Formal Methods for Java
Lecture 7: JML and Abstract Data Types

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

May 17, 2017
The Java Modelling Language (JML)

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

- 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();
}

The specification allows undesired things.

- After `removeFirst()` new value of `isEmpty()` is undefined.
Why is Specification Incomplete?

The specification allows undesired things.

- After `removeFirst()` new value of `isEmpty()` is undefined.
- In a correct implementation, after two `enqueue()` and one `removeFirst()` list is not empty.

Specification does not say so.
The specification allows undesired things.

- After `removeFirst()` new value of `isEmpty()` is undefined.
- In a correct implementation, after two `enqueue()` and one `removeFirst()` list is not empty. Specification does not say so.
- Problem: the internal state is not visible in spec.
Why is Specification Incomplete?

The specification allows undesired things.

- After `removeFirst()` new value of `isEmpty()` is undefined.
- In a correct implementation, after two `enqueue()` and one `removeFirst()` list is not empty. Specification does not say so.
- Problem: the internal state is not visible in spec.
- There is no internal state in an interface!
Adding Model Variables

Solution: add a model variable that records the size.

```java
public interface PriorityQueue {
    //@ public instance model int size;

    //@ public invariant size >= 0;

    /*@ public normal_behavior
        @ ensures size == \old(size) + 1;
        @*/
    public void enqueue(Comparable o);

    /*@ public normal_behavior
        @ requires !isEmpty();
        @ ensures size == \old(size) - 1;
        @*/
    public Comparable removeFirst();

    /*@ public normal_behavior
        @ ensures \result == (size == 0);
        @*/
    public /*@pure@*/ boolean isEmpty();
}
```
Model Variables

//@ public instance model int size;

- The keyword `instance` is the opposite of `static`.
- The keyword `model` denotes a variable that only exists in the specification.
Model Variables

```java
//@ public instance model int size;
```

- The keyword `instance` is the opposite of `static`.
- The keyword `model` denotes a variable that only exists in the specification.
- Public model variables can be accessed by other classes.
//@ public instance model int size;

- The keyword `instance` is the opposite of `static`.
- The keyword `model` denotes a variable that only exists in the specification.
- Public model variables can be accessed by other classes.
- Only specification can access model variables (read-only).
Model Variables

```java
//@ public instance model int size;
```

- The keyword `instance` is the opposite of `static`.
- The keyword `model` denotes a variable that only exists in the specification.
- Public model variables can be accessed by other classes.
- Only specification can access model variables (read-only).
- If a model variable is accessed in code, the compiler complains.
Visibility in JML

```java
//@ public instance model int size;
...
/*@ public normal_behavior
   @ ensures result == (size == 0);
   @*/
public /*@pure@*/ boolean isEmpty();
```

Why is size public?
Visibility in JML

//@ public instance model int size;
...
/*@ public normal_behavior
   @ ensures \result == (size == 0);
   @*/
public/*@pure*/ boolean isEmpty();

Why is size public?

- The external interface must be public.
Visibility in JML

```java
//@ public instance model int size;
...
/*@ public normal_behavior @
   @ ensures \result == (size == 0);
   @*/
public /*@pure@*/ boolean isEmpty();
```

Why is size public?

- The external interface must be public.
- The specification is part of the interface.
Visibility in JML

```java
//@ public instance model int size;
...
/*@ public normal_behavior
   @ ensures \result == (size == 0);
   @*/
public /*@pure@*/ boolean isEmpty();
```

Why is size public?

- The external interface must be public.
- The specification is part of the interface.
- To understand the specification one needs to know about size.
- Therefore, size is public.
public class Heap implements PriorityQueue {
    private Comparable[] elems;
    private int numElems;

    public void enqueue(Comparable o) {
        elems[numElems++] = o;
        ...
    }

    public Comparable removeFirst() {
        ...
        return elems[--numElems];
    }

    public isEmpty() {
        return numElems == 0;
    }
}
public class Heap implements PriorityQueue {
    private Comparable[] elems;
    private int numElems;

    //@ private represents size <- numElems;

    public void enqueue(Comparable o) {
        elems[numElems++] = o;
        ...
    }

    public Comparable removeFirst() {
        ...
        return elems[--numElems];
    }

    public isEmpty() {
        return numElems == 0;
    }
}
Every model variable in a concrete class must be represented:
//@ private represents size <- numElems;
Representing Model variables

Every model variable in a **concrete** class must be represented:

```java
//@ private represents size <- numElems;
```

The expression can also call pure functions:

```java
//@ private represents size <- computeSize();
```
How to Model Internal Structure?

