Formal Methods for Software Development

Java Modeling Language, Part II

Wolfgang Ahrendt

05 October 2018

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 *only in specification* 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 *only in specification* 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 *only in specification* by **`spec_public`**

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

(Different solution: use specification-only fields; not covered in this course, but see Sect. 7.7 in [JML Tutorial], see Literature slide.)

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 Modifiers: Purity Cont'd

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

JML Expressions ≠ 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 `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 ≤ 2.
- The variable `m` holds the maximum entry of array `arr`.
- All `Account` objects in the array `allAccounts` are stored at the index corresponding to their respective `accountNumber` field.
- All instances of class `BankCard` have different `cardNumbers`.

First-order Logic in JML Expressions

JML **`boolean`** expressions extend JAVA **`boolean`** expressions by:

- implication
- equivalence

First-order Logic in JML Expressions

JML `boolean` expressions extend Java `boolean` expressions by:

- implication
- equivalence
- quantification

boolean JML Expressions

boolean JML expressions are defined recursively:

boolean JML Expressions

- each side-effect free **boolean** JAVA expression is a **boolean** JML expression
- if a and b are **boolean** JML expressions, and x is a variable of type t, then the following are also **boolean** JML expressions:
  - `!a` ("not a")
  - `a && b` ("a and b")
  - `a || b` ("a or b")
  - `a ==> b` ("a implies b")
  - `a <==> b` ("a is equivalent to b")
  - `(\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"

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` 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 ≤ 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 `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.

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

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`.

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

Generalized Quantifiers

JML offers also generalized quantifiers:

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

returning the maximum, minimum, product, or sum of the values of a given expressions (with variables in a given range)

Generalized Quantifiers

JML offers also generalized quantifiers:

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

returning the maximum, minimum, product, or sum of the values of a given expressions (with variables in a given range)

Examples (with their value):

`(\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 a given expressions (with variables in a given range)

Examples (with their value):

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

`(\product int i; 0 < i && i < 5; (2*i)+1)` $= 3 * 5 * 7 * 9$

Generalized Quantifiers

JML offers also generalized quantifiers:

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

returning the maximum, minimum, product, or sum of the values of a given expressions (with variables in a given range)

Examples (with their value):

`(\sum int i; 0 <= i && i < 5; i)`	$= 0 + 1 + 2 + 3 + 4$
`(\product int i; 0 < i && i < 5; (2*i)+1)`	$= 3 * 5 * 7 * 9$
`(\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 a given expressions (with variables in a given range)

Examples (with their value):

`(\sum int i; 0 <= i && i < 5; i)`	$= 0 + 1 + 2 + 3 + 4$
`(\product int i; 0 < i && i < 5; (2*i)+1)`	$= 3 * 5 * 7 * 9$
`(\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)  {/*...*/}
```

`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 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 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 is that sufficient?

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)
- Why is that sufficient?
- We assume sortedness in prestate

Consequence of Sortedness for Implementer

method `contains`

- Can employ binary search (logarithmic complexity)
- Why is that sufficient?
- 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 is that sufficient?
- 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 is that sufficient?
- 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 is that sufficient?
- 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 is that sufficient?
- 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
}
```

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 of objects other than `this`, 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 of objects other than `this`, like '`o.size`'

In classes, **instance is default**. In interfaces, **static is default**.
If **`instance`** or **`static`** is omitted for invariants
⇒ instance invariant in classes, static invariant in interfaces

Static JML Invariant Example

```
public class BankCard {

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

  private /*@ spec_public @*/ int cardNumber;

  // rest of class follows

}
```

Class Invariants: Intuition, Notions & Scope

Class invariants must be

- established by
  - constructors (instance invariants)
  - static initialisation (static invariants)

Class Invariants: Intuition, Notions & Scope

Class invariants must be

- established by
  - constructors (instance invariants)
  - static initialisation (static invariants)
- preserved by all (non-helper) methods
  - assumed in prestate (implicit preconditions)
  - ensured in poststate (implicit postconditions)
  - can be violated during method execution

