The Java Modeling Language – JML

Néstor Cataño

ncatano@puj.edu.co

Faculty of Engineering

Pontificia Universidad Javeriana

Lecture Plan

1. An Introduction to JML
2. JML's Specification Pragmas
3. JML's Advanced Features
4. JML Specification Examples
5. Tool Support for JML

Lecture Plan

1. An Introduction to JML
2. JML's Specification Pragmas
3. JML's Advanced Features
4. JML Specification Examples
5. Tool Support for JML

The Java Modeling Language

- Formal specification language for Java programs
  - record design and implementation decisions
  - specify behaviour of classes
- JML specifications describe
  - interfaces (names and static information found in Java declarations)
  - behaviour (how classes and interfaces act when used)

JML's Goals

- Easy to understand for any Java programmer
  - Notation close to Java syntax

JML's Goals

- Easy to understand for any Java programmer
  - Notation close to Java syntax
- Rigorous and formal
  - A formal semantics for JML exists

JML's Goals

- Easy to understand for any Java programmer
  - Notation close to Java syntax
- Rigorous and formal
  - A formal semantics for JML exists
- Amenable to tool support
  - KRAKATOA, JACK, ESC/JAVA, JML, LOOP

JML's Goals

- Easy to understand for any Java programmer
  - Notation close to Java syntax
- Rigorous and formal
  - A formal semantics for JML exists
- Amenable to tool support
  - KRAKATOA, JACK, ESC/JAVA, JML, LOOP
- Practical, effective for detailed designs

JML Notation

- JML specifications
  - Predicates in First-Order Logic
  - Java boolean expressions
  - Specification-purpose boolean JML constructs
- Specifications are added between `/*@ ... @*/` or after `//@` in Java code
  - preconditions
  - postconditions
  - invariants

JML Operators

- Java logical operators (`||`, `&&`, `!`)
- Java relational and bitwise operators (`<`, `<=`, ..., `<<`,`>>`)
- Logical forward and reverse implication (`==>`, `<==`)
- Logical equivalence and non-equivalence (`<==>`, `<=!=>`)
- Universal and existential quantification (`\forall`, `\exists`)

JML Predicates

- JML predicates are expressed in first order logic, constructed using
  - side-effect-free Java booleans,
  - extended with a few operators (`\old`, `\result`, `\forall`, `\exists`, `\max`, etc.),
  - and using a few keywords (`requires`, `ensures`, `invariant`, etc.)

JML by Example

```
public class IntMathOps {                                    // 1
                                                             // 2
 /*@ public normal_behavior                                 // 3
   @  requires y >= 0;                                       // 4
   @  assignable \nothing;                                   // 5
   @  ensures 0 <= \result &&                                // 6
   @          \result * \result <= y &&                       // 7
   @          y < (\reult + 1) * (\result + 1);              // 8
   @*/                                                       // 9
  public static int isqrt(int y)                             //10
  {                                                          //11
    return (int) Math.sqrt(y);                               //12
  }                                                          //13
}                                                            //14
```

JML by Example

- `requires`. It is used for specifying preconditions.
- `ensures`. It is used for specifying postconditions.
- `normal_behavior`. The method terminates normally, without throwing an exception.
- `exceptional_behavior`. The method throws an exception.
- `behavior`. The method may terminate normally or abruptly.

Lightweight Specifications

- They do not use a **`behavior`** keyword
- They are written in individual lines after **`//@`**
- They are typical of ESC/JAVA

```
public class IntMathOps3 {
  //@ requires y >= 0;
  public static int isqrt(int y)
  {
    return (int) Math.sqrt(y);
  }
}
```

Lightweight Specifications

- If a **`requires`** keyword is missed, a **`requires true;`** specification is assumed
- If a **`assignable`** keyword is missed, an **`assignable \everything;`** specification is assumed
- If a **`ensures`** keyword is missed, an **`ensures true;`** specification is assumed

JML and Design-by-Contract

- Methods pre- and postconditions define a contract between the method (the class) and the caller of the method
- This contract stipulates that
  1. Methods may assume preconditions and must ensure postconditions
  2. Clients must ensure preconditions and may assume postconditions

Contracts and Proof Obligations

```
class Decimal {
 int intPart, decPart;
 //@ invariant decPart >= 0;

 /*@ requires m != null;
   @ ensures decPart == m.decPart &&
   @         intPart == \old(intPart);
   @*/
  void setDecimal(Decimal m) {
    decPart = m.decPart;
  }
 }
```

Contracts and Proof Obligations

For method `setDecimal`:

- If `m != null` and `m.decPart >= 0` then
  - `decPart == m.decPart`, and
  - `intPart == \old(intPart)`

Contracts and Proof Obligations

For method `setDecimal` :

- If `m != null` and `m.decPart >= 0` then
  - `decPart == m.decPart`, and
  - `intPart == \old(intPart)`

For the caller :

- `o != null` and `o.decPart >= 0` must be true in every place where the method call `setDecimal(o)` is made
- `setDecimal`'s postcondition may be assumed in those places

