Formal Specification of Software

Bernhard Beckert

Adaptation of slides by
Wolfgang Ahrendt
Chalmers University, Gothenburg, Sweden

Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General Philosophy

Integrate

- specification and
- implementation

in one single language.

$\Rightarrow$ JML is not external to JAVA

JML

is

Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General Philosophy

Integrate

- specification and
- implementation

in one single language.

⇒ JML is not external to JAVA

JML
is
JAVA

Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General Philosophy

Integrate

- specification and
- implementation

in one single language.

$\Rightarrow$ JML is not external to JAVA

JML

is

JAVA + FO Logic + pre/post-conditions, invariants + more ...

Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General Philosophy

Integrate

- specification and
- implementation

in one single language.

⇒ JML is not external to JAVA

JML
is
JAVA + FO Logic + pre/post-conditions, invariants + more ...

Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General Philosophy

Integrate

- specification and
- implementation

in one single language.

$\Rightarrow$ JML is not external to JAVA

JML
is
JAVA + FO Logic + pre/post-conditions, invariants + more ...

JML Annotations

JML extends JAVA by annotations.

JML annotations include:

- ✔ preconditions
- ✔ postconditions
- ✘ intermediate assertions
- ✔ class invariants
- ✔ additional modifiers
- ✘ 'specification-only' field declarations
- ✘ 'specification-only' field conditions
- ✘ 'specification-only' field assignments
- ✘ ...

✔: in this course, ✘: not in this course

JML/Java integration

JML annotations are attached to JAVA programs
by
writing them directly into the JAVA source code files!

But to not confuse the JAVA compiler:

JML annotations live in in special comments,
ignored by JAVA, but recognised by JML.

JML/Java integration

JML annotations are attached to JAVA programs
by
writing them directly into the JAVA source code files!

But to not confuse the JAVA compiler:

JML annotations live in in special comments,
ignored by JAVA, but recognised by JML.

JML Example 1

from the file `ATM.java`

```
⋮

/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....

⋮
```

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

Everything between `/*` and `*/` is invisible for JAVA.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

But:

A JAVA comment with '@' as its first character
is *not* a comment for JML.

(Non-JAVA) JML annotations appear in JAVA comments starting with @.

How about "//" comments?

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

But:

A JAVA comment with '@' as its first character
is *not* a comment for JML.

(Non-JAVA) JML annotations appear in JAVA comments starting with @.

How about "//" comments?

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

But:

A Java comment with '@' as its first character
is *not* a comment for JML.

(Non-Java) JML annotations appear in Java comments starting with @.

How about "//" comments?

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

But:

A JAVA comment with '@' as its first character
is *not* a comment for JML.

(Non-JAVA) JML annotations appear in JAVA comments starting with @.

How about "//" comments?

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

is equivalent to:

```
//@ public normal_behavior
//@ requires !customerAuthenticated;
//@ requires pin == insertedCard.correctPIN;
//@ ensures customerAuthenticated;
public void enterPIN (int pin) {
    if ( ....
```

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

What about the intermediate '@'s?

Within a JML annotation, a '@' is ignored:

- if it is the first (non-white) character in the line
- if it is the last character before '*/'.

⇒ The blue '@'s are not *required*, but it's a *convention* to use them.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

What about the intermediate '@'s?

Within a JML annotation, a '@' is ignored:

- if it is the first (non-white) character in the line
- if it is the last character before '*/'.

⇒ The blue '@'s are not *required*, but it's a *convention* to use them.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

What about the intermediate '`@`'s?

Within a JML annotation, a '`@`' is ignored:

- if it is the first (non-white) character in the line
- if it is the last character before '`*/`'.

⇒ The blue '`@`'s are not *required*, but it's a *convention* to use them.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

This is a `public` specification case, meaning it:

1. is visible from all classes and interfaces
2. can only mention `public` fields/methods of this class

2. is normally a problem. Solution later in the lecture.

In this course: only `public` specifications.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

This is a **public** specification case, meaning it:

1. is visible from all classes and interfaces
2. can only mention `public` fields/methods of this class

2. is normally a problem. Solution later in the lecture.

In this course: only `public` specifications.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

This is a **public** specification case, meaning it:

1. is visible from all classes and interfaces
2. can only mention `public` fields/methods of this class

2. is normally a problem. Solution later in the lecture.

In this course: only `public` specifications.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

Each keyword ending on **behavior** opens a ‘specification case’.

normal_behavior opens a ‘normal behavior specification case’:
The method guarantees normal termination if the caller guarantees all preconditions of this specification case.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

This specification case has two preconditions (marked by `requires`)

1. `!customerAuthenticated`
2. `pin == insertedCard.correctPIN`

Here, the preconditions are `boolean` JAVA expressions.

