

# Software Engineering using Formal Methods

## Java Modeling Language

Wolfgang Ahrendt

29 September 2015

# Role of JML in the Course

programming/modelling language	property/specification language	verification technique
PROMELA	LTL	model checking
JAVA	JML	deductive verification

# Unit Specifications

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

# Unit 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:

# Unit 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:

- ▶ application level – application level
- ▶ application level – library level
- ▶ library level – library level

# Unit Specifications

**In the object-oriented setting:**

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

We start with **method** specifications.

# Unit Specifications

**In the object-oriented setting:**

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

We start with **method** specifications.

Method specifications *potentially* refer to:

# Unit Specifications

**In the object-oriented setting:**

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

We start with **method** specifications.

Method specifications *potentially* refer to:

- ▶ initial values of formal parameters,



# Unit Specifications

**In the object-oriented setting:**

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

We start with **method** specifications.

Method specifications *potentially* refer to:

- ▶ initial values of formal parameters,
- ▶ result value,

# Unit Specifications

**In the object-oriented setting:**

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

We start with **method** specifications.

Method specifications *potentially* refer to:

- ▶ initial values of formal parameters,
- ▶ result value,
- ▶ pre-state and post-state

# Unit Specifications

**In the object-oriented setting:**

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

We start with **method** specifications.

Method specifications *potentially* refer to:

- ▶ initial values of formal parameters,
- ▶ result value,
- ▶ visible 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 (Meyer, 1992, Eiffel)

# 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 (Meyer, 1992, Eiffel)

Contract between *caller* and *callee* (called method)

*callee guarantees certain outcome provided caller guarantees prerequisites*

## Running Example: ATM.java

```
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 () { ... }  
  
}
```

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 as authenticated.*

# Getting More Precise: Specification as Contract

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



# 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

# 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

# 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

# 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` has been increased by 1  
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` has been increased by 1  
user is not authenticated

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

# 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` has been 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. In case of termination, it may be normal or abrupt.



# 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. In case of termination, it may be normal or abrupt.

non-termination and abrupt termination  $\Rightarrow$  next lecture

# Formal Specification

Natural language specs are very important and widely used

# Formal Specification

Natural language specs are very important and widely used, we focus on

## Formal Specification

Describe contracts with mathematical rigour

# Formal Specification

Natural language specs are very important and widely used, we focus on

## Formal Specification

Describe contracts with mathematical rigour

## Motivation

- ▶ High degree of precision
  - ▶ formalization often exhibits omissions/inconsistencies
  - ▶ avoid ambiguities inherent to natural language
- ▶ Potential for **automation** of program analysis
  - ▶ monitoring
  - ▶ test case generation
  - ▶ **program verification**

# 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

# 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

JAVA

JML  
is

# 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

# 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 Annotations

JML **extends** JAVA by **annotations**.

## JML annotations include:

- ✓ preconditions
- ✓ postconditions
- ✓ class invariants
- ✓ additional modifiers
- ✗ 'specification-only' fields
- ✗ 'specification-only' methods
- ✓ loop invariants
- ✓ ...
- ✗ ...

✓: 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

Ensures compatibility with standard JAVA compiler:

JML annotations live in special JAVA comments,  
ignored by JAVA compiler, recognized by JML tools

# JML by Example

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

from the file ATM.java

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

```
⋮
```

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

# JML by Example

```
/*@ 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 ( ...
```

But:

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

# 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 tools.

JML annotations appear in JAVA comments starting with @.



# 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 tools.

JML annotations appear in JAVA comments starting with @.

How about “//” comments?

# JML by Example

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

*equivalent to:*

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

# JML by Example

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

*equivalent to:*

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

The easiest way to [comment out JML](#):

# JML by Example

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

*equivalent to:*

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

The easiest way to [comment out JML](#):

```
/*_@ public normal_behavior ... */
```

# JML by Example

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

*equivalent to:*

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

The easiest way to [comment out JML](#):

```
/*_@ public normal_behavior ... */
//_@ public normal_behavior
//_@ requires !customerAuthenticated;
...

