Chapter 7
Formal Specification with the Java Modeling Language

Marieke Huisman, Wolfgang Ahrendt, Daniel Grahl, and Martin Hentschel

This text is a general, self contained, and tool independent introduction into the *Java Modeling Language*, JML. It appears in a book about the KeY approach and tool, because JML is the dominating starting point of KeY style Java verification. However, this chapter does not depend on KeY, nor any other specific tool, nor on any specific verification methodology. With this text, the authors aim to provide, for the time being, the definitive, general JML tutorial.

Other chapters in this book discuss the particular usage of JML in KeY style verification.[1] In this chapter, however, we only refer to KeY in very few places, without relying on it. This introduction is written for all readers with an interest in formal specification of software in general, and anyone who wants to learn about the JML approach to specification in particular. A preliminary version of this chapter appeared as a technical report [Huisman et al., 2014].

Introduction

The *Java Modeling Language*, JML, is an increasingly popular specification language for Java software, that has been developed as a community effort since 1999. The nature of such a project entails that language details change, sometimes rapidly, over time and there is no ultimate reference for JML. Fortunately, for the items that we address in this introduction, the syntax and semantics are for the greatest part already settled by Leavens et al. [2013]. Basic design decisions have been described in [Leavens et al., 2006b],[2] who outline these three overall goals:

- "JML must be able to document the interfaces and behavior of existing software, regardless of the analyses and design methods to create it. [...]

[1] Chapter 8 defines a translation of the JML variant supported by KeY into Java dynamic logic, and thereby defines a (translational) semantics of JML. Appendix A provides a language reference for the exact JML variant supported by KeY, presenting syntax, as well as more details on the semantics. Chapter 9 is entirely dedicated to *modular* specification and verification using JML and KeY. Chapter 16 is a tutorial on KeY, using JML in a very intuitive manner only.

[2] This 2006 journal publication is a revised version of a technical report that first appeared in 1998.

- The notation used in JML should be readily understandable by Java programmers, including those with only standard mathematical training. [...]
- The language must be capable of being given a rigorous formal semantics, and must also be amenable to tool support."

This essentially means two things to the specification language: Firstly, it needs to express properties about the special aspects of the Java language, e.g., inheritance, object initialization, or abrupt termination. Secondly, the specification language itself heavily relies on Java; its syntax extends Java's syntax and its semantics extend Java's semantics. The former makes it convenient to talk about such features in a natural way, instead of defining auxiliary constructs or instrumenting the code as in other specification methodologies. The latter can also come in handy since, with a reasonable knowledge of Java, little theoretical background is needed in order to use JML. This has been one of the major aims in the design of JML. It however bears the problem that reasoning about specifications in a formal and abstract way becomes more difficult as even simple expressions are evaluated w.r.t. the complex semantics of Java.

History and Background

Assertions in source code to prove correctness of the implementation have already been proposed long time ago by Floyd [1967]. However, assertions were not widely used in practice—the `assert` statement in Java only first appeared in version 1.4, in 2002. Other programming languages adopted assertions earlier: Bertrand Meyer introduced the concept of *Design by Contract* (DbC) in 1986 with the Eiffel language [Meyer, 1992, 1997]. DbC is a programming methodology where the behavior of program components is described as a *contract* between the provider and the clients of the component. The client only has to study the component's contract, and this should tell him or her exactly what he or she can expect from the component. The provider is free to choose any implementation, as long as it respects the component's contract. Design by Contract has become a popular methodology for object-oriented languages. In this case, the components are the program's classes. Contracts naturally correspond with the object-oriented paradigm to hide (or encapsulate) the internal state of an object.

The Eiffel compiler came with a special option to check validity of a contract at runtime. Subsequently, the same ideas where applied to reason about other programming languages (including Modula-3, C++, and Smalltalk, that were all handled in the Larch project [Guttag and Horning, 1993, Leavens and Cheon, 1993]). With the growing popularity of Java, several people decided to develop a specification language for Java. Gary T. Leavens and his students at Iowa State University used their experience from the Larch project, and started work on a DbC specification language for Java in 1998. They proposed a specification language, and simultaneously developed a JML runtime assertion checker, that could be used to validate the contracts at runtime. At more or less the same time, K. Rustan M. Leino and his team at the DEC/Compaq research center started working on a tool for static

code analysis. For the *Extended Static Checker for Java*, ESC/Java [Leino et al., 2000], they developed a specification language that was more or less a subset of JML. A successor, ESC/Java2 [Cok and Kiniry, 2005], finally adopted JML as it is now. Several projects have been targeting tool supported formal verification of Java programs: the LOOP project [van den Berg and Jacobs, 2001], the Krakatoa project [Marché et al., 2004], and of course KeY. While in KeY originally specifications had been written in the Object Constraint Language (OCL), that is part of UML, from version 0.99 (released in 2005) on, JML has been the primary input language.

Ever since, the community has worked on adopting a single JML language, with a single semantics—and this is still an ongoing process. Over the years, JML has become a very large language, containing many different specification constructs, some of which are only sensible in a single analysis technique. Because of the language being so large, not for all constructs the semantics is actually understood and agreed upon, and moreover all tools that support JML in fact only support a subset of it. There have been several suggestions of providing a formal semantics [Jacobs and Poll, 2001, Engel, 2005, Darvas and Müller, 2007, Bruns, 2009], but as of 2015, there is no final consensus. Moreover, JML suffers from the lack of support for current Java versions; currently there are no specifications for Java 5 features, such as enums or generic types. Dedicated expressions to deal with enhanced foreach loops have been proposed by Cok [2008].

How to Read this Chapter

When introducing JML, we mix a top-down and a bottom-up approach. At first, we introduce the probably most important concept of JML (and similar languages), method contracts, in a high-level manner (Section 7.1). We then jump to the most elementary building blocks of JML specifications, JML expressions (Section 7.2), which are needed to discuss method contracts in more detail (Section 7.3). Then, we lift the granularity of contracts from to the method to the class level (Section 7.4). After discussing the treatment of the `null` reference, and of exceptions (Sections 7.5,7.6), we turn to measures for increasing the separation of specification and implementation, specification-only fields, methods, and variables (Section 7.7). Subtle complications of the integer semantics deserve their own, brief discussion (Section 7.8). Finally, we show that JML is not only used to specify desired behavior, but also to support the verification process through auxiliary assertions (Section 7.9). An overview of JML tools and a comparison with other specification languages (Section 7.10) conclude this tutorial.

During the course of this chapter, the reader may want to experiment with the examples (available from www.key-project.org/thebook2), using various tools, like KeY and OpenJML, among others. This is strongly encouraged. However, there are differences, unfortunately, concerning which language features and library methods are supported, and different restrictions on the way JML is written. Some of these difference are a bit arbitrary, others are more fundamental. For instance, runtime verification tools impose restrictions on JML which are not present in static verification

tools, e.g., that invariants and postconditions have to be executable. The reader may therefore encounter examples that cannot be (without changes) processed by every JML tool.

7.1 Introduction to Method Contracts

Specifications, whether they are presented in natural language or some formalism, can express properties about system artifacts on various levels of granularity; like for instance the overall system, some intermediate level, like architectural components, or, on an even finer level of granularity, source code units. JML is designed for *unit* specification. In Java, those units are:

- methods, where JML specifies the effect of a single method invocation;
- classes, where JML merely specifies constraints on the internal structure of an object; and
- interfaces, where JML specifies the external behavior of an object.

Specifications of these units serve as *contracts* for their implementers, fixing what they can rely upon, and what they have to deliver in return, following the aforementioned Design by Contract paradigm.

We start by introducing method specifications in this section. While we go along, we will also introduce more general concepts, such as JML expressions, that are later used for class and interface specifications as well.

7.1.1 Clauses of a Contract

Contracts of methods are an agreement between the *caller* of the method and the *callee*, describing what guarantees they provide to each other. More specifically, it describes what is expected from the code that calls the method, and it provides guarantees about what the method will actually do. While in our terminology, 'contract' refers to the complete behavioral specification, written JML specifications usually consist of *specification cases*.[3] These specification cases are made up of several *clauses*.

The expectations on the caller are called the *preconditions* of the method. Typically, these will be conditions on the method's parameters, e.g., an argument should be a nonnull reference; but the precondition can also describe that the method should only be called when the object is in a particular state. In JML, each precondition is preceded by the keyword `requires`, and the conjunction of all requires clauses forms the method's precondition. We would like to emphasize that it is not the method

[3] In the context of KeY, what is called a *contract* approximately corresponds to a specification case in JML. What is called 'the contract' in JML (i.e., the complete specification) is considered as a set of multiple contracts for the same target in KeY. For details see Section 8.2.4.

implementer's responsibility to check or handle a violation of the precondition. Instead, this is the responsibility of the caller, and the whole point of contracts is to make this distribution of responsibilities explicit, and checkable. Having said that, it can be a difficult design decision when the caller should be responsible for 'good' parameters and prestates, and when the called method should check and handle this itself. We refer to Section 7.1.2 for a further discussion of *defensive* versus *offensive* specifications and implementations.

The guarantees provided by a method are called the *postcondition* of the method. They describe how the system state is changed by the method, or what the expected return value of the method is. A method only guarantees its postcondition to hold whenever it is called in a state that respects the precondition. If it is called in a state that does not satisfy the precondition, then no guarantee is made at all. In JML, every postcondition expression is preceded by the keyword **`ensures`**, and the conjunction of all ensures clauses forms the method's postcondition.

JML specifications are written as special comments in the Java code, starting with `/*@` or `//@`. The `@` symbol allows the JML parser to recognize that the comment contains a JML specification. Sometimes, JML specifications are also called *annotations*, because they annotate the program code. Preconditions and postconditions are basically just Java expressions (of Boolean type). This is done on purpose: if the specifications are written in a language that the programmer is already familiar with, they are easier for him or her to write and to read. JML extends Java's syntax; almost every side effect free Java expression, i.e., that does not modify the state and has no observable interaction with the outside world, (see [Gosling et al., 2013]) is also a valid JML expression. See Section 7.2 for a detailed discussion of JML expressions.

Example 7.1. Figure 7.1 contains an example of a basic JML specification. It contains specification cases for the methods in an interface `Student`, modeling a typical student at some university.

We discuss the different aspects of this example in full detail. To specify a certain method with JML, requires and ensures clauses are placed immediately before that method, within a JML comment, starting with `/*@` or `//@`. For instance, the method `changeStatus` is specified in JML using two pre- and two postconditions.

The `@` symbol is not only used at the beginning of a JML comment, but possibly also at the beginning of each line of the JML specification, and before the `*/`. This is not necessary, but helps to highlight the JML specifications better. In general, an `@` is ignored within a JML annotation if it is the first (nonwhite) character in the line, or if it is the last character before '`*/`.'

The requires and ensures clauses always consist of the keyword **`requires`** or **`ensures`**, respectively, followed by a Boolean expression. Note that a specification case must at least contain one **`ensures`** clause and that **`requires`** clauses may only appear at the beginning of a specification case.

For method `getName`, we specify that it is a **`pure`** method, i.e., it may not have any (visible) side effects. Also, it must terminate unconditionally (possibly with an exception). Only pure methods may be used in specification expressions, because these should not have side effects, and always terminate.

```
1  public interface Student {
2
3      public static final int bachelor = 0;
4      public static final int master = 1;
5
6      public /*@ pure @*/ String getName();
7
8      //@ ensures \result == bachelor || \result == master;
9      public /*@ pure @*/ int getStatus();
10
11     //@ ensures \result >= 0;
12     public /*@ pure @*/ int getCredits();
13
14     //@ ensures getName().equals(n);
15     public void setName(String n);
16
17     /*@ requires c >= 0;
18       @ ensures getCredits() == \old(getCredits()) + c;
19       @*/
20     public void addCredits(int c);
21
22     /*@ requires getCredits() >= 180;
23       @ requires getStatus() == bachelor;
24       @ ensures getCredits() == \old(getCredits());
25       @ ensures getStatus() == master;
26       @*/
27     public void changeStatus();
28
29
30 }
```

Listing 7.1 First JML example specification

Method `getStatus` is also specified as being pure. In addition, we specify that its result may only be one of two values: `bachelor` or `master`. To denote the return value of the method, the reserved JML keyword `\result` is used.

