Dynamic Logic for Practical Program Verification

Set 3

Wolfgang Ahrendt

Chalmers University of Technology, Gothenburg, Sweden

VTSA Summer School, Luxembourg, 2024

Part I

Java Modeling Language

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)`':

> *Checks whether the* `pin` *belongs to the bank card currently inserted in the ATM. If a wrong* `pin` *is received three times in a row, the card is confiscated. After receiving 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 < 3` 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 < 3` 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 < 3` and pin is incorrect
postcondition	`wrongPINCounter` has been increased by `1`, user is not authenticated
precondition	card is inserted, user not yet authenticated, `wrongPINCounter >= 3` 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 < 3` and pin is incorrect
postcondition	`wrongPINCounter` has been increased by 1, user is not authenticated
precondition	card is inserted, user not yet authenticated, `wrongPINCounter >= 3` and pin is incorrect
postcondition	card is confiscated user is not authenticated

Meaning of Pre/Postcondition 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/Postcondition 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/Postcondition 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 $\Rightarrow$ later

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;
  @*/
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 comes later.

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;
  @*/
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 < 3;
  @ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
  @*/
public void enterPIN (int pin) {
```

JML by Example

```
/*@ <spec-case1> also
  @
  @ public normal_behavior
  @ requires !customerAuthenticated;
  @ requires pin != insertedCard.correctPIN;
  @ requires wrongPINCounter < 3;
  @ 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 prestate 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 >= 3;
  @ 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`"

JML by Example

Question:

Could it be

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

instead of

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

??

JML by Example

Question:

Could it be

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

instead of

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

??

A: No. The second says that, after the method, the current value of field `invalid` (of the object formerly referred to by `insertCard`) is `false`.

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 poststate?

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 poststate?

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 poststate?

Recall: fields of class `ATM`:

```
insertedCard
customerAuthenticated
wrongPINCounter
```

What happens with `insertCard` and `wrongPINCounter`?

Assignable Clause

Unsatisfactory to add

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

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

Assignable Clause

Unsatisfactory to add

JML

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

JML

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

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

JML

```
@ assignable loc_1,...,loc_n;
```

JML

Specification Cases with Assignable

completing spec-case-1:

JML

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

JML

Specification Cases with Assignable

completing spec-case-2:

JML

```
@ public normal_behavior
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter < 3;
@ ensures wrongPINCounter == \old(wrongPINCounter) + 1;
@ assignable wrongPINCounter;
```

JML

Specification Cases with Assignable

completing spec-case-3:

JML

```
@ public normal_behavior
@ requires insertedCard != null;
@ requires !customerAuthenticated;
@ requires pin != insertedCard.correctPIN;
@ requires wrongPINCounter >= 3;
@ ensures insertedCard == null;
@ ensures \old(insertedCard).invalid;
@ assignable insertedCard,
@            insertedCard.invalid,
```

JML

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 Modifiers

JML extends the JAVA modifiers by additional modifiers

The most important ones are:

- `spec_public`
- `pure`
- `nullable`
- `non_null`
- `helper`

JML Modifiers: `spec_public`

In `enterPIN` example, pre/postconditions 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/postconditions 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 needed, make them public *in specification, only* by **`spec_public`**

JML Modifiers: `spec_public`

In `enterPIN` example, pre/postconditions 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 needed, make them public *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/postconditions 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 needed, make them public *in specification, only* by **`spec_public`**

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

(Alternative 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 must 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 must not themselves change the state!

Definition ((Strictly) Pure method)

A method is pure iff it always terminates and has no visible side effects on existing objects.
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 must not themselves change the state!

Definition ((Strictly) Pure method)

A method is pure iff it always terminates and has no visible side effects on existing objects.
A method is strictly pure if it is pure and does not create new objects.

JML expressions may contain calls to (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 must not themselves change the state!

Definition ((Strictly) Pure method)

A method is pure iff it always terminates and has no visible side effects on existing objects.
A method is strictly pure if it is pure and does not create new objects.

JML expressions may contain calls to (strictly) pure methods.

Pure methods are annotated by **`pure`** or **`strictly_pure`** resp.

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

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 ≤ 9.

Beyond `boolean` Java expressions

How to express the following?

