-list*
implements-clause ::= `implements` *name-weakly-list*
name-weakly-list ::= *name* [`weakly`] [`,` *name* [`weakly`]] . . .
class-block ::= `{` [*field*] . . . `}`

B.1.3 Field Syntax

The following gives the syntax of fields.

field ::= [*doc-comment*] . . . *modifiers member-decl*
| [*doc-comment*] . . . *modifiers jml-declaration*
| [*method-specification*] [`static`] *compound-statement*
| *method-specification* `static_initializer`
| *method-specification* `initializer`
| `axiom` *predicate* `;`
| `;`
member-decl ::= *variable-decls* `;` | *method-decl*
| *class-definition* | *interface-definition*
variable-decls ::= *type-spec variable-declarators* [*jml-var-assertion*]
variable-declarators ::= *variable-declarator* [`,` *variable-declarator*] . . .
variable-declarator ::= *ident* [*dims*] [`=` *initializer*]
initializer ::= *expression* | *array-initializer*
array-initializer ::= `{` [*initializer-list*] `}`
initializer-list ::= *initializer* [`,` *initializer*] . . . [`,`]
method-decl ::= *method-specification*
[*type-spec*] *method-head method-body*
| [*type-spec*] *method-head*
[*method-specification*]
method-body
method-head ::= *ident* `(` [*param-declaration-list*] `)`
[*dims*] [*throws-clause*]
method-body ::= *compound-statement* | `;`
throws-clause ::= `throws` *name* [`,` *name*] . . .
param-declaration-list ::= *param-declaration* [`,` *param-declaration*] . . .

param-declaration ::= [`final`] *type-spec ident* [*dims*]

B.1.4 Behavioral Specification Syntax for Types

The following gives the syntax of behavioral specifications for types.

jml-var-assertion ::= `initially` *predicate*
    | `readable_if` *predicate*
    | `monitored_by` *spec-expression-list*
jml-declaration ::= *invariant* | *history-constraint*
    | *depends-decl* | *represents-decl*
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* [`.` *ident*] . . .
    | `new` *reference-type*
method-ref-start ::= `super` | `this` | *ident* | `\other`
param-disambig-list ::= *param-disambig* [`,` *param-disambig*] . . .
param-disambig ::= *type-spec* [*ident* [*dims*]]
depends-decl ::= *depends-keyword store-ref* `<-` *store-ref-list* `;`
depends-keyword ::= `depends` | `depends_redundantly`
represents-decl ::= *represents-keyword*
        store-ref `<-` *spec-expression* `;`
    | *represents-keyword*
        store-ref `\such_that` *predicate* `;`
represents-keyword ::= `represents` | `represents_redundantly`

B.1.5 Store Ref Syntax

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*
    | `\fields_of` `(` *spec-expression* [`,` `reference-type` [`,` *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* | '`[`' *spec-array-ref-expr* '`]`'
spec-array-ref-expr ::= *spec-expression*
    | *spec-expression* `..` *spec-expression*
    | `*`
store-ref-keyword ::= `\nothing` | `\everything` | `\not_specified`

B.1.6 Behavioral Specification Syntax for Methods

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
specification ::= spec-case-seq
            [ subclassing-contract ]
            [ redundant-spec ]
        | subclassing-contract
            [ redundant-spec ]
        | redundant-spec
spec-case-seq ::= spec-case [ also spec-case ] . . .
spec-case ::= generic-spec-case | behavior-spec | model-program
extending-specification ::= also specification
        | and conjoinable-spec-seq
            [ subclassing-contract ]
            [ redundant-spec ]
conjoinable-spec-seq ::= conjoinable-spec [ and conjoinable-spec ] . . .
conjoinable-spec ::= generic-conjoinable-spec | behavior-conjoinable-spec
generic-conjoinable-spec ::= [ spec-var-decls ] simple-spec-body
behavior-conjoinable-spec ::= [ privacy ] behavior
            [ spec-var-decls ]
            simple-spec-body
        | [ privacy ] exceptional_behavior
            [ spec-var-decls ]
            exceptional-simple-spec-body
        | [ privacy ] normal_behavior
            [ spec-var-decls ]
            normal-simple-spec-body
privacy ::= public | protected | private
exceptional-simple-spec-body ::= assignable-clause [ assignable-clause ] . . .
           [ signals-clause ] . . .
           [ diverges-clause ] . . .
        | signals-clause [ signals-clause ] . . .
           [ diverges-clause ] . . .
normal-simple-spec-body ::= assignable-clause [ assignable-clause ] . . .
           [ ensures-clause ] . . .
           [ diverges-clause ] . . .
        | ensures-clause [ ensures-clause ] . . .
           [ diverges-clause ] . . .
redundant-spec ::= implications [ examples ] | examples
implications ::= implies_that spec-case-seq
examples ::= for_example example [ also example ] . . .
```

The following is the syntax of generic specification cases. These are the least verbose and most general specification cases.