For method `getCredits` we also specify that it is pure, and in addition we specify that its return value must be nonnegative; a student thus never can have a negative amount of credits.

Method `setName` is nonpure, i.e., it may have side effects. Its postcondition is expressed in terms of the pure methods `getName` and `equals`: it ensures that after termination the result of `getName` is equal to the parameter `n`.

Method `addCredits`'s precondition states a condition on the method parameters, namely that only a positive number of credits can be added. The postcondition specifies how the credits change. Again, this postcondition is expressed in terms of a pure method, namely `getCredits`. Notice the use of the keyword `\old`. An expression `\old(`E`)` in the postcondition actually denotes the value of expression E in the state where the method call started, the *prestate* of the method. Thus the postcondition of `addCredits` expresses that the number of credits only increases:

after evaluation of the method, the value of `getCredits` is equal to the old value of `getCredits`, i.e., before the method was called, plus the parameter `c`.

Method `changeStatus`'s precondition specifies that this method only may be called when the student is in a particular state, namely when they have obtained a sufficient amount of credits to pass from the Bachelor status to the Master status. Moreover, the method may only be called when the student is still having a Bachelor status. The postcondition expresses that the number of credits is not changed by this operation, but the status is. Notice that the two preconditions and the two postconditions of `changeStatus` are written as separate **`requires`** and **`ensures`** clauses, respectively. Implicitly, these are each joined in conjunction, thus the specification is equivalent to the following specification:

```
/*@ requires getCredits() >= 180 &&
  @          getStatus() == bachelor;
  @ ensures getCredits() == \old(getCredits()) &&
  @         getStatus() == master;
  @*/
public void changeStatus();
```

The reader might have wondered why not all method specifications in `Student` have a pre- and a postcondition. Implicitly though, they have. For every specification clause, there is a default. For pre- and postconditions this is the predicate **`true`**, i.e., no constraints are placed on the caller of the method, or on the method's implementation.

Example 7.2. Thus for example the specification of method `getStatus` actually is the following:

—— Java + JML ——

```
/*@ requires true;
  @ ensures status == bachelor || status == master;
  @*/
public int getStatus() {
   return status;
}
```

—— Java + JML ——

7.1.2 Defensive Versus Offensive Method Implementations

An important point about method contracts is that they can be used to avoid *defensive* programming. Consider the specification of method `addCredits` in Listing 7.1,

This method assumes that its argument is nonnegative, and otherwise it is not going to function correctly. When one uses a defensive programming style, one would first test the value of the argument and throw an exception if this was negative. This clutters up the code, and in many cases it is not necessary. Instead, using specifications, one can use an 'offensive' coding style. The specification states what

the method requires from its caller. It only guarantees to function correctly if the caller also fulfills its part of the contract. When validating the application, one checks that every call of the method is indeed within the bounds of its specification, and thus the explicit test in the code is not necessary. Thus, making good use of specifications can avoid adding many parameter checks in the code. Such checks are only necessary when the parameters cannot be controlled—for example, because they are given via an external user.

7.1.3 Specifications and Implementations

Method specifications are written independently of possible implementations. Classes that implement this interface may choose different implementations, as long as they respect the specification. Method specifications do not always have to specify the exact behavior of a method; they give minimal requirements that the implementation should respect.

Example 7.3. Considering the specification in Listing 7.1 again, the method specification for `changeStatus` prescribes that the credits may not be changed by this method. However, method `addCredits` is free to update the status of the student. So for example, an implementation that silently updates the status from Bachelor to Master is appropriate according to the specification. The specification case is repeated here for understandability and that it is not required and recommended to copy specifications of interfaces in classes that realize them.

Java + JML

```
/*@ requires c >= 0;
  @ ensures getCredits() == \old(getCredits()) + c;
  @*/
public void addCredits(int c) {
   credits = credits + c;
   if (credits >= 180) {status = master;}
}
```

Java + JML

According to the specification, both `addCredits` and `changeStatus` would be free to change the name of the student, even though we would typically not expect this to happen. A way to avoid this, is to add explicitly conditions `getName()==\old(getName())` to all postconditions. Later, in Section 7.9.1, we will see how **`assignable`** clauses can be used to explicitly disallow these unwanted changes in a more convenient way.

7.2 Expressions

We have already seen that standard Java expressions can be used in JML specifications. These expressions have to be side effect free, thus for example assignments, or increment/decrement operators, are not allowed. As also mentioned above, JML expressions may contain method calls to pure methods.

In addition, JML defines several specification-specific constructs, to be used in expressions. The use of the `\result` and `\old` keywords has already been demonstrated in Listing 7.1, and the official language specification contains a few more of these. Besides Java's logical operators, such as conjunction `&`, disjunction `|` and negation `!`, also other logical operators are allowed in JML specifications, e.g., implication `==>`, and logical equivalence `<==>`. Since expressions are not supposed to have side effects or terminate exceptionally, in JML in many cases the difference between logical operators such as `&` and `|`, and short circuit operators, such as `&&`, and `||` is not important. However, sometimes the short circuit operators have to be used to ensure an expression is well-defined. For instance, `y != 0 & x/y == 5` may not be a well-defined expression, while `y != 0 && x/y == 5` is.

7.2.1 Quantified Boolean Expressions

For specifying interesting properties, purely propositional Boolean expressions are too limited. How could one for instance express any of the following properties with just propositional connectors?

- An array `arr` is sorted.
- The variable `m` holds the maximum entry of array `arr`.
- All `Account` objects in an array `allAccounts` are stored at the index corresponding to their respective `accountNumber` field.

Given that the arrays in these examples have a statically unknown length, propositional connectives are not enough to express any of the above. What we need here is *quantification*. For that, Boolean JML expressions are extended by the following constructs.[4]

- `(\forall` *T x*`;` *b*`)`
  'for all *x* of type *T*, *b* holds'
- `(\forall` *T x*`;` *a*`;` *b*`)`
  'for all *x* of type *T fulfilling a*, *b* holds'
- `(\exists` *T x*`;` *b*`)`
  'there exists an *x* of type *T* such that *b* holds'
- `(\exists` *T x*`;` *a*`;` *b*`)`
  'there exists an *x* of type *T fulfilling a*, such that *b* holds'

[4] The JML keywords `\forall` and `\exists` correspond to ∀ and ∃ in textbook notation.

Here, T is a Java (primitive or reference) type, x is any name (hereby declared to be of type T), and a and b are Boolean JML expressions. The a is called *range predicate*. The two forms using a range predicate are not strictly needed, as they can be expressed without. `(\forall` T x`;` a`;` b`)` is logically equivalent to `(\forall` T x`;` a `==>` b`)`, and `(\exists` T x`;` a`;` b`)` is logically equivalent to `(\exists` T x`;` a `&&` b`)`. However, the range predicates have a certain pragmatics not shared by their logical counterparts. In `(\forall` T x`;` a`;` b`)`, as well as in `(\exists` T x`;` a`;` b`)`, the Boolean expression a is used intuitively to restrict range of x further than T does.

Example 7.4. Using quantifiers, we can specify that an array should be sorted, for instance in a precondition for a logarithmic `lookup` method that assumes sorting.

JML
```
//@ requires (\forall int i, j;
//@                   0 <= i & i < j & j < a.length;
//@                   a[i] <= a[j]);
public int lookup(int elem) {...
```
JML

The first argument `int i,j` is the declaration of the variables over that the quantification ranges. The second argument `0 <= i & i < j & j < a.length` defines the range of the values for this variable, and the third argument is the actually universally quantified formula (`a[i] <= a[j]` in this case).

Example 7.5. An alternative, but less preferred, way to phrase the specification in Example 7.4 is the following:

JML
```
//@ requires (\forall int i, j;
//@           0 <= i & i < j & j < a.length ==> a[i] <= a[j]);
public int lookup(int elem) {...
```
JML

Besides supporting readability, the range predicate form helps certain JML tools to 'execute' quantified formulas where possible. This is less important for theorem provers, like KeY. But a runtime verification tool would need to operationalize the precondition, by looping through all `i,j` fulfilling `0<=i & i<j & i<a.length`, instead of looping through all `i,j` between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`.

Example 7.6. To specify that a method returns the index of an integer array `arr` holding the maximum entry, we can write the following postcondition.

JML
```
//@ ensures (\forall int i; 0 <= i &&
//@          i < arr.length; \result >= arr[i]);
```
JML

But is that enough? (The reader may briefly reflect before reading on.) This single line only specifies that the result is larger than any other element. An implementation always returning `Integer.MAX_VALUE` would satisfy the above postcondition[5]. We therefore need an additional postcondition that states that the result is actually an element of the array:

JML
```
//@ ensures arr.length > 0 ==>
//@    (\exists int i; 0<=i && i<arr.length; \result==arr[i]);
```
JML

Example 7.7. The following Boolean JML expressions say that all `Account` objects in an array `allAccounts` are stored at the index corresponding to their respective `accountNumber` field.

JML
```
(\forall int i; 0 <= i && i < allAccounts.length;
                allAccounts[i].accountNumber == i)
```
JML

Such an expression could for instance be used in an invariant, see Section 7.4.1.

7.2.2 Numerical Comprehensions

In addition to the Boolean quantified expressions, JML offers so called *generalized quantifiers* `\sum`, `\product`, `\min`, `\max`, and `\num_of`. Those are actually numerical comprehensions (or higher-order functions) with bound variables; see Section 2.3.1. The postcondition in Example 7.6 can alternatively be given as:

```
//@ ensures \result ==
//@         (\max int i; 0 <= i && i < arr.length; arr[i]);
```

The above is syntactically similar to a quantified formula: the `\max` operator binds a variable `i`, and a Boolean guard expression restricts it to be within the range of the array's indices. The type of the `\max` expression is the type of its body; here it is `int`. The intuitive semantics is obviously that the result is the maximum of all `arr[i]` where `i` is in the array range. However, the `\max` construct is not total, i.e., it is not always a well-defined expression. In case `arr` has zero length, for instance, there is no maximum. A similar case appears with a noncompact range, e.g., the set of all mathematical integers (represented by the JML type `\bigint`, see Section 7.8): `(\max \bigint i; true; i)`.

Another comprehension operator is the summation operator `\sum`, of which we make use in Example 7.9 on page 224 since the exact number of summands is not known:

[5] See also Section 7.8 for a discussion on Java integers.

```
(\sum int i; 0 <= i && i < s1.length; s1[i].getCredits())
```

This expression corresponds to $\sum_{i=0}^{\texttt{s1.length}-1} \texttt{s1[i].getCredits()}$ in mathematical notation. More generally, sum comprehensions in JML can have several bound variables that range over sets of values. The general pattern is `(\sum` T x`;` P`;` Q`)` where T is a type, P a Boolean expression and Q an integer expression corresponds to $\sum_{x\in\{y\in T|P\}} Q$. Likewise the `\product` operator is used to express product comprehensions. Since addition (as multiplication) is commutative and associative, there is no particular order in which elements are summed up. Sums with empty ranges have value 0 by definition, empty products have value 1.

Expressions using the `\num_of` operator, that gives the cardinality of a finite set, can be expressed in terms of sums: `(\num_of` T x`;` P`)` is syntactic sugar for `(\sum` T x`;` P`; 1)`.