Contracts and Proof Obligations

For method `setDecimal` :

- If `m != null` and `m.decPart >= 0` then
  - `decPart == m.decPart`, and
  - `intPart == \old(intPart)`

For the caller :

- `o != null` and `o.decPart >= 0` must be true in every place where the method call `setDecimal(o)` is made
- `setDecimal`'s postcondition may be assumed in those places

Additionally :

- `decPart >= 0` must be an invariant of `Decimal` class

Lecture Plan

1. An Introduction to JML
2. JML's Specification Pragmas
3. JML's Advanced Features
4. JML Specification Examples
5. Tool Support for JML

Lecture Plan

1. An Introduction to JML
2. JML's Specification Pragmas
3. JML's Advanced Features
4. JML Specification Examples
5. Tool Support for JML

Java Card's `Decimal` class

```
public class Decimal extends Object {
  public static final short MAX_DECIMAL_NUMBER = (short) 32767;
  public static final short PRECISION = (short) 1000;
  private short intPart = (short) 0;
  private short decPart = (short) 0;

  public Decimal setValue(Decimal d) throws DecimalException {
   return setValue(d.getIntPart(),d.getDecPart());
  }

  public Decimal setValue(short i, short d) throws DecimalException {
   if(i < 0 || d < 0) decimal_exception.throwIt(DECIMAL_OVERFLOW);
   intPart = i;
   decPart = d;
   return this;
  }
  ...
}
```

\forall and \exists

```
(\forall int i; a[i] != null);

(\forall int i; (0 <= i && i < length) ==>
                  src[srcOff + i] == dest[destOff + i]);

(\exists int i; (0 <= i && i < length) &&
                (\forall int j; (0 <= j && j < i) ==>
                                  src[srcOff+j] == dest[destOff+j]));
```

assert

- **`assert P`**
  - **P** must hold at certain point in a method body
  - **P** is a valid JML first-order logic property

assert

```
if (i <= 0 || j < 0) {
  ...
}
else if (j < 5) {
//@ assert i > 0 && 0 < j && j < 5;
 ...
}
else {
//@ assert i > 0 && j > 5;
  ...
}
```

`spec_public`

- JML does not allow **`private`** fields in **`public`** specifications
- Putting **`/*@ spec_public @*/`** before a **`private`** field declaration causes the field to be included in the scope of every specification

spec_public

```
public class Decimal extends Object {
  public static final short MAX_DECIMAL_NUMBER = (short) 32767;
  public static final short PRECISION = (short) 1000;
  /*@ spec_public @*/ private short intPart = (short) 0;
  /*@ spec_public @*/ private short decPart = (short) 0;
   ...
}
```

`requires`, `ensures` and `normal_behavior`

- `requires R`.
  - method precondition `R`
- `ensures Q`
  - method postcondition `Q`
  - `Q` must hold if method terminates normally, i.e., without throwing a `java.lang.Exception`
- `normal_behavior` (total correctness)
  - If precondition holds in the pre-state then
  - method terminates in a normal state, and
  - normal post-condition holds in this state

`requires`, `ensures` and `normal_behavior`

```
public class Decimal extends Object {

 /*@ normal_behavior;
   @ requires i>=0 && d>=0;
   @ ensures intPart == i && decPart == d;
   @*/
  public Decimal setValue(short i, short d) throws DecimalException {
    if(i < 0 || d < 0) Exception.throwIt(DECIMAL_OVERFLOW);
    intPart = i;
    decPart = d;
    return this;
  }
}
```

`signals`, `exceptional_behavior`

- `signals (E e) R`
  - exceptional postcondition `R`
  - `R` must hold if method throws an exception `e` that is a subclass of `E`
- `exceptional_behavior` (total correctness)
  - If precondition holds in the pre-state then
  - method will always terminate in a exceptional state throwing a `java.lang.Exception`, and
  - the exceptional post-condition will hod in that state

`signals` and `exceptional_behavior`

```
public class Decimal extends Object {

 /*@ exceptional_behavior;
   @ requires i < 0 || d < 0;
   @ signals (DecimalException e) true;
   @*/
  public Decimal setValue(short i, short d) throws DecimalException {
   if(i < 0 || d < 0) Exception.throwIt(DECIMAL_OVERFLOW);
   intPart = i;
   decPart = d;
   return this;
  }
}
```

`assignable`

- **`assignable L`**
  - method may only modify the set of memory locations denoted by **`L`**
  - any other location not listed may therefore not be modified
  - this is true for both normal and exceptional postconditons

assignable

```
public class Decimal extends Object {
 /*@ normal_behavior;
   @   requires i>=0 && d>=0;
   @   assignable intPart, decPart;
   @   ensures true;
   @*/
  public Decimal setValue(short i, short d) throws DecimalException {
    if(i < 0 || d < 0) Exception.throwIt(DECIMAL_OVERFLOW);
    intPart = i;
    decPart = d;
    return this;
  }
}
```

