Software Engineering using Formal Methods

Java Modeling Language, Part II

Wolfgang Ahrendt & Josef Svenningsson & Meng Wang

4 October 2012

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`

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 is true")
  - `(\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 is true")
  - `(\exists t x; a)` ("there exists x of type t such that a")
  - `(\forall t x; a; b)` ("for all x of type t fulfilling a, b is true")
  - `(\exists t x; a; b)` ("there exists an x of type t fulfilling a, such that b")

JML Quantifiers

in

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

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

a called "range predicate"

JML Quantifiers

in

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

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

a called "range predicate"

those forms are redundant:

`(\forall t x; a; b)`
equivalent to
`(\forall t x; a ==> 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` used to restrict range of `x` further than `t`

Pragmatics of Range Predicates

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

widely used

pragmatics of range predicate:

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

example: "`arr` is sorted at indexes between 0 and 9":

Pragmatics of Range Predicates

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

widely used

pragmatics of range predicate:

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

example: "`arr` is sorted at indexes between 0 and 9":

```
(\forall 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` 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` 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 ≤ 2

Using Quantified JML expressions

How to express:

- an array `arr` only holds values ≤ 2

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

Using Quantified JML expressions

How to express:

- an array `arr` only holds values ≤ 2

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

Using Quantified JML expressions

How to express:

- an array `arr` only holds values ≤ 2

```
(\forall int i; 0 <= i && i < arr.length; arr[i] <= 2)
```

Using Quantified JML expressions

How to express:

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

Using Quantified JML expressions

How to express:

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

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

Using Quantified JML expressions

How to express:

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

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

is this enough?

Using Quantified JML expressions

How to express:

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

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

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

Using Quantified JML expressions

How to express:

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

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

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

Using Quantified JML expressions

How to express:

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

Using Quantified JML expressions

How to express:

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

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

Using Quantified JML expressions

How to express:

- all created instances of class `BankCard` have different `cardNumbers`

Using Quantified JML expressions

How to express:

- all created instances of class `BankCard` have different `cardNumbers`

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

Using Quantified JML expressions

How to express:

- all created instances of class `BankCard` have different `cardNumbers`

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

note:

- JML quantifiers range also over non-created objects

Using Quantified JML expressions

How to express:

- all created instances of class `BankCard` have different `cardNumbers`

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

note:

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

Using Quantified JML expressions

How to express:

- all created instances of class `BankCard` have different `cardNumbers`

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

note:

- JML quantifiers range also over non-created objects
- same for quantifiers in KeY!
- in JML, restrict to created objects with `\created`

Using Quantified JML expressions

How to express:

- all created instances of class `BankCard` have different `cardNumbers`

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

note:

- JML quantifiers range also over non-created objects
- same for quantifiers in KeY!
- in JML, restrict to created objects with **`\created`**
- in KeY?

Using Quantified JML expressions

How to express:

- all created instances of class `BankCard` have different `cardNumbers`

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

note:

- JML quantifiers range also over non-created objects
- same for quantifiers in KeY!
- in JML, restrict to created objects with **`\created`**
- in KeY? ($\Rightarrow$ coming lecture)

Generalized Quantifiers

JML offers also generalized quantifiers:

- `\max`
- `\min`
- `\product`
- `\sum`

returning the maximum, minimum, product, or sum of the values of the expressions given, where the variables satisfy the given range.

Examples (all these expressions are `true`):

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

Generalized Quantifiers

JML offers also generalized quantifiers:

- `\max`
- `\min`
- `\product`
- `\sum`

returning the maximum, minimum, product, or sum of the values of the expressions given, where the variables satisfy the given range.

Examples (all these expressions are `true`):

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

Generalized Quantifiers

JML offers also generalized quantifiers:

- `\max`
- `\min`
- `\product`
- `\sum`

returning the maximum, minimum, product, or sum of the values of the expressions given, where the variables satisfy the given range.

Examples (all these expressions are `true`):

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

Generalized Quantifiers

JML offers also generalized quantifiers:

- `\max`
- `\min`
- `\product`
- `\sum`

returning the maximum, minimum, product, or sum of the values of the expressions given, where the variables satisfy the given range.

Examples (all these expressions are `true`):

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
(\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
(\max int i; 0 <= i && i < 5; i) == 4
(\min int i; 0 <= i && i < 5; i-1) == -1
```