```
generic-spec-case ::= [ spec-var-decls ] spec-header [ generic-spec-body ]
        | [ spec-var-decls ] generic-spec-body
spec-header ::= requires-clause [ requires-clause ] . . .
           [ when-clause ] . . .
```

```
            [ measured-clause ] ...
        | when-clause [ when-clause ] ...
            [ measured-clause ] ...
        | measured-clause [ measured-clause ] ...
    generic-spec-body ::= simple-spec-body
        | {| generic-spec-case-seq |}
    generic-spec-body-seq ::= generic-spec-case [ also generic-spec-case ] ...
    simple-spec-body ::= assignable-clause [ assignable-clause ] ...
            [ ensures-clause ] ...
            [ signals-clause ] ...
            [ diverges-clause ] ...
        | ensures-clause [ ensures-clause ] ...
            [ signals-clause ] ...
            [ diverges-clause ] ...
        | signals-clause [ signals-clause ] ...
            [ diverges-clause ] ...
        | diverges-clause [ diverges-clause ] ...
```

The following gives the syntax of specification cases that start with one of the `behavior` keywords.

```
    behavior-spec ::= [ privacy ] behavior generic-spec-case
        | [ privacy ] exceptional_behavior exceptional-spec-case
        | [ privacy ] normal_behavior normal-spec-case
    exceptional-spec-case ::= [ spec-var-decls ] spec-header
                [ exceptional-spec-body ]
        | [ spec-var-decls ] exceptional-spec-body
    exceptional-spec-body ::= assignable-clause [ assignable-clause ] ...
            [ signals-clause ] ...
            [ diverges-clause ] ...
        | signals-clause [ signals-clause ] ...
            [ diverges-clause ] ...
        | {| exceptional-spec-case-seq |}
    exceptional-spec-case-seq ::= exceptional-spec-case
            [ also exceptional-spec-case ] ...
    normal-spec-case ::= [ spec-var-decls ] spec-header
            [ normal-spec-body ]
        | [ spec-var-decls ] normal-spec-body
    normal-spec-body ::= assignable-clause [ assignable-clause ] ...
            [ ensures-clause ] ...
            [ diverges-clause ] ...
        | ensures-clause [ ensures-clause ] ...
            [ diverges-clause ] ...
        | {| normal-spec-case-seq |}
    normal-spec-case-seq ::= normal-spec-case [ also normal-spec-case ] ...
```

The following gives the syntax of subclassing contracts.