`model`, `in`, and `represents`

- `model` fields
  - specification purpose fields
  - do not exist as program variables
  - used to write abstract specifications
  - can be mentioned in `assignable` specifications
- `in`
  - declares that an abstract value is related to some others abstract or concrete values
  - used in checking `assignable` specifications

`model`, `in`, and `represents`

- `represents` clauses
  - make the relation between abstract and concrete values explicit
  - relation may be functional, `<-`, or relational `such_that`

`model`, `in`, and `represents`

```
public class Decimal extends Object {
  ...
  public short intPart = (short) 0;  //@ in decimal;
  public short decPart = (short) 0;  //@ in decimal;

 /*@ model int decimal;
   @ represents decimal <- intPart * PRECISION + decPart; @*/
   ...
}
```

- `decimal` is a function over `intPart`, `PRECISION` and `decPart`
- If `intPart` or `decPart` is modified, then `decimal` is modified

`model`, `in`, and `represents`

```
public class Decimal extends Object {
  ...
 public short intPart = (short) 0;  //@ in decimal;
 public short decPart = (short) 0;  //@ in decimal;

 /*@ model int decimal;
   @ represents decimal <- intPart * PRECISION + decPart; @*/

 /*@ behavior
   @  requires true;
   @  assignable decimal;
   @  ensures decimal == i * PRECISION + d;
   @  signals (DecimalException e) i<0 || d<0;
   @*/
  public Decimal setValue(short i, short d) throws DecimalException {
    ...
  }
}
```

`\old` and `\fresh`

- `\old(e)`
  - value of expression `e` in the pre-state
- `\fresh(x)`
  - `x` is not null
  - `x` was not allocated in the pre-state
- `\old` and `\fresh` can only be used in normal or exceptional postconditions

\old and \fresh

```
/*@ behavior
  @  requires true;
  @  assignable decimal;
  @  ensures decimal == i * PRECISION + d;
  @  signals (DecimalException e) (i<0 || d<0) &&
  @                                decimal == \old(decimal) &&
  @                                \fresh(e);
  @*/
 public Decimal setValue(short i, short d) throws DecimalException {
   ...
 }
```

\result

- represents the value returned by a method
- has the same type as the method return type
- can only be used in normal or exceptional postconditions

\result

```
/*@ normal_behavior
  @  requires true;
  @  ensures \result <==>
  @          (\exists int i; i>=0 && i<MAX_DATA && data[i]==cur);
  @*/
boolean contens(byte cur) {
  boolean resu = false;
  byte i = (byte)0;

  boolean found = false;

  while(i < MAX_DATA && ! resu) {
    if(data[i] == cur) resu = true;
    else i++;
  }
  return resu;
}
```

`pure`

- methods in JML specifications must be side-effects free
  - for instance, `x++`, `y=x+1`, are not allowed in JML specs
- a side-effects free method is called `pure`
- `pure` methods are implicitly `assignable \nothing;`

pure

```
public class Decimal extends Object {
 //@ assignable \nothing;
 public short getIntPart(){
   return intPart;
 }
}
```

pure

```
public class Decimal extends Object {

 public /*@ pure @*/ short getIntPart(){
   return intPart;
 }
}
```

invariant

- it must hold in all visible state
  - Beginning and end of any method invocation
  - End of a constructor invocation
- it is implicitly assumed at the beginning of a method declaration
- it does not have to hold during the execution of a method, so it can temporally be broken

invariant

```
public class Decimal extends Object {
  public static final short MAX_DECIMAL_NUMBER = (short) 32767;
  public static final short PRECISION = (short) 1000;
   public short intPart = (short) 0;
   public short decPart = (short) 0;

  /*@ invariant 0 <= intPart && intPart <= MAX_DECIMAL_NUMBER &&
    @           0 <= decPart && decPart < PRECISION;
    @*/
   ...
}
```

non_null

- replaces an invariant specification about an object being non-null
- useful short-hand for many purposes

non_null

```
public class SalerID extends Object implements PartnerID {
  ...
  //@ invariant data != null;
  public byte[] data = new byte[ID_LENGTH];
 ...
}
```

non_null

```
public class SalerID extends Object implements PartnerID {
  ...

  public /*@ non_null @*/ byte[] data = new byte[ID_LENGTH];

 ...
}
```

Recapitulation

- `\forall`, `\exists`
- `assert`
- `requires`, `ensures`, `signals`
- `normal_behavior`, `exceptional_behavior`
- `assignable`
- `model`, `represents`
- `\old`, `\fresh`
- `\result`
- `pure`
- `invariant`, `non_null`

Resources

- JML's Web-site http://www.jmlspecs.org
- The JML Tool (G. Leavens et al.)
  http://sourceforge.net/project/showfiles.php?group_id=65346
- The ESC/JAVA Tool (K. Leino et al., J. Kiniry et al.)
  http://secure.ucd.ie/products/opensource/ESCJava2/
- The KRAKATOA Tool (Ch. Paulin et al.) http://krakatoa.lri.fr/
- The JACK Tool (J.-L. Lanet et al.)
  http://www-sop.inria.fr/everest/soft/Jack/jack.html