Getting started with JML

Improve your Java programs with JML annotation

Joe Verzulli

March 18, 2003

The Java Modeling Language (JML) is a notation for detailed design that encourages a new way of thinking about methods and classes. In this primer, Java programming consultant Joe Verzulli introduces JML and some of its most important declarative constructs.

One of the principles of object-oriented analysis and design is that procedural thinking should be postponed as long as possible. Most of us adhere to this rule only up to the point of method implementation. Once the classes and their interfaces have been identified and it's time to start implementing methods, we switch over to procedural thinking. After all, what alternative is there? As with most languages, when writing Java code we need to provide a step-by-step procedure for computing the results of each method.

On its own, procedural notation just says how to do something without ever saying what it is we're trying to do. It would be useful to know what we're trying to achieve before we start doing it, but the Java language doesn't provide a way to explicitly incorporate that information into our code.

The Java Modeling Language (JML) adds annotations to Java code that let us specify what methods do without saying how they do it. With JML, we can describe a method's intended functionality without regard to implementation. In this way, JML extends the object-oriented principle of postponing procedural thinking into the stage of method design.

JML introduces a number of constructs for declaratively describing behavior. These include model fields, quantifiers, visibility scoping for assertions, preconditions, postconditions, invariants, contract inheritance, and specifications of normal versus exceptional behavior. These constructs make JML very powerful, but it isn't necessary to understand or use all of them, or to use all of them at once. You can learn and use JML incrementally, starting out very simply.

In this article, we'll take a gradual approach to learning JML. We'll start with an overview of the benefits of using JML, talking specifically about its impact on the development and compilation process. Next, we'll discuss some JML constructs for preconditions, postconditions, model fields, quantification, side effects, and exceptional behavior. Along the way, examples will provide motivation for each of the JML constructs. At the end of this article, you will have a conceptual understanding of how JML works, and be able to apply it in your own programs.
JML overview

Using JML to declaratively describe the desired behavior of classes and methods notably improves the overall development process. The benefits of adding modeling notation to your Java code include the following:

- More precise description of what the code does
- Efficient discovery and correction of bugs
- Reduced chance of introducing bugs as the application evolves
- Early discovery of incorrect client usage of classes
- Precise documentation that is always synced with application code

JML annotations are always within Java comments, so they have no impact on code that is compiled normally. The open-source JML compiler (see Related topics) can be used when we want to compare the actual behavior of a class to its JML specifications. Code compiled with the JML compiler will raise a JML exception at runtime if the code does not do what the specification says it should. Not only can this catch bugs in the code but it also ensures that the documentation (in the form of JML annotations) stays in sync with the code.

In the following sections I’ll use the PriorityQueue interface and the BinaryHeap class from the open source Jakarta Commons Collection Component (JCCC) to illustrate the features of JML. You will find the complete, annotated PriorityQueue and BinaryHeap source code in the Related topics section.

Requirements and responsibilities

The source code for this article (see Related topics) includes the PriorityQueue interface from the JCCC. The PriorityQueue interface contains method signatures that specify the data types of arguments and return values but don’t say anything about the behavior of the methods. We’d like to be able to specify the semantics of a PriorityQueue so that all classes implementing it behave in the desired way. (Without a behavioral specification we could end up with a stack class that implements the PriorityQueue interface, or some other odd situation like that.)

Consider the pop() method in PriorityQueue. What are the requirements on pop() for a priority queue? There are three basic ones. First, pop() should not be called unless there is at least one element in the queue. Second, it should return the element in the queue that has highest priority. Finally, it should remove the element it returns from the queue.

Listing 1 shows the JML annotation expressing the first requirement:

Listing 1. JML annotation for a pop() method requirement

```java
/*@ 
public normal_behavior 
@  requires ! isEmpty(); 
@*/
Object pop() throws NoSuchElementException;
```
As previously mentioned, JML annotations are written inside Java comments. Multi-line comments containing JML start with the characters `/*@`. JML ignores any `@` signs that are the first non-white space character in a line. JML can also be written inside single-line comments starting with `//@`.

The `public` keyword here means the same thing in JML as it does in the Java language. `public` indicates that this JML specification is visible to all other classes in the application. Public specifications can only refer to public methods and member variables. JML also allows specifications to have `private-`, `protected-`, or `package-` level scope. The scoping rules for JML are analogous to the corresponding rules in the Java language.

The `normal_behavior` keyword indicates that this specification describes the case in which `pop()` returns normally without throwing an exception. Later we'll talk about how exceptional behavior is specified.

