Formal Specification and Verification
Java Modeling Language

Bernhard Beckert

Based on a lecture by Wolfgang Ahrendt and Reiner Hähnle at Chalmers University, Göteborg
Road-map

Deductive Verification of Java source code
Deductive Verification of Java source code

1. foundations: proving in first-order logic (done)
Deductive Verification of Java source code

1. foundations: proving in first-order logic (done)

3. proving Java programs correct (later)
1. foundations: proving in first-order logic (done)
2. specifying Java programs (comes now)
3. proving Java programs correct (later)
What kind of Specifications

system level specifications
(requirements analysis, GUI, use cases)
important, but
not subject of this course
What kind of Specifications

system level specifications
(requirements analysis, GUI, use cases)
important, but
not subject of this course

instead:
unit specification—contracts among implementers on various levels:
What kind of Specifications

*system level specifications*

(required analysis, GUI, use cases)

important, but

*not subject of this course*

instead:

*unit specification*—*contracts among implementers* on various levels:

- application level ↔ application level
- application level ↔ library level
- library level ↔ library level
in the object-oriented setting:
units to be specified are interfaces, classes, and their methods

first focus on methods

methods specified by potentially referring to:
**Unit Specifications**

*in the object-oriented setting:*

units to be specified are *interfaces*, *classes*, and their *methods*

first focus on methods

methods specified by *potentially* referring to:

- result value,
in the object-oriented setting:

units to be specified are interfaces, classes, and their methods

first focus on methods

methods specified by potentially referring to:

- result value,
- initial values of formal parameters,
in the object-oriented setting:
units to be specified are interfaces, classes, and their methods

first focus on methods

methods specified by potentially referring to:

- result value,
- initial values of formal parameters,
- pre-state and post-state
in the object-oriented setting:
units to be specified are interfaces, classes, and their methods

first focus on methods

methods specified by potentially referring to:

- result value,
- initial values of formal parameters,
- accessible part of pre/post-state
 Specifications as Contracts

to stress the different roles – obligations – responsibilities in a specification:

widely used analogy of the specification as a contract

“Design by Contract” methodology
Specifications as Contracts

to stress the different roles – obligations – responsibilities in a specification:

widely used analogy of the specification as a contract

“Design by Contract” methodology

contract between caller and callee of method

callee guarantees certain outcome provided caller guarantees prerequisites
public class ATM {

    // fields:
    private BankCard insertedCard = null;
    private int wrongPINCounter = 0;
    private boolean customerAuthenticated = false;

    // methods:
    public void insertCard (BankCard card) { ... }
    public void enterPIN (int pin) { ... }
    public int accountBalance () { ... }
    public int withdraw (int amount) { ... }
    public void ejectCard () { ... }
}

Informal Specification

very informal Specification of ‘enterPIN (int pin)’:

Enter the PIN that belongs to the currently inserted bank card into the ATM. If a wrong PIN is entered three times in a row, the card is confiscated. After having entered the correct PIN, the customer is regarded is authenticated.
Getting More Precise: Specification as Contract

Contract states what is guaranteed under which conditions.
Contract states **what is guaranteed under which conditions.**

**precondition**
- card is inserted, user not yet authenticated,
- pin is correct
Contract states what is guaranteed under which conditions.

**precondition**  
card is inserted, user not yet authenticated,  
    pin is correct

**postcondition**  
user is authenticated
Contract states *what is guaranteed under which conditions.*

**precondition** card is inserted, user not yet authenticated, pin is correct

**postcondition** user is authenticated

**precondition** card is inserted, user not yet authenticated, wrongPINCounter < 2 and pin is incorrect

**postcondition** wrongPINCounter is increased by 1, user is not authenticated

**postcondition** card is confiscated, user is not authenticated
Getting More Precise: Specification as Contract

Contract states **what is guaranteed under which conditions.**

**precondition** card is inserted, user not yet authenticated, pin is correct

**postcondition** user is authenticated

**precondition** card is inserted, user not yet authenticated, wrongPINCounter < 2 and pin is incorrect

**postcondition** wrongPINCounter is increased by 1 user is not authenticated
Contract states **what is guaranteed under which conditions.**

**precondition** card is inserted, user not yet authenticated, pin is correct

**postcondition** user is authenticated

**precondition** card is inserted, user not yet authenticated, wrongPINCounter < 2 and pin is incorrect

**postcondition** wrongPINCounter is increased by 1
user is not authenticated

**precondition** card is inserted, user not yet authenticated, wrongPINCounter >= 2 and pin is incorrect
Contract states what is guaranteed under which conditions.

**precondition**
- card is inserted, user not yet authenticated, pin is correct

**postcondition**
- user is authenticated

**precondition**
- card is inserted, user not yet authenticated, wrongPINCounter < 2 and pin is incorrect

**postcondition**
- wrongPINCounter is increased by 1
  - user is not authenticated

**precondition**
- card is inserted, user not yet authenticated, wrongPINCounter >= 2 and pin is incorrect

**postcondition**
- card is confiscated
  - user is not authenticated
Meaning of Pre/Post-condition pairs

Definition

A pre/post-condition pair for a method m is satisfied by the implementation of m if:

When m is called in any state that satisfies the precondition then in any terminating state of m the postcondition is true.
Meaning of Pre/Post-condition pairs

Definition

A pre/post-condition pair for a method \( m \) is satisfied by the implementation of \( m \) if:

When \( m \) is called in any state that satisfies the precondition then in any terminating state of \( m \) the postcondition is true.