However, like for maximum, sum comprehensions are not always well-defined. For instance, the expression `(\sum \bigint i; 0 <= i; i)` corresponds to $\sum_{i=0}^{\infty} i$, the value of which is undefined since it diverges. In some tools—including KeY—effective reasoning about these comprehensions is therefore restricted to closed integer intervals, for which sums, etc., are always defined. In particular, KeY only interprets sums of the shape `(\sum int i;` ℓ `<= i && i <` u`;` Q`)`, where the lower bound ℓ is included and the upper bound u is excluded. This restricted form using intervals has the advantage of having a simple induction schema to define these comprehensions, that lays the foundation to reasoning about sums and products. More details about this are discussed in Section 8.1.

7.2.3 Evaluation in the Prestate

As indicated in the introductory example, JML allows us to mark any expression e in a postcondition with `\old(`e`)`, which means that e is not evaluated in the current (post)state of the method, but in its prestate. In most cases, `\old(`e`)` is a subexpression of some bigger expression, and it is important to be aware that all parts of the expression not included in `\old(`…`)` construct are evaluated in the current (post)state. This is fairly obvious in many examples, like `ensures getCredits() == \old(getCredits()) + c;` in Figure 7.1. For a more subtle example, consider an ATM scenario, where an `insertedCard` (represented by an object with a Boolean field `invalid`) is 'confiscated' after too many failed attempts to enter the correct PIN, specified by

```
//@ ...
//@ ensures \old(insertedCard).invalid;
//@ ...
```

We encourage the reader, before reading on, to reflect on the difference between `\old(insertedCard).invalid` and `\old(insertedCard.invalid)`.

Writing `\old(insertedCard.invalid)` would mean that the method implementation has to guarantee that the `invalid` field of the old `insertedCard` object

was true *before* the method's execution. This makes no sense, as a method implementation can never influence its prestate. However, `\old(insertedCard).invalid` makes much more sense, as an implementation can, for instance, set the `invalid` field of the old `insertedCard` object to `true`. To demand the invalidation of the object `insertedCard` in the poststate, `\old(insertedCard).invalid` refers to the *current* field of the object *formerly* referred to by `insertedCard`.

7.3 Method Contracts in Detail

Now that the reader is familiar with the particular features of JML expressions, we are ready to continue the presentation of method contracts. Among other things, we will introduce specification visibility, much more structure, and more semantics, in contracts.

7.3.1 Visibility of Specifications

So far, the specifications have not specified anything about the values of an object's fields. Typically, these are declared private, which limits also their use within specifications. Basically JML uses the same access rules like Java which means that elements used within specifications have to be visible to it and that a specification itself also has a visibility. The access modifiers `public`, `protected`, and `private` are explicitly used to define specifications visibility. If none of these modifiers is used a specification has the default (package) visibility.

In addition to the Java access rules, JML forbids the usage of elements within specifications that are less visible than the specification itself. The reason of this restriction is to avoid to expose implementation details to the clients (information hiding). As a consequence, it is not possible to use private variables directly within protected or public specifications. However, it is possible to change their visibility only for the specification layer via `spec_protected` or `spec_public`. These modifiers have to be used with care and only if the adjusted field fits the abstraction level of the specification.

Example 7.8. If we specify the instance variables of `CStudent` to be `spec_public`, then its constructor can also be specified as in Listing 7.2.

A second restriction of specification visibility to keep in mind is that specifications that constrain a field must have at least the visibility of the field. The reason is that otherwise a user of a field would not see the constraints to maintain. This is especially important for invariants and constraints, discussed in Sections 7.4.1 and 7.4.3.

```
class CStudent implements Student {

    /*@ spec_public @*/ private String name;
    /*@ spec_public @*/ private int credits;
    /*@ spec_public @*/ private int status;

    ⋮

    /*@ requires c >= 0;
      @ ensures credits == c;
      @ ensures status == bachelor;
      @ ensures name = n;
      @*/
    public CStudent (int c, String n) {
        credits = c;
        name = n;
        status = bachelor;
    }
}
```

Listing 7.2 Class `CStudent` with `spec_public` variables

7.3.2 Specification Cases

When specifying a method, it is often useful—and sometimes necessary—to describe the behavior separately for different parts of the prestate/input space. The structuring mechanism for that is the *specification case*, each of which is specific for a particular precondition. Specification cases are combined by the `also` keyword. The above method contracts consisted of only one specification case. We now give an example where two specification cases are given for one method.

Example 7.9. Listing 7.3 shows the specification of a class implementing a set of integers, with a limited capacity that is fixed at the time when the integer set object is constructed.

Here, method `add` is specified by two specification cases, one for the case, where the set is not full and the element to be added is not contained (`size < limit && !contains(elem)`); and one for the case, where the set is full or the element to be added is already contained (`size == limit || contains(elem);`). Note that it is possible to specify `add` with only one specification case. Confer to [Raghavan and Leavens, 2000] for a procedure to produce flat specifications.

Listing 7.3 is furthermore an example for extensive usage of quantification. Moreover, it demonstrates the power of pure methods. Without the ability to use `contains` in the specification of the other methods, all occurrences of `contains` would need to be replaced by the existentially quantified JML expression specifying `contains`, resulting in a much more complicated specification. We will extend on this example when discussing class invariants.

```
1  public class LimitedIntegerSet {
2    public final int limit;
3    /*@ spec_public @*/ private int arr[];
4    /*@ spec_public @*/ private int size = 0;
5
6    public LimitedIntegerSet(int limit) {
7      this.limit = limit;
8      this.arr = new int[limit];
9    }
10
11   /*@ requires size < limit && !contains(elem);
12     @ ensures \result == true;
13     @ ensures contains(elem);
14     @ ensures (\forall int e;
15     @                  e != elem;
16     @                  contains(e) <==> \old(contains(e)));
17     @ ensures size == \old(size) + 1;
18     @
19     @ also
20     @
21     @ requires size == limit || contains(elem);
22     @ ensures \result == false;
23     @ ensures (\forall int e;
24     @                  contains(e) <==> \old(contains(e)));
25     @ ensures size == \old(size);
26     @*/
27   public boolean add(int elem) {/*...*/}
28
29   /*@ ensures !contains(elem);
30     @ ensures (\forall int e;
31     @                  e != elem;
32     @                  contains(e) <==> \old(contains(e)));
33     @ ensures \old(contains(elem))
34     @         ==> size == \old(size) - 1;
35     @ ensures !\old(contains(elem))
36     @         ==> size == \old(size);
37     @*/
38   public void remove(int elem) {/*...*/}
39
40   /*@ ensures \result == (\exists int i;
41     @                            0 <= i && i < size;
42     @                            arr[i] == elem);
43     @*/
44   public /*@ pure @*/ boolean contains(int elem) {/*...*/}
45
46     ⋮
47 }
```

Listing 7.3 Specifying limited size integer set

7.3.3 Semantics of Normal Behavior Specification Cases

An important question is when a method specification is actually satisfied. And in particular, if a method does not terminate, does it then satisfy its specification? The specifications as we have seen here implicitly state that the method must always terminate, i.e., they specify a *total correctness* condition, see [Hoare, 1969]. If method `m` is specified as follows:

```
/*@ requires P;
  @ ensures Q;
  @*/
public ... m(...) { ...
```

this means the following: If method `m` is executed in a prestate where P holds, *then* execution of method `m` from this prestate terminates, *and*—if it terminates normally[6]—in the final state the postcondition Q holds. Section 8.2 provides a more formal account on contract semantics.

Nontermination and Exceptions

To specify that a method may not terminate under some precondition, one can add an explicit **`diverges`** clause. A diverges clause specifies under which conditions a method may not terminate, for example to express that for certain parameters a method may not terminate. As we have seen above, the default is **`false`**, i.e., a method must always terminate. Like **`requires`**, **`diverges`** clauses are evaluated in the prestate; a **`diverges`** clause thus describes a precondition that is necessary for nontermination.

```
/*@ requires P;
  @ ensures Q;
  @ diverges x < 0;
  @*/
public ... m(int x) { ...
```

Sometimes we wish to exclude the case that a method may terminate because of an exception. In this case, the respective specification case is preceded by the keyword **`normal_behavior`**, which states that the method execution must terminate normally, and in the final state the postcondition must hold.

Lightweight and Heavyweight Specification

The JML reference manual [Leavens et al., 2013]further distinguishes between so called *lightweight* and *heavyweight* specifications. Heavyweight specification

[6] A method is said to terminate normally if either it reached the end of its body, in a normal state, or it terminated because of a **`return`** instruction. In Section 7.6 we discuss how we can specify methods that terminate because of an exception.

cases are preceded by one of the keywords **behavior**, **normal_behavior**, or **exceptional_behavior** (see Section 7.6); all others are lightweight. The difference is that, in lightweight specifications, there are no standardized defaults—except for **diverges** whose default is always **false**. Instead, every tool is free to choose its own semantics. KeY takes the choice of applying the same defaults as for heavyweight specifications. The visibility of a lightweight specification case in JML is always the one of the method they specify.

7.3.4 Specifications for Constructors

Constructors can be considered as special methods. In the prestate of a constructor, the object does not yet exist. Thus a precondition of a constructor can only put constraints on the constructor parameters, it cannot require anything about the internal state of the object—as the object does not exist yet when the constructor is called. However, the postcondition of the constructor can specify constraints on the state of the object. Typically, it will relate the object state to the constructor's parameters.

Example 7.10. Suppose we have a class `CStudent` implementing the `Student` interface. It could have the following constructor:

—— Java + JML ——

```
/*@ requires c >= 0;
  @ ensures getCredits() == c;
  @ ensures getStatus() == bachelor;
  @ ensures getName() == n;
  @*/
CStudent (int c, String n) {
   credits = c;
   name = n;
   status = bachelor;
}
```

—— Java + JML ——

Thus, it would be incorrect to specify **requires** `getCredits() >= 0;` or **requires** `getStatus() == bachelor`; these specifications are meaningless at the moment that the constructor is invoked.

7.3.5 Notions of Purity

Above in Section 7.1.1, we have said that only pure methods may be used in a method specification, and purity was defined as terminating unconditionally and having no

visible side effects. 'No visible side effects' means that the state that was allocated on the heap before the method call may not be changed. Thus, this does not exclude that a method creates a new object and initializes it. In the same way, constructors are pure if they only operate on fields of the object they initialize, not touching the state that was allocated before the call to the constructor. If it, however, changes other parts of the state it is not pure. Later, in Section 9.4.4, we will see how purity annotations help to verify programs in a modular way. For clarity, this notion of purity in JML is sometimes known as *weak purity*. This is in contrast to *strict purity* that requires that the heap is not changed in any way. While weakly and strictly pure methods have the same observable behavior, reasoning about hidden changes in weakly pure methods can make a proof more complicated. In KeY's dialect of JML, strict purity is indicated by the modifier `strictly_pure`.

Apart from that, there are situations where methods are technically speaking not pure, but from a client point of view may be considered to be so. Consider for an example the function that computes a hash code. The first time this function is called on an object, a field of the object will be written, so that the next calls can be evaluated by looking up this field. Because of this, different notions of purity and *observational purity* exist in the literature [Barnett et al., 2004, 2005b, Darvas and Müller, 2006, Darvas and Leino, 2007, Naumann, 2007, Cok and Leavens, 2008].

For the scope of this chapter, it is sufficient to define purity simply as not having any observable side effects.

While pure methods must terminate under any circumstance, they may still raise exceptions or have a nontrivial precondition. In these cases, the value of a pure method invocation is not always well-defined. Therefore, it is a best practice to have `true` as precondition of pure methods and to rule out exceptions and not defined return values.

7.4 Class Level Specifications

Consider again the specification of `Student` in Listing 7.1. If we look carefully at the specifications and the description that we give about the student's credits, we notice that we implicitly assume some properties about the value of `getCredits` that hold throughout. For example, we wrote above:

> "a student thus never can have a negative amount of credits"

and also

> "the number of credits only increases."