```

# JML by Example

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

What about the intermediate '@'s?

# JML by Example

```
/*@ 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 '\*/'.

# JML by Example

```
/*@ 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.



# 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

# 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.

# JML by Example

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

Each keyword ending with **behavior** opens a 'specification case'.

## normal\_behavior Specification Case

The method guarantees to *not* throw any exception

# JML by Example

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

Each keyword ending with **behavior** opens a 'specification case'.

## **normal\_behavior** Specification Case

The method guarantees to *not* throw any exception (on the top level),

# JML by Example

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

Each keyword ending with **behavior** opens a 'specification case'.

## **normal\_behavior** Specification Case

The method guarantees to *not* throw any exception (on the top level), *if the caller guarantees all preconditions of this specification case.*

# JML by Example

```
/*@ 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

# JML by Example

```
/*@ 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*

# JML by Example

```
/*@ 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*

in general:

preconditions are *boolean JML expressions* (see below)



# JML by Example

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

specifies only the case where **both** preconditions are true in pre-state  
the above is equivalent to:

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

# JML by Example

```
/*@ 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`

# JML by Example

```
/*@ 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*

# JML by Example

```
/*@ 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)

# JML by Example

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 ( ...
```

# JML by Example

```
/*@ <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) { ...
```

for the first time, JML expression not a JAVA expression

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

*E* can be any (arbitrarily complex) JML expression.

# JML by Example

```
/*@ <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) { ...
```

two postconditions state that:

'Given the above preconditions, enterPIN guarantees:

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

## Question:

could it be

```
@ ensures \old(insertedCard.invalid);
```

instead of

```
@ ensures \old(insertedCard).invalid;
```

??



# 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?

# 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
```

# 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
```

what happens with `insertCard` and `wrongPINCounter`?

# Completing Specification Cases

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:

```
@ 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);
```

# Assignable Clause

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 loc1, ..., locn;
```



# 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 loc1, ..., locn;
```

Meaning: **No location other than  $loc_1, \dots, loc_n$  can be assigned to.**

# 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 loc1, ..., locn;
```

Meaning: **No location other than  $loc_1, \dots, loc_n$  can be assigned to.**

Special cases:

**No** location may be changed:

```
@ assignable \nothing;
```

# 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 loc1, ..., locn;
```

Meaning: **No location other than  $loc_1, \dots, loc_n$  can be assigned to.**

Special cases:

**No** location may be changed:

```
@ assignable \nothing;
```

**Unrestricted**, method allowed to change anything:

```
@ assignable \everything;
```

# Specification Cases with Assignable

completing spec-case-1:

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

# Specification Cases with Assignable

completing spec-case-2:

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

# Specification Cases with Assignable

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 insertedCard,
@           insertedCard.invalid,
```

# Assignable Groups

You can specify groups of locations as assignable, using '\*'.

example:

```
@ assignable o.*, a[*];
```

makes all fields of object o and all positions of array a assignable.

JML extends the JAVA modifiers by additional modifiers

The most important ones are:

- ▶ **spec\_public**
- ▶ **pure**
- ▶ **nullable** (next lecture)
- ▶ **non\_null** (next lecture)
- ▶ **helper** (next lecture)



## JML Modifiers: `spec_public`

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

**But:** `public` specifications can access only `public` fields

**Not desired:** make all fields mentioned in specification `public`

## JML Modifiers: `spec_public`

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

**But:** `public` specifications can access only `public` fields

**Not desired:** make all fields mentioned in specification `public`

### Control visibility with `spec_public`

- ▶ Keep visibility of JAVA fields `private/protected`
- ▶ If necessary make them visible in specification only by `spec_public`

## JML Modifiers: `spec_public`

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

**But:** `public` specifications can access only `public` fields

**Not desired:** make all fields mentioned in specification `public`

### Control visibility with `spec_public`

- ▶ Keep visibility of JAVA fields `private/protected`
- ▶ If necessary make them visible in specification only by `spec_public`

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

## JML Modifiers: `spec_public`

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

**But:** `public` specifications can access only `public` fields

**Not desired:** make all fields mentioned in specification `public`

### Control visibility with `spec_public`

- ▶ Keep visibility of JAVA fields `private/protected`
- ▶ If necessary make them visible in specification only by `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; not covered in this course)

# JML Modifiers: Purity

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

Examples:

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

But: specifications may not themselves change the state!