1. No guarantees are given when the precondition is not satisfied.
2. Termination may or may not be guaranteed.
3. Terminating state may be reached by normal or by abrupt termination (cf. exceptions).

Formal Specification and Verification: Java Modeling Language

B. Beckert 10 / 1
Meaning of Pre/Post-condition pairs

Definition

A pre/post-condition pair for a method \( m \) is satisfied by the implementation of \( m \) if:

When \( m \) is called in any state that satisfies the precondition then in any terminating state of \( m \) the postcondition is true.

1. No guarantees are given when the precondition is not satisfied.
2. Termination may or may not be guaranteed.
3. Terminating state may be reached by normal or by abrupt termination (cf. exceptions).

non-termination and abrupt termination ⇒ next lecture
What kind of Specifications

Natural language specs are very important.
What kind of Specifications

Natural language specs are very important.

but this course’s focus:

“formal” specifications:

Describing contracts of units in a mathematically precise language.
What kind of Specifications

Natural language specs are very important.

but this course’s focus:

“formal” specifications:

Describing contracts of units in a mathematically precise language.

Motivation:

- higher degree of precision.
What kind of Specifications

Natural language specs are very important.

but this course’s focus:

“formal” specifications:
Describing contracts of units in a mathematically precise language.

Motivation:

- higher degree of precision.
- eventually: *automation* of program analysis of various kinds:
  - static checking
  - program verification
JML is a specification language tailored to JAVA.

General JML Philosophy

Integrate

- specification
- implementation

⇒ JML is not external to JAVA
JML is a specification language tailored to JAVA.

General JML Philosophy

Integrate

- specification
- implementation

in one single language.

⇒ JML is not external to JAVA

JML is JAVA
JML is a **specification language** tailored to **JAVA**.

General JML Philosophy

Integrate

- specification
- implementation

in **one single language**.

⇒ JML is not external to **JAVA**

JML

is

**JAVA + FO Logic**
Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General JML Philosophy
Integrate

- specification
- implementation

in one single language.

⇒ JML is not external to JAVA

JML is JAVA + FO Logic + pre/post-conditions, invariants
Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General JML Philosophy

Integrate

- specification
- implementation

in one single language.

⇒ JML is not external to JAVA

JML is

JAVA + FO Logic + pre/post-conditions, invariants + more ...
JML extends Java by annotations.

<table>
<thead>
<tr>
<th>JML annotations include:</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ preconditions</td>
</tr>
<tr>
<td>✓ postconditions</td>
</tr>
<tr>
<td>✓ class invariants</td>
</tr>
<tr>
<td>✓ additional modifiers</td>
</tr>
<tr>
<td>✓ ‘specification-only’ fields</td>
</tr>
<tr>
<td>✗ ‘specification-only’ methods</td>
</tr>
<tr>
<td>✓ loop invariants</td>
</tr>
<tr>
<td>✓ ...</td>
</tr>
<tr>
<td>✗ ...</td>
</tr>
</tbody>
</table>

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

to not confuse Java compiler:

JML annotations live in special comments, ignored by Java, recognized by JML.
from the file ATM.java

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

public void enterPIN (int pin) {
  if ( ....

  :

  :
JML by Example

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

public void enterPIN (int pin) {
   if ( ....

Everything between /* and */ is invisible for JAVA.
public void enterPIN (int pin) {
    if ( ....
JML by Example

/*@ 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
it is not a comment for JML.
JML by Example

```java
/*@ 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
it is not a comment for JML.

JML annotations appear in JAVA comments starting with @.
```
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
it is not a comment for JML.

JML annotations appear in JAVA comments starting with @.

How about ‘//’ comments?
JML by Example

```java
public void enterPIN (int pin) {
    if ( ....
}

equivalent to:

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

public void enterPIN (int pin) {
    if ( ....

What about the intermediate ‘@’s?
/*
 * @ 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 ‘*/’.
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.
JML by Example

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

public void enterPIN (int pin) {
    if ( ....

This is a public specification case:

1. it is accessible from all classes and interfaces
2. it can only mention public fields/methods of this class
JML by Example

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

public void enterPIN (int pin) {
    if ( ....

This is a public specification case:

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

2. Can be a problem. Solution later in the lecture.
```
JML by Example

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

public void enterPIN (int pin) {
   if ( ....

This is a public specification case:
   1. it is accessible from all classes and interfaces
   2. it can only mention public fields/methods of this class

2. Can be a problem. Solution later in the lecture.

In this course: mostly public specifications.
Each keyword ending on behavior opens a ‘specification case’.

**normal_behavior Specification Case**

The method guarantees to *not* throw any exception,
JML by Example

```java
/*@ 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 Specification Case**

The method guarantees to *not* throw any exception, if the caller guarantees all preconditions of this specification case.
public void enterPIN (int pin) {
    if ( ....
}

This specification case has two preconditions (marked by requires)
1. !customerAuthenticated
2. pin == insertedCard.correctPIN
/*@ 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:
preconditions are boolean JAVA expressions
This specification case has two preconditions (marked by requires)

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

Here:
preconditions are boolean JAVA expressions

In general:
preconditions are boolean JML expressions (see below)
specifies only the case where both preconditions are true in pre-state

the above is equivalent to:

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

This specification case has one postcondition (marked by ensures)

    customerAuthenticated

here:
postcondition is boolean JAVA expressions
/*@ 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

here:
postcondition is boolean JAVA expressions

in general:
postconditions are boolean JML expressions (see below)
different specification cases are connected by ‘also’.