But if we would like to make explicit that we assume that these properties always hold, we would have to add this to *all* specifications in `Student`, and thus in particular, also to all methods that do not relate at all to the number of credits. Thus for example, we would get the following specification:

Java + JML
```
/*@ requires getCredits() >= 0;
  @ ensures \result == bachelor || \result == master;
  @ ensures getCredits() >= 0;
  @*/
/*@ pure @*/ public int getStatus();
```
Java + JML

Clearly, this is not desired, because specifications would get very large, and besides describing the intended behavior of that particular method, they also describe properties over the lifetime of the object. Therefore, JML provides also class level specifications, such as invariants, history constraints, and initially clauses. These specify properties over the internal state of an object, and how the state can evolve during the object's lifetime.

7.4.1 Invariants

One of the most important and widely-used specification elements in object-orientation are type *invariants*[7], also called *class* or *interface invariants*, depending on where they are defined. An invariant is a Boolean (JML) expression over the object state, and can be seen as a condition to constrain the state an instance can be in. In addition, any constructor has to ensure that the invariant is established. Methods can be except from this scheme by adding the modifier `helper` to their declaration.

Example 7.11. Listing 7.4 shows three possible invariants that can be added to interface `Student`. These specify that credits are never nonnegative; a student's status is always either Bachelor or Master, and nothing else; and if a student's status is Master, he or she has earned more than 180 credits. The pure methods are used in the invariants.[8]

Of course, instead of specifying invariants, one could also add these specifications to all pre- and postconditions explicitly. However, this means that if you add a method to a class, you have to remember to add these pre- and postconditions yourself. Moreover, invariants are also inherited by subclasses (and by implementations of interfaces). Thus any method that overrides a method from a superclass still has to respect the invariants. And any method that is added to the subclass also has to respect the invariants from the superclass. This leads to a very nice separation of concerns.

[7] Not to be confused with loop invariants. Those will be discussed in Section 7.9.2.

[8] There is an unresolved discussion about whether methods that are used in invariants have to be `helper`, or how to otherwise avoid potential circularity between showing and assuming invariants. We choose to not mark public methods as `helper`, because helper methods are designed for local usage. Please note, though, that some tools, like OpenJML, require methods used in invariants to be `helper`.

```
interface Student {

    public static final int bachelor = 0;
    public static final int master = 1;

    /*@ instance invariant getCredits() >= 0;
      @ instance invariant getStatus() == bachelor ||
      @                    getStatus() == master;
      @ instance invariant getStatus() == master ==>
      @                    getCredits() >= 180;
      @
      @ instance initially getCredits() == 0;
      @ instance initially getStatus() == bachelor;
      @
      @ instance constraint getCredits() >= \old(getCredits());
      @ instance constraint \old(getStatus()) == master ==>
      @                     getStatus() == master;
      @ instance constraint \old(getName()) == getName();
      @*/

    public /*@ pure @*/ String getName();

    public /*@ pure @*/ int getStatus();

    public /*@ pure @*/ int getCredits();

   /*@ requires c >= 0;
     @ ensures getCredits() == \old(getCredits()) + c;
     @*/
    public void addCredits(int c);

    /*@ requires getCredits() >= 180;
      @ requires getStatus() == bachelor;
      @ ensures getCredits() == \old(getCredits());
      @ ensures getStatus() == master;
      @*/
    public void changeStatus();


}
```

Listing 7.4 Interface `Student` with class level specifications

An important point to realize is that invariants have to hold only in all states in which a method is called or terminates. Thus, inside the method, the invariant may be temporarily broken. Note that the kind of termination of a method does not matter. Regardless of terminating normally, exceptionally, or erroneously, a method has to meet the invariant.

Example 7.12. The following possible implementation of `addCredits` is correct, even though it breaks the invariant that a student can only be studying for a Master

if they have earned more than 180 points inside the method: if `credits + c` is sufficiently high, the status is changed to Master. After this assignment the invariant does not hold, but because of the next assignment, the invariant is reestablished before the method terminates.

Java + JML

```
/*@ requires c >= 0;
  @ ensures getCredits() == \old(getCredits()) + c;
  @*/
public void addCredits(int c) {
  if (credits + c>= 180) {status = master;} // invariant broken
  credits = credits + c;
}
```

Java + JML

However, if a method calls another method on the same object, it has to ensure that the invariant holds before this callback. Why this is necessary, is best explained with an example.

```
interface CallBack {

    //@ instance invariant getX() > 0;
    //@ instance invariant getY() > 0;

    /*@ pure @*/ public int getX();
    /*@ pure @*/ public int getY();

    //@ ensures getX() == x;
    public void setX(int x);

    //@ ensures getY() == y;
    public void setY(int y);

    //@ ensures \result == getX() % getY();
    public int remainder();

    public int longComputation();

}
```

Listing 7.5 Interface `CallBack`

Example 7.13. Consider the interface `CallBack` in Listing 7.5. Typically, correctness of the method `remainder` crucially depends on the value of `getY` being greater than 0. Suppose we have an implementation of the `CallBack` interface, where the method `longComputation` is sketched in Listing 7.6.

```
public int longComputation(){
   ...
   if (getY() ...) {
      setY(0); // invariant broken
   }
   ...
   int r = remainder(); // callback
   ...
   setY(r + 1); // invariant reestablished
   ...
   return ...
}
```

Listing 7.6 Invariant broken during callback

Naively, one could think that the fact that the invariant about `getY()` is broken inside this method, is harmless, because the invariant is reestablished by the `setY(r + 1)` statement. However, the call to method `remainder` is a callback, and the invariant should hold at this point. In fact, correct functioning of this method call depends on the invariant holding. The invariant implicitly is part of `remainder`'s precondition. If the invariant does not hold at the point of the callback, this means that `remainder` is called outside its precondition, and no assumption can be made about its result as well.

Although invariants are always specified within a class or interface, their effective scope is global. A method of some specific class is obliged to respect invariants of all other classes. There is a way to avoid the requirement that the invariant has to hold upon callback, by specifying that a method is a **`helper`** method. Such methods must not depend on the invariant to hold, and they do not guarantee that the invariant will hold afterwards. Typically, only private methods should be specified as helper methods, because one does not want that any other object can directly invoke a helper method. Finally we note that, while a *pure* helper method cannot assume the invariant to hold when it is called, it does *preserve* any invariant because of purity.

Where Do Invariants Come From?

Sometimes invariants are imposed by the domain which is modeled by the code. The interface `Student` in Listing 7.4 is such an example. Students can only have a positive number of credits, they must be either Master or Bachelor students, and so forth. Another common motivation for invariants is efficiency. Efficient computations often require to organize data in a specific way. One way is introducing redundancy, like for instance in an index of a book, mapping words to pages where they occur. Such an index is redundant (we can always search through the whole book to find the occurrences of a word), but it enables efficient look-up. On the downside, redundancy opens up for inconsistencies. The countermeasure is to use invariants, formalizing the consistency conditions (like each word in an index appearing in the text as well, at

the page given by the index). Other ways to increase efficiency limit the organization of data to comply to certain restrictions. A prominent example of that is sortedness, which allows for quicker look-up. In the following, we demonstrate how sortedness can be expressed with an invariant.

Example 7.14. We turn the `LimitedIntegerSet` (Listing 7.3) into a sorted data structure, by adding the invariant

JML
```
/*@ public invariant (\forall int i;
  @                           0 < i && i < size;
  @                           arr[i-1] <= arr[i]) ;
  @*/
```
JML

to that class. With that, the implementer of each method can *rely* on sortedness in the prestate, and the implementer of each (impure) method has to *guarantee* sortedness in the poststate.

Static Invariants vs. Instance Invariants

Class invariants may or may not refer to the object `this` and its instance (i.e., nonstatic) fields or methods. For example, the class invariant in Example 7.14 refers to the instance field `arr`. Such invariants are also called *instance invariants*, and can be declared as such with the `instance` modifier. This is however not necessary, as class invariants are instance invariants per default. If, on the other hand, a class invariant does *not* refer to `this`, neither to any instance field or instance method, we can highlight that (and potentially help verification tools) by declaring the invariant as *static*, using the `static` modifier. Please note that, since instance methods might change static variables, static invariants have to be respected by instance methods as well.

Similarly, interface invariants may or may not refer to instance (i.e., nonstatic) methods. For example, all invariants in Listings 7.4 and 7.5 mention instance methods, and are therefore *instance* invariants. The reader may have noted that invariants in Listings 7.4 and 7.5 are explicitly declared as `instance invariant`. This is necessary because, for interfaces, the default is different from classes: invariants are static, if not declared otherwise.

Semantics of Invariants

Defining a precise semantics for invariants is still an active area of research, see, e.g., [Poetzsch-Heffter, 1997, Leino and Müller, 2004, Barnett et al., 2004, Müller et al., 2006, Bruns, 2009]. A complication is that, although invariants are declared in a particular class, not only instances of that class have to respect it, but all objects in the

system. An alternative approach, that is used in the Spec# framework, is to explicitly add specification statements `unpack` and `pack` for invariants. An invariant may only be broken if it has been explicitly unpacked. When the invariant is reestablished, it has to be explicitly be packed again, and this only succeeds if the invariant indeed holds at this point. Every method can then specify explicitly whether it assumes invariants to hold (i.e., to be packed) or not. This approach is sometimes referred to as the *Boogie methodology* [Barnett et al., 2006].

Similar to the Boogie methodology, in the KeY system, invariants are not implicitly added to specifications. Instead, the specification must make explicit which specific invariants are included, and which are not. This specification may be more verbose, but it is clear from the given specification that invariants are assumed or established. See Section 9.2.1.3 for further discussion. The invariant for an object `o` can be referred to through `\invariant_for(o)`. This allows fine-grained usage of invariants in specifications. Unlike in Boogie, explicit packing/unpacking instructions in the code are not necessary. Instead, the specifier has to specify a set of locations the invariant depends on at most (`accessible` clause). Usually, methods rely at least on the invariant of the current receiver. For convenience, this invariant is implicitly included for non`helper` methods (see Section 8.2 on proof obligations).

Finally, it is important to realize that the notion of invariants that we discussed here only makes sense in a sequential setting. In a multithreaded setting, there always may be another thread accessing the object simultaneously, and one cannot talk about initial and final states of a method invocation anymore. Instead, in a multithreaded setting, one sometimes specifies *strong invariants* that may never be broken. For instance, Zaharieva-Stojanovski and Huisman [2014] present a modular specification and verification technique for class invariants in a concurrent setting.

7.4.2 Initially Clauses

Sometimes, one explicitly wishes to specify the conditions that are satisfied by an object upon creation. Each (nonhelper) constructor[9] of the object has to establish the predicate specified by the *initially clause*. Another advantage of initially clauses is that they are inherited; that means that also constructors of subclasses have to fulfill them. Constructors in Java itself are not inherited. As a consequence, a constructor can rely on the guarantees provided by a called super constructor but does not have to maintain them.

Example 7.15. Listing 7.4 shows some possible initially clauses for the `Student` interface.

Again, it would be possible to specify this property as a postcondition of all constructors, instead of as a single initially clause. But in this way, any additional constructor has to respect the initially clause, and we ensure that also subclasses respect it.

[9] Again, typically only private constructors would be annotated as a helper constructor.

7.4.3 History Constraints

Invariants as we discussed above define a predicate that every state of the object should respect. However, sometimes one also wishes to specify how an object may evolve over time, i.e., the relationship that exists between the prestate and the poststate of a method call. This could be seen as a sort of general postcondition that has to be respected by every method, however the definition is actually more fine grained than that. For this, *history constraints* (usually *constraints* for short) have been introduced by Liskov and Wing [1993]. Constraints can be seen as implicit postconditions, but just as invariants and initially clauses, they have the advantage that they are inherited, and immediately are required to hold for any additional methods. Constraints may rely on syntactical features that are used to measure changes between states such as the `\old` operator. Assigning suitable semantics to history constraints is nontrivial; a possibility would be to see them as special two-state model methods (see Section 9.2.2). This is not yet implemented in KeY at the time of publishing this book.