# JML Modifiers: Purity

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

Examples:

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

But: specifications may not themselves change the state!

## Definition ((Strictly) Pure method)

A method is **pure** iff it has no visible side effects on existing objects and always terminates.

A method is **strictly pure** if it is pure and does not create new objects.

# JML Modifiers: Purity

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

Examples:

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

But: specifications may not themselves change the state!

## Definition ((Strictly) Pure method)

A method is **pure** iff it has no visible side effects on existing objects and always terminates.

A method is **strictly pure** if it is pure and does not create new objects.

JML expressions may call (strictly) pure methods.

# JML Modifiers: Purity

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

Examples:

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

But: specifications may not themselves change the state!

## Definition ((Strictly) Pure method)

A method is **pure** iff it has no visible side effects on existing objects and always terminates.

A method is **strictly pure** if it is pure and does not create new objects.

JML expressions may call (strictly) pure methods.

Pure methods are annotated by **pure** or **strictly\_pure** resp.

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



## JML Modifiers: Purity Cont'd

- ▶ **pure** puts obligation on implementor not to cause side effects
- ▶ It is possible to **formally verify** that a method is pure
- ▶ **pure** implies **assignable \nothing;**  
(may create new objects)
- ▶ **assignable \strictly\_nothing;**  
expresses that no new objects are created
- ▶ Assignable clauses are local to a specification case
- ▶ **pure** is global to the method

# JML Expressions $\neq$ 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”)

# JML Expressions $\neq$ 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”)
  - ▶ ...
  - ▶ ...
  - ▶ ...
  - ▶ ...

# Beyond boolean JAVA expressions

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`

# 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 `allAccounts` 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 `allAccounts` are stored at the index corresponding to their respective `accountNumber` field
- ▶ all instances of class `BankCard` have different `cardNumbers`

# First-order Logic in JML Expressions

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

**boolean** JML expressions are defined recursively:

## boolean JML Expressions

- ▶ 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  $b$ ”)
  - ▶  $(\backslash\text{forall } t \ x; \ a)$  (“for all  $x$  of type  $t$ ,  $a$  holds”)
  - ▶  $(\backslash\text{exists } t \ x; \ a)$  (“there exists  $x$  of type  $t$  such that  $a$ ”)

# boolean JML Expressions

**boolean** JML expressions are defined recursively:

## boolean JML Expressions

- ▶ 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  $b$ ”)
  - ▶  $(\backslash\text{forall } t \ x; \ a)$  (“for all  $x$  of type  $t$ ,  $a$  holds”)
  - ▶  $(\backslash\text{exists } t \ x; \ a)$  (“there exists  $x$  of type  $t$  such that  $a$ ”)
  - ▶  $(\backslash\text{forall } t \ x; \ a; \ b)$  (“for all  $x$  of type  $t$  **fulfilling**  $a$ ,  $b$  holds”)
  - ▶  $(\backslash\text{exists } t \ x; \ a; \ b)$  (“there exists an  $x$  of type  $t$  **fulfilling**  $a$ , such that  $b$ ”)

# JML Quantifiers

in

```
(\forallall t x; a; b)
```

```
(\existsexists t x; a; b)
```

**a** is called “range predicate”

# JML Quantifiers

in

```
(\forallall t x; a; b)
```

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

**a** is called “range predicate”

those forms are redundant:

```
(\forallall t x; a; b)
```

equivalent to

```
(\forallall t x; a ==> b)
```

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

equivalent to

```
(\exists t x; a && b)
```

# Pragmatics of Range Predicates

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

widely used

*pragmatics of range predicate:*

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

widely used

*pragmatics of range predicate:*

`a` is used to restrict range of `x` further than `t`

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

```
(\forall int i,j;
```



# Pragmatics of Range Predicates

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

widely used

*pragmatics of range predicate:*

`a` is used to restrict range of `x` further than `t`

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

```
(\forall int i,j; 0<=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` is used to restrict range of `x` further than `t`

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

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

# Using Quantified JML expressions

How to express:

- ▶ an array `arr` only holds values  $\leq 2$

# Using Quantified JML expressions

How to express:

- ▶ an array `arr` only holds values  $\leq 2$