**Preconditions and postconditions**

The JML keyword `requires` is used for preconditions. A precondition is a condition that must be satisfied before calling a method. Listing 1 says that the precondition of `pop()` is that `isEmpty()` returns `false`; that is, that the queue contains at least one item.

The postcondition of a method specifies the responsibilities of the method; that is, when the method returns, the postcondition should be true. In our example, `pop()` should return the element in the queue that has highest priority. We would like to specify this postcondition so it can be checked by JML at runtime. To do this we need to keep track of all values that have been added to the priority queue in order to determine which one should be returned by `pop()`. We might consider adding a member variable to `PriorityQueue` to store the values in the queue, but there are two problems with this:

- `PriorityQueue` is an interface, so it should be compatible with different implementations such as a binary heap, Fibonacci heap, or calendar queue. The JML annotations for `PriorityQueue` shouldn't say anything about implementation.
- As an interface, `PriorityQueue` can only contain static member variables.

JML provides a way out of this dilemma with its concept of `model fields`.

**Model fields**

Model fields are like member variables that can only be used in behavior specifications. Here's an example of a model field declaration used in `PriorityQueue`:

```java
//@ public model instance JMLObjectBag elementsInQueue;
```

This declaration says that there is a model field called `elementsInQueue` whose data type is `JMLObjectBag` (a bag data type that’s part of JML). The `instance` keyword indicates that, even though the field is declared in an interface, a separate non-static copy of the field will be placed in each instance of classes implementing `PriorityQueue`. Like all JML annotations, the declaration of `elementsInQueue` appears in a Java comment, so `elementsInQueue` can't be used in regular Java code. No object will contain an `elementsInQueue` field at runtime.
Specification vs. implementation

It seems inefficient to use a to store the elements because then every element must be examined to find the one with highest priority. However, the bag is part of the specification, not part of the implementation. The purpose of the specification is to describe the behavioral interface of PriorityQueue; that is, the external behavior that its clients can rely on. The implementation of PriorityQueue can use a more efficient approach as long as it gives the same results as the specification. For example, the JCCC includes a BinaryHeap class that implements PriorityQueue by using a binary heap stored in an array.

Although specifications don't need to be written with efficiency in mind, the efficiency of the JML runtime assertion checker is important, since running with assertion checking enabled does have a performance impact.

elementsInQueue contains the values added to the priority queue. Listing 2 shows how the specification for pop() makes use of elementsInQueue:

**Listing 2. A model field used in the postcondition of pop()**

```java
/*@ 
@ public normal_behavior 
@   requires ! isEmpty(); 
@   ensures 
@     elementsInQueue.equals(((JMLObjectBag) 
@          \old(elementsInQueue)) 
@     .remove(\result)) && 
@     \result.equals(\old(peek())); 
@*/
Object pop() throws NoSuchElementException;
```

The ensures keyword indicates that what follows is a postcondition that must be satisfied when pop() returns. \result is a JML keyword that is equal to the value returned by pop(). \old() is a JML function that returns the value its argument had before pop() was called.

The ensures clause contains two postconditions. The first says that the value returned by pop() is removed from elementsInQueue. The second says that the returned value is the same as the value returned by peek().

Class-level invariants

We've seen that JML allows us to specify preconditions and postconditions for methods. It also allows us to specify class-level invariants. Class-level invariants are conditions that must be true on entry and exit of every method in a class. For example, //@ public instance invariant elementsInQueue != null; is an invariant for PriorityQueue that says that elementsInQueue cannot be null at any time after a constructor of a class implementing PriorityQueue has returned.

Quantification

In the earlier specification for pop(), we said that its return value was equal to the value that peek() returns, but we didn't look at the specification for peek(). The specification for peek() in PriorityQueue is shown in Listing 3:
Listing 3. Specification for peek() in PriorityQueue

```java
/*@ public normal_behavior
  @   requires ! isEmpty();
  @   ensures elementsInQueue.has(result);
  @*/
/*@ pure @*/ Object peek() throws NoSuchElementException;
```

This JML annotation says that `peek()` should only be called when there is at least one element in the queue. It also says that the value returned by `peek()` must be in `elementsInQueue`; that is, it must be a value that was previously inserted into the queue.

The `/*@ pure @*/` comment indicates that `peek()` is a pure method. A *pure method* is one that doesn't have side effects. JML only allows assertions to use pure methods. We declare `peek()` to be pure so it can be used in the postcondition of `pop()`. If JML allowed non-pure methods in assertions then we could inadvertently write specifications that had side effects. This could result in code that works when compiled with assertion checking enabled but doesn't work when assertion checking is disabled. We'll talk more about side effects later on.