Example 7.16. Listing 7.4 defines several constraints for the `Student` interface. The first constraint specifies that the amount of credits can never decrease. The second constraint specifies that if a student has obtained the Master status, he or she will remain a Master student, and cannot be downgraded to a Bachelor student again. Finally, the third constraint specifies that a student's name can never change.

When specifying constraints, it is important that they should denote a reflexive relation, i.e., it should be possible to respect a constraint without actually changing the state. In particular, any pure method should be able to respect the constraint. Therefore, one should not specify the following strict constraint:

```
constraint \old(getCredits()) < getCredits();
```

as it is impossible to respect this constraint with a pure method. Typically, constraints will also be transitive, so that when you consecutively call two methods from the same object, you also know the relationship that holds between the prestate of the first method, and the poststate of the second method.

Example 7.17. Consider the possible implementation of `addCredits` in Listing 7.7. To show that the constraint is respected, it has to hold for the following state pairs:

- (prestate, call-state `changeStatus`)
- (call-state `changeStatus`, return-state `changeStatus`)
- (return-state `changeStatus`, poststate)

Notice that if the constraint is transitive, the relationship also holds for the pair of prestate and poststate, which is indeed what we want.

Again, in a multithreaded setting, the meaning of constraints would become less clear. Because any interleaving is possible, all intermediate states must be assumed to be visible to other threads. However, a constraint such as that `getName` returns a constant value could still be meaningful also in a multithreaded setting (except

```
//@ constraint \old(getCredits()) <= getCredits();

/*@ requires c >= 0;
  @ ensures getCredits() == \old(getCredits()) + c;
  @*/
// prestate
public void addCredits(int c) {
   credits = credits + c;
   if (credits >= 180) {
      // call-state changeStatus
      changeStatus();
      // return-state changeStatus
   }
} // poststate
```

Listing 7.7 Checking history constraints

that the number of possible visible state pairs that have to be considered might grow exponentially). Therefore, in a concurrent setting one could imagine a notion of *strong* constraints*strong history constraints*, i.e., a relationship that has to hold for any pair of consecutive states.

7.4.4 Initially Clauses and History Constraints: Static vs. Instance

Just as class invariants (see Section 7.4.1), also initially clauses and history constraints have instance as well as static versions, which can be declared with the `instance` and `static` modifier, respectively. The static variants cannot explicitly mention an instance (i.e., nonstatic) field or method, neither can they refer to `this` itself. The instance variants, on the other hand, have no such restriction.

In *classes*, the default for initially clauses and history constraints is `instance`, meaning this modifier can be omitted. For *interfaces*, the default for initially clauses and history constraints is `static`. Note that, in interface `Student` (Listing 7.4), all initially clauses and history constraints mention nonstatic methods or fields. They can therefore not be static (which is the default), and have to be marked as `instance` explicitly.

7.4.5 Inheritance of Specifications

Design by Contract allows one to impose the concept of *behavioral subtyping* [Liskov, 1988], that is usually defined by the *Liskov substitution principle*, or Liskov principle for short [Liskov and Wing, 1994]. A type T' is a behavioral subtype of type T if every observable behavior of T is also observable on T'. In an object-oriented

program, this means that any subclass may be used wherever a superclass is expected. Behavioral subtyping expresses the idea that a subclass thus should behave as the superclass (at least, when it is used in a superclass context). Subclasses in Java do not always define behavioral subtypes. They can be used simply for the purpose of code reuse.

However, the substitution principle as originally stated by Liskov [1988] can sometimes be too strong in practice (see [Leavens, 1988]). For instance, what exactly is the refined behavior of a *linked* list, as compared to a list in general? Surely, there is no nondeterminism that can be refined. This means there cannot be strict behavioral subtypes regarding all behaviors. Instead, we focus on the client perspective again and define behavior subtypes regarding contracts (and invariants). This means that a class C' is a behavioral subtype of a super class C, if for every method m implemented in both C and C' (i.e., the implementation in C' is overriding), every specification case for $C :: m$ is also a specification case for $C' :: m$, and that the contract of $C :: m$ is refined by the contract of $C' :: m$. A full formalization of this definition of behavioral subtyping can be found in [Leavens and Naumann, 2006].

To ensure that a subclass indeed defines a behavioral subtype, specification inheritance can be used [Dhara and Leavens, 1995, Leavens and Dhara, 2000]: In JML, every (nonprivate) method in the subclass inherits the overridden method's specification cases defined in the superclass. And in addition, all invariants of the superclass are inherited by the subclass. Notice that this same approach applies for interfaces and implementing classes. An interface can be specified with its desired behavior. Every class that implements this interface should be a behavioral subtype of the interface, i.e., it should satisfy all the specifications of the interface. Concretely, this means the following:

- every method that overrides a method from a superclass, or implements from an interface, has to respect the method specification from the superclass;
- every class that implements an interface has to respect the specifications of the interface; and
- every class that extends another class has to respect the specifications of that class.

Still, it is possible to refine specifications in subclasses (or implementing classes), in addition to what is inherited. Any additional specification of an inherited method (whether or not the implementation is overridden) is added to the inherited specifications from the superclass, using the `also` keyword.

```
/*@ also
  @ <subclass-specific-spec-cases>
  @*/
public void method () { ...
```

Note that the JML annotation starts with `also`, not preceded by anything. This is because the inherited specification cases are still there, even if implicit, to be extended here by whatever is written after the `also`.

Invariants are also fully, and implicitly, inherited. Extending the set of inherited invariants by additional invariants specific for a subclass is easy, by simply writing

them in the subclass, using the normal syntax for invariants. The same applies also to initially clauses and constraints.

The idea of behavioral subtypes is crucial for the correctness of object-oriented programs. We can specify the behavior of a class in an abstract way. For example, in class `Average` in Listing 7.8, we have an array of `Student` instances; the concrete instances that are stored in the array may have different implementations, but we know that they all implement the methods specified in the interface `Student` in Listing 7.1. This means that we can rely on the specification case of `Student#getCredits()` in Line 11 of `Average#averageCredits()`.

Respecting inherited specifications is a good practice, but it does not guarantee behavioral subtyping per se. JML allows us to make program elements more visible in the specification than they are in the implementation (through the **`spec_public`** modifier, see Section 7.3.1). In this way, specifications may expose implementation details. While it is also a good practice to declare those specifications private, in many cases, this would disable us from giving *any* meaningful specification. A solution to this dilemma is abstraction, that will be covered in Section 7.7.1 below.

7.5 Nonnull Versus Nullable Object References

In Java, the set of values of reference type include the null reference. (Note that the same is true for the values of array type, because each array type is also a subtype of `Object`.) But even if the type system always allows **`null`**, the specifier may want to exclude the null reference in many cases. Whether or not null is allowed can be expressed by means of simple (in)equations, like, for instance, `o` **`!= null`**, in pre/-postconditions or invariants. However, this issue is of so dominant importance that JML offers two special modifiers just for that, **`non_null`** and **`nullable`**. Class members (i.e., fields), method parameters, and method return values can be declared as **`non_null`** (meaning null is forbidden), or **`nullable`** (in which case null is allowed, but not enforced).

Here are some examples for forbidding null values.

```
private /*@ non_null @*/ String name;
```

adds the *implicit* invariant `invariant name != null;` to the class at hand.

```
public void setName(/*@ non_null @*/ String n) {...
```

adds the *implicit* precondition `requires n != null;` to each specification case of `setName`.

```
public /*@ non_null @*/ String getName() {...
```

adds the *implicit* postcondition `ensures \result != null;` to each specification case of `getName`.

The reader can imagine that **`non_null`** modifiers can easily bloat the specification. Therefore, JML has built-in **`non_null`** as the *default* for all fields, method parameters, and return types, such that all **`non_null`** modifiers in the above examples are actually redundant. By only writing the following, without any explicit **`non_null`**, we get

exactly the same implicit invariants, preconditions, and postconditions as mentioned above.

```
private String name;

public void setName(String n) {...

public String getName() {...
```

But how can we allow null anyway? We can avoid the restrictive nonnull default by the aforementioned modifier **nullable**. In the above examples, we could allow null (and thereby avoid the implicit conditions), by writing

```
private /*@ nullable @*/ String name;

public void setName(/*@ nullable @*/ String n) {...

public /*@ nullable @*/ String getName() {...
```

Notice that the nonnull by default also can have some unwanted effects, as illustrated by the following example.

Example 7.18. Consider the following declaration of a `LinkedList`.

—— Java + JML ——

```
public class LinkedList {
   private Object elem;
   private LinkedList next;
   ...
}
```

—— Java + JML ——

Because of the nonnull by default behavior of JML, this means that all elements in the list are nonnull. Thus the list must be cyclic, or infinite.[10] This is usually not the intended behavior, and thus the `next` reference should be explicitly annotated as **nullable**.

—— Java + JML ——

```
public class LinkedList {
   private Object elem;
   private /*@ nullable @*/ LinkedList next;
   ....
}
```

—— Java + JML ——

In short, it is important to remember that for all class fields, method parameters, and method results, the null reference is forbidden wherever we do not state otherwise with the JML modifier **nullable**.

[10] A linked data structure having infinite length is indeed a contradiction. At runtime, there are only finitely many created objects on the heap.

In the context of allowing vs. forbidding the null reference, handling of arrays deserves special mentioning. The additional question here is whether, or not, the prohibition of null holds for the *elements* of the array. Without loss of generality, we consider the following array typed field declaration: `String[] arr;`. Because of nonnull being the default, this is equivalent to writing `/*@ non_null @*/ String[] arr;`. Now, in both cases, the prohibition of null references extends, in JML, to the elements of the array! In other words, both the above forms have the same meaning as if the following invariants were added:

—— Java + JML ——

```
//@ invariant arr != null;
//@ invariant (\forall int i;
//@                     i >= 0 && i < arr.length;
//@                     arr[i] != null);
```

—— Java + JML ——

Again, no such invariant is needed for disallowing null; writing `String[] arr;` is enough. We can, however, allow null for *both*, the whole array and its elements (at first), by writing `/*@ nullable @*/ String[] arr;`. To that, we can add further restrictions. For instance, if only the elements may be null, but not the whole array, we can write:

—— Java + JML ——

```
//@ invariant arr != null;
/*@ nullable @*/ String[] arr;
```

—— Java + JML ——

7.6 Exceptional Behavior

So far, we have only considered normal termination of methods. But in some cases, exceptions cannot be avoided. Therefore JML also allows one to specify explicitly under what conditions an exception may occur.

The `signals` and `signals_only` clauses are introduced to specify *exceptional postconditions*. In addition, one can give an `exceptional_behavior` method. Exceptional postconditions have the form `signals (`E `e)` P, where E is a subtype of `Throwable`, and the following meaning: *if* the method terminates because of an exception that is an instance of type E, then the predicate P has to hold. The variable name `e` can be used to refer to the exception in the predicate. Note the implication direction: a `signals` clause does *not* specify under which condition an exception may occur by itself, neither that it *must* occur. Such specification patterns can only be obtained in combination with `requires` and `ensures` clauses. The `signals` clause describes a necessary condition, but not a sufficient one. For a formal account on contract semantics, see Section 8.2 in the following chapter.