```java
/*@ 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 ( ....
```
also

public normal_behavior
requires !customerAuthenticated;
requires pin != insertedCard.correctPIN;
requires wrongPINCounter < 2;
ensures wrongPINCounter == \old(wrongPINCounter) + 1;

for the first time, JML expression not a JAVA expression

\old(E) means: \textit{E evaluated in the pre-state of enterPIN.}

\textit{E} can be any (arbitrarily complex) JAVA/JML expression.
 pública 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) {
...

two postconditions state that:
‘Given the above preconditions, enterPIN guarantees:
insertedCard == null and \old(insertedCard).invalid'
consider spec-case-1:

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

what does spec-case-1 *not* tell about post-state?
 Specification Cases Complete?

consider spec-case-1:

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

what does spec-case-1 not tell about post-state?

recall: fields of class ATM:

insertedCard
customerAuthenticated
wrongPINCounter
consider spec-case-1:

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

what does spec-case-1 *not* tell about post-state?

recall: fields of class ATM:

- insertedCard
- customerAuthenticated
- wrongPINCounter

what happens with insertCard and wrongPINCounter?
completing spec-case-1:

@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin == insertedCard.correctPIN;
@ ensures customerAuthenticated;
@ ensures insertedCard == \old(insertedCard);
@ ensures wrongPINCounter == \old(wrongPINCounter);
Completing Specification Cases

completing spec-case-2:

```plaintext
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter < 2;
@ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
@ ensures insertedCard == \old(insertedCard);
@ ensures customerAuthenticated
@   == \old(customerAuthenticated);
```
Completing Specification Cases

completing spec-case-3:

© public normal_behavior
© requires insertedCard != null;
© requires !customerAuthenticated;
© requires pin != insertedCard.correctPIN;
© requires wrongPINCounter >= 2;
© ensures insertedCard == null;
© ensures \old(insertedCard).invalid;
© ensures customerAuthenticated
©   == \old(customerAuthenticated);
© ensures wrongPINCounter == \old(wrongPINCounter);
unsatisfactory to add

@ ensures loc == \old(loc);

for all locations loc which do not change
Assignable Clause

unsatisfactory to add

@ ensures loc == \old(loc);

for all locations loc which *do not* change

instead:
add assignable clause for all locations which *may* change

@ assignable loc_1, ..., loc_n;
Assignable Clause

unsatisfactory to add

\[ @ \textbf{ensures} \ loc == \old(loc); \]

for all locations \textit{loc} which \textit{do not} change

instead:
add \textit{assignable clause} for all locations which \textit{may} change

\[ @ \textbf{assignable} \ loc_1, \ldots, loc_n; \]

Meaning: No location other than \textit{loc}_1, \ldots, \textit{loc}_n can be assigned to.
completing spec-case-1:

@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin == insertedCard.correctPIN;
@ ensures customerAuthenticated;
@ assignable customerAuthenticated;
completing spec-case-2:

@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter < 2;
@ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
@ assignable wrongPINCounter;
completing spec-case-3:

@ public normal_behavior
@ requires insertedCard != null;
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter >= 2;
@ ensures insertedCard == null;
@ ensures old(insertedCard).invalid;
@ assignable wrongPINCounter,
@ insertedCard,
@ insertedCard.invalid;
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 (enterPIN) example, pre/post-conditions made heavy use of class fields

But: public specifications can only talk about public fields.

Not desired: make all fields public.

/*@
@*/
in (enterPIN) example, pre/post-conditions made heavy use of class fields

But: public specifications can only talk about public fields.

Not desired: make all fields public.

one solution:
- keep the fields private/protected
- make those needed for specification `spec_public`
JML Modifiers: \texttt{spec\_public}

in (\texttt{enterPIN}) example, pre/post-conditions made heavy use of class fields

But: public specifications can only talk about public fields.

Not desired: make all fields public.

one solution:

\begin{itemize}
  \item keep the fields 	exttt{private}/protected
  \item make those needed for specification \texttt{spec\_public}
\end{itemize}

\begin{verbatim}
private /*@ spec_public @*/ BankCard insertedCard = null;
private /*@ spec_public @*/ int wrongPINCounter = 0;
private /*@ spec_public @*/ boolean customerAuthenticated = false;
\end{verbatim}
JML Modifiers: spec_public

in (enterPIN) example, pre/post-conditions made heavy use of class fields

But: public specifications can only talk about public fields.

Not desired: make all fields public.

one solution:

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

private /*@ spec_public @*/ BankCard insertedCard = null;
private /*@ spec_public @*/ int wrongPINCounter = 0;
private /*@ spec_public @*/ boolean customerAuthenticated = false;

(different solution: use specification-only fields)
in (enterPIN) example, pre/post-conditions made heavy use of class fields

But: public specifications can only talk about public fields.

Not desired: make all fields public.

One solution:
- keep the fields private/protected
- make those needed for specification spec_public

