2. Background on JML

The *Java Modeling Language* (*JML*) [Leavens et al., 2006a, 2008] is a language for writing formal specifications about Java programs. Its authors classify it as a *behavioural interface specification language* [Hatcliff et al., 2009], i.e., a language for specifying the functional behaviour of the interfaces of program modules (such as, in an object-oriented setting, methods and classes). Using a metaphor proposed by Meyer [1992], one can also call it a *design by contract* language: its specifications can naturally be understood as "legal contracts" between the clients and the implementers of each module, where the responsibility for overall correctness is split between the contracting parties. Some other examples for this general type of specification language are Larch [Guttag et al., 1993], Eiffel [Meyer, 2000], SPARK [Barnes, 2003], Spec# [Barnett et al., 2005] and OCL [OCL 2006]. In the area of design-by-contract style specification for Java, JML can today be considered the de facto standard, and it is supported by a large number of tools (refer to Burdy et al. [2005] for a slightly outdated overview of JML tools).

The KeY tool accepts JML as one of its input formats, and translates it into its verification logic JavaDL. Alternatively, specifications can be written in JavaDL directly, but using JML usually means better readability and less verbosity, and requires less special knowledge. The present chapter serves as an introduction to the parts of JML relevant for this thesis, and, by this example, to the basics of design-by-contract style specification for object-oriented software in general. Particular emphasis is placed on the difficult issue of *data abstraction* in specifications. i.e., the question how classes and interfaces can be specified in an abstract way, without referring to the concrete data structures used in any particular implementation of the specified functionality. These issues motivate the extensions to JML proposed in Chapter 3, and the changes in the definition of JavaDL in Chapter 5, compared to the traditional definition of this logic as given by Beckert [2001]; Beckert et al. [2007]. Just as the definition of JML [Leavens et al., 2008] is written in natural language rather than in some formal notation, both the present chapter and Chapter 3 have a less formal nature than later chapters of this thesis. Formalisations of parts of JML are explored by Leavens et al. [2006b] and by Bruns [2009].

Outline We start with a look at JML's expression sub-language in Section 2.1. Section 2.2 then introduces what is arguably JML's most elementary specification feature, namely basic *method contracts* consisting of *pre- and postconditions*, *modifies clauses*, and sometimes *diverges clauses*. Besides method contracts, a second main feature of design by contract specification for object-oriented software in general and of JML in particular is that of *object invariants*, which are the topic of Section 2.3. In Section 2.4 we take a look at the interplay between *subtyping* and *inheritance* on one hand and JML specifications on the other hand. Section 2.5 deals with the question of *data abstraction* in specifications, and presents JML's answers to this question, namely *ghost fields*, *model fields*, *data groups*, and *pure methods* with *depends clauses*. We conclude in Section 2.6.

2.1. Expressions

JML is a specification language dedicated to specifying Java programs. As such, it aims to provide a flat learning curve for experienced Java programmers new to formal specification, and strives to be as close to the Java language as possible. Its expressions are mostly just Java expressions, which for pre- and postconditions and invariants are of type **`boolean`**. The only restriction compared to Java expressions is that JML expressions must not have side-effects on the program state, in order to give them a logical, rather than imperative, character. This means that Java expressions like `i++` or calls to state-changing methods are not legal JML expressions.

On the other hand, JML features a number of extensions over the expressions allowed in Java. Two important such extensions are the universal quantifier **`\forall`** and the existential quantifier **`\exists`**. An expression `(`**`\forall`**` T x; b1; b2)`, where `T` is a type, where `x` is a variable identifier bound in `b1` and `b2`, and where `b1` and `b2` are **`boolean`** expressions, means that for all instances `x` of type `T` for which the expression `b1` holds, expression `b2` must also hold. Analogously, an expression `(`**`\exists`**` T x; b1; b2)` means that *there is* an instance of type `T` such that *both* `b1` and `b2` hold. One can also write `(`**`\forall`**` T x; b)` and `(`**`\exists`**` T x; b)` to abbreviate the expressions `(`**`\forall`**` T x; `**`true`**`; b)` and `(`**`\exists`**` T x; `**`true`**`; b)`, respectively.

Some other extensions are only available in method postconditions. Here, the method's return value, if applicable, can be referred to using the keyword **`\result`**. The value that an expression `e` had in the pre-state before method execution (but after assigning values to its formal parameters) can be denoted as **`\old`**`(e)`. For an expression `e` of a reference type, **`\fresh`**`(e)` is a **`boolean`** expression which is true if and only if the object to which `e` evaluates was not yet created in the pre-state.

JML also features a meta-type **`\TYPE`** whose instances are the types themselves, and a few operators for this type. Firstly, for every type `T`, **`\type`**`(T)` is an expression of type **`\TYPE`** evaluating to `T`. Also, for every expression `e`, **`\typeof`**`(e)` is an expression of type **`\TYPE`** which evaluates to the dynamic type of the value of `e`. Thus, for example, "**`\typeof`**`(3+4) == `**`\type`**`(Object)`" is a JML expression that always evaluates to false.

A problem with using Java expressions directly for specification is that in Java, expressions do not have a defined value in every state; sometimes, trying to evaluate an expression instead results in an exception being thrown. For example, evaluating the expression `x/y` in a state where the sub-expression `y` has the value 0 will trigger an exception of type `java.lang.ArithmeticException`. Because of this, Java's **`boolean`** expressions used in JML to formulate, for example, pre- and postconditions and invariants, are essentially *three-valued* formulas, which in every state evaluate to either "true", "false", or "undefined" (if an exception is thrown).

In specifications, we may nevertheless want to stay within the world of classical two-valued logic, which is generally easier to handle in verification tools. A common approach to achieve this, which used to be employed by JML, is known as *underspecification* [Gries and Schneider, 1995]. Here, we use fixed, but unknown ("underspecified") regular values instead of a dedicated "undefined" value. This works largely well, because "good" specifications avoid cases of undefinedness anyway. For example, specifications may use Java's short-cut evaluation of the operators `||` and `&&` to avoid throwing exceptions: in "`y == 0 || x/y == 3`", the value (or lack thereof) of `x/0` is irrelevant, because the subexpression `x/y` is never evaluated in a state where `y` has the value 0.