Class Invariants: Intuition, Notions & Scope

Class invariants must be

- established by
  - constructors (instance invariants)
  - static initialisation (static invariants)
- preserved by all (non-helper) methods
  - assumed in prestate (implicit preconditions)
  - ensured in poststate (implicit postconditions)
  - can be violated during method execution

Scope of invariant

- not limmited to it's class/interface
- depends on visibility (`private` vs. `public`) of local state

$\Rightarrow$ An invariant must not be violated by any code in any class

The JML modifier: **`helper`**

JML helper methods

```
T /*@ helper @*/ m(T p1, ..., T pn)
```

Neither assumes nor ensures any invariant by default.

The JML modifier: `helper`

JML helper methods

```
T /*@ helper @*/ m(T p1, ..., T pn)
```

Neither assumes nor ensures any invariant by default.

Pragmatics & Usage examples of helper methods

- Helper methods are usually **`private`**.
- Used for structuring implementation of public methods
  (e.g. factoring out reoccurring steps)
- Used in constructors
  (where invariants have not yet been established)

The JML modifier: `helper`

JML helper methods

```
T /*@ helper @*/ m(T p1, ..., T pn)
```

Neither assumes nor ensures any invariant by default.

Pragmatics & Usage examples of helper methods

- Helper methods are usually **`private`**.
- Used for structuring implementation of public methods
  (e.g. factoring out reoccurring steps)
- Used in constructors
  (where invariants have not yet been established)

Additional purpose in KeY context

Normal form, used when translating JML to Dynamic Logic.
(See later lecture)

Referring to Invariants

Aim: refer to invariants of arbitrary objects in JML expressions.

Referring to Invariants

Aim: refer to invariants of arbitrary objects in JML expressions.

- **\invariant_for**(o) is a boolean JML expression

Referring to Invariants

Aim: refer to invariants of arbitrary objects in JML expressions.

- **\invariant_for**(o) is a boolean JML expression
- **\invariant_for**(o) is true in a state where all invariants of o are true, otherwise false

Referring to Invariants

Aim: refer to invariants of arbitrary objects in JML expressions.

- **\invariant_for**(o) is a boolean JML expression
- **\invariant_for**(o) is true in a state where all invariants of o are true, otherwise false

Pragmatics:

- Use **\invariant_for(this)** when local invariant is intended but *not* implicitly given, e.g., in specification of **helper** methods.

Referring to Invariants

Aim: refer to invariants of arbitrary objects in JML expressions.

- **\invariant_for**(o) is a boolean JML expression
- **\invariant_for**(o) is true in a state where all invariants of o are true, otherwise false

Pragmatics:

- Use **\invariant_for(this)** when local invariant is intended but *not* implicitly given, e.g., in specification of **helper** methods.
- Put **\invariant_for(o)**, where **o** $\neq$ this, into local **requires**/**ensures** clause or **invariant** to assume/guarantee or maintain invariant of **o** locally

Examples of Referring to Invariants

```
public class Database {
  ...
  /*@ public normal_behavior
    @ requires ...;
    @ ensures  ...;
    @*/
  public void add (Set newItems) {
    ... <rough adding at first> ...;
    cleanUp();
  }
  ...
```

Examples of Referring to Invariants

```
public class Database {
  ...
  /*@ public normal_behavior
    @ requires ...;
    @ ensures  ...;
    @*/
  public void add (Set newItems) {
    ... <rough adding at first> ...;
    cleanUp();
  }
  ...
  /*@ private normal_behavior
    @ ensures \invariant_for(this);
    @*/
  private /*@ helper @*/ void cleanUp() { ... }
  ...
```