```java
class BankCard {
    private /*@ spec_public @*/ BankCard insertedCard = null;
    private /*@ spec_public @*/ int wrongPINCounter = 0;
    private /*@ spec_public @*/ boolean customerAuthenticated = false;
}
```

(Bug note:
in KeY 1.4, spec_public fields are only visible within their class)
It can be handy to use method calls in JML annotations. Examples:

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

allowed if, and only if method is guaranteed to have no side effects

In JML, you can specify methods to be ‘pure’:

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

The ‘pure’ modifier puts an additional obligation on the implementer (not to cause side effects), but allows to use the method in annotations.
boolean JML Expressions (to be completed)

- 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")
JML Expressions ≠ Java Expressions

boolean JML Expressions (to be completed)

- 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")
  - ...
  - ...
  - ...
  - ...
  - ...
How to express the following?

- an array `arr` only holds values \( \leq 2 \)
Beyond boolean Java expressions

How to express the following?

- an array arr only holds values $\leq 2$
- the variable $m$ holds the maximum entry of array $arr$
How to express the following?

- an array \( arr \) only holds values \( \leq 2 \)
- the variable \( m \) holds the maximum entry of array \( arr \)
- all Account objects in the array accountProxies are stored at the index corresponding to their respective accountNumber field
Beyond boolean Java expressions

How to express the following?

- an array arr only holds values $\leq 2$
- the variable m holds the maximum entry of array arr
- all Account objects in the array accountProxies are stored at the index corresponding to their respective accountNumber field
- all created instances of class BankCard have different cardNumbers
JML boolean expressions extend JAVA boolean expressions by:

- implication
- equivalence
First-order Logic in JML Expressions

JML boolean expressions extend JAVA boolean expressions by:

- implication
- equivalence
- quantification
boolean JML expressions are defined recursively:

- 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 \(b\)"")
  - \(a \rightarrow b\) ("\(a\) implies \(b\)"")
  - \(a \leftrightarrow b\) ("\(a\) is equivalent to \(b\)"")
  - \((\forall t \, x; \, a)\) ("for all \(x\) of type \(t\), \(a\) is true")
  - \((\exists t \, x; \, a)\) ("there exists \(x\) of type \(t\) such that \(a\)"
JML Expressions

Each side-effect free boolean JML 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 \(b\)"")
- \(a \mid\mid b\) ("\(a\) or \(b\)"")
- \(a \Rightarrow b\) ("\(a\) implies \(b\)"")
- \(a \Leftrightarrow b\) ("\(a\) is equivalent to \(b\)"")
- \((\forall t \ x; \ a)\) ("for all \(x\) of type \(t\), \(a\) 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 an \(x\) of type \(t\) fulfilling \(a\), such that \(b\)")
JML Quantifiers

in

(\forall t x; a; b)

(\exists t x; a; b)

a called “range predicate”
JML Quantifiers

in

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

a called “range predicate”

those forms are redundant:

(\forall t x; a; b)
equivalent to
(\forall t x; a \implies b)

(\exists t x; a; b)
equivalent to
(\exists t x; a \land b)
Pragmatics of Range Predicates

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

widely used

*pragmatics of range predicate:*

a used to restrict range of x further than t
Pragmatics of Range Predicates

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

widely used

*pragmatics of range predicate*:

\( a \) used to restrict range of \( x \) further than \( t \)

example: “\( arr \) is sorted at indexes between 0 and 9”:
Pragmatics of Range Predicates

(\forall t x; a; b) \text{ and } (\exists t x; a; b)

widely used

\textit{pragmatics of range predicate:}

a used to restrict range of x further than t

example: “arr is sorted at indexes between 0 and 9”:

(\forall \text{int } i,j;
Pragmatics of Range Predicates

(\forall t \ x; \ a; \ b) \text{ and } (\exists t \ x; \ a; \ b)

widely used

**pragmatics of range predicate:**

a used to restrict range of x further than t

example: “arr is sorted at indexes between 0 and 9”:

(\forall \text{ int } i,j; \ 0\leq i \ \&\& \ i<j \ \&\& \ j<10;
Pragmatics of Range Predicates

\( (\forall t \ x; a; b) \) and \( (\exists t \ x; a; b) \) widely used

pragmatics of range predicate:

a used to restrict range of \( x \) further than \( t \)

example: “arr is sorted at indexes between 0 and 9”:

\( (\forall \text{int} \ i,j; \ 0\leq i \ \&\& \ i<j \ \&\& \ j<10; \ \text{arr}[i] \leq \text{arr}[j]) \)
Using Quantified JML expressions

How to express:

- an array `arr` only holds values ≤ 2
Using Quantified JML expressions

How to express:

- an array \texttt{arr} only holds values $\leq 2$

$(\forall \texttt{int } i; \texttt{arr}[i] \leq 2)$
Using Quantified JML expressions

How to express:

- an array arr only holds values ≤ 2

\[
\text{forall int } i; \quad 0 \leq i \land i < \text{arr.length};
\]
Using Quantified JML expressions

How to express:

▶ an array arr only holds values \( \leq 2 \)

\( \forall \text{int } i; \ 0 \leq i < \text{arr.length}; \ \text{arr}[i] \leq 2 \)
Using Quantified JML expressions

How to express:

- the variable $m$ holds the maximum entry of array $arr$
Using Quantified JML expressions

How to express:

- the variable \( m \) holds the maximum entry of array \( arr \)

\[
(\forall \text{int } i; \ 0 \leq i \land i < \text{arr.length}; \ m \geq \text{arr}[i])
\]
How to express:

- the variable $m$ holds the maximum entry of array $arr$

$$\forall \text{int } i; 0\leq i \land i<\text{arr.length}; m \geq \text{arr}[i]$$

is this enough?
Using Quantified JML expressions

How to express:

- the variable \( m \) holds the maximum entry of array \( arr \)

\[
(\forall \text{int } i; \ 0 \leq i \land i < \text{arr.length}; \ m \geq \text{arr}[i])
\]

\[
(\exists \text{int } i; \ 0 \leq i \land i < \text{arr.length}; \ m = \text{arr}[i])
\]
Using Quantified JML expressions

How to express:

- the variable $m$ holds the maximum entry of array $arr$

$$\forall \text{int } i; 0 \leq i \&\& i < arr.length; m \geq arr[i]$$

$arr.length > 0 \implies$

$$\exists \text{int } i; 0 \leq i \&\& i < arr.length; m = arr[i]$$
Using Quantified JML expressions

How to express:

- all Account objects in the array accountProxies are stored at the index corresponding to their respective accountNumber field
How to express:

- all Account objects in the array accountProxies are stored at the index corresponding to their respective accountNumber field

\[
\forall \text{int } i; \ 0 \leq i \land i < \text{maxAccountNumber}; \\
\text{accountProxies}[i].\text{accountNumber} == i
\]
Using Quantified JML expressions

How to express:

- all created instances of class BankCard have different cardNumbers
Using Quantified JML expressions

How to express:

- all created instances of class BankCard have different cardNumbers

(\forall BankCard p1, p2; 
created(p1) && created(p2); 
p1 != p2 ==> p1.cardNumber != p2.cardNumber)
Using Quantified JML expressions

How to express:

- all created instances of class BankCard have different cardNumbers

(\forall \text{BankCard } p1, p2; \\
\text{created}(p1) \land \text{created}(p2); \\
p1 \neq p2 \implies p1.cardNumber \neq p2.cardNumber)

note:
- JML quantifiers range also over non-created objects
Using Quantified JML expressions

How to express:

- all created instances of class BankCard have different cardNumbers

\( (\forall \text{BankCard } p1, p2; \ \
\text{created}(p1) \land \text{created}(p2); \ \ p1 \neq p2 \implies p1.\text{cardNumber} \neq p2.\text{cardNumber}) \)

note:

- JML quantifiers range also over non-created objects
- same for quantifiers in KeY!
Using Quantified JML expressions

How to express:

- all created instances of class BankCard have different cardNumbers

(\forall \text{BankCard } p_1, p_2; \\
\text{created}(p_1) \land \text{created}(p_2); \\
p_1 \neq p_2 \implies p_1\text{.cardNumber} \neq p_2\text{.cardNumber})

note:

- JML quantifiers range also over non-created objects
- same for quantifiers in KeY!
- in JML, restrict to created objects with \text{created}
Using Quantified JML expressions

How to express:

- all created instances of class BankCard have different cardNumbers

\( (\forall \text{BankCard } p1, p2; \\\ \\ created(p1) && created(p2);\\ \\ p1 \neq p2 \Rightarrow p1.\text{cardNumber} \neq p2.\text{cardNumber}) \)

note:

- JML quantifiers range also over non-created objects
- same for quantifiers in KeY!
- in JML, restrict to created objects with \(\text{\_created}\)
- in KeY?
Using Quantified JML expressions

How to express:

- all created instances of class BankCard have different cardNumbers

(\forall \text{BankCard } p1, p2;
  \text{created}(p1) \land \text{created}(p2);
  p1 \neq p2 \implies p1.cardNumber \neq p2.cardNumber)

Note:

- JML quantifiers range also over non-created objects
- same for quantifiers in KeY!
- in JML, restrict to created objects with \text{created}
- in KeY? (⇒ coming lecture)
Example: Specifying LimitedIntegerSet