**About inheritance**

JML specifications are inherited by subclasses and classes implementing interfaces (unlike the assert statement in J2SE 1.4). The JML keyword indicates that a specification is combined with specifications inherited from ancestor classes and from interfaces being implemented. Therefore, the specification for `peek()` in the `PriorityQueue` interface applies to the `peek()` in `BinaryHeap` as well. This means that the value returned by `BinaryHeap.peek()` must be in `elementsInQueue`, even though this is not explicitly stated in the specification of `BinaryHeap.peek()`.

**Min and max heaps**

One thing is conspicuously missing from the specification for `peek()`. It never says that the value returned is the one with highest priority. It turns out that the `PriorityQueue` interface in the JCCC can be used for both min heaps and max heaps. For a min heap the highest-priority element is the one that is smallest, whereas for a max heap the highest-priority element is the one that is largest. Because `PriorityQueue` does not know whether it is being used with a min heap or a max heap, the part of the specification indicating which element is returned must go in the class that implements `PriorityQueue`.

In the JCCC, the `BinaryHeap` class implements `PriorityQueue`. `BinaryHeap` allows clients to specify whether it should behave as a min heap or a max heap via a parameter in its constructor. We use the boolean model variable `isMinimumHeap` to indicate whether the `BinaryHeap` is behaving as a min or max heap. The specification of `peek()` in `BinaryHeap` uses `isMinimumHeap`, as shown in Listing 4:
### Adding quantifiers

The postcondition in Listing 4 consists of two parts, one for min heaps and one for max heaps. The $\Rightarrow$ symbol means "implies." $x \Rightarrow y$ is true if (and only if) either $y$ is true or $x$ is false. For a min heap the following condition applies:

$$\forall Object\ obj;\ elementsInQueue.has(obj);\ compareObjects(result,\ obj)\ \leq\ 0$$

\forall is a JML quantifier. The above $\forall$ expression is true if for all $Object\ obj$, such that $elementsInQueue.has(obj)$ is true, $compareObjects(result,\ obj)$ returns a value less than or equal to zero. In other words, when $compareObjects()$ is used to compare values, the value returned by $peek()$ is less than or equal to every element in $elementsInQueue$. Additional JML quantifiers include $\exists$, $\sum$, and $\min$.

### Adding comparators

The `BinaryHeap` class allows elements to be compared in two different ways. One approach is to rely on the natural ordering of the elements using the `Comparable` interface. The other is for clients to pass a `Comparator` object to the `BinaryHeap` constructor. That `Comparator` will then be used for ordering. We use the model field `comparator` to indicate the `Comparator` object, if any. The `compareObjects()` method in the `peek()` postcondition uses whichever comparison approach the client has selected. `compareObjects()` is defined as shown in Listing 5:
**Listing 5. The compareObjects() method**

```java
/*@ 
@ public normal_behavior 
@ ensures 
@   (comparator == null) ==> 
@     (result == ((Comparable) a).compareTo(b)) && 
@   (comparator != null) ==> 
@     (result == comparator.compare(a, b)); 
@
@ public pure model int compareObjects(Object a, Object b) 
@ { 
@   if (m_comparator == null) 
@     return ((Comparable) a).compareTo(b); 
@   else 
@     return m_comparator.compare(a, b); 
@ }
@*/
```

The keyword `model` in the declaration of `compareObjects` indicates that it is a model method. **Model methods** are JML methods that can only be used in specifications. They are declared in Java comments and cannot be used in regular Java implementation code.

If a client of `BinaryHeap` has requested that a specific `Comparator` be used, then `m_comparator` will refer to that `Comparator`; otherwise, it will be null. `compareObjects()` checks the value of `m_comparator` and uses the appropriate comparison operation.

### How model fields get values

We showed the postcondition for `peek()` in Listing 4. The postcondition verifies that the return value has a priority greater than or equal to the priority of all elements in the model field `elementsInQueue`. This raises a question: how do model fields such as `elementsInQueue` get their values? Preconditions, postconditions, and invariants must be side-effect free, so they cannot set the value of model fields.

JML uses a `represents` clause to associate model fields with concrete implementation fields. For example, the following `represents` clause is used for the model field `isMinimumHeap`:

```java
//@ private represents isMinimumHeap <- m_isMinHeap;
```

This clause says that the value of the model field `isMinimumHeap` is equal to the value of `m_isMinHeap`. `m_isMinHeap` is a private boolean member field in the `BinaryHeap` class. Whenever the value of `isMinimumHeap` is required, JML will substitute the value of `m_isMinHeap`.

