15-819M: Data, Code, Decisions

06a: Java Modeling Language

André Platzer

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

```java
class JavaProgram {
    public int[] next() {
        for (int i = plength - 1; i >= 0; 
            i (--p[i] == n)
        } else
            return p;
    }
    throw new NoSuchElementException();
```
Outline

1. Overview

2. Unit Specification
   - Running Example
   - Informal Specification

3. Java Modeling Language (JML)
   - JML by Example
   - Assignable Locations
   - JML Modifiers
   - JML Expressions

4. Literature
Outline

1 Overview

2 Unit Specification
   - Running Example
   - Informal Specification

3 Java Modeling Language (JML)
   - JML by Example
   - Assignable Locations
   - JML Modifiers
   - JML Expressions

4 Literature
Deductive Verification of Java source code
Road-map

Deductive Verification of Java source code

1. foundations: proving in first-order logic (done)
Deductive Verification of \texttt{JAVA} source code

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

3. proving \texttt{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

1. Overview

2. Unit Specification
   - Running Example
   - Informal Specification

3. Java Modeling Language (JML)
   - JML by Example
   - Assignable Locations
   - JML Modifiers
   - JML Expressions

4. Literature
**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 ↔ application level
- application level ↔ library level
- library level ↔ library level
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,
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,
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
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 \textit{caller} and \textit{callee} of method

\textit{Callee} guarantees certain outcome \textit{provided} \textit{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 () { ... } 
}

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.
Contract states \textit{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

**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
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

- wrongPINCounter is increased by 1

- user is not authenticated

- card is confiscated

- 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
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
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.

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 non-termination and abrupt termination.
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

1. Overview

2. Unit Specification
   - Running Example
   - Informal Specification

3. Java Modeling Language (JML)
   - JML by Example
   - Assignable Locations
   - JML Modifiers
   - JML Expressions

4. Literature
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**

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.

⇒ JML is not external to Java

JML is Java + FO Logic
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.

**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 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!

Let’s not confuse Java compiler:

JML annotations live in in special comments, ignored by Java, recognized by JML.
JML/\texttt{Java} integration

JML annotations are attached to \texttt{Java} programs by writing them directly into the \texttt{Java} source code files!

Let’s not confuse \texttt{Java} compiler:

JML annotations live in special comments, ignored by \texttt{Java}, recognized by JML.

Neat proposed variant:

JML annotations live in \texttt{Java @annotations}.
/*@ public normal_behavior
   @ requires !customerAuthenticated;
   @ requires pin == insertedCard.correctPIN;
   @ ensures customerAuthenticated;
   @*/

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

public void enterPIN (int pin) {
  if ( ....
/*@ 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.
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)

```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?
/**
   * 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 ( ....
public void enterPIN (int pin) {
    if ( ....

What about the intermediate `@`'s?
```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)

```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.
```
/*@ 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
/*@ 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.
/*
  @public\_behavior
  @requires \!customerAuthenticated;
  @requires \text{pin} == \text{insertedCard.correctPIN};
  @ensures customerAuthenticated;
  @*/

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

This is a \textbf{public} specification case:
  \begin{enumerate}
    \item it is accessible from all classes and interfaces
    \item it can only mention public fields/methods of this class
  \end{enumerate}

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)

```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 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
/*@ 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
/*@ 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;
  @*/
/*@ 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 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
public void enterPIN (int pin) {
    if ( ....

This specification case has one postcondition (marked by \texttt{ensures})

- customerAuthenticated

here:
postcondition is \texttt{boolean JAVA expressions}

in general:
postconditions are \texttt{boolean JML expressions} (see below)
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 (......

/*@ <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’
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?
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:

```plaintext
@ 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?
Consider spec-case-1:

\[
\text{@ public normal_behavior}
\]
\[
\text{@ requires !customerAuthenticated;}
\]
\[
\text{@ requires pin == insertedCard.correctPIN;}
\]
\[
\text{@ ensures customerAuthenticated;}
\]

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 Spec-Cases

Completing spec-case-1:

\[\begin{align*}
@ & \text{public} \ \text{normal \_behavior} \\
@ & \text{requires} \ \neg \text{customerAuthenticated}; \\
@ & \text{requires} \ \text{pin} == \text{insertedCard.correctPIN}; \\
@ & \text{ensures} \ \text{customerAuthenticated}; \\
@ & \text{ensures} \ \text{insertedCard} == \text{\textbackslash old}(\text{insertedCard}); \\
@ & \text{ensures} \ \text{wrongPINCounter} == \text{\textbackslash old}(\text{wrongPINCounter});
\end{align*}\]
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);
Unpleasant if we have to add

@\textbf{ensures} \quad loc == \texttt{old}(loc);

for all locations \textit{loc} which \textbf{do not} change
Assignable Clause: Fighting Frames

Unpleasant if we have to add

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

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

Instead:
add assignable clause for all locations which may change

\[ \text{@ assignable } \, \text{loc}_1, \ldots, \text{loc}_n; \]
Assignable Clause: Fighting Frames

Unpleasant if we have to add

@ \texttt{ensures } \texttt{loc} == \texttt{old(loc)};

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

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

@ \texttt{assignable } \texttt{loc}_1, \ldots, \texttt{loc}_n;

Meaning: No location other than \texttt{loc}_1, \ldots, \texttt{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.
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.
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`
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

```java
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

```java
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

```java
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’:

```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.
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 \( a \)"")
  - \( a \land b \) ("\( a \) and \( b \)"")
  - \( a \lor b \) ("\( a \) 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 \texttt{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`
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`

Stay tuned for next lecture . . .
Outline

1. Overview

2. Unit Specification
   - Running Example
   - Informal Specification

3. Java Modeling Language (JML)
   - JML by Example
   - Assignable Locations
   - JML Modifiers
   - JML Expressions

4. Literature
Literature for this Lecture

Essential reading:


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/