In general, pre/postconditions and invariants are `boolean` JML expressions.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

This specification case has two preconditions (marked by `requires`)

1. `!customerAuthenticated`
2. `pin == insertedCard.correctPIN`

Here, the preconditions are `boolean` JAVA expressions.

In general, pre/postconditions and invariants are `boolean` JML expressions.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
```

This specifies only the case where both preconditions are true in the prestate.

I.e., the above is equivalent to:

```
/*@ public normal_behavior
  @ requires (     !customerAuthenticated
  @              && pin == insertedCard.correctPIN );
  @ ensures customerAuthenticated;
  @*/
```

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

This specification case has one postcondition (marked by **`ensures`**)

- `customerAuthenticated`

Again, the postcondition is a `boolean` JAVA expressions.

Again, in general pre/postconditions and invariants are `boolean` JML expressions.

Discussion Example 1

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

This specification case has one postcondition (marked by **ensures**)

- customerAuthenticated

Again, the postcondition is a boolean JAVA expressions.

Again, in general pre/postconditions and invariants are boolean JML expressions.

Discussion Example 1

Different specification cases are connected by ‘also’.

```
/*@ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin == insertedCard.correctPIN;
  @ ensures customerAuthenticated;
  @
  @ also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2;
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) {
    if ( ....
```

Discussion Example 1

```
/*@ <spec-case1> also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 2;
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) { ...
```

Now, for the first time, we have a JML expression which is not a Java expression.

`\old(`*E*`)` is: *E* evaluated in the prestate of `enterPIN`.

E can be any (arbitrarily complicated) Java/JML expression.

Discussion Example 1

```
/*@ <spec-case1> also <spec-case2> also
  @
  @ public normal_behavior
  @ requires insertedCard != null;
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter >= 2;
  @ ensures insertedCard == null;
  @ ensures \old(insertedCard).invalid;
  @*/
public void enterPIN (int pin) { ...
```

Ths specification case has two postconditions, stating that:

'Given the above preconditions, `enterPIN` guarantees:

`(insertedCard == null && \old(insertedCard).invalid)`'

JML Modifiers

JML extends the JAVA modifiers by additional modifiers.

The most important ones are:

- `spec_public`
- `pure`

Aim: admitting more class elements to be used in JML expressions.

JML Modifiers: `spec_public`

In Example 1 (`enterPIN`), pre- and postconditions made heavy use of class fields.

But: `public` specifications can only talk about `public` fields.

Not desired: make all fields public.

Solution:

- keep the fields `private/protected`
- make those needed for specification `spec_public`

```
private /*@ spec_public @*/ boolean customerAuthenticated;
private /*@ spec_public @*/ int wrongPINCounter;
```

JML Modifiers: `spec_public`

In Example 1 (`enterPIN`), pre- and postconditions made heavy use of class fields.

But: `public` specifications can only talk about `public` fields.

Not desired: make all fields public.

Solution:

- keep the fields `private`/`protected`
- make those needed for specification `spec_public`

```
private /*@ spec_public @*/ boolean customerAuthenticated;
private /*@ spec_public @*/ int wrongPINCounter;
```

JML Modifiers: `spec_public`

In Example 1 (`enterPIN`), pre- and postconditions made heavy use of class fields.

But: `public` specifications can only talk about `public` fields.

Not desired: make all fields public.

Solution:

- keep the fields `private`/`protected`
- make those needed for specification `spec_public`

```
private /*@ spec_public @*/ boolean customerAuthenticated;
private /*@ spec_public @*/ int wrongPINCounter;
```

JML Modifiers: `pure`

It can be handy to use method calls in JML annotations.
Examples:

- `o1.equals(o2)`
- `li.contains(elem)`
- `li1.max() < li2.min()`

This is allowed if, and only if, the method call is guaranteed to have no side effects.

In JML, you can specify methods to be '`pure`':