```
(\forall int i;
```

# Using Quantified JML expressions

How to express:

- ▶ an array `arr` only holds values  $\leq 2$

```
(\forall int i; 0 <= i && i < arr.length;
```

# Using Quantified JML expressions

How to express:

- ▶ an array `arr` only holds values  $\leq 2$

```
(\forall int i; 0 <= i && i < arr.length; arr[i] <= 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 int i; 0 <= i && i < arr.length; m >= arr[i])
```



# Using Quantified JML expressions

How to express:

- ▶ the variable `m` holds the maximum entry of array `arr`

```
(\forall int i; 0 <= i && i < arr.length; m >= arr[i])
```

is this enough?

# Using Quantified JML expressions

How to express:

- ▶ the variable `m` holds the maximum entry of array `arr`

```
(\forall int i; 0 <= i && i < arr.length; m >= arr[i])
```

```
(\exists int i; 0 <= i && i < arr.length; m == arr[i])
```

# Using Quantified JML expressions

How to express:

- ▶ the variable `m` holds the maximum entry of array `arr`

```
(\forall int i; 0 <= i && i < arr.length; m >= arr[i])
```

```
arr.length > 0 ==>
```

```
(\exists int i; 0 <= 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

# 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

```
(\forall int i; 0 <= i && i < maxAccountNumber;  
    accountProxies[i].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;  
    p1 != p2 ==> p1.cardNumber != p2.cardNumber)
```

# Generalized Quantifiers

JML offers also **generalized 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 these expressions are true):

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
```



# Generalized Quantifiers

JML offers also **generalized 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 these expressions are true):

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
```

```
(\product int i; 0 < i && i < 5; i+2) == 3 * 4 * 5 * 6
```

# Generalized Quantifiers

JML offers also **generalized 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 these expressions are true):

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
```

```
(\product int i; 0 < i && i < 5; i+2) == 3 * 4 * 5 * 6
```

```
(\max int i; 0 <= i && i < 5; i) == 4
```

# Generalized Quantifiers

JML offers also **generalized 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 these expressions are true):

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
```

```
(\product int i; 0 < i && i < 5; i+2) == 3 * 4 * 5 * 6
```

```
(\max int i; 0 <= i && i < 5; i) == 4
```

```
(\min int i; 0 <= i && i < 5; i-1) == -1
```

## Example: Specifying LimitedIntegerSet

```
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
}
```

## Prerequisites: Adding Specification Modifiers

```
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()

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

## Specifying contains()

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

has no effect on the state, incl. no exceptions

## Specifying contains()

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

has no effect on the state, incl. no exceptions

how to specify result value?



# Result Values in Postcondition

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

```
/*@ public normal_behavior  
   @ ensures \result ==
```

# Result Values in Postcondition

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

```
/*@ public normal_behavior  
  @ ensures \result == (\exists int i;  
  @
```

# Result Values in Postcondition

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

```
/*@ 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**.

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

## Specifying add() (spec-case1) – new element can be added

```
/*@ 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) {/*...*/}
```

## Specifying add() (spec-case2) – new element cannot be added

```
/*@ 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()

```
/*@ 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) {/*...*/}
```

# Literature for this Lecture

*essential reading:*

**New JML Tutorial** M. Huisman, W. Ahrendt, D. Grahl, M. Hentschel:  
Formal Specification with the Java Modeling Language.  
Chapter in the new KeY book, to appear  
(see “JML” on literature/tools page)

*further reading*, all available at

<http://www.eecs.ucf.edu/~leavens/JML/documentation.shtml>:

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

*JML Reference Manual*

**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*