```java
public class LimitedIntegerSet {
    public final int limit;
    private int arr[];
    private int size = 0;

    public LimitedIntegerSet(int limit) {
        this.limit = limit;
        this.arr = new int[limit];
    }

    public boolean add(int elem) { /*...*/ }

    public void remove(int elem) { /*...*/ }

    public boolean contains(int elem) { /*...*/ }

    // other methods
}
```
public class LimitedIntegerSet {
    public final int limit;
    private /*@ spec_public @*/ int arr[];
    private /*@ spec_public @*/ int size = 0;

    public LimitedIntegerSet(int limit) {
        this.limit = limit;
        this.arr = new int[limit];
    }
    public boolean add(int elem) { /*...*/ }

    public void remove(int elem) { /*...*/ }

    public /*@ pure @*/ boolean contains(int elem) { /*...*/ }

    // other methods
}
Specifying contains()

```java
public /*@ pure @*/ boolean contains(int elem) {/*...*/}
```
Specifying `contains()`

```java
public /*@ pure @*/ boolean contains(int elem) { /*...*/ }
```

has no effect on state
public /*@ pure @*/ boolean contains(int elem) {/*/...*/}

has no effect on state

how to specify result value?
In postconditions, one can use ‘\result’ to refer to the return value of the method.

/*@ public normal_behavior
  @ ensures \result ==
*/
In postconditions, one can use ‘\result’ to refer to the return value of the method.

```java
/*@ public normal_behavior
  @ ensures \result == (\exists int i; @
  @
```
In postconditions, one can use ‘\result’ to refer to the return value of the method.

```java
/*@ public normal_behavior
  @ ensures \result == (\exists int i;
  @          0 <= i && i < size;
  @
```
Result Values in Postcondition

In postconditions, one can use ‘\result’ to refer to the return value of the method.