The **signals_only** clause is optional in a method specification. Its syntax is **signals_only** E_1, E_2, ..., E_n, meaning that if the method terminates because of an exception, the dynamic type of the exception has to be a subclass of E_1, E_2, ..., or E_n. If **signals_only** is left out, only the exception types that are declared in the method's **throws** clause and *unchecked exceptions*, i.e., instances of `Error` and `RuntimeException`, are permitted. These are exactly the exception types that are permitted by Java's type system.

```
1  class Average {
2
3      /*@ spec_public @*/ private Student[] sl;
4
5      /*@ signals_only ArithmeticException;
6        @ signals (ArithmeticException e) sl.length == 0;
7        @*/
8      public int averageCredits() {
9          int sum = 0;
10         for (int i = 0; i < sl.length; i++) {
11             sum = sum + sl[i].getCredits();
12         };
13         return sum/sl.length;
14     }
15 }
```

Listing 7.8 Class `Average`

Example 7.19. Consider for example class `Average` in Listing 7.8. The specification of method `averageCredits` states that the method may only terminate normally, or with an `ArithmeticException`—and thus, it will not throw an `ArrayIndexOutOfBoundsException`. Moreover, if an `ArithmeticException` occurs, then in this exceptional state the length of `sl` is 0.

Notice that it is incorrect in this case to use an **ensures** clause, instead of a **signals** clause: an **ensures** clause specifies a normal postcondition, that only holds upon normal termination of the method.

Above, in Section 7.1 we discussed **normal_behavior** specifications. Implicitly, these state that the method has to terminate normally. Similarly, JML also features an **exceptional_behavior** method specification. This specifies that, if the method terminates, then this must be due to an exception.[11] In contrast, a plain **behavior** specification may well contain both **ensures** clauses and **signals** or **signals_only** clauses, whereas a normal behavior specification may not contain these, and an exceptional behavior specification may not contain an **ensures** clause. As mentioned above in Section 7.3.2, a single method can be specified with several method specifications, joined with **also**. Exceptional behavior specifications are typically used in this case.

[11] Remember that an explicit **diverges** clause still permits nontermination.

Example 7.20. Consider the more detailed specification for `averageCredits` in Listing 7.9. This states that if `sl.length > 0`, i.e., there are students in the list,

```
class Average2 {

    /*@ spec_public @*/ private Student[] sl;

    /*@ normal_behavior
      @ requires sl.length > 0;
      @ ensures \result ==
      @         (\sum int i; 0 <= i && i < sl.length;
      @                      sl[i].getCredits())/sl.length;
      @ also
      @ exceptional_behavior
      @ requires sl.length == 0;
      @ signals_only ArithmeticException;
      @ signals (ArithmeticException e) true;
      @*/
    public int averageCredits() {
        int sum = 0;
        for (int i = 0; i < sl.length; i++) {
            sum = sum + sl[i].getCredits();
        };
        return sum/sl.length;
    }
}
```

Listing 7.9 Class `Average2`

then the method terminates and the result is the average value of the credits obtained by these students. If `sl.length == 0` then the method will terminate exceptionally, with an `ArithmeticException`.

In this example, the two preconditions together cover the complete state space for the value of `sl.length`. If `sl.length` could be less than `0`, the method's behavior would not be specified.

Finally, it is important to realize that invariants and constraints also must hold when a method terminates exceptionally. This might seem strange at first: something goes wrong during the execution, so why would it be necessary that the object stays in a good state. But in many cases, the execution can recover from the exception, and normal execution can be resumed. But this means that it is necessary that also when an exception occurs, the object stays in a 'well-defined' state, i.e., a state in which the invariants hold, and that evolves according to the constraints.

A Note on `false`

The Boolean expression **`false`** is used frequently to exclude certain behaviors. For instance, the clause

```
signals (Throwable e) false;
```

states that the method at hand must not terminate exceptionally. Because, if it did, the property `false` would need to hold, which is never the case. Therefore, exceptional termination is never able to satisfy such a specification. Similarly, if one specifies a postcondition `ensures false;` this states that a method must not terminate normally. Thus a method specification:

```
ensures false;
signals (Throwable e) false;
diverges true;
```

implicitly says that a method must never terminate (neither normally, nor exceptionally). Finally, a method can also be specified with a precondition `requires false;`. This means that the method may not be invoked, as no caller can fulfill the precondition of the method.

7.7 Specification-Only Class Members

The previous sections shows how the behavior of code members is specified in JML. But sometimes it is easier or even required to introduce new members only for specification. Model fields, as discussed in Section 7.7.1, allow to provide abstraction from the concrete program state. For each abstract state, a relationship to the concrete program state can be defined. In addition to model fields, sometimes it is also useful to define model methods, i.e., methods that are used in specifications only.

This section also introduces ghost variables (Section 7.7.2). These can be used to extend the state space with specification-only information. They do not provide abstraction, but can record extra information. The use of model and ghost fields is often confused, and therefore Section 7.7.3 compares both approaches, and highlights their differences. For an in-depth account on model field and model method semantics, their encoding in KeY, and how to use them in verification, the reader is kindly referred to Section 9.2.

7.7.1 Model Fields and Model Methods

An important feature of specifications is that they provide abstraction over the concrete implementations. *Model fields* serve as an abstraction feature in a familiar guise. They are declared like regular fields, but within JML specifications and with the modifier keyword `model`. Model fields can be read from like regular fields, but there are no assignments to them since they do not have a state of their own. Instead, to make sure that the concrete implementation corresponds to the abstract specification, a link between the two has to be made. For this purpose, the *represents clause* defines how the value of the abstract variable is defined in terms of the values

of the concrete entities. In the so called functional form, the represents clause, that is a class member, appears similar to an assignment, as can be seen in the following example taken from [Breunesse et al., 2005].

Example 7.21. Class `Decimal` implements decimal variables using an `intPart` and `decPart` variable, but the specification is given in terms of a single model field that represents the value of the composed decimal number.

Java + JML

```
class Decimal {
    public static final short PRECISION = (short) 1000;
    /*@ spec_public @*/ private short intPart = (short) 0;
    /*@ spec_public @*/ private short decPart = (short) 0;

    //@ model int value;
    //@ represents value = intPart * PRECISION + decPart;
}
```

Java + JML

Sometimes, a represents clause cannot be defined directly as a translation into concrete variables; sometimes a (nonfunctional) relation between the abstract and the concrete state can be expressed, sometimes only a dependency relation. JML provides a way to define nonfunctional represents clauses. Instead of the assignment operator, they consist of the keyword **`\such_that`** followed by a Boolean expression. It means that the model field points to some value such that this condition is satisfied.

Example 7.22. Consider class `MatrixImplem` in Listing 7.10. It implements a matrix as a single array (on some platforms, like JavaCard, only one-dimensional arrays are allowed). A model variable matrix is declared, that specifies the abstract representation of the matrix. Unfortunately, no functional represents clause can be specified for this. Instead, the `such_that` keyword is used to define a relational represents clause, that enables to write the specifications of the matrix methods in terms of the abstract matrix variable.

Model fields are useful in many cases. Typical examples are specifications of interfaces. The behavior of an interface is specified in terms of model variables, and the classes implementing the interface define represents clauses for these model variables, relating them to their own concrete implementation. Because of the flexible connection between concrete and abstract state using the **`represents`** clause, this does not impose any restriction on the internal state of a class implementing the interface. Note that in interfaces, model field declarations are **`static`** by default, nonstatic model field declarations must use the modifier **`instance`**.

Example 7.23. Listing 7.11 gives an alternative specification for interface `Student` using model fields. It shows the specification for an implementing class `CCStudent`. Note that it does not declare the model variables, but only defines the represents clause.

```
public class MatrixImplem {

    //@ public model int[][] matrix;
    private int x;
    private int y;
    private int[] matrix_implem;
    /*@ represents matrix \such_that
      @ (\forall int i; i >= 0 && i < x;
      @   (\forall int j; j >= 0 && j < y;
      @     matrix[i][j] == matrix_implem[x * j + i]));
      @*/

    /*@ ensures
      @ (\forall int i; i >= 0 && i < x;
      @   (\forall int j; j >= 0 && j < y;
      @     matrix[i][j] == 0));
      @*/
    public MatrixImplem(int x, int y) {
        this.x = x;
        this.y = y;
        matrix_implem = new int [x * y];
    }

    //@ ensures \result == matrix[i][j];
    public /*@ pure @*/ int get (int i, int j) {
        return matrix_implem[x * j + i];
    }

    /*@ ensures \result >= 0 && \result < x
      @     ==> matrix[\result][coordY(elem)] == elem;
      @*/
    public /*@ pure @*/ int coordX (int elem) {
        for (int i = 0; i < matrix_implem.length; i++)
            if (matrix_implem[i] == elem)
                return i % x;
        return -1;
    }

    /*@ ensures \result >= 0 && \result < y
      @     ==> matrix[coordX(elem)][\result] == elem;
      @*/
    public /*@ pure @*/ int coordY (int elem) {
        for (int i = 0; i < matrix_implem.length; i++)
            if (matrix_implem[i] == elem)
                return i / x;
        return -1;
    }
}
```

Listing 7.10 Relational represents clause

```
public interface Student {

    /*@ public instance model int status;
      @ public instance model int credits;
      @ represents status = (credits < 180 ? bachelor : master);
      @*/

    /*@ public instance invariant status == bachelor || status == master;
      @ public instance invariant credits >= 0;
      @*/

    public static final int bachelor = 0;
    public static final int master = 1;

    /*@ pure @*/ public String getName();

    //@ ensures \result == status;
    /*@ pure @*/ public int getStatus();

    //@ ensures \result == credits;
    /*@ pure @*/ public int getCredits();

    //@ ensures getName().equals(n);
    public void setName(String n);

    /*@ requires c >= 0;
      @ ensures credits == \old(credits) + c;
      @*/
    public void addCredits(int c);

    /*@ requires credits >= 180;
      @ requires status == bachelor;
      @ ensures credits == \old(credits);
      @ ensures status == master;
      @*/
    public void changeStatus();
}

class CCStudent implements Student {

    private int[] creditList;

    /*@ private represents credits =
      @     (\sum int i; 0 <= i && i < creditList.length; creditList[i]);
      @*/

    // rest of class continued...
}
```

Listing 7.11 Interface `Student` with model fields and an implementation.

Sometimes, to complete a specification, one needs a method that only is intended for specification. To support this, JML provides *model methods*. A model method is defined as part of the specification. It can be implemented, but it may also be abstract. And the behavior of a model method is typically defined in terms of its pre- and postconditions again. Typical usages for model methods are:

- if the specification needs a method that is not related to the code, for example to sum all the elements in an array;
- if the specification needs a method that cannot be implemented easily, but that can be specified without any problem.

7.7.2 Ghost Variables

Sometimes the information needed in specifications is not provided by the source code itself. Typical examples are specifications that express something about the control flow, e.g., how often or in which order methods are called, or about the used resources, e.g., to limit the number of objects. This additional knowledge can be modeled with ghost variables.

A ghost variable in JML can be defined as a class/instance member or as a local variable. In both cases, it is declared like a normal Java variable, but inside a JML annotation preceded by the keyword `ghost`. The used type may be a specification-only type such as `\bigint` (see Section 7.8). The initial value of a ghost variable can be directly assigned at its declaration. Its value can be updated during method execution by a `set` statement. This is a JML annotation statement within a method body, consisting of a keyword `set` followed by an assignment. The left-hand side of the assignment has to be a ghost variable and the right sight can be any side-effect-free JML expression.

Example 7.24. Consider class `LinkedList` in Listing 7.12, that represents a linked data structure. In general, this structure could be circular. To specify that it really is a list, i.e., that it is finite and noncircular, we use a ghost field `length` to represent the length of a list. Since there may be more elements than Java's primitive `int` type can accommodate, we use the specification-only type `\bigint`. The invariant states that `length` is always positive and that the `length` of the tail is always smaller than the current one. From this, we may conclude the above property.