Examples of Referring to Invariants

Example

If all (non-helper) methods of `ATM` shall maintain invariant of object stored in `insertedCard`:

```
public class ATM {
  ...
/*@ private invariant
  @ insertedCard != null ==> \invariant_for(insertedCard);
  @*/
  private BankCard insertedCard;
  ...
```

Examples of Referring to Invariants

Alternatively more fine grained:

Example

If method `withdraw` of `ATM` relies on invariant of `insertedCard`:

```
public class ATM {
  ...
  private BankCard insertedCard;
  ...
  /*@ public normal_behavior
    @ requires \invariant_for(insertedCard);
    @ requires <other preconditions>;
    @ ensures <postcondition>;
    @*/
  public int withdraw (int amount) { ... }
  ...
```

Notes on \invariant_for

- For non-helper methods, **\invariant_for(this)** *implicitly* added to pre- and postconditions!

Notes on `\invariant_for`

- For non-helper methods, **`\invariant_for(this)`** *implicitly* added to pre- and postconditions!
- **`\invariant_for`**`(expr)` returns true iff `expr` satisfies the invariant of its static type:
  - Given `class B extends A`
  - After executing initialiser `A o = new B();`
    `\invariant_for(o)` is true when `o` satisfies invariants of `A`

Notes on `\invariant_for`

- For non-helper methods, **`\invariant_for(this)`** *implicitly* added to pre- and postconditions!
- **`\invariant_for`**`(expr)` returns true iff `expr` satisfies the invariant of its static type:
  - Given `class B extends A`
  - After executing initialiser `A o = new B();`
    `\invariant_for(o)` is true when `o` satisfies invariants of `A` ,
    `\invariant_for((B)o)` is true when `o` satisfies invariants of `B`.

Notes on `\invariant_for`

- For non-helper methods, **`\invariant_for(this)`** *implicitly* added to pre- and postconditions!
- **`\invariant_for`**`(expr)` returns true iff `expr` satisfies the invariant of its static type:
  - Given `class B extends A`
  - After executing initialiser `A o = new B();`
    `\invariant_for(o)` is true when `o` satisfies invariants of `A` ,
    `\invariant_for((B)o)` is true when `o` satisfies invariants of `B`.
- If `o` and `this` have different types, `\invariant_for(o)` only covers **`public`** invariants of `o`'s type.
  E.g., `\invariant_for(insertedCard)` refers to **`public`** invariants of `BankCard`.

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 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 **pre**state:

- enterPIN *must* throw an exception ('**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

$$\texttt{signals\_only}\ \texttt{E}_1\texttt{,...,}\texttt{E}_n\texttt{;}$$

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

signals_only Clause: General Case

An exceptional specification case can have one clause of the form

$$\texttt{signals\_only}\ E_1,\ldots,E_n;$$

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

Meaning:

If an exception is thrown, it is of type E_1 or ... or 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 afterwards

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 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!

Final Remarks on `non_null` and `nullable`

`non_null` as default in JML only since some years.

$\Rightarrow$ 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 some 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 some 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 some 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 () { ...
```

General Behaviour Specification Case

Complete Behavior Specification Case

```
behavior
  forall T1 x1; ...  forall Tn xn;
  old U1 y1 = F1; ...  old Uk yk = Fk;
  requires P;
  measured_by Mbe if Mbp;
  diverges D;
  when W;
  accessible R;
  assignable A;
  callable p1(...), ..., pl(...);
  captures Z;
  ensures Q;
  signals_only E1, ..., Eo;
  signals (E e) S;
  working_space Wse if Wsp;
  duration De if Dp;
```

gray not in this course

green in this course

General Behaviour Specification Case

Meaning of a behavior specification case in JML

An implementation of a method *m* satisfying its behavior spec. case must ensure: If property *P* holds in the method's prestate, then one of the following must hold