```
    subclassing-contract ::= subclassing_contract
            accessible-clause [ accessible-clause ] ...
            [ callable-clause ] ...
        | subclassing_contract
```

> *callable-clause* [*callable-clause*] . . .
>
> *accessible-clause* ::= *accessible-keyword object-ref-list* `;`
> *object-ref-list* ::= *object-ref* [`,` *object-ref*] . . .
> | *store-ref-keyword*
> *object-ref* ::= *store-ref-expression*
> | `\other` [*store-ref-name-suffix*] . . .
> *accessible-keyword* ::= `accessible` | `accessible_redundantly`
> *callable-clause* ::= *callable-keyword callable-methods-list* `;`
> *callable-keyword* ::= `callable` | `callable_redundantly`
> *callable-methods-list* ::= *method-name-list* | *store-ref-keyword*

B.1.7 Model Program Syntax

The following gives the syntax of model programs, adapted from the refinement calculus [Back88] [Back-vonWright89a] [Morgan94] [Morris87]. See Section B.1.11 [Statement and Annotation Statement Syntax], page 61 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 signalling 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 ::= *assignable-clause* [*assignable-clause*] . . .
            [*ensures-clause*] . . .
            [*signals-clause*] . . .
            [*diverges-clause*] . . .
            [*continues-clause*] . . .
            [*breaks-clause*] . . .
            [*returns-clause*] . . .
        | *ensures-clause* [*ensures-clause*] . . .
            [*signals-clause*] . . .
            [*diverges-clause*] . . .
            [*continues-clause*] . . .
            [*breaks-clause*] . . .
            [*returns-clause*] . . .
        | *signals-clause* [*signals-clause*] . . .
            [*diverges-clause*] . . .
            [*continues-clause*] . . .
            [*breaks-clause*] . . .
            [*returns-clause*] . . .
        | *diverges-clause* [*diverges-clause*] . . .
            [*continues-clause*] . . .
            [*breaks-clause*] . . .
            [*returns-clause*] . . .
        | *continues-clause* [*continues-clause*] . . .
            [*breaks-clause*] . . .
            [*returns-clause*] . . .
        | *breaks-clause* [*breaks-clause*] . . .
            [*returns-clause*] . . .
        | *returns-clause* [*returns-clause*] . . .

abrupt-spec-case ::= [*spec-var-decls*] *spec-header*
            [*abrupt-spec-body*]
        | [*spec-var-decls*] *abrupt-spec-body*
abrupt-spec-body ::= *assignable-clause* [*assignable-clause*] . . .
            [*diverges-clause*] . . .
            [*continues-clause*] . . .
            [*breaks-clause*] . . .
            [*returns-clause*] . . .
        | *diverges-clause* [*diverges-clause*] . . .
            [*continues-clause*] . . .
            [*breaks-clause*] . . .
            [*returns-clause*] . . .
        | *continues-clause* [*continues-clause*] . . .
            [*breaks-clause*] . . .

[*returns-clause*] . . .
| *breaks-clause* [*breaks-clause*] . . .
[*returns-clause*] . . .
| *returns-clause* [*returns-clause*] . . .
| `{|` *abrupt-spec-case-seq* `|}`
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`

B.1.8 Example Syntax

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 ::= *assignable-clause* [*assignable-clause*] . . .
[*signals-clause*] . . .
[*diverges-clause*] . . .
| *signals-clause* [*signals-clause*] . . .
[*diverges-clause*] . . .
normal-example-body ::= *assignable-clause* [*assignable-clause*] . . .
[*ensures-clause*] . . .
[*diverges-clause*] . . .
| *ensures-clause* [*ensures-clause*] . . .
[*diverges-clause*] . . .

B.1.9 Method Specification Clause Syntax

The following gives the syntax of clauses that occur in method specifications.

spec-var-decls ::= *forall-var-decls* [*let-var-decls*]
| *let-var-decls*
forall-var-decls ::= *forall-var-decl* [*forall-var-decl*] . . .
forall-var-decl ::= `forall` *quantified-var-decl* `;`
let-var-decls ::= `old` *local-spec-var-decl* [*local-spec-var-decl*] . . .
local-spec-var-decl ::= `model` *type-spec* *spec-variable-declarators* `;`
| `ghost` *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`
when-clause ::= *when-keyword pred-or-not* `;`
when-keyword ::= `when` | `when_redundantly`
measured-clause ::= *measured-by-keyword* `\not_specified` `;`
  | *measured-by-keyword spec-expression* [`if` *predicate*] `;`
measured-by-keyword ::= `measured_by` | `measured_by_redundantly`
assignable-clause ::= *assignable-keyword conditional-store-ref-list* `;`
assignable-keyword ::= `assignable` | `assignable_redundantly`
  | `modifiable` | `modifiable_redundantly`
  | `modifies` | `modifies_redundantly`
conditional-store-ref-list ::= *conditional-store-ref*
    [`,` *conditional-store-ref*] . . .
conditional-store-ref ::= *store-ref* [`if` *predicate*]
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`

B.1.10 Predicate and Specification Expression Syntax

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 B.1.12 [Expression Syntax], page 62 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_modified` `(` *store-ref-list* `)`
  | `\fresh` `(` *spec-expression-list* `)`
  | `\reach` `(` *spec-expression* [`,` `reference-type` [`,` *store-ref-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 ::= spec-primary-expr [ . ident ] . . . . has ( ident )
                && predicate

spec-quantified-expr ::= ( quantifier quantified-var-decl ; [ [ predicate ] ; ]
                spec-expression )
quantifier ::= \forall | \exists | \max | \min | \num_of | \product | \sum
quantified-var-decl ::= 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
```

B.1.11 Statement and Annotation Statement Syntax

The following gives the syntax of statements. These are the standard Java statements, with the addition of annotations, the *because-statement*, *assert-redundantly-statement*, *assume-statement*, *set-statement*, and *unreachable-statement*, and the various forms of *model-prog-statement*. See Section B.1.7 [Model Program Syntax], page 57 for the syntax of *model-prog-statement*, which is only allowed in model programs.