Example: Specifying `LimitedIntegerSet`

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

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

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

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

  // other methods
}
```

Prerequisites: Adding Specification Modifiers

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

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

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

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

  // other methods
}
```

Specifying `contains()`

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

Specifying `contains()`

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

has no effect on the state, incl. no exceptions

Specifying `contains()`

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

has no effect on the state, incl. no exceptions

how to specify result value?

Result Values in Postcondition

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

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

Result Values in Postcondition

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

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

Result Values in Postcondition

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

```
/*@ public normal_behavior
  @ ensures \result == (\exists int i;
  @                               0 <= i && i < size;
  @
```

Result Values in Postcondition

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

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

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

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

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

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

Specifying remove()

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

Specifying Data Constraints

So far:
JML used to specify method specifics.

Specifying Data Constraints

So far:
JML used to specify method specifics.

How to specify constraints on class data?

Specifying Data Constraints

So far:
JML used to specify method specifics.

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

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

Specifying Data Constraints

So far:
JML used to specify method specifics.

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

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

data constraints are global:
all methods must preserve them

Consider LimitedSortedIntegerSet

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

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

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

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

  // other methods
}
```

Consequence of Sortedness for Implementations

`method contains`

- can employ binary search (logarithmic complexity)

Consequence of Sortedness for Implementations

`method contains`

- can employ binary search (logarithmic complexity)
- why is that sufficient?

Consequence of Sortedness for Implementations

`method contains`

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

Consequence of Sortedness for Implementations

method `contains`

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

method `add`

- searches first index with bigger element, inserts just before that

Consequence of Sortedness for Implementations

method `contains`

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

method `add`

- searches first index with bigger element, inserts just before that
- thereby tries to establish sortedness in post-state

Consequence of Sortedness for Implementations

method `contains`

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

method `add`

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

Consequence of Sortedness for Implementations

method `contains`

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

method `add`

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

Consequence of Sortedness for Implementations

method `contains`

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

method `add`

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

method `remove`

- (accordingly)

Specifying Sortedness with JML

recall class fields:

```
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 pre-state

Specifying Sorted `contains()`

can assume sortedness of pre-state

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

Specifying Sorted `contains()`

can assume sortedness of pre-state

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

`contains()` is *pure*

⇒ sortedness of post-state trivially ensured

Specifying Sorted `remove()`

can assume sortedness of pre-state

must ensure sortedness of post-state

```
/*@ 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;

  /*@ public 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/post-conditions
}
```

JML Class Invariant

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

Instance vs. Static Invariants

instance invariants

can refer to instance fields of `this` object

(unqualified, like '`size`', or qualified with '**`this`**', like '**`this`**`.size`')

JML syntax: **`instance invariant`**

Instance vs. Static Invariants

instance invariants

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

static invariants

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

Instance vs. Static Invariants

instance invariants

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

static invariants

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

both

can refer to
– static fields
– instance fields via explicit reference, like '`o.size`'

Instance vs. Static Invariants

instance invariants

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

static invariants

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

both

can refer to
– static fields
– instance fields via explicit reference, like '`o.size`'

in classes: **instance is default** (static in interfaces)
if **`instance`** or **`static`** is omitted ⇒ instance invariant!

