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 its 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 `\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)
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();
}

How to Model Internal Structure?

- Specification is incomplete.
- Which values are returned by removeFirst()?
- We need a model variable representing the queue.
- JML defines useful types to model complex data structures.
Example: Model for Internal Structure

//@ model import org.jmlspecs.models.JMLObjectBag;
public interface PriorityQueue {
    //@ public instance model JMLObjectBag queue;

    //@ public normal_behavior
    @ ensures queue.equals(old(queue).insert(o));
    @ modifies queue;
    @*/
    public void enqueue(Comparable o);

    //@ public normal_behavior
    @ requires !isEmpty();
    @ ensures old(queue).has(result)
    @ and queue.equals(old(queue).remove(result))
    @ and forall java.lang.Comparable o;
    @ queue.has(o); result.compareTo(o) <= 0);
    @ modifies queue;
    @*/
    public Comparable removeFirst();

    //@ public normal_behavior
    @ ensures result == (queue.isEmpty());
    @*/
    public /*@pure@*/ boolean isEmpty();
}
What is JMLObjectBag?

- `org.jmlspecs.models.JMLObjectBag` is a **pure** class. It has pure function and no references to non-pure classes.
- Therefore, it can be used in specifications.
- There are lot of other classes:
  
How Does It Work?

For objects, e.g., \( \texttt{old}(\texttt{this}) == \texttt{this} \), since \( \texttt{old}(\texttt{this}) \) is the old pointer not the old content of the object.

Why does it work as expected with \( \texttt{old}(\texttt{queue}) \)?

- JMLObjectBag is immutable
- The \texttt{insert} method is declared as
  
  \begin{verbatim}
  public /*@pure@*/ JMLObjectBag insert(/*@nullable@*/ Object elem)
  \end{verbatim}

  Compare this to the \texttt{add} method of List:
  
  \begin{verbatim}
  public boolean add(/*@nullable@*/ Object elem)
  \end{verbatim}

  - \texttt{insert} returns a reference to a new larger list.
  - The content of \( \texttt{old}(\texttt{queue}) \) never changes, but \texttt{queue} changes.
Representing by a Pure Function

```java
import org.jmlspecs.models.JMLObjectBag;
public class Heap implements PriorityQueue {
    private Comparable[] elems;  //@ in queue;
    private int numElems;        //@ in queue;

    //@ private represents queue <- computeQueue();

    //@
    private model pure non_null JMLObjectBag computeQueue() {
        JMLObjectBag bag = new JMLObjectBag();
        for (int i = 0; i < numElems; i++) {
            bag = bag.insert(elems[i]);
        }
        return bag;
    }
    @*/

    ...
}
```
Representing by a Ghost Variable

```java
import org.jmlspecs.models.JMLObjectBag;
public class Heap implements PriorityQueue {
    private Comparable[] elems;  //@ in queue;
    private int numElems;        //@ in queue;

    //@ private ghost JMLObjectBag ghostQueue; in queue;
    //@ private represents queue <- ghostQueue;

    public void enqueue(Comparable o) {
        //@ set ghostQueue = ghostQueue.insert(o);
        ...
    }

    public Comparable removeFirst() {
        ...
        //@set ghostQueue = ghostQueue.remove(first);
        return first;
    }
}
```
The assignable Problem

//@ model import org.jmlspecs.models.JMLObjectBag;

public interface PriorityQueue {
//@ public instance model JMLObjectBag queue;

/*@ normal_behavior
@ ensures queue.equals(old(queue).insert(o));
@*/
  public void enqueue(/*@non_null@*/ Comparable o);
...
Adding assignable.

What does the function enqueue change?
It changes the model variable `queue` and nothing else.

```
//@ model import org.jmlspecs.models.JMLObjectBag;

public interface PriorityQueue {
    //@ public instance model JMLObjectBag queue;

    //@ normal_behavior
    @ ensures queue.equals(\old(queue).insert(o));
    @ assignable queue;
    @*/
    public void enqueue(/*@ non_null */ Comparable o);

    ...
```

However, when compiling Heap.java:

File "Heap.java", line 50, character 29 error: Field "numElems"
is not assignable by method "Heap.enqueue( java.lang.Comparable )";
only fields and fields of data groups in set "{queue}" are assignable [JML]
Mapping Variables To Model Variables.

We have to tell JML that \textit{elem} and \textit{numElems} are the implementation of the model variable \textit{queue}.

There is a special JML syntax:
\begin{verbatim}
import org.jmlspecs.models.JMLObjectBag;

public class Heap implements PriorityQueue {
    private Comparable[] elems; //@ in queue;
    private int numElems;     //@ in queue;

    /*@ private represents queue <- computeQueue(); */
    ...
\end{verbatim}
Datagroups

- Every model variable forms a data group.
- Other variables in the class or in sub-classes can be associated with this data group.
- Functions with specification assignable queue, where queue is a datagroup, may modify any variable in this group.
More About Datagroups

- There is a special data group `objectState`, which should represent the object state.
- All variables should be added to this group (but they are rarely).
- Adding a datagroup to another datagroup works recursively:
  ```java
  //@ model import org.jmlspecs.models.JMLOBJECTBag;

  public interface PriorityQueue {
    //@ public instance model JMLOBJECTBag queue; //@ in objectState;
  }
  ```

After this change `numElems` and `elems` are also automatically contained in `objectState`.
Datagroups are useful to group variables.

```java
class Calendar {
    //@ model JMLDataGroup datetime; in objectState;
    //@ model JMLDataGroup time, date; in datetime;
    int day, month, year; //@ in date;
    int hour, min, sec; //@ in time;
    int timezone; //@ in objectState;
    Locale locale; //@ in objectState;

    ...  
    //@ assignable datetime;
    void setDate(Date date);

    //@ assignable timezone;
    void setTimeZone();
}
```

This avoids listing the variables again.
Datagroups and model variables are useful for visibility issues:

```java
class Tree {
    //@ public model JMLDataGroup content; in objectState

    private Node rootNode; //@ in content

    //@ assignable content;
    public void insert(Object o);
}
```

Using `assignable rootNode` would produce an error.