However, if the person writing the specification makes a mistake, specifications can be insufficiently guarded against undefinedness, and in these cases simple underspecification can lead to unexpected results. For example, the expression "`x/y == 3 || y == 0`" throws an exception in Java if `y` has the value 0, but evaluates to true with underspecification. Another example is "`x/0 == x/0`", which always throws an exception in Java, but which is always true if underspecification is used: the value of `x/0` may be unknown, but it is the same on both sides of the equality operator.

After these discrepancies between the underspecification semantics of JML and the semantics of Java had been criticised by Chalin [2007a], JML switched to an approach called *strong validity* [Chalin, 2007b]. Here, a top-level pre- or postcondition or invariant evaluates to true if it logically evaluates to true *and* if it would not throw an exception in Java; it evaluates to false otherwise. In other words, Java's three-valued semantics is used, but at the top level of a specification expression, the value "undefined" is converted into the value "false". For example, a top-level expression "`x/0 == x/0`" and its negation "`!(x/0 == x/0)`" both always evaluate to false: in both cases, the subexpression `x/0` is "undefined", making the overall expression "undefined" also, which is then converted into the value “false”. Short-circuit evaluation is respected, so “**true** || x/0 == x/0” is always true, while “x/0 == x/0 || **true**” is always false. Note that even though—in contrast to the underspecification approach—a third truth value is involved, this does not imply that a verification system supporting JML must necessarily use many-valued logic; see for example the work by Darvas et al. [2008].

2.2. Method Contracts

JML specifications can be written directly into Java source code files, where they appear as comments that start with the character “@” to distinguish them from other, non-JML comments. A first example for a Java class annotated with JML is shown in Figure 2.1.

Those comments that precede method declarations in Figure 2.1 specify *method contracts* (sometimes called just *contracts*, or *specification cases* in JML) for the respective method. As a first approximation, we can see method contracts as lists consisting of an arbitrary number of *preconditions* and *postconditions* for the method. A method can be annotated with multiple contracts, separated by the **also** keyword. For example, method get in line 27 has two contracts, one starting with “**public normal_behaviour**” and one starting with “**public exceptional_behaviour**”.

As demonstrated by these occurrences of the keyword **public**, Java’s *visibility* system extends also to JML specifications. Specification elements, such as contracts, may refer only to fields and methods of higher (more public) visibility than the specification element itself. However, a field or method may be marked as **spec_protected** or **spec_public** (as in lines 2 and 4 of Figure 2.1) to increase its visibility in JML specifications, without changing it on the level of Java. Beyond this syntactical requirement, visibilities do not carry semantical meaning. They may however influence the behaviour of a modular verifier (Chapter 6).

The elements of method contracts that we consider in this section are *pre- and postconditions* (Subsection 2.2.1), *modifies clauses* (Subsection 2.2.2), and *diverges clauses* (Subsection 2.2.3). Another element occurring as a part of method contracts, the *depends clause*, is introduced later in Section 2.5.

2.2.1. Pre- and Postconditions

Method preconditions are declared with the keyword **requires**, followed by a **boolean** JML expression. Postconditions are declared with the keyword **ensures**. Their meaning is that if all the preconditions of a contract hold before method execution and if the method terminates normally—i.e., it terminates not by throw-

Java + JML

```java
public class ArrayList {
    private /*@spec_public nullable@*/ Object[] array
                                                 = new Object[10];
    private /*@spec_public@*/ int size = 0;

    /*@ private invariant array != null;
      @ private invariant 0 <= size && size <= array.length;
      @ private invariant (\forall int i; 0 <= i && i < size;
      @                                   array[i] != null);
      @ private invariant \typeof(array) == \type(Object[]);
      @*/

    /*@ public normal_behaviour
      @   ensures \result == size;
      @*/
    public /*@pure@*/ int size() {
        return size;
    }

    /*@ public normal_behaviour
      @   requires 0 <= index && index < size;
      @   ensures \result == array[index];
      @ also public exceptional_behaviour
      @   requires index < 0 || size <= index;
      @   signals_only IndexOutOfBoundsException;
      @*/
    public /*@pure@*/ Object get(int index) {
        if(index < 0 || size <= index) {
            throw new IndexOutOfBoundsException();
        } else {
            return array[index];
        }
    }

    /*@ public normal_behaviour
      @   assignable array, array[*];
      @   ensures size == \old(size) + 1 && array[size - 1] == o;
      @   ensures (\forall int i; 0 <= i && i < size - 1;
      @                           array[i] == \old(array[i]));
      @*/
     public void add(Object o) {...}
}
```

Java + JML

Figure 2.1.: Java class `ArrayList` with JML specifications

ing an exception—then all the postconditions declared with **`ensures`** must hold afterwards. For example, lines 21 and 22 of Figure 2.1 state that if the method `get` is called with an argument `index` that lies between `0` and `size`, and if the method terminates normally, then its return value is the object contained in the array `array` at position `index`.

Postconditions for the case of *exceptional* termination (i.e., termination by throwing an exception) are specified using clauses of the form "**`signals`**`(E e) b`", where `E` is some subtype of class `java.lang.Exception`, where `e` is a variable identifier bound in `b`, and where `b` is a boolean JML expression. The meaning of such a **`signals`** clause is the following: if the method terminates by throwing an exception `e` of type `E` (and if the preconditions held on method entry), then `b` must evaluate to true in the post-state.

The keyword **`normal_behaviour`** at the beginning of a contract is essentially "syntactic sugar" for a postcondition "**`signals`**`(Exception e)` **`false`**": it means that the method must not throw any exception if the contract's preconditions hold. Conversely, the keyword **`exceptional_behaviour`** is syntactic sugar for "**`ensures false`**", meaning that the method must not terminate *normally*. Another abbreviation are clauses of the form "**`signals_only`** $\texttt{E}_1$`,...,`$\texttt{E}_n$", which can be spelled out as

`signals``(Exception e) e` **`instanceof`** $\texttt{E}_1$ `|| ... || e` **`instanceof`** $\texttt{E}_n$`;`

For example, the second contract of method `get` in Figure 2.1 states that if the method is called in a state where the argument `index` is out of bounds, then the method may terminate only by throwing an exception of the type `java.lang.IndexOutOfBoundsException` (or of a subtype). Like **`requires`** and **`ensures`** clauses, multiple **`signals`** clauses are connected by conjunction, so if a method terminates by throwing an exception of type `E`, then *all* signals clauses about `E` or about any of its supertypes apply.