The `represents` clause is not limited to member fields on the right side of the `<-`. Consider the following `represents` clause for `elementsInQueue`:

**Listing 6. A represents clause for elementsInQueue**

```java
/*@ private represents elementsInQueue 
@   <- JMLObjectBag.convertFrom(
@     Arrays.asList(m_elements) 
@     .subList(1, m_size + 1)); 
@*/
```
This clause says that elementsInQueue is equal to the values in m_elements[], from indices 1 through m_size inclusive. m_elements[] is a private member variable in BinaryHeap that is used to store the elements in the priority queue. m_size is the number of elements that are currently used in m_elements[]. The BinaryHeap does not use m_elements[0]; this simplifies the index calculations for the binary heap. JMLObjectBag.convertFrom() converts a List to a JMLObjectBag, which is the data type needed for elementsInQueue.

Whenever the JML runtime assertion checker needs the value of elementsInQueue it will look to the code fragment on the right side of the <-> in the represents clause.

Now, we are ready to talk about side effects and exceptional behavior.

**Side effects**

Recall the postcondition of pop() from Listing 2:

```java
ensures elementsInQueue.equals(((JMLObjectBag)
\old(elementsInQueue))
 .remove(\result) &&
\result.equals(\old(peek()));
```

This says that pop() has the side effect of removing an element from elementsInQueue. However, it does not say that there are no other side effects. For example, an implementation of pop() could modify m_isMinHeap to change a min heap to a max heap. Such a modification would cause no runtime assertion failures as long as it also returned the correct value, but it would weaken our overall JML specification.

We can strengthen the postcondition to disallow any side effects other than modifying elementsInQueue, as shown in Listing 7:

**Listing 7. A postcondition for side effects**

```java
ensures elementsInQueue.equals(((JMLObjectBag)
\old(elementsInQueue))
 .remove(\result) &&
\result.equals(\old(peek())) &&
isMinimumHeap == \old(isMinimumHeap) &&
comparator == \old(comparator);
```

Adding a postcondition of the form x == \old(x) closes the loophole. On the downside, this approach eventually leads to cluttered specifications because every method must have clauses in its postcondition for every field that isn't changed by the method. It also makes maintenance more difficult because if we add a new field to the class we'll have to revise the postconditions of all methods to say that they don't modify the new field.

JML provides a better way to indicate the side effects of a method, with the assignable clause.

**The assignable clause**

The assignable clause lets us write the specification for pop() as shown in Listing 8:
Listing 8. An assignable clause for a method specification

```java
/*@ public normal_behavior
@   requires ! isEmpty();
@   assignable elementsInQueue;
@   ensures
@     elementsInQueue.equals(((JMLObjectBag)
@       \old(elementsInQueue))
@                        .remove(\result)) &&
@     \result.equals(\old(peek()));
@*/
Object pop() throws NoSuchElementException;
```

Only the fields listed in an assignable clause can be modified by a method (additional conditions are discussed below). The assignable clause for pop() says that pop() can modify elementsInQueue but it cannot modify other fields such as isMinimumHeap or comparator. It would be an error if the implementation of pop() modified m_isMinHeap. (Note that the current version of the JML compiler does not check that methods only modify fields in their assignable clause, however.)

Modifying locations

To say that a method can only modify fields listed in an assignable clause is a bit of a simplification. The actual rules allow a method to modify a location (loc) if any of the following factors is true:

- loc is mentioned in an assignable clause.
- A location mentioned in an assignable clause depends on loc. (For example, consider the case of "assignable isMinimumHeap;." The model field isMinimumHeap depends on the concrete field m_isMinHeap, so this assignable clause would allow a method to modify m_isMinHeap as well as isMinimumHeap.)
- loc was not allocated when the method started execution.
- loc is a local variable of the method or is a formal parameter of the method.

The last case allows a method to modify its arguments even if the arguments do not appear in an assignable clause. At first glance, this might seem to allow a method to modify a variable in its caller if that variable is passed as an argument to the method. For example, suppose we have a method foo(Bar obj) that contains the statement obj = anotherBar. While the statement changes the value of obj, it doesn't affect any variables in the caller of foo() because the obj in foo() is distinct from the variables in the caller of foo() even though they refer to the same Bar instance.

What if foo(Bar obj) contains the statement obj.x = 17? This would cause a change visible to the calling function. There is a catch, however. The rules for the assignable clause allow a method to assign a new value to an argument without having to mention that variable in an assignable clause. But they do not allow it to assign a new value to a field of an argument such as obj.x. If foo() needs to modify obj.x then it must have an assignable clause of the form assignable obj.x;

Two JML keywords can be used with the assignable clause: \nothing and \everything. We can indicate that a method does not have any side effects by writing assignable \nothing.
Similarly, we can indicate that a method is allowed to modify anything by writing `assignable \everything`. The `pure` keyword described earlier in the context of the `peek()` method is equivalent to `assignable \nothing;`.