7.7.3 Ghost Variables Versus Model Fields

It is important to understand the difference between model and ghost variables. Both are variables that are used for specification purposes only, and they do not occur during the execution of the program.

```
1 public class LinkedList {
2     private /*@ spec_public @*/ int value;
3     private /*@ spec_public nullable @*/ LinkedList next;
4
5     //@ public ghost \bigint length;
6     //@ public invariant 0 < length;
7     //@ public invariant next == null || next.length+1 == length;
8 }
```

Listing 7.12 Using a ghost field to track recursion depth

However, model variables provide an abstract representation of the state. If the underlying state changes, implicitly the model variable also changes. Often it is possible to define this relationship explicitly as a translation, but sometimes it can only be given in a nonconstructive manner (or even as a dependency relation).

In contrast, ghost variables extend the state. They provide some additional information that cannot be directly related to the object state. Ghost variables are often used to keep track of the events that have happened on an object, e.g., which methods have been called, how often have these methods been called etc. There also exists work where ghost variables have been used to keep track of the resources used by the program: every time a new object is created, there is an associated `set` annotation that increases a resource counter, modeled as a ghost variable [Barthe et al., 2005]. In this way, the specification can state something about the number of objects that have been created by the program. This information allows then to define a resource analysis over the application.

7.8 Integer Semantics

Since JML incorporates Java expressions, specifications also adhere to the semantics of the Java numerical data types. This means in particular that always special care has to be taken regarding overflows in integer operations[12]. Undoubtedly, dealing with finite numerical data types is a very common source of programming errors. The most infamous example from the real world is the maiden flight of Ariane 5, where conversion of 64-bit floating-point data to 16-bit integers finally caused the spacecraft to be destroyed just seconds after lift off [Nuseibeh, 1997]. It is thus desirable to detect such errors and to not repeat them in the specification. We will show how to avoid this problem through the use of JML's `\bigint` data type, that represents the mathematical integers. This section does not discuss semantics of integral data types in general; those can be found in [Beckert et al., 2007, Chapter 12] or (more elaborate) in [Schlager, 2002].

[12] Similar issues arise with rounding in floating-point operations, which however will not be covered here.

Example 7.25. Regard the short method `mult()` below; it returns `a*b`, but this is not multiplication in the mathematical sense, since an overflow may occur.

```
public int mult (int a, int b) { return a*b; }
```

The naive specification `ensures \result == a*b;` would be trivially true since JML uses the very same overflow semantics as in Java.

This example shows a feature of Java that may be a large source of confusion. Integer operators in Java are often misunderstood to equal their mathematical counterparts, see, e.g., the survey by Chalin [2003]. But the actual mathematical functionality[13] represented by, e.g., `a*b` (where both are `int` expressions) is $((a+2^{31})\cdot(b+2^{31}) \mod 2^{32})-2^{31})$. In addition, these operators are overloaded—the `*` operator has different semantics if one operand is of type `long` (64-bit integers). This means that, in many situations, naive specifications are just incorrect due to the presence of overflows. For instance, in Listing 7.4, the invariant that credits are nonnegative can be broken by method `addCredits()`, that does not check for overflows.

Example 7.26. To display even more obscure characteristics of overflow semantics, the following Boolean JML expression is trivially true. We leave it to the reader to find out with which element the quantifier would be instantiated.

JML

```
(\exists int x; x-1 > x
                && (\forall int y; x <= y))
                && x == -x
                && x != 0 && x * 2 == 0);
```

JML

Besides Java's bounded integer types (also known as *bit vector* types), JML offers the specification only primitive type `\bigint` that represents the mathematical integers $\mathbb{Z}$. 'Specification only' means that, besides variables bound by a quantifier, only ghost variables and ghost/model fields can be declared with type `\bigint`. The Java standard library also provides a type called `BigInteger`, that represents arbitrary precision integers. While `\bigint` is a primitive type with an infinite number of elements, `BigInteger` is just a regular Java object type. This means, in particular, that instances of `BigInteger` must be created through constructors and that quantification makes little sense since it only ranges over the (finitely many) created instances. It is therefore inadequate for specification purposes.

Let us come back to Example 7.25. How can we specify that there is no overflow? In Java, all arithmetic operations are *unchecked*, i.e., an overflow is not indicated in any way, e.g., by exceptions. A precondition like `a*b <= Integer.MAX_VALUE` is trivially true. Instead, we can apply numerical conversion to `\bigint` to expressions

[13] More mathematically speaking, the `int` data type with operators `+` and `*` forms a finite Abelian ring that is isomorphic to $\mathbb{Z}/\mathbb{Z}_{2^{32}}$. This means that addition and multiplication are commutative, associative, and distributive; but there are zero-dividers—as shown in Example 7.26.

of type `int`. Note that this kind of conversion, a widening, has no effect on the *values* of `a` and `b`, but on the semantics of the `*` operator. Under the preconditions that the (mathematical) product of a and b is within the bounds of `int`, we can ensures that the result is indeed the mathematical product:

Java + JML
```
//@ requires Integer.MIN_VALUE <= (\bigint) a * (\bigint) b;
//@ requires Integer.MAX_VALUE >= (\bigint) a * (\bigint) b;
//@ ensures \result == (\bigint) a * (\bigint) b;
public int mult (int a, int b) { return a*b; }
```
Java + JML

Because this specification is tedious to write and even more horrible to read, classes and methods can be annotated in JML with *math modifiers* [Chalin, 2004]. The default integer semantics in specifications can be changed by declaring the method `spec_bigint_math`, that achieves the above while saving to write down casts explicitly.

Java + JML
```
//@ requires Integer.MIN_VALUE <= a * b;
//@ requires Integer.MAX_VALUE >= a * b;
//@ ensures \result == a * b;
public /*@ spec_bigint_math @*/ int mult (int a, int b) {
    return a*b;
}
```
Java + JML

An even simpler way to express the absence of overflows is to change the semantics of the Java implementation through the `code_safe_math` modifier. It causes the program to be interpreted as if operations were checked, leading to an exception in case of overflow. The only thing left to show is that there are no exceptions:

Java + JML
```
//@ signals_only \nothing;
public /*@ code_safe_math @*/ int mult (int a, int b) {
    return a*b;
}
```
Java + JML

There are six math modifiers in total, declaring integer expressions in specifications or code to be interpreted as either Java integers with default operations, mathematical integers, or Java integers with checked operations. While these modifiers are currently not directly supported, the KeY prover offers to select different integer semantics with a similar effect; see Section 15.2.3 on page 531 and Section 5.4.

7.9 Auxiliary Specification for Verification

The previously discussed specification constructs are essential to the Design by Contract philosophy and relevant to all analysis techniques. However, for static verification of Java programs it is typically required to provide some additional information, like the locations a method might access (Section 7.9.1); guidance for the verification tool in the presence of loops via loop invariants (Section 7.9.2); or in general via assert statements (Section 7.9.3).

7.9.1 Framing

An important aspect of verification is modularity. Each method is verified in isolation, and any method call inside a body is abstracted by its method specification. To achieve this, it is not enough to specify what a method does; it is also required to specify what a method does *not* do. This is known as the *frame problem* [Borgida et al., 1995, Müller et al., 2003]. Basically, for modular verification one needs to know what is the *frame* of a method, i.e., what are the variables that may be changed at most by the method, and what is the *antiframe*, i.e., which variables must not be changed by the method.

To specify this, JML uses the *assignable clause*. This provides a set of variable locations that may be modified by a method (thus, it may be an over-approximation of the actual set of locations that is modified by the method). Location sets can be given through comma separated lists of single variables or one of the special keywords `\nothing` (only locations of newly allocated objects may be changed, corresponds to weak purity, see Section 7.3.5), `\everything` (any location may be changed), `this.*` (all locations provided by the current object), and `array[*]` or `array[i..j]` (all elements in the array or between indices `i` and `j`). Whereas assignable clauses are attached to single specification cases, pure methods are defined to have an empty frame under *any* precondition. The extension to JML that is used in KeY provides additional constructs to specify frames, offering more flexibility; see Section 9.3.2. Most importantly, the keyword `\strictly_nothing` denotes strictly the empty set of locations; strictly pure methods are annotated with `strictly_pure`, see Section 7.3.5.

JML also allows one to add an `accessible` clause to method specifications, Section 9.9.10 of the JML reference manual [Leavens et al., 2013]. This clause provides a set of variable locations on which the observable behavior of the method depends. The way this clause is used in KeY differs from and considerably goes beyond standard JML. We postpone explanation of `accessible` clauses to Sections 8.3.2 and 9.3.

Example 7.27. Listing 7.13 contains the specification of Listing 7.1, but with assignable clauses added. Method `addCredits` increases the achieved credits, which means that it may have to update the master flag to maintain the invariant. There-

fore, the assignable clause of this method lists the instance variables `credits` and `master`. Even though the variables are not modified directly by the method, it is required to list them in the assignable clause, because they may be modified during the method execution. Methods `updateCredits`, `changeToMaster` and `setName` modify only one instance variable, that is listed in the assignable clause of their method specifications. Finally, method `getName` is specified as a pure method, that automatically implies that the assignable clause is `\nothing` by default.

Of course, it would be possible to add the information in the assignable clause to the postcondition, explicitly specifying that the variables not mentioned in the assignable clause are not changed. But this is not a satisfactory solution: a class might have many variables and only a few are typically changed by a method. Moreover, when a new variable is added, for every method that does not change it, an additional postcondition about this variable not being changed would have to be added. As one can imagine, this is error-prone, and leads to overly verbose specifications.

For readers who would like to dive further into the topic of modularity, Chapter 9 is entirely dedicated to aspects of modularity in specification and verification. In particular, it introduces a specification-only type `\locSet`, which represents sets of program locations as first class subjects.

7.9.2 Loop Invariants

A verification tool typically needs some guidance in presence of loops to verify that a method implementation complies to its specification. This is due to the general impossibility to statically evaluate the loop body repeatedly until the loop condition evaluates to false. The number of iterations is not static but depends on dynamic input parameters and initial states. In program verification, the dominating solution to this problem is the usage of a *loop invariant* [Floyd, 1967, Hoare, 1969]. This is a formula whose validity is preserved by the loop body (given the loop condition was true before). From this we can conclude that, if the entire loop starts in a state where the loop invariant holds, then it will still hold once the loop terminates in addition to the negated loop condition[14].

There exit approaches to automated invariant generation [German and Wegbreit, 1975, Karr, 1976] (see also Chapter 6), and the recent years saw a very dynamic development in this area. Yet, much more needs to be done to automatically find good invariants, and to integrate that into verification tools. (The bottleneck is currently not to generate formulas that are invariant over the loop body, but to identify those that contribute to the overall correctness proof.) For the time being, finding loop invariants that allow us to verify some code unit is still a largely manual task. Guidance on how to write loop invariants is beyond our scope here. But the reader can refer to Section 16.3 in this book.

[14] In fact, to reason about Java, it is required to also support abrupt loop termination, caused by an exception or programmatically by a `return`, `break` or `continue` statement.

```
public class Student {
    private /*@ spec_public @*/ String name;

    /*@ public invariant credits >= 0;
      @*/
    private /*@ spec_public @*/ int credits;

    /*@ public invariant credits < 180 ==> !master &&
      @                  credits >= 180 ==> master;
      @*/
    private /*@ spec_public @*/ boolean master;

    /*@ requires c >= 0;
      @ ensures credits == \old(credits) + c;
      @ assignable credits, master;
      @*/
    public void addCredits(int c) {
        updateCredits(c);
        if (credits >= 180) {
            changeToMaster();
        }
    }

    /*@ requires c >= 0;
      @ ensures credits == \old(credits) + c;
      @ assignable credits;
      @*/
    private void updateCredits(int c) {
        credits += c;
    }

    /*@ requires credits >= 180;
      @ ensures master;
      @ assignable master;
      @*/
    private void changeToMaster() {
        master = true;
    }

    /*@ ensures this.name == name;
      @ assignable this.name;
      @*/
    public void setName(String name) {
        this.name = name;
    }

    /*@ ensures \result == name;
      @*/
    public /*@ pure @*/ String getName() {
        return name;
    }
}
```

Listing 7.13 Full specification of `Student` with assignable clauses

In the first place, loop invariants are proof artifacts, comparable to induction hypotheses in inductive proofs. But JML offers the possibility to annotate loops, in the source code, with invariants, to be used by verification tools during the proof process. The corresponding keyword is `maintaining` or `loop_invariant`, followed by a Boolean JML expression. The JML comment that contains this must be placed directly in front of the loop. Notice that a loop invariant may contain an `\old(`*E*`)` expression. This refers to the value of the expression *E* before the method started, *not* to the value of *E* at the previous iteration of the loop.