Note that JML specifications can only talk about the method's post-state in case of normal termination, or in case of termination by throwing an object whose type is a subtype of `java.lang.Exception`. The possibility that an object could be thrown that is of some other subtype of `java.lang.Throwable`, in particular a subtype of `java.lang.Error`, is silently ignored. The rationale behind this is that such events typically represent "external" problems, which are not directly caused by the program itself. Two examples are `java.lang.OutOfMemoryError` and `java.lang.UnknownError`. As we consider these errors to be out of the program's control, we do not demand that the program guarantees that they do not occur, or that any conditions hold afterwards if they do occur.

Yet another abbreviation supported by JML is the keyword **`non_null`**. Affixing this e.g. to a method parameter `x` or to a method return type means to add to all contracts of the method an implicit precondition "**`requires`**` x != `**`null`**" or an implicit postcondition “`\result != null`”, respectively. Moreover, as proposed by Chalin and Rioux [2006], `non_null` has become the default in JML: now the `non_null` modifier is always implicitly present, unless something is explicitly labelled as `nullable`. Thus, both contracts of method `get` in Figure 2.1 have an invisible postcondition “`\result != null`”, and the contract of method `add` has an invisible precondition “`o != null`”.

2.2.2. Modifies Clauses

If a method contract is to be useful for modular verification, where method calls are dealt with only by looking at the called method's contract instead of its implementation, it must constrain *what* part of the state may be changed by the method in addition to just *how* it is changed. This act of constraining is often referred to as *framing*, the part of the state that may be changed as the *frame* of the method, and the whole issue of framing as the *frame problem* [Borgida et al., 1995].

In principle, framing can be done with a postcondition that just lists all the unchanged memory locations, and uses something like the `\old` operator to state that their current values are the same as their values in the pre-state. However, doing so is at best cumbersome due to the typically high number of unchanged locations, and at worst impossible if, in a modular setting, the program context is not entirely available. One solution to this problem is to allow universal quantification over memory locations. Another common solution, which is employed in JML, is to extend method contracts with *modifies clauses* [Guttag et al., 1993]. A modifies clause is a list of the (typically few) locations that may be modified, implying that all the others may not be modified.

In JML, modifies clauses are also called *assignable clauses*. This difference in nomenclature hints at a subtle semantical difference: when defining the precise meaning of modifies clauses, there is a choice on whether *temporary modifications* to locations not in the modifies clause are to be allowed or not. If, as above, modifies clauses are viewed as being essentially postconditions that frame the overall effect of the method, then any temporary modification that is undone before method termination can have no effect on the validity of the modifies clause. However, as the name *assignable* clause suggests, JML imposes a stricter policy: a method satisfies its assignable clause only if every single assignment executed during its execution is covered by the assignable clause. In a concurrent setting, this interpretation is advantageous, because then, modifies clauses also constrain the intermediate states of execution, which allows them to be used for reasoning about non-interference between threads. On the other hand, for sequential programs the classical, more semantic, interpretation is completely sufficient, and in this thesis we stick with it.

Modifies clauses are declared in JML with the keyword **`assignable`**. The expression after this keyword is not a normal expression, but a list of so-called “store ref expressions” denoting sets of memory locations. The overall modifies clause evaluates to the union of these individual sets. The store ref expression `o.f`, where `f` is some field defined for the type of expression `o`, denotes the singleton location set consisting of the field `f` of the object which is the value of `o` (and not, as in a normal expression, the *value* of this location). Similarly, the store ref expression `a[i]` denotes the singleton set containing the `i`th component of array `a`. A range of array components, or all components of an array, can be denoted as `a[i..j]` or `a[*]`, respectively. By `o.*` we can refer to the set of all fields of a single object `o`. There are also the keywords **`\nothing`** and **`\everything`**, standing for the border cases of an empty set of locations and the set of all locations in the program, respectively.

We see an example for a modifies clause in line 36 of Figure 2.1. If the contract’s precondition holds when calling method `add` (i.e., if the method’s argument is different from **`null`**), then the method may modify only the field `array` of the receiver object **`this`**, and the array components of the array pointed to by **`this`**`.array`. Note that modifies clauses are always evaluated in the pre-state, so the modifies clause refers to the array which is pointed to by **`this`**`.array` at the beginning of the execution, which may be a different one than at the end of the method. Another important thing to note is that modifies clauses only constrain changes to locations on the heap (i.e., object fields and array components), but never changes to local variables: these variables are internal to a method, and any changes to them can be neither relevant nor visible to any of the method’s callers anyway. Modifies clauses also never constrain the allocation of new objects, or assignments to locations belonging to newly allocated objects. Thus, method `add` is free to allocate a new array object and assign to its components (e.g., if the old array is filled up completely), even though these locations are not (and cannot be) mentioned in the modifies clause.

Specifying that a method has *no* side effects is an important border case of modifies clauses. Such methods are called *pure*, and JML has a special keyword **`pure`** which can be attached to method declarations in order to designate a method as pure. Examples for pure methods are `size` and `get` in Figure 2.1. The meaning of the **`pure`** modifier is essentially the same as adding an implicit **`assignable \nothing`** to all contracts of the method (and using **`diverges false`** for all contracts, see Subsection 2.2.3 below). It also determines whether a method may be used in specifications or not.

As JML’s modifies clauses implicitly allow the allocation and initialisation of new objects, its pure methods are free to create, modify and even return a newly allocated object. This definition of purity is known as *weak purity*, as opposed to *strong purity*, where pure methods must not have an effect on the heap at all. The more liberal approach of weak purity is useful in practice: for example, imagine a method that is to return a pair of integers, which it can do by creating, initialising and returning an object of a class `Pair`, while being otherwise free from side effects. However, weak purity complicates the semantics of expressions. It is for example debatable whether the specification expression `newObject() == newObject()`, where the weakly pure method `newObject` creates and returns a fresh object, should be considered as always true (because the entire expression is evaluated in the same state and so both method calls return the same object), or as always false (because, as in Java, the post-state of the first method call is the pre-state of the second, and so the second call will allocate a different object), or neither. It can also be argued that such an expression should not be permitted in specifications at all [Darvas and Leino, 2007; Darvas, 2009].

