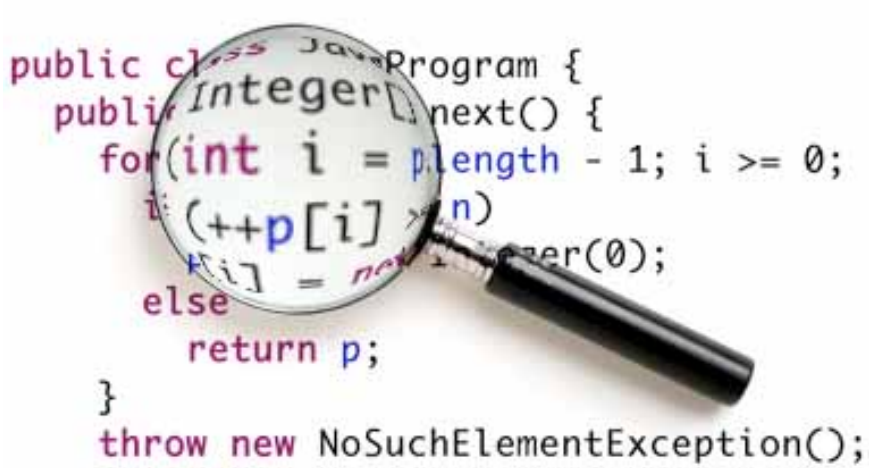
15-819M: Data, Code, Decisions

06a: Java Modeling Language

André Platzer

aplatzer@cs.cmu.edu
Carnegie Mellon University, Pittsburgh, PA



Outline

Outline

Road-map

Deductive Verification of JAVA source code

Road-map

Deductive Verification of JAVA source code

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

Road-map

Deductive Verification of JAVA source code

1. foundations: proving in first-order logic (done)
3. proving JAVA programs correct (later)

Road-map

Deductive Verification of JAVA source code

1. foundations: proving in first-order logic (done)
2. *specifying* JAVA programs (comes now)
3. proving JAVA programs correct (later)

Outline

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

instead:

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

- application level $\leftrightarrow$ application level
- application level $\leftrightarrow$ library level
- library level $\leftrightarrow$ library level

Unit Specifications

For object-oriented programs:

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

first focus on methods

methods specified by *potentially* referring to:

Unit Specifications

For object-oriented programs:

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

first focus on methods

methods specified by *potentially* referring to:

- result value,

Unit Specifications

For object-oriented programs:

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,

Unit Specifications

For object-oriented programs:

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

Unit Specifications

For object-oriented programs:

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

Stressing different roles – obligations – responsibilities in a specification:

widely used analogy of specification = *contract*

"Design by Contract" methodology [Bertrand Meyer]

Specifications as Contracts

Stressing different roles – obligations – responsibilities in a specification:

widely used analogy of specification = *contract*

"Design by Contract" methodology [Bertrand Meyer]

Contract between *caller* and *callee* of 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 () { ... }

}
```

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 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` is 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` is 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` 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 (exceptions).

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 (exceptions).

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

What kind of Specifications

Natural language specs are very important.

What kind of Specifications

Natural language specs are very important.

but here:

"Formal" specifications:

Describing contracts of units in a mathematically precise language.

What kind of Specifications

Natural language specs are very important.

but here:

"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 here:

"Formal" specifications:

Describing contracts of units in a mathematically precise language.

Motivation:

- higher degree of precision
- *automation* of program analysis of various kinds:
  - static checking
  - program verification

Outline

Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General JML Philosophy

Integrate

- specification
- implementation

in one single language.

$\Rightarrow$ 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.

$\Rightarrow$ JML is not external to JAVA

JML
is
JAVA

Java Modeling Language (JML)

JML is a specification language tailored to JAVA.

General JML Philosophy

Integrate

- specification
- implementation

in one single language.

$\Rightarrow$ 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.

$\Rightarrow$ 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.

$\Rightarrow$ JML is not external to JAVA

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

JML Annotations

JML extends JAVA by annotations.

JML annotations include:

- ✔ preconditions
- ✔ postconditions
- ✔ 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!

JML/JAVA integration

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

Let's not confuse JAVA compiler:

JML annotations live in in special comments,
ignored by JAVA, recognized by JML.

JML/JAVA integration

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

Let's not confuse JAVA compiler:

JML annotations live in in special comments,
ignored by JAVA, recognized by JML.

Neat proposed variant:

JML annotations live in Java @annotations.

JML by Example (ATM.java)

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

JML by Example (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 (ATM.java)

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

JML by Example (ATM.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
is *not* a comment for JML.

JML by Example (ATM.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 is *not* a comment for JML.

JML annotations appear in JAVA comments starting with @.

JML by Example (ATM.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
is *not* a comment for JML.

JML annotations appear in JAVA comments starting with @.

How about "//" comments?

JML by Example (ATM.java)

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

equivalent to:

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

JML by Example (ATM.java)

```
/*@ 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 (ATM.java)

```
/*@ 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-blank) character in the line
- if it is the last character before '*/'.

JML by Example (ATM.java)

```
/*@ 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-blank) 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 (ATM.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 (ATM.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 (ATM.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.

In this course: mostly `public` specifications.

JML by Example (ATM.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,

JML by Example (ATM.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.

JML by Example (ATM.java)

```
/*@ 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 (ATM.java)

```
/*@ 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 (ATM.java)

```
/*@ 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 later)

JML by Example (ATM.java)

```
/*@ 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 (ATM.java)

```
/*@ 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 (ATM.java)

```
/*@ 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 (ATM.java)

```
/*@ 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 (ATM.java)

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 (ATM.java)

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

Now JML expression not a JAVA expression

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

E can be any (arbitrarily complex) JAVA/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`'

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`?

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`? Frame prob.

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: Fighting Frames

Unpleasant if we have to add

```
@ ensures loc == \old(loc);
```

for all locations *loc* which *do not* change

Assignable Clause: Fighting Frames

Unpleasant if we have 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: Fighting Frames

Unpleasant if we have 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;
```

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

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 wrongPINCounter,
@            insertedCard,
@            insertedCard.invalid;
```

JML Modifiers

JML extends the JAVA modifiers by additional modifiers.

The most important ones are:

- `spec_public`
- `pure`

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

JML Modifiers: `spec_public`

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

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

No solution: make all fields public.

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.

No solution: make all fields public.

One solution:

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

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.

No solution: 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;
```

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.

No solution: 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)

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.

No solution: 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;
```

(Bug note:
in KeY1.4, spec_public fields are only visible within their class)

JML Modifiers: `pure`

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

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

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

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

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

The '`pure`' modifier puts an additional obligation on the implementer (not to cause side effects), but allows to use the method in annotations.

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")

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")
  - ...
  - ...
  - ...
  - ...

Beyond `boolean` JAVA expressions

How to express the following?

- an array `arr` only holds values ≤ 2

Beyond `boolean` JAVA expressions

How to express the following?

- an array `arr` only holds values ≤ 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 ≤ 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 ≤ 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`

Beyond `boolean` JAVA expressions

How to express the following?

- an array `arr` only holds values ≤ 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`

Stay tuned for next lecture . . .

Outline

Literature for this Lecture

Essential reading:

in KeY Book A. Roth and Peter H. Schmitt: Formal Specification. Chapter 5 only sections 5.1,5.3, In: B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, vol 4334 of *LNCS*. Springer, 2006.

Further reading:

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

`http://www.eecs.ucf.edu/~leavens/JML/`