```
public /*@ pure @*/ int max() { ...
```

The '`pure`' modifier puts an additional obligation on the implementer (no to use side effects), but allows to use the method in annotations.

JML Expressions and FO Logic

So far: pre/postconditions did not use first-order logic formulae, but simply `boolean` JAVA expressions.

But: last lecture motivated the need for more powerful features, foremost quantification[1].

⇒ many specification frameworks employ *formulas* of some `logic`

Not so JML!

Design decision taken in JML

Instead of going from JAVA `boolean` expressions to a more expressive logic, make the `boolean` expressions more expressive themselves.

[1]see `List::set()`

JML Expressions and FO Logic

So far: pre/postconditions did not use first-order logic formulae, but simply `boolean` JAVA expressions.

But: last lecture motivated the need for more powerful features, foremost quantification[1].

⇒ many specification frameworks employ *formulas* of some `logic`

Not so JML!

Design decision taken in JML

Instead of going from JAVA `boolean` expressions to a more expressive logic, make the `boolean` expressions more expressive themselves.

[1] see `List::set()`

JML Expressions and FO Logic

⇒ JML `boolean` expressions extend JAVA `boolean` expressions by:

- implication
- quantification
- (more ...)

Instead of a formula being valid, or not valid, in a certain model, we speak about a `boolean` expression being true or false in a certain state.

`boolean` JML Expressions

`boolean` JML expressions are defined recursively:

Formulae

- each side-effect free `boolean` JAVA expression is a `boolean` JML expression
- if `a` and `b` are `boolean` JML expressions, and `x` is a variable of type `t`, then the following are also `boolean` JML expressions:
  - `!a` ("not a")
  - `a && b` ("a and b")
  - `a || b` ("a or b")
  - `a ==> b` ("a implies b")
  - `a <==> b` ("a is equivalent to b")
  - `(\forall t x; a)` ("for all `x` of type `t` is true")
  - `(\exists t x; a)` ("there exists `x` of type `t` such that a")
  - `(\forall t x; a; b)` ("for all `x` of type `t` fulfilling a, b is true")
  - `(\exists t x; a; b)` ("there exists `x` of type `t` fulfilling a, such that b")

`boolean` JML Expressions

`boolean` JML expressions are defined recursively:

Formulae

- each side-effect free `boolean` Java expression is a `boolean` JML expression
- if `a` and `b` are `boolean` JML expressions, and `x` is a variable of type `t`, then the following are also `boolean` JML expressions:
  - `!a` (“not a”)
  - `a && b` (“a and b”)
  - `a || b` (“a or b”)
  - `a ==> b` (“a implies b”)
  - `a <==> b` (“a is equivalent to b”)
  - `(\forall t x; a)` (“for all `x` of type `t` is true”)
  - `(\exists t x; a)` (“there exists `x` of type `t` such that a”)
  - `(\forall t x; a; b)` (“for all `x` of type `t` fulfilling a, b is true”)
  - `(\exists t x; a; b)` (“there exists `x` of type `t` fulfilling a, such that b”)

JML Quantifiers

In the two last quantifier expressions:

`(\forall t x; a; b)` and `(\exists t x; a; b)`

`a` is called the 'range predicate'

These forms are redundant:

```
(\forall t x; a; b)
```
is equivalent to
```
(\forall t x; a ==> b)
```

and

```
(\exists t x; a; b)
```
is equivalent to
```
(\exists t x; a && b)
```

Pragmatics of Range Predicates

Even if the forms

`(\forall t x; a; b)` and `(\exists t x; a; b)`

are redundant, they are widely used.

Pragmatics of the range predicate:

`a` is used to restrict the range of `x` further than its type `t` does.

```
(\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])
```

says that `a` is sorted at indexes between 0 and 9.

The quantifiers for `i` and `j` 'range' over values making the expression between `;` and `;` true.

Generalized Quantifiers

JML offers generalised quantifiers:

- `\max`
- `\min`
- `\product`
- `\sum`

returning the maximum, minimum, product, or sum of the values of the expressions given, where the variables satisfy the given range.

Examples (all formulae are `true`):

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
(\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
(\max int i; 0 <= i && i < 5; i) == 4
(\min int i; 0 <= i && i < 5; i-1) == -1
```

Result Values in Postcondition

```
/*@ public normal_behavior
  @ ensures (\forall int j; j >= 0 && j < a.length;
  @                         \result >= a[j]);
  @*/
public static /*@ pure @*/ int max(int[] a) {
    if (...
```

In a postcondition:
one can use '`\result`' to refer to the return value of the method.

But is the above postcondition sufficient?

Result Values in Postcondition

```
/*@ public normal_behavior
  @ ensures (\forall int j; j >= 0 && j < a.length;
  @                         \result >= a[j]);
  @*/
public static /*@ pure @*/ int max(int[] a) {
    if (...
```

In a postcondition:
one can use '`\result`' to refer to the return value of the method.

But is the above postcondition sufficient?

Result Values in Postcondition

```
/*@ public normal_behavior
  @ ensures (\forall int j; j >= 0 && j < a.length;
  @                         \result >= a[j]);
  @*/
public static /*@ pure @*/ int max(int[] a) {
    if (...
```

In a postcondition:
one can use '**\result**' to refer to the return value of the method.

But is the above postcondition sufficient?

