Rigorous Software Development
CSCI-GA 3033-009

Instructor: Thomas Wies

Spring 2013

Lecture 6

Java Modeling 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*
- Combines ideas from two approaches:
  - Eiffel with its built-in language for Design by Contract
  - Larch/C++ a BISL for C++

JML Syntax: Method Specifications

In JML the specification precedes the method in `/*@ ... @*/`.

- **`requires`** `formula`:
  - The specification only applies if `formula` holds when method is called.
  - Otherwise behavior of method is undefined.
- **`ensures`** `formula`:
  - If the method exits normally, `formula` has to hold.

JML Syntax: Formulas

A JML formula is a Java Boolean expression. The following list shows some operators of JML that do not exist in Java:

- `\old(expression)`:
  - the value of expression before the method was called (used in ensures clauses)
- `\result`:
  - the return value (used in ensures clauses).
- `F ==> G`:
  - states that `F` implies `G`. This is an abbreviation for `!F || G`.
- `\forall Type t; condition; formula:`
  - states that `formula` holds for all `t` of type `Type` that satisfy `condition`.

JML Example: Factorial

A simple JML method contract

```
/*@ requires n >= 0;
  @ ensures \result >= 1;
  @*/
public static int factorial(int n) {
  int result = n;
  while (--n > 0)
    result *= n;
  return result;
}
```

JML Syntax: Method Specifications

In JML the specification precedes the method in `/*@ ... @*/`.

- **`requires`** `formula`
- **`ensures`** `formula`
- **`modifies`** `variables`:
  - The method only changes values of `variables`
- **`signals`** `(exception) formula`:
  - If the method signals `exception` then `formula` holds.
- **`signals_only`** `exceptions`:
  - The method may only throw exceptions that are a subtype of one of the `exceptions`.
  - If omitted, method can signal only exceptions that appear in **`throws`** clause.
- **`diverges`** `formula`:
  - The function may only diverge if `formula` holds.

Specifying Side Effects

- Side effects of method calls are not restricted to the state of the object on which the method is invoked.
- A method can change the heap in an unpredictable way.
- How can we specify side effects?
- We add frame conditions to contracts that specify which parts of the heap are not affected by a method call.

Specifying Side Effects

The **assignable** clause restricts the possible changes to the heap.

The specification

```
/*@ requires x >= 0;
  @ modifies \nothing;
  @ ensures \result <= Math.sqrt(x) &&
  @         Math.sqrt(x) < \result + 1;
  @*/
public static int isqrt(int x) {
  body
}
```

expresses that `isqrt` has no side effects.

Structuring Specifications with **also**

```
/*@ requires x > 0;
  @ ensures \return = 2*x;
  @ also
  @ requires x <= 0;
  @ ensures \return = 0;
  @*/
public int foo (int x) { body }
```

Specifying Exceptions

```
/*@ signals (IllegalArgumentException e) x < 0;
  @ signals_only IllegalArgumentException;
  @*/
public static int isqrt(int x) { body }
```

- If `IllegalArgumentException` is thrown, `x < 0` holds.
- `IllegalArgumentException` is the only type of exception that is thrown.
- If no **`signals_only`** clause is specified, JML assumes a sane default value: the method may throw only exceptions it declares with the **`throws`** keyword (in this case none).
- The code is still allowed to throw an error like an `OutOfMemoryError` or a `ClassNotFoundError`.

Making Exceptions Explicit

```
/*@ public normal_behavior
  @ requires x >= 0;
  @ modifies \nothing;
  @ ensures \result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
  @ also
  @ public exceptional_behavior
  @ requires x < 0;
  @ modifies \nothing;
  @ signals (IllegalArgumentException e) true;
  @*/
public static int isqrt(int x) throws IllegalArgumentException {
  if (x < 0) throw new IllegalArgumentException();
  body
}
```

Making Exceptions Explicit

- If several specifications are given with **`also`**, the method must fulfill **all** of these specifications.
- A specification with **`normal_behavior`** implicitly has the clause

  ```
  signals (java.lang.Exception) false
  ```

  so the method may not throw an exception.
- A specification with **`exceptional_behavior`** implicitly has the clause

  ```
  ensures false
  ```

  so the method may not terminate normally.