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

| *assume-statement*
| *set-statement*
| *unreachable-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 ::= `model` | `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*] `;`

The following gives the syntax of JML annotations that can be used on statements. See Section B.1.7 [Model Program Syntax], page 57, 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 ;`
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`

B.1.12 Expression Syntax

The JML syntax for expressions extends the Java syntax with several operators and primitives. See Section 3.1 [Extensions to Java Expressions for Predicates], page 43 for a brief description of the meaning of the JML syntax added to Java expressions.

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 Section B.1.10 [Predicate and Specification Expression Syntax], page 60). 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-opt 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
builtInType ::= 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 ‘]’ ] . . .
dims ::= ‘[’ ‘]’ [ ‘[’ ‘]’ ] . . .
array-initializer ::= { [ initializer [ , initializer ] . . . [ , ] ] }
```

initializer ::= *expression*
    | *array-initializer*

B.1.13 Deprecated Syntax

The following describes *deprecated* syntax; that is syntax that is currently allowed, but which will not be legal in the future. The JML parser warns the user when deprecated syntax is used.

The following syntax for *let-var-decls* that uses the keyword `let` is deprecated. Such code should use the keyword `old` instead.

let-var-decls ::= `let` *local-spec-var-decl* [*local-spec-var-decl*] . . .

B.2 Microsyntax or Lexical Grammar

Throughout the figures for the lexical grammar below, grammatical productions are to be understood lexically; that is, this grammar concerns individual characters, not tokens. Another way of thinking of this is that no *white-space* may intervene between the characters of a token.

The microsyntax of JML is described by the production *microsyntax* in the grammar 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*

B.2.1 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 are special in that they cannot appear in some contexts where other whitespace can appear, and are also used to end C++-style (`//`) comments. This is described formally below.

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

B.2.2 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*] . . .

B.2.3 Comments

Both kinds of Java comments are allowed in JML: old C-style comments and new 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-star-slash*] . . .
  `|` `+` *non-at* [*non-star-slash*] . . .
  `|` stars-non-slash [non-star-slash] . . .
non-star-slash ::= *non-star*
  `|` *stars-non-slash*
stars-non-slash ::= `*` [`*`] . . . *non-slash*
non-at-plus-star ::= any character except `@`, `+`, or `*`
non-at ::= any character except `@`
non-star ::= any character except `*`
non-slash ::= any character except `/`
C-style-end ::= [`*`] . . . `*/`
C++-style-comment ::= `//` [`+`] *end-of-line*
  `|` `//` *non-at-plus-newline* [*non-newline*] . . . *newline*
  `|` `//+` *non-at* [*non-newline*] . . . *newline*
non-newline ::= any character except a newline or carriage return
non-at-plus-newline ::= any character except `@`, `+`, newline, or carriage return

B.2.4 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 newline, and between pairs of `/*@` and `@*/` as *annotations*.

Annotations must hold entire grammatical units of JML specifications. For example the following is illegal, because the *postcondition* is split over two annotations, and thus each contains a fragment instead of a complete grammatical unit.

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

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 (i.e., ignore) the character sequences that are *annotation-markers*: `//@`, `//+@`, `/*@`, `/*+@`, and `@+*/`, `@*/`. The at-sign (`@`) in `@*/` is optional, and more than one at-sign may appear in the other annotation markers. However, JML will recognize *jml-keywords* only within annotations.

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

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

B.2.5 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 (i.e., not `*`) character on a line determines its interpretation. In particular, this means that the *jml-pre* and *pre-jml* tokens that start and end the *jml-specs* portion of a documentation coment are only recognized at the beginning of a line (following any leading `*` and whitespace).

> *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* ::= *pre-jml* [*method-specification*] *jml-pre*
>   [*pre-jml* [*method-specification*] *jml-pre*] . . .

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*, is 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-newline* [*non-end-of-line*] . . .
  however, the start of the line must not match *pre-jml* or *jml-pre*
pre-jml ::= *pre-tag* [*doc-non-nl-ws*] . . . *jml-tag*
jml-pre ::= *end-jml-tag* [*doc-non-nl-ws*] . . . *end-pre-tag*
pre-tag ::= `<pre>` | `<PRE>`
end-pre-tag ::= `</pre>` | `</PRE>`
jml-tag ::= `<jml>` | `<JML>` | `<esc>` | `<ESC>`
end-jml-tag ::= `</jml>` | `</JML>` | `</esc>` | `</ESC>`

B.2.6 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-keyword*s. 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-keywords* represents the reserved words in Java (as in the JDK version 1.4). JML keywords are only recognized as such if they occur outside of a *spec-expression* but within either an annotation or a file that is a JML file (with suffixes '`.jml`' '`.jml-refined`', or '`.refines-jml`'). JML predicate keywords are, as their name implies, used within *spec-expression*s; they are also used in *store-ref-list*s and *constrained-list*s. The details are given below.

keyword ::= *java-keyword* | *jml-predicate-keyword* | *jml-keyword*
jml-predicate-keyword ::= `\elemtype` | `\everything`
   | `\exists` | `\fields_of` | `\forall`
   | `\fresh` | `\invariant_for` | `\is_initialized`
   | `\lblneg` | `\lblpos` | `\lockset`
   | `\max` | `\min` | `\nonnullelements`
   | `\nothing` | `\not_modified` | `\not_specified`
   | `\num_of` |`\old` | `\other`
   | `\product` | `\reach` | `\result`
   | `\such_that` | `\sum` | `\type`
   | `\typeof` | `\TYPE`
jml-keyword ::= `abrupt_behavior` | `accessible_redundantly` | `accessible`
   | `also` | `and` | `assert_redundantly`