2.2.3. Diverges Clauses

JML allows one to specify when a method must terminate (either normally or by throwing an exception) using so-called *diverges clauses*. A diverges clause is declared in a contract using the keyword **`diverges`**, followed by a **`boolean`** expression. The meaning of a diverges clause **`diverges`** `b` is that the method may refrain from terminating (may “diverge”) only if `b` held in the pre-state. In contrast to other clauses in contracts, the default used in case of a missing diverges clause is not the most liberal possibility, but rather the most strict one, namely “**`diverges false`**”, which says that the method must terminate under all circumstances (provided that the contract’s preconditions held in the pre-state). Leavens et al. [2008] do not define how multiple **`diverges`** clauses are to be understood; however, the natural way seems to be combining them disjunctively, i.e., multiple **`diverges`** clauses allow the method to diverge if at least one of the conditions holds in the pre-state.

2.3. Object Invariants

An *object invariant* (sometimes called *class invariant* or just *invariant*) is a consistency property on the data of the objects of some class that we want to hold “always” during program execution. Where method contracts constrain the behaviour of individual methods, object invariants are intended to constrain the behaviour of a class as a whole. The concept of object invariants goes back to Hoare [1972]. In JML, object invariants are declared with the keyword **`invariant`**. Examples are the invariants of class `ArrayList` in lines 6–10 of Figure 2.1: for the objects of this class, we want the array **`this`**`.array` to always be different from **`null`**; we want the value of the location **`this`**`.size` to always be between `0` and the array’s length; we want the array components between these bounds to be different from **`null`**; and finally, we want the dynamic type of the array to always be exactly `Object[]`, not a subtype: otherwise, method `add` might trigger a `java.lang.ArrayStoreException`, because the type of its argument might not be compatible with the type of the array.

Like method parameters and return values, field declarations of a reference type can be annotated with **`non_null`** and **`nullable`** to specify whether **`null`** values are allowed or not. For a **`non_null`** field `f`, an object invariant "**`this`**`.f != `**`null`**" is implicitly added to the specification. And like before, **`non_null`** is the default that is used when neither annotation is given explicitly. Thus, the invariant in line 6 of Figure 2.1 would not be necessary if we had not used **`nullable`** for the declaration in line 2. The reason for the use of **`nullable`** is that on variables of a reference array type, the effect of **`non_null`** goes beyond demanding that the pointer to the array is not **`null`**: it also demands that all *components* of the array be different from **`null`**. But since this is too strict for the example, where some of the array components *should* be allowed to be **`null`**, we use **`nullable`** to suppress the default, and spell out the appropriate invariants explicitly.

Above we stated that, in a correct program, invariants hold "always". As the quotation marks suggest, this is not entirely the truth: as soon as invariants mention more than one field, requiring them to hold absolutely always is too strict for practical purposes. For example, imagine a class that declares a field `x` of type **`int`** and a field `negative` of type **`boolean`**, where we want `negative` to be true exactly when `x` is negative. We can specify this in a natural way with an invariant "`negative == (x<0)`". Since the two fields cannot be updated both at the same time, this invariant must sometimes get violated when the value of `x` changes, at least for the short moment before a subsequent assignment to `negative`.

Another complication with invariants is that—in contrast to method contracts, which confine a single method (or, at most, a hierarchy of overriding methods, Section 2.4)—they are effectively *global* properties: for an invariant to hold anything near to "always" for some object *o*, it must be respected not only by all methods called on *o*, or even only by all methods defined for the class of *o*. Rather, it must be respected by all methods in the entire program. This may be surprising at first, because object invariants (as introduced above) are usually intended as constraints on a single class or on its objects, not on the entire program. One might hope that methods outside the class of *o* have no way of violating an invariant on *o*. But—unless special measures are taken—this is wrong. For example, the invariant could mention **`public`** fields, which can be assigned to anywhere in the program; or, more subtly, it could depend on other objects that can be manipulated directly, circumventing *o*. In Figure 2.1, if a method outside `ArrayList` were to obtain a reference to the array object referenced by the field `array` of some `ArrayList` object, it could modify this array directly, and thereby break the invariant in line 8.

Because of these issues, the exact semantics of object invariants is much less straightforward than it appears on first sight, and getting it "right" is a target of active research (see for example the work of Poetzsch-Heffter [1997]; Barnett et al. [2004]; Leino and Müller [2004]; Roth [2006]; Müller et al. [2006]). JML's current answer, as defined by Leavens et al. [2008], is its *visible state semantics*. Essentially (we omit **`static`** methods, **`static`** invariants, and finalisers for the sake of simplicity), this requires an invariant to hold for an object *o* in all states that are *visible* for *o*. A state is called *visible* for an object *o* if it occurs either (i) at the end of a constructor call on *o*, or (ii) at the beginning or at the end of a method call on *o*, or (iii) whenever neither such call is in progress. In other words, the invariants of *o* must hold always, except in situations where a method or constructor call on *o* is currently in progress, and where also we are neither in the post-state of a constructor call on *o* nor in pre- or post-state of a method call on *o*.

Returning to the above example, the visible state semantics solves the problem that the fields which the invariant "`negative == (x < 0)`" depends on cannot be updated simultaneously, because it permits a method to temporarily violate the invariant for its **`this`** object, as long as no other methods are called on **`this`** before the invariant is re-established. In cases where such a method call is necessary, one can annotate the called method with the keyword **`helper`**. It is then exempt from the visible state semantics, i.e., the receiver object's invariants do not have to hold in its pre- or post-state. The disadvantage of doing so is that **`helper`** methods may not rely on the invariant for satisfying their contracts. In contrast, non-**`helper`** methods are only required to satisfy their contracts for pre-states where the receiver's invariants hold. For example, method `get` in Figure 2.1 relies on the value of `array` being different from **`null`** as guaranteed by the invariant, and would not satisfy its **`normal_behaviour`** contract if it were a **`helper`** method.