Lightweight vs. Heavyweight Specifications

A lightweight specification

```
/*@ requires P;
  @ modifies X;
  @ ensures Q;
  @*/
public void foo() throws IOException;
```

is an abbreviation for the heavyweight specification

```
/*@ public normal_behavior
  @ requires P;
  @ diverges false;
  @ modifies X;
  @ ensures Q;
  @ signals_only IOException
  @*/
public void foo() throws IOException;
```

Pure Methods

The specification

```
public /*@ pure @*/ boolean isEmpty () { body }
```

is an abbreviation for the specification

```
/*@ modifies \nothing;
  @ diverges false;
  @*/
public boolean isEmpty () { body }
```

Null References

The specification

```
public void foo (/*@non_null*@/ Object o);
```

is an abbreviation for the specification

```
//@ requires o != null;
public void foo (Object o);
```

By default, all references are `non_null`, i.e. nullable references have to be specified explicitly:

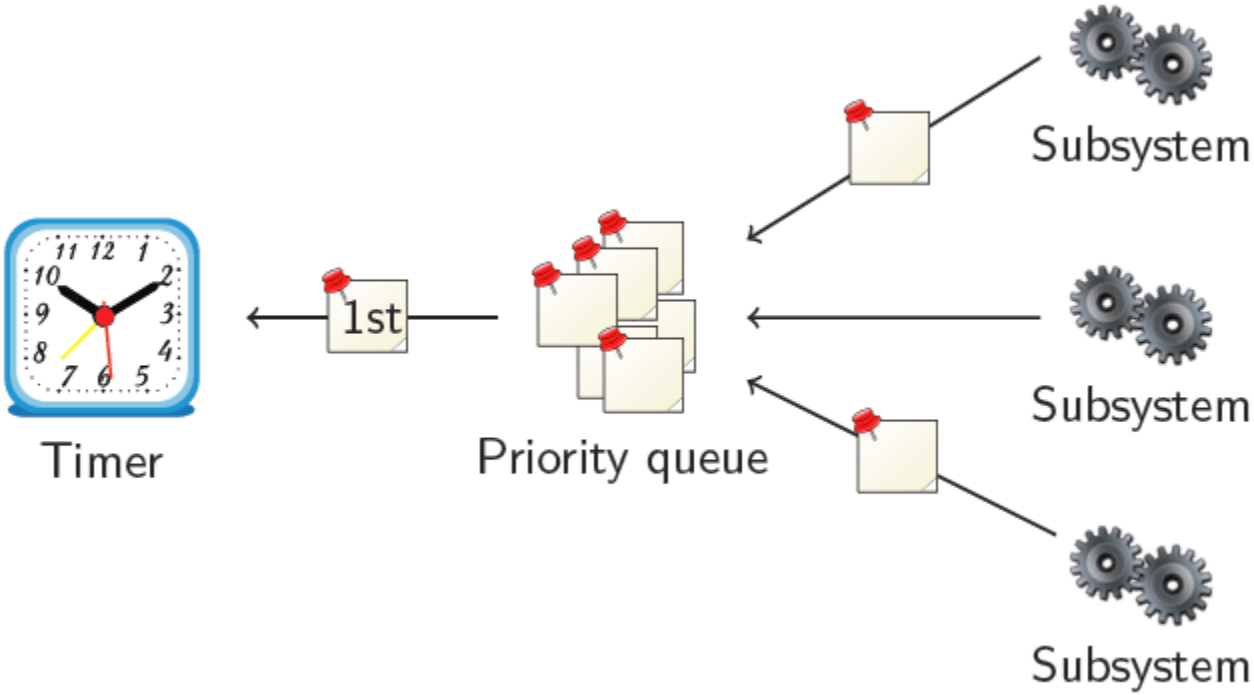
```
public void foo (/*@nullable*@/ Object o);
```

JML Syntax: Class Specifications

In JML class invariants are also in `/*@ ... @*/`.

- **`invariant`** `formula`:
  - Whenever a method is called or returns, then `formula` has to hold.
- **`constraint`** `formula`:
  - `formula` defines a history constraint, i.e. a relation between any states *s* and *s'* such that *s'* occurs after *s* in an execution of the program.

Case Study: Priority Queue



- Subsystems request timer events and queue them.
- First timer event is passed to the timer.
- Priority queue maintains events in its internal data structure.

Priority Queue Interface

```
public interface PriorityQueue {
  public void enqueue(Comparable o);
  public Comparable removeFirst();
  public boolean isEmpty();
}
```

Adding Specifications: 1st Attempt

```
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();
}
```

Specification Is Incomplete

The specification allows undesired behavior:

- After `removeFirst()` new value of `isEmpty()` is undefined.
- In a correct implementation, after two `enqueue()` and one `removeFirst()` the queue is not empty.
- The specification does not say so.
- Problem:
  - The internal state is not visible in the specification
  - There is not even internal state in an interface!