As long as no `diverges` clause (see Section 7.3.3) is defined, it is required to prove that a method terminates. In presence of a loop this is only possible if a `decreasing` clause (also named *variant*) is provided together with the loop invariant. The decreasing term must be well-founded, which means that it cannot decrease forever. For the decreasing clause, it has to be shown that it is strictly decreasing for each loop iteration and that it evaluates to a nonnegative value in any state satisfying the invariant. Therefore, this is sufficient to conclude that the loop terminates. In JML the decreasing term is specified via keyword `decreasing`, followed by an expression of type integer.

Example 7.28. The loop invariant in method `search` in Listing 7.14 shows a very common loop invariant pattern for methods iterating over an array. All the elements that have been examined so far respect a certain property, and the loop terminates at least when all the elements in the array have been examined. Variable `found` indicates in this example whether the element to search is contained in the already examined elements or not. A loop invariant restricting the range of loop variables is typically always needed, but not sufficient alone. In this example, the range of loop variable `i` is limited to valid array indices (`0 <= i && i <= a.length`). Finally, a well-founded decreasing clause is provided, that allows one to prove termination.

```
1  /*@ normal_behavior
2    @ requires a != null;
3    @ ensures \result == (\exists int i;
4    @                              0 <= i && i < a.length; a[i] == val);
5    @*/
6  public boolean search(int[] a, int val) {
7      int i = 0;
8      /*@ maintaining !(\exists int j;  0 <= j && j < i; a[j] == val);
9        @ maintaining 0 <= i && i <= a.length;
10       @ decreasing a.length - i;
11       @*/
12     while (i < a.length) {
13         if (a[i] == val)
14             return true;
15         i++;
16     }
17     return false;
18 }
```

Listing 7.14 Loop invariant example to search an element in an array

Loop invariants are sensitive to the frame problem as discussed for method calls in Section 7.9.1. Basically, it is necessary to specify which variable locations might be changed by a loop and which not. In KeY this is done with the assignable clause. Only locations have to be specified since local variables changed by the loop are computed automatically by KeY. Note that a loop assignable clause refers to all locations that are possibly changed by *any* loop iteration, not just a single one. For instance, if an array `a` is manipulated at a (variable) index `i`, it is not enough to specify **`assignable a[i];`** but instead **`assignable a[*];`** refers to any element.

Example 7.29. Method `sum` of Listing 7.15 computes the sum of the values provided by an array using a for-each loop. The assignable clause is explicitly set to **`\strictly_nothing`** to make sure that no objects are created during loop execution. Local variables are not listed in the assignable clause since they are automatically added by KeY.

```
1  /*@ requires array != null;
2    @ ensures \result == (\sum int i;
3    @                          0 <= i && i < array.length; array[i]);
4    @*/
5  public static int sum(int[] array) {
6      int sum = 0;
7      /*@ maintaining sum == (\sum int j;
8        @                           0 <= j && j < \index; array[j]);
9        @ maintaining \index >= 0 && \index <= array.length;
10       @ decreasing array.length - \index;
11       @ assignable \strictly_nothing;
12       @*/
13     for (int value : array) {
14         sum += value;
15     }
16     return sum;
17 }
```

Listing 7.15 Loop invariant example to compute the sum of an array

Java 1.5 introduced so called *enhanced for* loops (also called *foreach* loops, see [Gosling et al., 2013, Section 14.14]) that iterate over elements of an array or a collection. Here, the index variable is only implicit. As proposed by Cok [2008], the keyword **`\index`** refers to this value. An example is also shown in Listing 7.15.

7.9.3 Assertions and Block Contracts

Sometimes, the program verifier needs some additional guidance in proving a contract. This can be given as an intermediate *assertion*: `assert` P`;`[15] We have to prove that P is true in this intermediate state. Afterwards, we can use this additional knowledge to prove the overall proof obligation. In this way, assertions in the code are similar to cuts in proofs. JML also provides a dual `assume` statement. It is supposed to be assumed to be true without verifying it.

While the intuition behind these constructs is clear, they perturb the concept of design by contract. In particular, the statement `assume false;` would make any contract trivially satisfied. For this reason, in KeY `assert` and `assume` are replaced by the more flexible concept of *block contracts* [Wacker, 2012]. The behavior of any Java block can be specified in the same way as a method is specified (see Section 7.1) by placing the specification directly in front of the Java block. It can contain any clause that is available for method contracts. The only differences are: First, that `\old` represents the value before executing the block, and not the one before executing the method, and second, that the `\signals_only` definition must be explicitly specified, because a block has no throws definition from which it can be computed. Listing 7.16 shows the usage of a block contract within a longer method. The block itself swaps the value of the two variables `x` and `y`.

```
1  public void swapInBetween() {
2      ⋮
3      /*@ ensures x == \old(y);
4        @ ensures y == \old(x);
5        @ assignable x, y;
6        @ signals_only \nothing;
7        @*/
8      {
9          y = x + y;
10         x = y - x;
11         y = y - x;
12     }
13     ⋮
14 }
```

Listing 7.16 Usage of a block contract to swap two values

[15] JML `assert` statements are not to be confused with Java `assert` statements. The former are only present in specifications and meant to guide the prover. The latter is an actual program statement to be checked at runtime, that raises an exception upon failure.

7.10 Conclusion

This chapter has provided a short overview of the Java Modeling Language (JML), its main features and how it can be used to describe intended program behavior. More information about JML, including people involved in the community effort, the reference manual, tools supporting JML, teaching material, and relevant papers are available from the JML webpage jmlspecs.org.

To conclude, we briefly discuss other related program annotation languages, and the wide range of tool support that exists for JML.

7.10.1 Tool Support for JML

One of the strong points of JML is that many different kinds of tool support exist for it, covering the whole spectrum of formal methods. For an—unfortunately outdated—overview of JML tools, the reader may refer to [Burdy et al., 2003a]. We briefly describe a few, more information is available from the JML webpage. It should be noted that most recent tool development, including KeY, aims at combining different kinds of tool support within a single environment. In particular both the JMLEclipse [Chalin et al., 2010] and OpenJML [Cok, 2011] tool suites each include their own runtime checker, static analysis tool, and test case generator.

The original developers of JML started the work on JML with runtime checking in mind, i.e., JML should provide support to check pre- and postconditions *during* program execution. Many different tools exist that support this, for different subsets of JML, e.g., JMLRac [Cheon, 2003], AspectJML [Rebêlo et al., 2014], and as mentioned subtools of JMLEclipse and OpenJML. The runtime checking approach has also been the basis for model checking of JML annotated programs in Bogor: every program annotation is translated into an assertion, that is validated during the software model checking procedure [Robby et al., 2006].

JML is also used for test case generation. JMLunitNG [Zimmerman and Nagmoti, 2010] extends standard unit testing with knowledge derived from the program annotations. It is included in the OpenJML tool suite. The test case generation feature of KeY (see Chapter 12) uses information from the KeY prover to improve test case generation. As mentioned, also JMLEclipse provides support for test case generation, based on the JET tool [Cheon, 2007]. A recently developed test case generation tool is JMLOK2 [Milanez et al., 2014].

There are also several tools that support static checking of JML annotations, i.e., at compile time, without executing the program. These tools differ in the level of automation and the support they provide for manually constructing a proof. In general, the more user intervention is possible, the more complex properties can be verified. KeY is a typical example of a tool that can verify complex properties, but may require manual intervention. Other tools in this category are Krakatoa [Marché et al., 2004] and KIV [Balser et al., 2000, Stenzel, 2005].

ESC/Java [Leino et al., 2000] and its successor ESC/Java2 [Cok and Kiniry, 2005] follow the *auto-active verification* paradigm. They intend to provide automatic support for proving program correctness (if necessary, compromising soundness or completeness). Another tool that has been developed with automation in mind is JACK [Barthe et al., 2007], however it also provides support to fall back on interactive proving using Coq. Also the static verification subtools of JMLEclipse and OpenJML are developed with automation in mind. Finally, the VerCors tool set [Amighi et al., 2012] combines separation logic support for concurrent programs with JML annotations.

Last, it should be mentioned that there are also very different tools that support JML. There is a JMLdoc facility that allows one to generate web pages for JML annotations (similar to Javadoc). There also exist tools that generate JML annotations. These range from generating arbitrary JML specifications such as Daikon [Ernst et al., 2007], and Houdini [Flanagan and Leino, 2000] to tools that can generate one specific class of annotations, such as Chase [Cataño and Huisman, 2003]. The KeY project provides support for editing JML specifications in Eclipse. The Eclipse extension is called JML Editing and offers features such as syntax highlighting and refactoring. It is available at www.key-project.org/eclipse/JMLEditing.

7.10.2 Comparison to Other Program Annotation Languages

The JML language has been a pioneer in the area of *annotation based specification languages* dedicated to a single programming language. As explained above, in Section II, the intention of the developers was to provide a language to write assertions for Java programs. Its design has been inspired by earlier experiences of some of the developers on annotating Modula-3 [Leino and Nelson, 1998], and C++ (the Larch project) [Cheon and Leavens, 1994].

As a major difference to more abstract specification languages, such as Z [Spivey, 1992], VDM [Fitzgerald et al., 2008], Alloy [Jackson, 2003], the B method [Abrial, 1996], and UML [Rumbaugh et al., 2010], JML focuses solely on the phases of software development in which source code is written. It is also primarily intended to specify existing code, rather than to implement programs according to a preexisting specification. However, it should be noted that some work has been done on translating specifications in these high level languages into JML, e.g., for B [Cataño et al., 2012].

JML also has a number of similarities to the Object Constraint Language (OCL) [Warmer and Kleppe, 1999], a language for annotating UML class diagrams with constraints on object states. It is used for both meta modeling and application modeling. In the latter case, annotations are added to the fine design of the implementation, much like class and method specifications in JML. But unlike JML, OCL does not subscribe to any programming language, and therefore does not address language-specific concerns (like, e.g., exceptions). Earlier versions of KeY supported OCL as well [Beckert et al., 2007], but this has been discontinued.

JML has been an inspiration for many other program annotation languages that have emerged over the last years, such as the ANSI/ISO C Specification Language (ACSL) [Baudin et al., 2010], and the language of the VCC tool (formerly "Verifying C Compiler") [Cohen et al., 2009], Spec# for C# [Barnett et al., 2005a], and Dafny [Leino, 2010], that is an integrated annotation *and* programming language.

Recently, *separation logic* [O'Hearn et al., 2001, 2004] has become a popular alternative to Hoare logic to specify program behavior. Separation logic allows explicit reasoning about the heap, that makes it suitable for reasoning about pointer programs, and for concurrent programs. Several approaches exist that combine separation logic with JML (or JML like languages), to enable reasoning about pointers and/or concurrent programs, while maintaining the expressiveness of JML [Tuerk, 2009, Jacobs and Piessens, 2011, Amighi et al., 2012]. The *dynamic frame* approach [Kassios, 2011, Weiß, 2011] offers even more flexibility to specify and reason about complex heap modifications. KeY uses its own extension to JML, that makes use of dynamic frames; it is covered in Section 9.3.