2.4. Subtyping and Inheritance

A core element of object-orientation is the dynamic dispatch of method calls, where the method implementation to be executed is determined at run-time based on the dynamic type of the receiver object. Ideally, the caller does not have to know what this dynamic type is, because the objects of any subtypes of the receiver's static type can be used as if they were objects of the static type itself. This principle, which demands that subtypes conform to the behaviour of their supertypes when accessed using methods declared in a supertype, is known as *behavioural subtyping* [Liskov and Wing, 1994].

In formal specification, behavioural subtyping typically means that a method which overrides another method has a precondition that is implied by the overridden method's precondition, and, conversely, a postcondition that implies the overridden method's postcondition. If all subtypes are behavioural subtypes, then this enables modular reasoning about a dynamically bound method call using only the contract found in the static type of the receiver: if the precondition of this contract holds, then this implies that the precondition of any overriding method must also hold; and the postcondition found in the static type is guaranteed to also be established by any overriding method. For this reason, JML globally enforces behavioural subtyping, even though this limits the programmers' freedom to (mis-)use the subtyping mechanism of Java.

Behavioural subtyping is enforced in JML by inheriting method contracts to all subclasses: a method must always satisfy the contracts declared for any methods that it overrides. Subclasses are free to introduce additional contracts. We can see multiple contracts for the same method as syntactic sugar for a single, larger contract [Raghavan and Leavens, 2000]. With this view, adding contracts for an overriding method corresponds to weakening the precondition and strengthening the postcondition of the overall method contract, as above.

Object invariants, too, are inherited, and additional invariants may be introduced in subclasses. Note that the interplay between invariants and inheritance is not unproblematic for modular verification. For example, according to the visible state semantics, when calling a method on an object, the invariants of this object have to hold. In this situation, the invariant acts as a precondition for the method call. Adding invariants in a subtype corresponds to strengthening this precondition, which violates the principle that a behavioural subtype may only *weaken* method preconditions. Approaches for the modular verification of programs with invariants have to face these issues, in addition to those hinted at in Section 2.3 above. We return to this subject in Chapter 3.

2.5. Data Abstraction

Abstraction, i.e, the process of simplifying away unnecessary details while keeping something's essence, is a fundamental concept in computer science. In software development, abstraction is crucial because software systems are usually too complex to be conceived or understood by a human mind in their entirety at any point in time. Abstraction allows us to focus on some aspects of a system, while (temporarily) blocking out others. For abstraction from program structures, we can distinguish between *control abstraction*, which abstracts from control flow, and *data abstraction*, which abstracts from data structures.

A module interface is an example of abstraction: it provides a simplified outside view on the module's behaviour, freeing its clients from having to consider its internals and protecting them from being affected by changes to the internals, as long as the interface itself is not changed. This is known as the *principle of* *information hiding* [Parnas, 1972]. Design by contract specifications can be a part of such interfaces; in particular, a method contract can be seen as a description of the externally visible behaviour of a method, abstracting from how this behaviour is achieved. However, if such contracts are formulated directly over the internal data structures used in the method's implementation, then this abstraction is brittle, and the principle of information hiding is violated. For example, the contracts in Figure 2.1 are unsatisfactory in this regard, because they expose the internal data structures of `ArrayList` with the help of the **`spec_public`** modifier. If these internals are changed, then the specification is also affected. What is missing is some form of data abstraction in the specifications.

Where abstraction is the process of going from a detailed to a simple model, the notion of *refinement* refers to the other way round, i.e., going from an abstraction to a more detailed model. This process is common during software development, as the program being developed evolves from a vague idea to a running implementation. Various formalisations for such a refinement process exist, where the program is first specified formally on an abstract level using some form of "abstract variables"; where these are then refined into a representation on "concrete" variables in one or multiple refinement steps; and where the correctness of these steps is verified formally. Such techniques are for example described by Hoare [1972]; Morgan [1990]; Abrial [1996]; Hallerstede [2009].

Even without a formal concept of refinement, the phenomenon of refinement still occurs frequently in object-oriented software development. A typical case is adding new subclasses for existing classes or interfaces. For example, instead of providing class `ArrayList` in Figure 2.1 directly, we might have started with a Java interface `List`, which would then be implemented by `ArrayList`. We might also want to add other classes that use different implementation techniques than an array, such as a class `LinkedList`. In order to specify the abstract `List` interface independently of such future refinements, we again need data abstraction in specifications. And unlike the situation above, where we only aimed at specifying `ArrayList` itself in an abstract way, here using **`spec_public`** is not even an (albeit unsatisfactory) option: in a Java interface such as `List`, there are not yet any internal data structures which could be exposed to the outside world.

For data abstraction, JML features *ghost fields* (Subsection 2.5.1), as well as *model fields* and *data groups* (Subsection 2.5.2). Also, pure methods can be used for data abstraction in specifications, especially when combined with *depends clauses* (Subsection 2.5.3).

2.5.1. Ghost Fields

Ghost fields and *ghost variables* are fields and local variables that are declared, read and written solely in specifications. They are not visible to a regular compiler, and do not exist in the compiled program. Still, their semantics is exactly the same as that of ordinary fields and variables. In JML, ghost fields and variables are declared with the keyword **`ghost`**, and they are assigned to using the keyword **`set`**. An example is given in Figure 2.2, where the class `ArrayList` of Figure 2.1 is split up into a Java interface `List` and into a separate class `ArrayList` which implements the interface using an array. The behaviour of the interface is specified abstractly using a ghost field `contents`. The declaration of `contents` in line 2 uses the JML keyword **`instance`**; without it, a ghost field declaration in an interface creates a **`static`** field. This mirrors the behaviour of Java itself, where interfaces are not allowed to contain non-**`static`** fields at all, and where thus all field declarations in interfaces are assumed **`static`** by default.