### Exceptional behavior

The specifications given earlier for `peek()` and `pop()` require that the queue is not empty when each method is called. But in reality `peek()` and `pop()` can be called when the queue is empty. In such a case, either method would throw `NoSuchElementException`. We need to revise our specification to allow for this possibility, and for this we’ll use the JML’s `exceptional_behavior` clause.

Until now our specifications have started with `public normal_behavior`. The `normal_behavior` keyword indicates that these specifications are for the case where the method does not throw any exceptions. A `public exceptional_behavior` annotation can be used to describe the behavior when an exception is thrown. Listing 9 shows `exceptional_behavior` in the `PriorityQueue`’s specification for `peek()`:

**Listing 9. An exceptional_behavior annotation**

```java
/*@ 
  @ public normal_behavior 
  @   requires ! isEmpty(); 
  @   ensures elementsInQueue.has(\result); 
  @ also 
  @ public exceptional_behavior 
  @   requires isEmpty(); 
  @   signals (Exception e) e instanceof NoSuchElementException; 
/*@ pure */ Object peek() throws NoSuchElementException;
```

Like every other example we’ve seen so far, the first part of this specification starts with `public normal_behavior`. The second part starting with `public exceptional_behavior` is where the exceptional behavior is described. Just like the `normal_behavior` clause, the `exceptional_behavior` clause has a `requires` clause. The `requires` clause indicates what condition must be true in order for the exception listed in the `signals` clause to be thrown. In the above example, if `isEmpty()` is true then `peek()` will throw a `NoSuchElementException`.

### The signals clause

The general form of a `signals` clause is `signals(E e) R;` where `E` is `Exception` or a class derived from `Exception` and `R` is an expression. JML interprets a signal clause as follows: if an exception of type `E` is thrown then JML checks if `R` is true. If so then the method meets its specification. If `R` is false then JML throws an unchecked exception indicating that the `exceptional_behavior` specification was violated.

The `signals` clause in the above declaration of `peek()` requires that a `NoSuchElementException` be thrown if the queue is empty. If `peek()` throws any other exception (for example, an `IllegalStateException` which is an unchecked exception) then JML will catch this as an error.
because `e instanceof NoSuchElementException` will be false. If we wanted to allow `peek()` to return either `NoSuchElementException` or an unchecked exception, we would change the `signals` clause to `signals (NoSuchElementException e) true;`. This means that if `peek()` throws `NoSuchElementException` then `true` must be true, which is always the case. Because we haven't said anything about other exceptions, `peek()` may throw any exception allowed by its signature. The signature specifies `throws NoSuchElementException`, which means that either `NoSuchElementException` or an unchecked exception can be thrown.

If `peek()` is called when there is something in the queue and it throws a `NoSuchElementException` (or any other exception) then the JML runtime assertion checker will raise an unchecked exception indicating that the normal behavior postcondition failed.

**Conclusion**

In this article I've introduced JML, explained its contribution to the object-oriented analysis and design process, and shown you, by example, how to use JML annotations in your Java programs. You can download the complete source code for this article from Related topics, where you'll also find references to assist you in learning more about JML.

You can use the open source JML compiler to generate class files that automatically check your JML specifications at runtime. If a program doesn't do what its JML annotations say it should, JML will throw an unchecked exception indicating which part of the specification was violated. This is useful both for catching bugs and for helping to keep the documentation (in the form of JML annotations) in sync with the code.

The JML runtime assertion checking compiler is the first of a growing number of tools that work with JML. Others include `jmldoc` and `jmlunit`. `jmldoc` is similar to `javadoc` but it also includes JML specifications in the generated HTML documentation. `jmlunit` generates the skeleton of a test-case class that allows JML to be conveniently used in conjunction with JUnit. You'll find a reference for these and other tools in Related topics.

*Thanks to Gary Leavens and Yoonsik Cheon for answering my questions about JML and for helpful comments on a draft of this article.*
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j-jml.zip</td>
<td>10KB</td>
</tr>
</tbody>
</table>
Related topics

- Sourceforge is the home of the JML specifications, open source JML tools such as the JML compiler, jmldoc, and jmlunit, and related information.
- The PriorityQueue interface and BinaryHeap class are part of the Jakarta Commons Collection Component.
- Granville Miller's Java modeling column is an introduction to modeling object-oriented systems (developerWorks, 2002).