```java
/**
 * public normal_behavior
 *  @ ensures \result == (\exists i; 0 <= i && i < size; arr[i] == elem);
 */

public /*@ pure @*/ boolean contains(int elem) { /*...*/ }
```
Specifying add()  (spec-case1)

/*@ public normal_behavior
   @ requires size < limit && !contains(elem);
   @ ensures \result == true;
   @ ensures contains(elem);
   @ ensures (\forall int e;
     @ e != elem;
     @ contains(e) <=> \old(contains(e)));
   @ ensures size == \old(size) + 1;
   @
   @ also
   @
   @ <spec-case2>
   @*/

public boolean add(int elem) {;/*...*/}
/** @ public normal_behavior
   @
   @ <spec-case1>
   @
   @ also
   @
   @ public normal_behavior
   @ requires (size == limit) || contains(elem);
   @ ensures \result == false;
   @ ensures (\forall int e;
     @ contains(e) <==> \old(contains(e)));
   @ ensures size == \old(size);
   @*/

public boolean add(int elem) {/**<...*/}
Specifying `remove()`

```java
/*@ public normal_behavior
    @ ensures !contains(elem);
    @ ensures (\forall int e;
              @ e != elem;
              @ contains(e) <=> \old(contains(e)));
    @ ensures \old(contains(elem))
    @      ==> size == \old(size) - 1;
    @ ensures !\old(contains(elem))
    @      ==> size == \old(size);
    @*/

public void remove(int elem) {/*@...*/}
```
Specifying Data Constraints

So far:
JML used to specify method specifics.
Specifying Data Constraints

So far:
JML used to specify method specifics.

How to specify constraints on class data?
Specifying Data Constraints

So far:
JML used to specify method specifics.

How to specify constraints on class data, e.g.:
- consistency of redundant data representations (like indexing)
- restrictions for efficiency (like sortedness)
Specifying Data Constraints

So far:
JML used to specify method specifics.

How to specify constraints on class data, e.g.:
- consistency of redundant data representations (like indexing)
- restrictions for efficiency (like sortedness)

Data constraints are global:
all methods must preserve them
Consider **LimitedSortedIntegerSet**

```java
public class LimitedSortedIntegerSet {
    public final int limit;
    private int arr[];
    private int size = 0;

    public LimitedSortedIntegerSet(int limit) {
        this.limit = limit;
        this.arr = new int[limit];
    }

    public boolean add(int elem) { /*...*/ }

    public void remove(int elem) { /*...*/ }

    public boolean contains(int elem) { /*...*/ }

    // other methods
}
```
Consequence of Sortedness for Implementations

**method contains**

- can employ binary search (logarithmic complexity)
method contains

- can employ binary search (logarithmic complexity)
- why is that sufficient?
Consequence of Sortedness for Implementations

**method contains**

- can employ binary search (logarithmic complexity)
- why is that sufficient?
- it assumes sortedness in pre-state
Consequence of Sortedness for Implementations

**method contains**
- can employ binary search (logarithmic complexity)
- why is that sufficient?
- it assumes sortedness in pre-state

**method add**
- searches first index with bigger element, inserts just before that
Consequence of Sortedness for Implementations

**method contains**
- can employ binary search (logarithmic complexity)
- why is that sufficient?
- it assumes sortedness in pre-state

**method add**
- searches first index with bigger element, inserts just before that
- thereby tries to establish sortedness in post-state
Consequence of Sortedness for Implementations

**method contains**
- can employ binary search (logarithmic complexity)
- why is that sufficient?
- it **assumes sortedness** in pre-state

**method add**
- searches first index with bigger element, inserts just before that
- thereby tries to **establish sortedness** in post-state
- why is that sufficient?
Consequence of Sortedness for Implementations

**method contains**
- can employ binary search (logarithmic complexity)
- why is that sufficient?
- it assumes sortedness in pre-state

**method add**
- searches first index with bigger element, inserts just before that
- thereby tries to establish sortedness in post-state
- why is that sufficient?
- it assumes sortedness in pre-state
Consequence of Sortedness for Implementations

**method contains**
- can employ binary search (logarithmic complexity)
- why is that sufficient?
- it **assumes sortedness** in pre-state

**method add**
- searches first index with bigger element, inserts just before that
- thereby tries to **establish sortedness** in post-state
- why is that sufficient?
- it **assumes sortedness** in pre-state

**method remove**
- (accordingly)
Specifying Sortedness with JML

recall class fields:

```java
public final int limit;
private int arr[];
private int size = 0;
```

sortedness as JML expression:

\[
\forall \text{int } i; 0 < i && i < \text{size}; \quad \text{arr}[i-1] \leq \text{arr}[i]
\]
recall class fields:

```java
public final int limit;
private int arr[];
private int size = 0;
```

sortedness as JML expression:

```jml
(\forall int i; 0 < i && i < size;
   arr[i-1] <= arr[i])
```
Specifying Sortedness with JML

recall class fields:

```java
public final int limit;
private int arr[];
private int size = 0;
```

sortedness as JML expression:

```jml
(\forall int i; 0 < i && i < size; 
arr[i-1] <= arr[i])
```

(what’s the value of this if size < 2?)
recall class fields:

```java
public final int limit;
private int arr[];
private int size = 0;
```

sortedness as JML expression:

