What is JML?

- A Design by Contract (DBC) tool for Java
- Specifies agreement between a class and client code
  - Obligations/Rights of the class and the client
Contracts in Software

/*@ requires x >= 0.0;
@ ensures JMLDouble.approximatelyEqualTo(x,
@ \result * \result, eps);
@*/

public static double sqrt(double x) { … }

<table>
<thead>
<tr>
<th>Obligations</th>
<th>Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td></td>
</tr>
<tr>
<td>Passes non-negative number</td>
<td>Gets square root approximation</td>
</tr>
<tr>
<td>Implementor</td>
<td></td>
</tr>
<tr>
<td>Computes and returns square root</td>
<td>Assumes argument is non-negative</td>
</tr>
</tbody>
</table>

Nathan Jokel
JML Syntax: comments

- Specifications written in \textit{annotation comments}
- Single-line:
  \begin{verbatim}
  //@ assert x >= 0;
  \end{verbatim}
- Multi-line:
  \begin{verbatim}
 /*@ ensures kgs >= 0
     && weight == kgs + 10;
  @*/
  \end{verbatim}
- Comments:
  \begin{verbatim}
  //@ requires x > 0; (* x is positive *)
  \end{verbatim}
JML Syntax: Assertions

- Assertions are Java expressions that evaluate to a boolean value, but:
  - Cannot have side effects
    - No use of =, ++, --, etc., and
    - Can only call pure methods.

```java
public /*@ pure @*/ int getWeight();
```
J ML – Types of Assertions

- Class Invariants
- Loop Invariants
- Method Pre and Postconditions
  - Normal and exceptional postconditions
Class Invariants

- *invariant* keyword used
- Checked at the start and end of each method call to the class

```java
public class Person{
    private String name;
    //@ public invariant !name.equals("");
    ...
}
```

Nathan Jokel
Loop Invariants

- *assert* keyword used
- Checked at each iteration at the designated point in a loop

```java
for(i=0; i<n; i++) {
    //@ assert !list.isEmpty();
    list.remove(i);
}
```
Method Pre and Postconditions

- **requires** keyword used for preconditions
  - Checked immediately before method invocation

- **ensures** keyword used for normal postconditions
  - Checked immediately following method invocation

```java
/*@ requires n != null && !n.equals("");
@ ensures name.equals(n)
@*/
public setName(String n);
```
Exceptional Postconditions

- `signals` keyword used
- Checked when method throws an exception
  - multiple exceptional postconditions possible

```java
/*@ signals (IllegalArgumentException e)
@     e.getMessage() != null
@     && !(x > 0.0);
/*@*/
public static double sqrt(double x) throws IllegalArgumentException
```

Nathan Jokel
JML: Additional Syntax

JML has some extensions to Java syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\result</td>
<td>result of method call</td>
</tr>
<tr>
<td>a ==&gt; b</td>
<td>a implies b</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>b implies a</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>a iff b</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>!(a &lt;= b)</td>
</tr>
<tr>
<td>\old(E)</td>
<td>value of E in pre-state</td>
</tr>
</tbody>
</table>

/*@ ensures kgs >= 0

    @    && \result == \old(weight + kgs);
    @*/

public int addWeight(int weight);
JML: Quantification

JML also provides for quantification

/*@ requires a != null
@    && (\forall int i;
@      0 < i && i < a.length;
@      a[i-1] <= a[i];
@*/

int binarySearch(int[] a, int x) {...}
J ML Tools

- **jmlc**
  - parses annotation comments and creates Java bytecode
  - calls javac

- **jmlrac**
  - executes code with assertions, throws exception if assertion violated
  - calls java

Nathan Jokel
JML: Exercising Assertions

- Java program with “main” method required by jmlrac
- Test cases needed to exercise assertions
  - A method that is never called in a program can’t cause an assertion violation!
For more information

- www.jmlspecs.org

Nathan Jokel