Abstract specifications can often be formulated naturally using basic mathematical concepts such as sets or relations. For example, the desired behaviour of the `List` interface is similar to the behaviour of a finite mathematical sequence. Specification languages usually provide some form of mathematical vocabulary to facilitate writing such specifications. JML is no exception, but—in pursuit of its goal to stay as close to Java as possible—it does not introduce the mathematical notions into the language directly as additional primitive types. Rather, it comes with a library of so-called *model classes*, which try to sneak the mathematical concepts in through the back door by modelling them as ordinary Java classes. One such class, `JMLObjectSequence`, is used in Figure 2.2 as the type of the ghost field `contents`. The intuition behind this is to think of `contents` as a sequence of Java objects, and verification tools may attempt to map model classes like `JMLObjectSequence` directly to the mathematical concepts that they represent [Leavens et al., 2005; Darvas and Müller, 2007a]. Still, the elements of model classes are first and foremost Java objects. This implies, for example, that the `equals` method should be used for comparing two such elements instead of the regular equality operator “`==`”, as in line 31. Otherwise, the *references* to the objects are compared, which is rarely intended.

The specification of the `List` interface in Figure 2.2 has the same structure as the **`public`** specification of `ArrayList` in Figure 2.1. Instead of `size` and `array`, it uses calls to pure methods on the `contents` object, where method `int_size` returns the length of the sequence, and where method `get` retrieves an element out of the sequence.

The implementing class `ArrayList` in Figure 2.2 works exactly as in Figure 2.1. However, for satisfying its inherited contracts, which talk about the ghost field `contents` instead of the concrete data, all changes to the list must be applied to the sequence stored in the ghost field, too. The constructor initialises the ghost field to an empty sequence, and the `add` method appends its argument to the end of the sequence. Note that the expressions occurring in **`set`** statements are no exception to the rule that JML expressions must not have side effects; **`set`**

Java + JML

```
public interface List {
    //@ public ghost instance JMLObjectSequence contents;

    /*@ public normal_behaviour
      @   ensures \result == contents.int_size();
      @*/
    public /*@pure@*/ int size();

    /*@ public normal_behaviour
      @   requires 0 <= index && index < contents.int_size();
      @   ensures \result == contents.get(index);
      @ also ...
      @*/
    public /*@pure@*/ Object get(int index);

    /*@ public normal_behaviour
      @   assignable \everything; //imprecise
      @   ensures contents.int_size() == \old(contents.int_size())+1;
      @   ensures ...
      @*/
     public void add(Object o);
}

public class ArrayList implements List {
    private /*@nullable@*/ Object[] array = new Object[10];
    private int size = 0;

    public ArrayList() {/*@set contents = new JMLObjectSequence();@*/}

    /*@ /*first four invariants as in Figure 2.1*/
      @ private invariant contents.equals(
      @                  JMLObjectSequence.convertFrom(array, size));
      @*/

    public int size() {...}
    public Object get(int index) {...}
    public void add(Object o) {
        ...
        //@ set contents = contents.insertBack(o);
    }
}
```

Java + JML

Figure 2.2.: Java interface `List` specified using ghost fields, and class `ArrayList` implementing the interface

statements only modify the ghost field on the left hand side of the assignment. This fits with the use of class `JMLObjectSequence`, because the objects of this class are *immutable* in the sense that all their methods are pure. A method like `insertBack`, which is used in line 39 of Figure 2.2, returns a new object that incorporates the changes, instead of changing the original object itself.

The connection which is maintained between the abstract and the concrete representation of the data is recorded as an additional object invariant, which we can see in line 31 of Figure 2.2. It uses the `convertFrom` method of the class `JMLObjectSequence` as a convenient way to construct a sequence out of the first `size` elements of the array. The correctness of the methods `size` and `get` depends on this invariant, because the inherited contracts are formulated using the ghost field, while the methods read and return the actual data just as in Figure 2.1. The invariant bridges this gap. Invariants in such a role are known as *gluing invariants* in other contexts [Hallerstede, 2009].

An open problem in Figure 2.2 is the modifies clause of method `add` in line 17. Obviously, any implementation of `add` has to modify `contents`. However, it needs the license to modify more than just `contents`, namely, the concrete data structures used to implement the list. In order to specify this, we would need to enumerate the concrete data structures of all subclasses of `List` in the modifies clause of `add` within the interface. But this is not an option, because it would contradict the idea behind using an interface and an abstract specification in the first place. The specification in Figure 2.2 circumvents the problem by resorting to a trivial modifies clause of `\everything`. This makes the contract satisfiable by overriding methods in subclasses, but it also makes the contract effectively useless for modular reasoning about calls to `add`, because such a call must then be assumed to have an unknown effect on the entire program state. This shows that in the presence of data abstraction, framing becomes a difficult problem, which ghost fields alone cannot solve. One possible solution is to use *data groups*. These are connected to the concept of *model fields*, and the two notions are discussed together in Subsection 2.5.2 below.

2.5.2. Model Fields and Data Groups

Like ghost fields, *model fields* [Leino and Nelson, 2002; Cheon et al., 2005] are declared just as Java fields, but inside specifications. Reading a model field is done using the same notation as for reading a regular Java field or a ghost field, too. But at this point the similarity ends; despite being called "fields", model fields are in many ways more closely related to pure methods than to Java fields. Where a Java field or a ghost field represents an independent memory location (one per object), which has its own state that can be manipulated by assigning to the field, both pure methods and model fields *depend* on the state of memory locations, instead of being locations themselves. Just like the value returned by a pure method is determined by a method body, the value of a model field is determined by a *represents clause* (also known as an *abstraction function*). In JML, model fields are declared with the keyword **`model`**, and represents clauses with the keyword **`represents`**. A variation of the `List` and `ArrayList` types from Figure 2.2, where the specification uses a model field and a represents clause instead of a ghost field and **`set`** assignments, is shown in Figure 2.3.

The `List` interface in Figure 2.3 is unchanged over Figure 2.2, except that the keyword **`ghost`** is replaced by **`model`**, and except that the modifies clause of `add` now uses `contents` instead of **`\everything`**. Ignoring the modifies clause for the moment, we notice the represents clause for `contents` in line 26. The symbol `=` separates the model field to be defined and its defining expression. Here, the defining expression again uses the `convertFrom` method to construct a `JMLObjectSequence` from the array. Where in Figure 2.2 this relation between `contents` and the array was an object invariant, which was maintained by explicit **`set`** assignments (and which was sometimes broken in intermediate states, in accordance with the visible state semantics), here `contents` by definition adjusts itself immediately and automatically whenever the right hand side of the represents clause changes. This again mirrors the behaviour of (pure) methods, whose return value also immediately changes whenever a location is modified on which the return value of the method depends.