Adding Model Fields

Solution: add a model field that records the size.

```
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 Fields

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

- A model field only exists in specifications.
- Public model fields can be accessed by specifications of other classes.
- Only specifications can access model fields (they are read-only).
- If a model field is accessed in code, the compiler complains.

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`.
- The specification is part of the interface.
- To understand the specification, one needs to know about `size`.
- Therefore, `size` is `public`.

Implementing the Specification

```
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;
  }
}
```

Representing Model Fields

- Every model field in a concrete class must be represented:

```
//@ private represents size = numElems;
```

- The representing expression can also call pure methods:

```
//@ private represents size = computeSize();
```

Obtaining Complete Specifications

- The specification is still incomplete.
- Which values are returned by `removeFirst()`?
- We need a model field representing the queue.
- JML provides useful predefined types to model complex data structures.

Complete Specification of Priority Queue

```
//@ model import org.jmlspecs.models.JMLObjectBag;
public interface PriorityQueue {
  //@ public instance model JMLObjectBag queue;
  /*@ public normal_behavior
    @ ensures queue.equals(\old(queue).insert(o));
  public void enqueue(Comparable o);
  /*@ public normal_behavior
    @ requires !isEmpty();
    @ ensures \old(queue).has(\result) &&
    @         queue.equals(\old(queue).remove(\result)) &&
    @         (\forall java.lang.Comparable o;
    @                 queue.has(o); \result.compareTo(o) <= 0);
  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.
- A pure class has only pure methods and no references to non-pure classes.
- Therefore, it can be used in specifications.
- JML provides many predefined types:

  http://www.cs.iastate.edu/~leavens/JML-release/javadocs/org/jmlspecs/models/package-summary.html

How Does It Work?

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

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

- `JMLObjectBag` is immutable
- The `insert` method of `JMLObjectBag` is declared as

```
public /*@pure@*/ JMLObjectBag insert(/*@nullable@*/ Object elem)
```

- Compare this to the add method of List:

```
public boolean add(/*@nullable@*/ Object elem)
```

- `insert` returns a reference to a new larger bag.
- the content of `\old(queue)` and `queue` never change
- but `\old(queue)` and `queue` are references to different objects.

Representing queue using a Model Method

```
//@model import org.jmlspecs.models.JMLObjectBag;
public class Heap implements PriorityQueue {
  private Comparable[] elems;
  private int numElems;
  //@ 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 queue by a Ghost Field

```
//@ model import org.jmlspecs.models.JMLObjectBag;
public class Heap implements PriorityQueue {
  private Comparable[] elems;
  private int numElems;
  //@ private ghost JMLObjectBag ghostQueue;
  //@ 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;
  /*@ public normal_behavior
    @ ensures queue.equals(\old(queue).insert(o));
    @*/
  public void enqueue(/*@non_null@*/ Comparable o);
  ...
}
```

Compilation 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 an `assignable` clause!

Adding assignable

What does the method enqueue change?

It changes the model field queue and nothing else.

```
//@ model import org.jmlspecs.models.JMLObjectBag;
public interface PriorityQueue {
  //@ public instance model JMLObjectBag queue;
  /*@ public 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 Fields To Model Fields

We have to tell JML that `elem` and `numElems` are the implementation of the model field `queue`.

There is a special JML syntax to do this:

```
import org.jmlspecs.models.JMLObjectBag;
  public class Heap implements PriorityQueue {
  private Comparable[] elems; //@ in queue;
  private int numElems; //@ in queue;
  /*@ private represents queue = computeQueue(); @*/
  ...
}
```

Data Groups

- A data group gives a name to a set of locations without exposing implementation details.
- Every model field forms a data group.
- Other fields in the class or in sub-classes can be associated with this data group

```
private Comparable[] elems; //@ in queue;
private int numElems; //@ in queue;
```

- Methods with specification

```
    assignable queue
```

may modify any field in the data group `queue`.

More About Data Groups

- There is a special data group `objectState`, which should represent the object state.
- All representation fields should be added to this group.
- Adding a data group to another data group adds all subgroups recursively:

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

After this change `numElems` and `elems` are also automatically contained in `objectState`.

Grouping Fields with Data Groups

Data groups are useful to group fields.

```
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();
}
```

Data Groups and Visibility

Data groups and model fields are useful for resolving visibility issues:

```
class Tree {
  //@ public model JMLDataGroup content;
  //@ in objectState;
  private Node rootNode; //@ in content;
  //@ assignable content;
  public void insert(Object o);
}
```

Using `assignable rootNote` would produce an error.