Semantics of a Specification (formally)

A function satisfies the specification

\[
\text{requires } e_1 \\
\text{ensures } e_2
\]

iff for all executions

\[
(Norm, heap, lcl) \xrightarrow{body} (Ret, heap', lcl')
\]

with \((Norm, heap, lcl) \xrightarrow{e_1 \triangleright v_1} q_1, \; v_1 \neq 0\), the post-condition holds, i.e., there exists \(v_2, q_2\), such that

\[
(Norm, heap', lcl') \xrightarrow{e_2 \triangleright v_2} q_2, \; \text{where } v_2 \neq 0
\]

However we need a new rule for evaluating \(\backslash old\):

\[
(Norm, heap, lcl) \xrightarrow{e \triangleright v} q \; \quad \text{where } heap, lcl \text{ is the state of the program before } body \text{ was executed}
\]

\[
(Norm, heap', lcl') \xrightarrow{\backslash old(e) \triangleright v} q
\]
In JML side-effects in specifications are forbidden:
If $e$ is an expression in a specification and

$$(\text{Norm}, \text{heap}, \text{lcl}) \xrightarrow{\text{e} \triangleright} (\text{flow}, \text{heap}', \text{lcl}')$$

then $\text{heap} \subseteq \text{heap}'$ and $\text{lcl} = \text{lcl}'$.

Here, $\text{heap} \subseteq \text{heap}'$ indicates that the new heap may contain new (unreachable) objects.
Also $\text{flow} \neq \text{Norm}$ is possible. In that case the expression is considered to be false.
A tool should warn the user if $\text{flow} \neq \text{Norm}$ is possible.
There were some discussions on exceptions in JML specifications.

- `next == null || next.prev == this` is okay. It never throws a null-pointer exception.
- `next.prev == this || next == null` is not equivalent. It is not valid if `next` is null.

Specifications that can throw an exception should be avoided.
Lightweight vs. Heavyweight Specifications

A lightweight specification
/*@ requires \textit{P};
   @ assignable \textit{X};
   @ ensures \textit{Q};
   @*/
public void \textit{foo}() throws \textit{IOException};

is an abbreviation for the heavyweight specification
/*@ public behavior
   @ requires \textit{P};
   @ diverges false;
   @ assignable \textit{X};
   @ ensures \textit{Q};
   @ signals\_only \textit{IOException}
   @*/
public void \textit{foo}() throws \textit{IOException};

With the \texttt{behavior}-keyword there are no default values for \texttt{diverges}, \texttt{signals\_only}, and \texttt{assignable}. 
Making Exceptions Explicit

```java
/*@ public normal_behavior
@  requires x >= 0;
@  assignable \nothing;
@  ensures \result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
@ also
@ public exceptional_behavior
@  requires x < 0;
@  assignable \nothing;
@  signals (IllegalArgumentException) true;
@*/

public static int isqrt(int x) throws IllegalArgumentException {
    if (x < 0)
        throw new IllegalArgumentException();
    body
}
```
If several specifications are given with \textit{also}, the method must fulfill \textit{all} specifications.

Specifications with \texttt{normal\_behavior} implicitly have the clause
\begin{verbatim}
signals (java.lang.Exception) false
\end{verbatim}

so the method must not throw an exception.

Specifications with \texttt{exceptional\_behavior} implicitly have the clause
\begin{verbatim}
ensures false
\end{verbatim}

so the method must not terminate normally.
The Java Modelling Language (JML)

JML is a behavioral interface specification language (BISL) for Java

- Proposed by G. Leavens, A. Baker, C. Ruby: 
  *JML: A Notation for Detailed Design*, 1999
- It combines ideas from two approaches:
  - Eiffel with it’s built-in language for Design by Contract (DBC)
  - Larch/C++ a BISL for C++
The Roots of JML

- Ideas from Eiffel:
  - Executable pre- and post-condition (for runtime checking)
  - Uses Java syntax (with a few extensions).
  - Operator `\texttt{old}` to refer to the pre-state in the post-condition.

- Ideas from Larch:
  - Describe the state transformation behavior of a method
  - Model Abstract Data Types (ADT)
JML and Abstract Data Types
Subsystems request timer events and queue them.

First timer event is passed to the timer.

Priority queue maintains events in its internal data structure.
public interface PriorityQueue {

    public void enqueue(Comparable o);

    public Comparable removeFirst();

    public boolean isEmpty();

}
public interface PriorityQueue {

    /*@ public normal_behavior
        @  ensures !isEmpty();
        @*/
    public void enqueue(Comparable o);

    /*@ public normal_behavior
        @  requires !isEmpty();
        @*/
    public Comparable removeFirst();

    public /*@pure@*/ boolean isEmpty();
}