```
| assignable_redundantly | assignable
| assume_redundantly | assume | axiom
| behavior | breaks_redundantly | breaks
| callable_redundantly | callable | choose_if
| choose | constraint_redundantly | constraint
| continues_redundantly | continues
| decreases_redundantly | decreases | decreasing_redundantly
| decreasing | depends_redundantly | depends
| diverges_redundantly | diverges | ensures_redundantly
| ensures | example | exceptional_behavior
| exceptional_example | exsures_redundantly | exsures
| forall | for_example | ghost
| implies_that | hence_by_redundantly | hence_by
| initializer | initially | instance
| invariant_redundantly | invariant | let
| loop_invariant_redundantly | loop_invariant
| maintaining_redundantly | maintaining
| measured_by_redundantly | measured_by
| model_program | model | modifiable_redundantly
| modifiable modifies_redundantly | modifies
| monitored_by | monitored | non_null
| normal_behavior | normal_example | nowarn
| old | or | post | pre
| pure | readable_if | refine
| represents_redundantly | represents | requires_redundantly
| requires | returns_redundantly | returns
| set | signals_redundantly | signals
| spec_protected | spec_public | static_initializer
| subclassing_contract | uninitialized | unreachable
| weakly | when_redundantly | when
```

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

special-symbol ::= *java-special-symbol* | *jml-special-symbol*
java-special-symbol ::= *java-separator* | *java-operator*
java-separator ::= `(` | `)` | `{` | `}` | ‘`[`’ | ‘`]`’ | `;` | `,` | `.`
java-operator ::= `=` | `<` | `>` | `!` | `~` | `?` | `:`
    | `==` | `<=` | `>=` | `!=` | `&&` | ‘`||`’ | `++` | `--`
    | `+` | `-` | `*` | `/` | `&` | ‘`|`’ | `^` | `%` | `<<` | `>>` | `>>>`
    | `+=` | `-=` | `*=` | `/=` | `&=` | ‘`|=`’ | `^=` | `%=` | `<<=` | `>>=` | `>>>=`
jml-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 | 0X hex-digit
        | hex-numeral 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-numeral hex-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. The exact syntax is given below.

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

Bibliography

[Arnold-Gosling98]
Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[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. Technical Report Ser. A, No 92, Abo Akademi University, Department of Computer Science, Lemminkäinengatan 14, 20520 Abo, Finland, 1989. Appears in *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*, Mook, The Netherlands, May/June 1989, Spring-Verlag, LNCS 430, J. W. de Bakker, et al, (eds.), pages 42–66.

[Back-Mikhajlova-vonWright98]
Ralph Back, Anna Mikhajlova, and Joakim von Wright. Modeling component environments and interactive programs using iterative choice. Technical Report 200, Turku Centre for Computer Science, September 1998.
'`http://www.tucs.abo.fi/publications/techreports/TR200.html`'.

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

[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`'.

[Borgida-Mylopoulos-Reiter95]
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.

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

[Dhara-Leavens96]
Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 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`'.

[Ernst-etal01]
Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin Dynamically Discovering Likely Program Invariants to Support Program Evolution *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[Finney96] Kate Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.

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

[Gifford-Lucassen86]
David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, pages 28–38. ACM, August 1986.

[Gosling-Joy-Steele96]
James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[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-Horning93]
John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, N.Y., 1993.

[Hayes93] I. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, Inc., second edition, 1993.

[Hoare69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

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

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

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

[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 Conference Proceedings*, volume 33, number 10 of *ACM SIGPLAN Notices*, pages 329–340. October 1998.

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

[Jonkers91]
H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM '91 Formal Software Development Methods 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, Volume 1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*, pages 428–456. Springer-Verlag, New York, N.Y., October 1991.