Represents clauses that use the `=` symbol are said to be in *functional form.* There is also a *relational form* of represents clauses, which allows us to define abstraction *relations* instead of only abstraction functions. The JML keyword for relational represents clauses is **`\such_that`**. It is followed by a **`boolean`** expression which describes the possible values of the model field. The functional form can be reduced to the relational form. For example, the represents clause in line 26 of Figure 2.3 can equivalently be written as

```
private represents contents
 \such_that contents == JMLObjectSequence.convertFrom(array,size);
```

In Subsection 2.5.1, we observed that objects of model classes should usually be compared with the `equals` method. Yet, here using “`==`” (either explicitly as above, or hidden in the functional form of the represents clause) works as intended. The represents clause—unlike the invariant in Figure 2.2—holds by definition, and never needs to be actively established by the program. This gives us the freedom to assume the stronger proposition that the objects are even identical, instead of just being equal with respect to `equals`. Note that the remark on *weak purity* of pure methods in Subsection 2.2.2 directly extends to model fields, too. In fact, `contents` is only weakly pure, because—as defined by the represents clause—it allocates and returns a fresh object of type `JMLObjectSequence`.

Java + JML

```
public interface List {
    //@ public model instance JMLObjectSequence contents;

    //@ ...
    public /*@pure@*/ int size();

    //@ ...
    public /*@pure@*/ Object get(int index);

    /*@ public normal_behaviour
      @   assignable contents;
      @   ensures ...
      @*/
     public void add(Object o);
}

public class ArrayList implements List {
    private /*@nullable@*/ Object[] array = new Object[10];
        //@ in contents;
        //@ maps array[*] \into contents;

    private int size = 0; //@ in contents;

    //@ /*four invariants as in Figure 2.1*/

    /*@ private represents contents
      @     = JMLObjectSequence.convertFrom(array, size);
      @*/

    public int size() {...}
    public Object get(int index) {...}
    public void add(Object o) {...}
}
```

Java + JML

Figure 2.3.: Java interface `List` specified using model fields and data groups, and class `ArrayList` implementing the interface

Another intricacy that we observe "along the way", without being overly concerned with it here, is the question of what happens if the value of the model field cannot be chosen in every state such that the represents clause is satisfied. An extreme case is a represents clause "**`represents`** `x` **`\such_that false`**", which is obviously impossible to satisfy in any state at all. More subtly, a represents clause "**represents** x **\such_that** y == 3", where y is a Java field, is also problematic, because a satisfying value for x can be found only in states where y happens to contain the value 3. There is no "official" answer for how to understand such represents clauses in the JML documentation [Leavens et al., 2008]. The simplest approach is to just consider represents clauses to be assumptions that hold in all states by definition, to accept the fact that then an inconsistent represents clause makes the entire specification trivially satisfied, and to consider the person writing the represents clause responsible for avoiding such a situation. Other, more involved solutions are explored by Breunesse and Poll [2003], Leino and Müller [2006] and Leino [2008].

In line 11 of Figure 2.3, the model field contents is used in the modifies clause of method add. Without further explanation, this would seem to be nonsensical. After all, the model field is not a location that the program could assign to, and thus **this**.contents should not be considered a legal store-ref expression. The foundation for allowing the use of model fields in modifies clauses is the concept of *data groups* [Leino, 1998]. A data group is a name referring to a set of memory locations. In JML, model fields always have two faces: in addition to their regular meaning, they are also data groups. In every state, a model field can be evaluated both to a value, as we have seen before, and to a set of locations. When used in a normal expression, a model field stands for its value, whereas at the top level of a modifies clause, it stands for its set of locations. Thus, the modifies clause in line 11 of Figure 2.3 refers to the locations in the data group interpretation of contents, and it allows the add method to modify these locations.

Like the value of a model field is defined via a represents clause, its data group interpretation is defined by declaring locations to be part of the data group with the keyword **in**. As an example, the JML annotations in lines 19 and 22 of Figure 2.3 make the locations **this**.array and **this**.size part of the data group of **this**.contents. The **in** annotation must be placed directly after the declaration of the field to be added. This kind of inclusion, where a field of an object becomes part of a data group of the *same* object, is known as *static inclusion*. In addition, there are *dynamic inclusions*, where a field of an object becomes part of the data group of some *other* object. These are declared using the keywords **maps** and **\into**. For example, adding the location **this**.contents in line 19 of Figure 2.3 is not enough; the *components* of the array pointed to by **this**.contents must be included as well, which is achieved by the annotation in line 20. Dynamic data group inclusions make data groups depend on the state: if the array field is changed, then the locations denoted by contents also change, because array[*] afterwards denotes a different set of locations than before.

In situations where only the data group aspect of a model field is desired, the regular value can simply be ignored. The type of such a model field, which is used only as a data group, does not matter. For documenting the intention that only the data group is relevant, it is however customary to use the type `JMLDataGroup`, which is another member of JML's model class library (which also contains `JMLObjectSequence`). We can use such a model field to resolve the problem with the modifies clause for `add` in Figure 2.2, without switching from ghost fields to model fields completely: we declare a model field "**`model`** `JMLDataGroup footprint`" in addition to the ghost field `contents`, and use `footprint` in the modifies clause of the `add` method. The `ArrayList` class then needs to declare the data group inclusions for `footprint` as for `contents` in Figure 2.3, but no represents clause is necessary.

2.5.3. Pure Methods and Depends Clauses

As we have seen in Subsection 2.5.2, model fields are in many ways similar to pure methods. This correctly suggests that like model fields, we can also use pure methods to achieve data abstraction in specifications. Using pure methods has the appeal that, unlike model fields, methods are a native concept of the programming language, and the necessary methods may already be present in the program anyway. An example for this is again the `List` interface from Figures 2.2 and 2.3, which can also be specified using its own pure methods `size` and `get` as shown in Figure 2.4.