Static JML Invariant Example

```
public class BankCard {

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

  private /*@ spec_public @*/ int cardNumber;

  // rest of class follows

}
```

Recall Specification of enterPIN()

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

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

Recall Specification of `enterPIN()`

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

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

last lecture:
all 3 *spec-cases* were **normal_behavior**

Specifying Exceptional Behavior of Methods

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

Specifying Exceptional Behavior of Methods

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

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

Specifying Exceptional Behavior of Methods

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

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

keyword `signals` specifies *post-state*, depending on thrown exception

Specifying Exceptional Behavior of Methods

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

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

keyword **`signals`** specifies *post-state*, depending on thrown exception

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 pre-state

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

signals_only Clause: General Case

an exceptional specification case can have one clause of the form

signals_only $E_1, \ldots, E_n$;

where $E_1, \ldots, E_n$ are exception types

`signals_only` Clause: General Case

an exceptional specification case can have one clause of the form

`signals_only` $\texttt{E}_1\texttt{,...,E}_n\texttt{;}$

where $\texttt{E}_1\texttt{,...,E}_n$ are exception types

Meaning:

if an exception is thrown, it is of type $\texttt{E}_1$ or ... or $\texttt{E}_n$

signals Clause: General Case

an exceptional specification case can have several clauses of the form

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

where `E` is exception type, `b` is boolean expression

signals Clause: General Case

an exceptional specification case can have several clauses of the form

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

where `E` is exception type, `b` is boolean expression

Meaning:

if an exception of type `E` is thrown, `b` holds in post condition

Allowing Non-Termination

by default, both:

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

specification cases enforce termination

Allowing Non-Termination

by default, both:

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

specification cases enforce termination

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

```
diverges true;
```

Allowing Non-Termination

by default, both:

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

specification cases enforce termination

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

```
diverges true;
```

Meaning:

given the precondition of the specification case holds in pre-state, 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

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

implicit precondition

'`requires card != null;`'

added to each specification case of `insertCard`

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

implicit postcondition

'`ensures \result != null;`'

added to each specification case of `toString`

non_null is default in JML!

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

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

implicit invariant

'`public invariant name != null;`'

added to class

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

implicit precondition

'`requires card != null;`'

added to each specification case of `insertCard`

```
public String toString()
```

implicit postcondition

'`ensures \result != null;`'

added to each specification case of `toString`

nullable: Examples

To prevent such pre/post-conditions 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!

Final Remarks on `non_null` and `nullable`

`non_null` as default in JML only since a few years.

⇒ Older JML tutorial or articles may not use the **`non_null`** by default semantics.

Final Remarks on `non_null` and `nullable`

`non_null` as default in JML only since a few years.

⇒ Older JML tutorial or articles may not use the **`non_null`** by default semantics.

Pitfall!

Final Remarks on non_null and nullable

non_null as default in JML only since a few years.

⇒ Older JML tutorial or articles may not use the **non_null** by default semantics.

Pitfall!

```
/*@ non_null @*/ Object[] a;
```

is not the same as:

```
/*@ nullable @*/ Object[] a; //@ invariant a != null;
```

Final Remarks on non_null and nullable

non_null as default in JML only since a few years.

⇒ Older JML tutorial or articles may not use the **non_null** by default semantics.

Pitfall!

```
/*@ non_null @*/ Object[] a;
```

is not the same as:

```
/*@ nullable @*/ Object[] a; //@ invariant a != null;
```

because the first one also implicitly adds

```
(\forall int i; i >= 0 && i < a.length; a[i] != null)
```

I.e. extends **non_null** also to the elements of the array!

JML and Inheritance

All JML contracts, i.e.

- specification cases
- class invariants

are inherited down from superclasses to subclasses.

A class has to fulfill all contracts of its superclasses.

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

```
/*@ also
  @
  @ <subclass-specific-spec-cases>
  @*/
public void method () { ...
```

Tools

Many tools support JML (see `http://www.jmlspecs.org`).

On the course website you find a link how to install a JML checker for eclipse that works with newer JAVA versions.

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.
(e-version via Chalmers Library)

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

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

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

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