```
(\forall int i; 0 < i && i < size;
   arr[i-1] <= arr[i])
```

(what’s the value of this if size < 2?)

but where in the specification does the red expression go?
can assume sortedness of pre-state
Specifying Sorted contains()

can assume sortedness of pre-state

/*@ public normal_behavior  
  @ requires (\forall int \ i; 0 < \ i && \ i < size;  
  @ \ arr[i-1] <= arr[i]);  
  @ ensures \ result == (\exists int \ i;  
  @ 0 <= \ i && \ i < size;  
  @ arr[i] == elem);  
  @*/

class /*@ public pure */ boolean contains(int elem) {/*@...*/}
Specifying Sorted contains()

can assume sortedness of pre-state

/*@ public normal_behavior
  @ requires (∀ int i; 0 < i && i < size;
  @             arr[i-1] <= arr[i]);
  @ ensures \result == (∃ int i;
  @             0 <= i && i < size;
  @             arr[i] == elem);
  @*/
public /*@ pure @*/ boolean contains(int elem) {/*...*/}

contains() is pure
⇒ sortedness of post-state trivially ensured
Specifying Sorted `remove()`

can assume sortedness of pre-state
must ensure sortedness of post-state

```java
/*@ public normal_behavior
   @ requires (\forall int i; 0 < i && i < size;
               arr[i-1] <= arr[i]);
   @ ensures !contains(elem);
   @ ensures (\forall int e;
               e != elem;
               contains(e) <=> \old(contains(e)));
   @ ensures \old(contains(elem))
   @  ==> size == \old(size) - 1;
   @ ensures !\old(contains(elem))
   @  ==> size == \old(size);
   @ ensures (\forall int i; 0 < i && i < size;
               arr[i-1] <= arr[i]);
   @*/

public void remove(int elem) { /*...*/}
```
Specifying **Sorted** add()  (spec-case1)

```java
/*@ public normal_behavior
  @ requires (∀ int i; 0 < i && i < size;
  @       arr[i-1] <= arr[i]);
  @ requires size < limit && !contains(elem);
  @ ensures \result == true;
  @ ensures contains(elem);
  @ ensures (∀ int e;
  @       e != elem;
  @       contains(e) <=> \old(contains(e)));
  @ ensures size == \old(size) + 1;
  @ ensures (∀ int i; 0 < i && i < size;
  @       arr[i-1] <= arr[i]);

  @ also <spec-case2>
  @*/

public boolean add(int elem) {/*@...*/}
```
Specifying **Sorted add()** (spec-case2)

/*@
public normal_behavior
@
@ <spec-case1> also
@

@ public normal_behavior
@ requires (∀ int i; 0 < i && i < size;
@    arr[i-1] <= arr[i]);
@ requires (size == limit) || contains(elem);
@ ensures result == false;
@ ensures (∀ int e;
@    contains(e) <=> \old(contains(e)));
@ ensures size == \old(size);
@ ensures (∀ int i; 0 < i && i < size;
@    arr[i-1] <= arr[i]);
@*/

public boolean add(int elem) { /*...*/}
Factor out Sortedness

so far: ‘sortedness’ has swamped our specification
Factor out Sortedness

so far: ‘sortedness’ has swamped our specification

we can do better, using

JML Class Invariant

construct for specifying data constraints centrally
Factor out Sortedness

so far: ‘sortedness’ has swamped our specification

we can do better, using

JML Class Invariant

construct for specifying data constraints centrally

1. delete blue and red parts from previous slides
2. add ‘sortedness’ as JML class invariant instead
public class LimitedSortedIntegerSet {

    public final int limit;

    /*@ public invariant \(\forall \text{int } i;\)
        @ 0 < i && i < size;
        @ arr[i-1] <= arr[i];
    */

    private /*@ spec_public */ int arr[];
    private /*@ spec_public */ int size = 0;

    // constructor and methods,
    // without sortedness in pre/post-conditions
}
JML Class Invariant

- JML class invariant can be places anywhere in class
- (contrast: method contract must be in front of its method)
- custom to place class invariant in front of fields it talks about
instance invariants

can refer to instance fields of this object

(unqualified, like ‘size’, or qualified with ‘self’, like ‘self.size’)

JML syntax: instance invariant
**Instance vs. Static Invariants**

**instance invariants**
can refer to instance fields of this object
  (unqualified, like ‘size’, or qualified with ‘self’, like ‘self.size’)
JML syntax: **instance invariant**

**static invariants**
cannot refer to instance fields of this object
JML syntax: **static invariant**
Instance vs. Static Invariants

**instance invariants**
can refer to instance fields of this object
  (unqualified, like ‘size’, or qualified with ‘self’, like ‘self.size’)
JML syntax: `instance invariant`

**static invariants**
cannot refer to instance fields of this object
JML syntax: `static invariant`

**both**
can refer to
  – static fields
  – instance fields via explicit reference, like ‘o.size’
**Instance vs. Static Invariants**

**instance invariants**
can refer to instance fields of this object
  (unqualified, like ‘size’, or qualified with ‘self’, like ‘self.size’)
JML syntax: `instance invariant`

**static invariants**
cannot refer to instance fields of this object
JML syntax: `static invariant`

**both**
can refer to
  – static fields
  – instance fields via explicit reference, like ‘o.size’

**instance is default**
if `instance` or `static` is omitted \(\Rightarrow\) instance invariant!
public class BankCard {

    /*@ public static invariant */
    @ (\forall BankCard p1, p2;
    @    \created(p1) && \created(p2);
    @    p1!=p2 ==> p1.cardNumber!=p2.cardNumber)
    @*/

    private /*@ spec_public */ int cardNumber;

    // rest of class follows

}
Recall Specification of enterPIN()

```java
private /*@ spec_public @*/ BankCard insertedCard = null;
private /*@ spec_public @*/ int wrongPINCounter = 0;
private /*@ spec_public @*/ boolean customerAuthenticated = false;