- An array `arr` only holds values ≤ 9.
- 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 ≤ 9.
- 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 ≤ 9.
- 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")
  - `(\forall t x; a)` ("for all x of type t, a holds")
  - `(\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")
  - `(\forall t x; a)` ("for all x of type t, a holds")
  - `(\exists t x; a)` ("there exists x of type t such that a")
  - `(\forall t x; a; b)` ("for all x of type t fulfilling a, b holds")
  - `(\exists t x; a; b)` ("there exists an x of type t fulfilling a, such that b")

JML Quantifiers

In

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

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

a is called "range predicate"

JML Quantifiers

In

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

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

a is called "range predicate"

Range predicates are redundant:

```
(\forall t x; a; b)
```
equivalent to
```
(\forall 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 ≤ 9.

Using Quantified JML expressions

How to express:

- An array `arr` only holds values ≤ 9.

JML

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

Using Quantified JML expressions

How to express:

- An array `arr` only holds values ≤ 9.

JML

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

Using Quantified JML expressions

How to express:

- An array `arr` only holds values ≤ 9.

JML

```
(\forall int i; 0 <= i && i < arr.length; arr[i] <= 9)
```

JML

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`.

JML

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

JML

Using Quantified JML expressions

How to express:

- The variable `m` holds the maximum entry of array `arr`.

JML

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

JML

is this enough?

Using Quantified JML expressions

How to express:

- The variable `m` holds the maximum entry of array `arr`.

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

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

Using Quantified JML expressions

How to express:

- All `Account` objects in the array `accountArray` are stored at the index corresponding to their respective `accountNumber` field.

Using Quantified JML expressions

How to express:

- All `Account` objects in the array `accountArray` are stored at the index corresponding to their respective `accountNumber` field.

JML

```
(\forall int i; 0 <= i && i < maxAccountNumber;
                accountArray[i].accountNumber == i )
```

JML

Using Quantified JML expressions

How to express:

- All existing instances of class `BankCard` have different `cardNumbers`.

Using Quantified JML expressions

How to express:

- All existing instances of class `BankCard` have different `cardNumbers`.

JML

```
(\forall BankCard p1, p2;
        p1 != p2 ==> p1.cardNumber != p2.cardNumber)
```

JML

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

`contains` is `pure`: no effect on the state + terminates normally

Specifying `contains()`

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

`contains` is `pure`: no effect on the state + terminates normally

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

Specifying Data Constraints

So far:
JML used to specify method specifics.

Specifying Data Constraints

So far:
JML used to specify method specifics.

How to specify constraints on data?

Specifying Data Constraints

So far:
JML used to specify method specifics.

How to specify constraints on data, e.g.:

- consistency of redundant data representations (like indexing)
- restrictions for efficiency (like sortedness)

Specifying Data Constraints

So far:
JML used to specify method specifics.

How to specify constraints on data, e.g.:

- consistency of redundant data representations (like indexing)
- restrictions for efficiency (like sortedness)

Data constraints are global: all methods must preserve them

Consider LimitedSorted IntegerSet

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

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

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

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

  // other methods
```

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)
- Why does that even work?

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)
- Why does that even work?
- We assume sortedness in prestate

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)
- Why does that even work?
- We assume sortedness in prestate

method `add`

- Search first index with bigger element, insert just before that

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)
- Why does that even work?
- We assume sortedness in prestate

method `add`

- Search first index with bigger element, insert just before that
- Thereby try to establish sortedness in poststate

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)
- Why does that even work?
- We assume sortedness in prestate

method `add`

- Search first index with bigger element, insert just before that
- Thereby try to establish sortedness in poststate
- Why is that sufficient?

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)
- Why does that even work?
- We assume sortedness in prestate

method `add`

- Search first index with bigger element, insert just before that
- Thereby try to establish sortedness in poststate
- Why is that sufficient?
- We assume sortedness in prestate

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)
- Why does that even work?
- We assume sortedness in prestate

method `add`

- Search first index with bigger element, insert just before that
- Thereby try to establish sortedness in poststate
- Why is that sufficient?
- We assume sortedness in prestate

method `remove`

- (accordingly)

Specifying Sortedness with JML

Recall class fields:

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

Sortedness as JML expression:

Specifying Sortedness with JML

Recall class fields:

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

Sortedness as JML expression:

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

Specifying Sortedness with JML

Recall class fields:

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

Sortedness as JML expression:

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

(What's the value of this if `size < 2`?)

Specifying Sortedness with JML

Recall class fields:

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

Sortedness as JML expression:

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

(What's the value of this if `size < 2`?)

But where in the specification does the red expression go?