[Lano-Haughton94]
K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies.* The Object-Oriented Series. Prentice Hall, New York, N.Y., 1994.

[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*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

[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.

[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-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.

[Ledgard80] Henry. F. Ledgard. A human engineered variant of BNF. *ACM SIGPLAN Notices*, 15(10):57–62, October 1980.

[Leino95a] K. Rustan M. Leino. A myth in the modular specification of programs. Technical Report KRML 63, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, November 1995. Obtain from the author, at rustan.leino@compaq.com.

[Leino95] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[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`'.

[Lerner91] Richard Allen Lerner. Specifying objects of concurrent systems. Ph.D. Thesis CMU-CS-91-131, School of Computer Science, Carnegie Mellon University, May 1991.

[Liskov-Wing94] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Lucassen87] John M. Lucassen. Types and effects: Towards the integration of functional and imperative programming. Technical Report TR-408, Massachusetts Institute of Technology, Laboratory for Computer Science, August 1987.

[Lucassen-Gifford88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif.*, pages 47–57. ACM, January 1988.

[Luckham-vonHenke85] David Luckham and Friedrich W. von Henke. An overview of anna - a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[Luckham-etal87] David Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA - A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, N.Y., 1987.

[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.

[Morgan94]
Carroll Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[Morgan-Vickers94]
Carroll Morgan and Trevor Vickers, editors. *On the refinement calculus*. Formal approaches of computing and information technology series. Springer-Verlag, New York, N.Y., 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.

[Mueller01]
Peter Müller. Modular Specification and Verification of Object-Oriented Programs. Ph.D. Dissertation, Fernuniversität Hagen, Germany. To appear, 2001.

[Nielson-Nielson-Amtoft97]
H. R. Nielson, F. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In M. Dam, editor, *Proceedings of the Fifth LOMAPS Workshop*, number 1192 in *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Ogden-etal94]
William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct 1994.

[Owre-etal95]
Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[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 Univeristy of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, November 2000.

[Poetzsch-Heffter97]
Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[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`'.

[Sivaprasad95]
Gowri Sivaprasad. Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces. Technical Report 95-27a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 1995.

[Spivey92] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, New York, N.Y., second edition, 1992.

[Talpin-Jouvelot94]
Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[Tan94] Yang Meng Tan. Interface language for supporting programming styles. *ACM SIGPLAN Notices*, 29(8):74–83, August 1994. Proceedings of the Workshop on Interface Definition Languages.

[Tan95] Yang Meng Tan. *Formal Specification Techniques for Engineering Modular C Programs*, volume 1 of *Kluwer International Series in Software Engineering.* Kluwer Academic Publishers, Boston, 1995.

[Wahls-Leavens-Baker00]
Tim Wahls, Gary T. Leavens, and Albert L. Baker. Executing Formal Specifications with Concurrent Constraint Programming. *Automated Software Engineering*, 7(4):315-343, December, 2000.

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

[Whitehead-Russell25]
A. N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, London, second edition. edition, 1925.

[Wills94] Alan Wills. Refinement in Fresco. In Lano and Houghton [Lano-Haughton94], chapter 9, pages 184–201.

[Wing87] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[Wing90a] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, September 1990.

[Wing83] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Woodcock-Davies96]
Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, International Series in Computer Science, 1996.

[Wright92] Andrew K. Wright. Typing references by effect inference. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer-Verlag, New York, N.Y., 1992.

Example Index

Concept Index