Result Values in Postcondition

```
/*@ public normal_behavior
  @ ensures (\forall int j; j >= 0 && j < a.length;
  @                         \result >= a[j]);
  @ ensures a.length > 0 ==>
  @         (\exists int j; j >= 0 && j < a.length;
  @                         \result == a[j]);
  @*/
public static /*@ pure @*/ int max(int[] a) {
    if (...
```

JML Invariants

So far: attached pre/postconditions to methods.

Now: attaching invariants to classes.

We are free where to put it in the class (potentially close to fields the invariant talks about).

JML Invariants: Example

```
/*@ public invariant
  @         accountProxies != null;
  @ public invariant
  @         accountProxies.length == maxAccountNumber;
  @ public invariant
  @         (\forall int i; i >= 0 && i < maxAccountNumber;
  @             ( accountProxies[i] == null
  @               ||
  @               accountProxies[i].accountNumber == i ));
  @*/
private /*@ spec_public nullable@*/ final
    OfflineAccountProxy[] accountProxies
        = new OfflineAccountProxy [maxAccountNumber];
```

`non_null` and `nullable`

JML extends the JAVA modifiers by further modifiers:

- class fields
- method parameters
- method return types

can be declared as

- **`nullable`**: may or may not be `null`
- **`non_null`**: must not be `null`

non_null: Examples

`private /*@ spec_public non_null @*/ String name;`

invariant

‘`public invariant name != null;`’

implicitly added to class

`public void insertCard(/*@ non_null @*/ BankCard card) {..`

precondition

‘`requires card != null;`’

implicitly added to each specification case of `insertCard`

`public /*@ non_null @*/ String toString()`

postcondition

‘`ensures \result != null;`’

implicitly added to each specification case of `toString`

`non_null` is default in JML!

⇒ same effect even without explicit '`non_null`'s

`private /*@ spec_public @*/ String name;`

invariant
'`public invariant name != null;`'
implicitly added to class

`public void insertCard(BankCard card) {..`

precondition
'`requires card != null;`'
implicitly added to each specification case of `insertCard`

`public String toString()`

postcondition
'`ensures \result != null;`'
implicitly added to each specification case of `toString`

`nullable`: Examples

To prevent such pre/post conditions and invariants: '`nullable`'

```
private /*@ spec_public nullable @*/ String name;
```

no implicit invariant added

```
public void insertCard(/*@ nullable @*/ BankCard card) {..
```

no implicit precondition added

```
public /*@ nullable @*/ String toString()
```

no implicit postcondition added to specification cases of `toString`

LinkedList: non_null or nullable?

```
public class LinkedList {
    private Object elem;
    private LinkedList next;
    ....
```

In JML this means:

- all elements in the list are `non_null`
- the list is cyclic, or infinite!

LinkedList: non_null or nullable?

```
public class LinkedList {
    private Object elem;
    private LinkedList next;
    ....
```

In JML this means:

- all elements in the list are `non_null`
- the list is cyclic, or infinite!

LinkedList: non_null or nullable?

```
public class LinkedList {
    private Object elem;
    private LinkedList next;
    ....
```

In JML this means:

- all elements in the list are `non_null`
- the list is cyclic, or infinite!

LinkedList: non_null or nullable?

Repair:

```
public class LinkedList {
    private Object elem;
    private /*@ nullable @*/ LinkedList next;
    ....
```

⇒ Now, the list is allowed to end somewhere!

Final Remark on `non_null` and `nullable`

`non_null` as default in JML is fairly new.

$\Rightarrow$ Not yet well reflected in literature and tools.

JML and Inheritance

All JML contracts, i.e.

- specification cases
- class invariants

are inherited down from superclasses to subclasses.

A class has to fulfill all contracts of its superclasses.

Recall the `hashCode` problem from lecture 6.

Literature

This was an intro into JML essentials.
Two tutorial papers:

- Gary T. Leavens, Yoonsik Cheon.
  Design by Contract with JML
- Gary T. Leavens, Albert L. Baker, and Clyde Ruby.
  JML: A Notation for Detailed Design

Both go beyond today's lecture, but that doesn't hurt.
The reference manual, for look-up:

- Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, and Joseph Kiniry.
  JML Reference Manual

all available at
`www.eecs.ucf.edu/~leavens/JML/documentation.shtml`

Tools

Many tools support JML (see `www.eecs.ucf.edu/~leavens/JML/`).
Most basic tool set:

- `jml`, a syntax and type checker
- `jmlc`, JML/Java compiler. Compile runtime assertion checks into the code.
- `jmldoc`, like `javadoc` for Java + JML
- `jmlunit`, unit testing based on JML

We recommend to use `jml` to check the syntax.