Specifying Sorted `contains()`

Can assume sortedness of prestate

Specifying Sorted `contains()`

Can assume sortedness of prestate

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

Specifying Sorted `contains()`

Can assume sortedness of prestate

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

`contains()` is *pure*
⇒ sortedness of poststate trivially ensured

Specifying Sorted `remove()`

Can assume sortedness of prestate. Must ensure sortedness of poststate

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

Specifying Sorted add() (spec-case1) – can add

```
/*@ public normal_behavior
  @ requires (\forall int i; 0 < i && i < size;
  @                          arr[i-1] <= arr[i]);
  @ 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;
  @ ensures (\forall int i; 0 < i && i < size;
  @                          arr[i-1] <= arr[i]);
  @
  @ also <spec-case2>
  @*/
public boolean add(int elem) {/*...*/}
```

Specifying Sorted add() (spec-case2) – cannot add

```
/*@ public normal_behavior
  @
  @ <spec-case1> also
  @
  @ public normal_behavior
  @ requires (\forall int i; 0 < i && i < size;
  @                          arr[i-1] <= arr[i]);
  @ requires (size == limit)  contains(elem);
  @ ensures \result == false;
  @ ensures (\forall int e; contains(e) <==> \old(contains(e)));
  @ ensures size == \old(size);
  @ ensures (\forall int i; 0 < i && i < size;
  @                         arr[i-1] <= arr[i]);
  @*/
public boolean add(int elem) {/*...*/}
```

Factor out Sortedness

So far: ‘sortedness’ has swamped our specification

Factor out Sortedness

So far: ‘sortedness’ has swamped our specification

We can do better, using

JML Class Invariant

construct for specifying data constraints centrally

Factor out Sortedness

So far: ‘sortedness’ has swamped our specification

We can do better, using

JML Class Invariant

construct for specifying data constraints centrally

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

JML Class Invariant

```
public class LimitedSortedIntegerSet {

  public final int limit;

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

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

  // constructor and methods,
  // without sortedness in pre/postconditions
}
```

Revisit Specification of `enterPIN()`

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

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

Revisit Specification of `enterPIN()`

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

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

so far:
all 3 *spec-cases* were **normal_behavior**

Specifying Exceptional Behavior of Methods

`normal_behavior` specification case, with preconditions P,
forbids method to throw exceptions if prestate satisfies P

Specifying Exceptional Behavior of Methods

`normal_behavior` specification case, with preconditions P,
forbids method to throw exceptions if prestate satisfies P

`exceptional_behavior` specification case, with preconditions P,
requires method to throw exceptions if prestate satisfies P

Specifying Exceptional Behavior of Methods

`normal_behavior` specification case, with preconditions P,
forbids method to throw exceptions if prestate satisfies P

`exceptional_behavior` specification case, with preconditions P,
requires method to throw exceptions if prestate satisfies P

Keyword **`signals`** specifies *poststate*, depending on thrown exception

Specifying Exceptional Behavior of Methods

`normal_behavior` specification case, with preconditions P,
forbids method to throw exceptions if prestate satisfies P

`exceptional_behavior` specification case, with preconditions P,
requires method to throw exceptions if prestate satisfies P

Keyword **`signals`** specifies *poststate*, depending on thrown exception

Keyword **`signals_only`** limits types of thrown exception

Completing Specification of `enterPIN()`

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

Completing Specification of `enterPIN()`

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

In case `insertedCard == null` in prestate:

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

Allowing Non-Termination

- `normal_behavior`
- `exceptional_behavior`

both enforce termination *by default*

Allowing Non-Termination

- `normal_behavior`
- `exceptional_behavior`

both enforce termination *by default*

In each specification case, non-termination can be permitted via the clause

```
diverges true;
```

Allowing Non-Termination

- `normal_behavior`
- `exceptional_behavior`

both enforce termination *by default*

In each specification case, non-termination can be permitted via the clause

```
diverges true;
```

Meaning:

Given the precondition of the specification case holds in prestate,
the method may or may not terminate

Further Modifiers: `non_null` and `nullable`

JML extends the JAVA modifiers by further modifiers:

- class fields
- method parameters
- method return types

can be declared as

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

non_null: Examples

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

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

non_null: Examples

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

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

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

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

non_null: Examples

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

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

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

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

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

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

`non_null` Default

`non_null` is default in JML!

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

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

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

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

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

```
public String toString()
```

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

`nullable`: Examples

To prevent such pre/postconditions and invariants: '**`nullable`**'

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

No implicit invariant added

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

No implicit precondition added

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

No implicit postcondition added to specification cases of `toString`

LinkedList: non_null or nullable?

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

In JML this means:

LinkedList: non_null or nullable?

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

In JML this means:

- All elements in the list are **non_null**

LinkedList: non_null or nullable?

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

In JML this means:

- All elements in the list are **non_null**
- The list is cyclic, or infinite!

LinkedList: non_null or nullable?

Repair:

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

⇒ Now, the list is allowed to end somewhere!