/*@ <spec-case1> also <spec-case2> also <spec-case3> @*/
public void enterPIN (int pin) { ... }
```
Recall Specification of enterPIN()

private /*@ spec_public @*/ BankCard insertedCard = null;
private /*@ spec_public @*/ int wrongPINcounter = 0;
private /*@ spec_public @*/ boolean customerAuthenticated = false;

/*@ <spec-case1> also <spec-case2> also <spec-case3> @*/
public void enterPIN (int pin) { ...}

last lecture:
all 3 spec-cases were normal_behavior
**normal_behavior** specification case, with preconditions $P$, forbids method to throw exceptions if pre-state satisfies $P$
**normal_behavior** specification case, with preconditions $P$, forbids method to throw exceptions if pre-state satisfies $P$

**exceptional_behavior** specification case, with preconditions $P$, requires method to throw exceptions if pre-state satisfies $P$
Specifying Exceptional Behavior of Methods

**normal_behavior** specification case, with preconditions $P$, forbids method to throw exceptions if pre-state satisfies $P$

**exceptional_behavior** specification case, with preconditions $P$, requires method to throw exceptions if pre-state satisfies $P$

Keyword **signals** specifies *post-state*, depending on thrown exception
normal_behavior specification case, with preconditions $P$, forbids method to throw exceptions if pre-state satisfies $P$

exceptional_behavior specification case, with preconditions $P$, requires method to throw exceptions if pre-state satisfies $P$

keyword signals specifies post-state, depending on thrown exception

keyword signals_only limits types of thrown exception
Completing Specification of enterPIN()

/*@ <spec-case1> also <spec-case2> also <spec-case3> also @
    @ public exceptional_behavior
    @ requires insertedCard==null;
    @ signals_only ATMException;
    @ signals (ATMException) !customerAuthenticated;
    @*/

public void enterPIN (int pin) { ...
Completing Specification of enterPIN()

```java
/*@ <spec-case1> also <spec-case2> also <spec-case3> also
   @
   @ public exceptional_behavior
   @ requires insertedCard==null;
   @ signals_only ATMException;
   @ signals (ATMException) !customerAuthenticated;
   @*/
public void enterPIN (int pin) { ...
```

in case `insertedCard==null` in pre-state

- an exception *must* be thrown (‘exceptional_behavior’)
- it can only be an ATMException (‘signals_only’)
- method must then ensure `!customerAuthenticated` in post-state (‘signals’)

Formal Specification and Verification: Java Modeling Language

B. Beckert 119 / 1
an exceptional specification case can have one clause of the form

```
signals_only (E1, ..., En);
```

where $E_1, \ldots, E_n$ are exception types
an exceptional specification case can have one clause of the form

```
signals_only (E1, ..., En);
```

where $E_1, ..., E_n$ are exception types

Meaning:

if an exception is thrown, it is of type $E_1$ or $\ldots$ or $E_n$
an exceptional specification case can have several clauses of the form

```plaintext
  signals (E) b;
```

where E is exception type, b is boolean expression
an exceptional specification case can have several clauses of the form

\[
\text{signals (E) } b;
\]

where E is exception type, b is boolean expression

Meaning:

if an exception of type E is thrown, b holds in post condition
Allowing Non-Termination

by default, both:

- normal_behavior
- exceptional_behavior

specification cases enforce termination
Allowing Non-Termination

*by default*, both:

- **normal_behavior**
- **exceptional_behavior**

specification cases enforce termination

in each specification case, termination can be permitted via the clause

```
diverges true;
```
Allowing Non-Termination

by default, both:

▶ normal_behavior
▶ exceptional_behavior

specification cases enforce termination

in each specification case, termination can be permitted via the clause

\texttt{diverges true;}

Meaning:

given the precondition of the specification case holds in pre-state, the method may or \textit{may not} terminate
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;

implicit invariant
‘public invariant name != null;’
added to class

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

implicit precondition
‘requires card != null;’
added to each specification case of insertCard

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

implicit postcondition
‘ensures \result != null;’
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;

implicit invariant
‘public invariant name != null;’

added to class

public void insertCard(BankCard card) { ..

implicit precondition
‘requires card != null;’

added to each specification case of insertCard

public String toString()

implicit postcondition
‘ensures \result != null;’

added to each specification case of toString
To prevent such pre/post-conditions and invariants: ‘`nullable’’

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

    In JML this means:
public class LinkedList {
    private Object elem;
    private LinkedList next;
    ....

In JML this means:
- all elements in the list are non_null
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:

```java
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.

⇒ Not yet well reflected in literature and tools.
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.

in addition, the subclass may add further specification cases, using *also*:

```java
/*@ also
  @<subclass-specific-spec-cases>
/*@ *
```

```java
public void method () { ... }
```
Many tools support JML (see www.eecs.ucf.edu/~leavens/JML/). Most basic tool set:

- **jml**, a syntax and type checker
- **jm1c**, JML/Java compiler. Compile runtime assertion checks into the code.
- **jmldoc**, like javadoc for Java + JML
- **jmlunit**, unit testing based on JML

For the lab, we do not require using the tools, but we recommend to use **jml** to check the syntax.
Literature for this Lecture

essential reading:


further reading, all available at www.eecs.ucf.edu/~leavens/JML/documentation.shtml:


JML Tutorial  Gary T. Leavens, Yoonsik Cheon. Design by Contract with JML

JML Overview  Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design