- Specification is still incomplete.
How to Model Internal Structure?

- Specification is still incomplete.
- Which values are returned by removeFirst()?
How to Model Internal Structure?

- Specification is still incomplete.
- Which values are returned by `removeFirst()`?
- We need a model variable representing the queue.
How to Model Internal Structure?

- Specification is still incomplete.
- Which values are returned by \texttt{removeFirst()}?
- We need a model variable representing the \texttt{queue}.
- JML defines useful types to model complex data structures.
The JML Collection Classes

JML defines its own collection classes for several reasons:

- They were introduced before Java had its own collection classes.
- They are functional and have no side-effects.
- They are pure and can be used in specifications.
- They distinguish more cleanly between objects and values.
The base interface is *org.jmlspecs.models.JMLCollection*.  
- Similar to *java.util.Collection*.  
- Method *size()* returns the size of the collection.  
- Method *iterator()* returns an iterator.  
- Containment check is implemented by *has()*.
The Collection Classes


The collection classes are

- **JMLxxxBag**: corresponds to java.util.Collection.
- **JMLxxxSet**: corresponds to java.util.Set.
- **JMLxxxSequence**: corresponds to java.util.List.
- **JMLxxxToxxxRelation**: corresponds to java.util.Map.

The **xxx** is one of

- **Object** to denote that elements are compared with ==.
- **Equals** to denote that elements are compared with equals().
- **Value** to denote that elements are compared with equals() and are cloned before they are stored.
Running Example: Model for Internal Structure

```java
//@ 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)
     * @ old(queue).equals(old(queue).remove(result))
     * @ \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();
}
```
How Does It Work?

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

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

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

Why does it work as expected with `\old(queue)`?

- JMLObjectBag is immutable
How Does It Work?

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

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

- JMLObjectBag is immutable
- The insert method is declared as
  ```java
  public /*@pure*/ JMLObjectBag insert(/*@nullable*/ Object elem)
  ```

  Compare this to the add method of List:
  ```java
  public void add(/*@nullable*/ Object elem)
  ```
How Does It Work?

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

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

- JMLObjectBag is immutable
- The insert method is declared as
  
  ```java
  public /*@pure@*/ JMLObjectBag insert(/*@nullable@*/ Object elem)
  ```

  Compare this to the add method of List:
  
  ```java
  public void add(/*@nullable@*/ Object elem)
  ```

- insert returns a reference to a new larger list.
How Does It Work?

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

Why does it work as expected with `\old(queue)`?

- JMLObjectBag is immutable
- The `insert` method is declared as
  ```java
  public /*@pure@*/ JMLObjectBag insert(/*@nullable@*/ Object elem)
  ```
  Compare this to the `add` method of List:
  ```java
  public void add(/*@nullable@*/ Object elem)
  ```

- `insert` returns a reference to a new larger list.
- The content of `\old(queue)` never changes, but `queue` changes.
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 /*@pure@*/ JMLObjectBag computeQueue() {
        JMLObjectBag bag = new JMLObjectBag();
        for (int i = 0; i < numElems; i++) {
            bag = bag.insert(elems[i]);
        }
        return bag;
    }

    ...
}

...
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);

    ...

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);
   ...

When compiling it, it produced a warning:

>jmlc -Q PriorityQueue.java
File "PriorityQueue.java", line 7, character 24 caution:
A heavyweight specification case for a non-pure method has no assignable clause [JML]

Lets add a assignable clause.
Adding assignable.

What does the function `enqueue` change?
Adding assignable.

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

```java
//@ 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);
    ...
```
Adding assignable.

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

```java
//@ model import org.jmlspecs.models.JMLObj ectBag;

public interface PriorityQueue {
   //@ public instance model JMLObj ectBag 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(); @*/
...
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).
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.