The calls to methods of class `JMLObjectSequence` have been replaced by calls to `size` and `get` in Figure 2.4. The fact that these pure methods are now themselves the basic building blocks of the specification is emphasised by their self-referential postconditions in lines 6 and 13, which are trivially satisfied by any implementation. The modifies clause of `add` is as problematic here as it is in the approach based on ghost fields, because like ghost fields, pure methods do not provide a means to abstract over sets of locations. This is solved in Figure 2.4 with the help of a model field `footprint` that is used only in its role as a data group.

For modular reasoning about specifications that use pure methods, it is usually necessary to limit the *dependencies* of these pure methods, that is, the memory locations that may influence the result of a method invocation. An example is the code in Figure 2.5. The precondition of `m` tells us that before the call to `m`, the list is not empty. We expect that the list is still not empty in line 9, and that thus the precondition of the first contract of method `get` is satisfied. However, without looking into all implementations of `size` (thereby sacrificing modularity of reasoning) and concluding that they do not depend on `x`, we cannot be sure that the intervening change to `x` does not affect the result of `size`. This demonstrates a general problem when using pure methods in specifications [Leavens et al., 2007, Challenge 3]: without further measures, any change to the heap can affect the value returned by a pure method in an unknown way.

Java + JML

```java
public interface List {
    //@ public model instance JMLDataGroup footprint;

    /*@ public normal_behaviour
      @   accessible footprint;
      @   ensures \result == size();
      @*/
    public /*@pure@*/ int size();

    /*@ public normal_behaviour
      @   requires 0 <= index && index < size();
      @   accessible footprint;
      @   ensures \result == get(index);
      @ also ...
      @*/
    public /*@pure@*/ Object get(int index);

    /*@ public normal_behaviour
      @   assignable footprint;
      @   ensures size() == \old(size()) + 1;
      @   ensures get(size() - 1) == o;
      @   ensures ...
      @*/
     public void add(Object o) {...}
}

public class ArrayList implements List {
    private /*@nullable@*/ Object[] array = new Object[10];
        //@ in footprint;
        //@ maps array[*] into footprint;

    private int size = 0; //@ in footprint;

    //@ /*four invariants as in Figure 2.1*/

    public int size() {...}
    public Object get(int index) {...}
    public void add(Object o) {...}
}
```

Java + JML

Figure 2.4.: Java interface `List` specified using pure methods and data groups, and class `ArrayList` implementing the interface

Java + JML

```
 1 public class Client {
 2     public int x;
 3
 4     /*@ normal_behaviour
 5       @   requires 0 < list.size();
 6       @*/
 7     void m(List list) {
 8       x++;
 9       Object o = list.get(0);
10       ...
11     }
12 }
```

Java + JML

Figure 2.5.: Client code that uses the `List` interface of Figure 2.4

JML provides a (partial) solution to this problem, namely *depends clauses*, also known as *accessible clauses*. Depends clauses are a dual concept to modifies clauses. Where a modifies clause is used to specify which locations a method may modify (which locations a method may *write* to), a depends clause is used to specify which locations a method's result may depend on (which locations a method may *read* from). In JML, depends clauses are declared within method contracts using the keyword **`accessible`**. Lines 5 and 12 of Figure 2.4 give depends clauses for `size` and `get`. These use the already introduced data group `footprint`, because the locations to be read by `size` and `get` are the same as those that are to be modified by `add`. The method bodies in class `ArrayList` (which are still those shown in Figure 2.1) satisfy these depends clauses.

In the example shown in Figure 2.5, the depends clause of `size` reduces the problem of determining that the change to **`this`**`.x` does not affect the result of `list.size()` to the problem of determining that **`this`**`.x` is not an element of the data group `list.footprint`. This would be easy if only *static* data group inclusions were permitted: then, we could conclude from the lack of an **`in`** clause next to the declaration of `x` in line 2 of Figure 2.5 that `x` is not part of any data group. In the presence of dynamic data group inclusions, it is more difficult, because there could always be a **`maps`** `...` **`\into`** clause in some subclass of `List` that effectively puts **`this`**`.x` into `list.footprint` for some program states. We return to this problem in Chapter 3.

Note that even though we have not considered these dependency issues in Subsections 2.5.1 and 2.5.2, they are nevertheless present in all variations of the specification of `List`. In the previous approaches, the dependencies of the pure methods declared in `JMLObjectSequence` are relevant, as well as the contents of the involved data group.

As an aside, the usefulness of depends clauses goes well beyond reasoning about pure methods in the context of data abstraction. For example, a second (related) application of depends clauses is in the specification of *object immutability* [Haack et al., 2007]: an immutable object is an object that does not change, so all its methods must be pure. Additionally, one typically expects the return values of these pure methods to remain the same from state to state. This can be specified with depends clauses which express that the methods do not depend on any mutable state outside of the immutable object. Also, the problem of *secure information flow* [Sabelfeld and Myers, 2003] is in its basic form just a minor generalisation of the verification of depends clauses, where one is interested not only in the dependencies of the method return value, but in the dependencies of any number of locations.

2.6. Conclusion

In this chapter, we have reviewed the basic concepts of design by contract specification for object-oriented programs, on the example of the Java Modeling Language (JML). The main components of such specifications are method contracts and object invariants. Both contracts and invariants are inherited to subclasses, enforcing behavioural subtyping and facilitating modular reasoning about program correctness. To allow for information hiding in specifications, and for refinement in the sense of adding new subclasses to existing classes and interfaces without having to change the supertype's specification, data abstraction mechanisms are necessary. In this area, JML offers ghost fields, model fields, data groups, and the use of pure methods together with depends clauses.

By far not all features of JML have been covered in this chapter. Omitted features include simple in-code *assertions*, *loop invariants*, **`static`** object invariants, *history constraints*, *model programs*, and others. Also note that JML is constantly evolving. This chapter is based on the state described in the newest version of the reference manual available at the time of writing [Leavens et al., 2008].

We have touched on a few problems with the current state of JML that we investigate more deeply in Chapter 3, concerning the semantics of object invariants and the mechanism of data groups.