```
behavior
 requires P;
 diverges D;
 assignable A;
 ensures Q;
 signals_only
         E1,...,Eo;
 signals (E e) S;
```

General Behaviour Specification Case

Meaning of a behavior specification case in JML

An implementation of a method *m* satisfying its behavior spec. case must ensure: If property *P* holds in the method's prestate, then one of the following must hold

```
behavior
 requires P;
 diverges D;
 assignable A;
 ensures Q;
 signals_only
         E1,...,Eo;
 signals (E e) S;
```

- *D* holds in the prestate and method *m* does not terminate (default: *D*=`false`)
- ...

General Behaviour Specification Case

Meaning of a behavior specification case in JML

An implementation of a method *m* satisfying its behavior spec. case must ensure: If property *P* holds in the method's prestate, then one of the following must hold

```
behavior
 requires P;
 diverges D;
 assignable A;
 ensures Q;
 signals_only
          E1,...,Eo;
 signals (E e) S;
```

- ...
- in the reached (normal or abrupt) poststate: All of the following items must hold
  - only heap locations (static/instance fields, array elements) that did not exist in the prestate or are listed in *A* (assignable) may have been changed

General Behaviour Specification Case

Meaning of a behavior specification case in JML

An implementation of a method *m* satisfying its behavior spec. case must ensure: If property *P* holds in the method's prestate, then one of the following must hold

```
behavior
 requires P;
 diverges D;
 assignable A;
 ensures Q;
 signals_only
          E1,...,Eo;
 signals (E e) S;
```

- ...
- in the reached (normal or abrupt) poststate: All of the following items must hold
  - only heap locations ...
  - if *m* terminates normally, then in its poststate property *Q* holds (default: *Q*=true)

General Behaviour Specification Case

Meaning of a behavior specification case in JML

An implementation of a method *m* satisfying its behavior spec. case must ensure: If property *P* holds in the method's prestate, then one of the following must hold

```
behavior
 requires P;
 diverges D;
 assignable A;
 ensures Q;
 signals_only
          E1,...,Eo;
 signals (E e) S;
```

- ...
- in the reached (normal or abrupt) poststate: All of the following items must hold
  - only heap locations ...
  - if *m* terminates normally then ...
  - if *m* terminates abruptly then
    - with an exception listed in **`signals_only`** (default: all exceptions of *m*'s throws declaration + `RuntimeException` and `Error`) and
    - for matching **`signals`** clause, the exceptional postcondition *S* holds

General Behaviour Specification Case

Meaning of a behavior specification case in JML

An implementation of a method *m* satisfying its behavior spec. case must ensure: If property *P* holds in the method's prestate, then one of the following must hold

```
behavior
 requires P;
 diverges D;
 assignable A;
 ensures Q;
 signals_only
         E1,...,Eo;
 signals (E e) S;
```

- ...
- in the reached (normal or abrupt) poststate: All of the following items must hold
  - ...
  - **`\invariant_for(this)`** must be maintained (in normal or abrupt termination) by non-`helper` methods

Desugaring: Normal Behavior and Exceptional Behavior

Both **`normal_behavior`** and **`exceptional_behavior`** cases are expressible as general **`behavior`** cases:

Normal Behavior Case

- defaults to '`signals (Throwable e) false;`'
- forbids overwriting of `signals` and `signals_only`

Desugaring: Normal Behavior and Exceptional Behavior

Both **`normal_behavior`** and **`exceptional_behavior`** cases are expressible as general **`behavior`** cases:

Normal Behavior Case

- defaults to '`signals (Throwable e) false;`'
- forbids overwriting of `signals` and `signals_only`

Exceptional Behavior Case

- defaults to '`ensures false`'
- forbids overwriting of `ensures`

Desugaring: Normal Behavior and Exceptional Behavior

Both **`normal_behavior`** and **`exceptional_behavior`** cases are expressible as general **`behavior`** cases:

Normal Behavior Case

- defaults to '`signals (Throwable e) false;`'
- forbids overwriting of `signals` and `signals_only`

Exceptional Behavior Case

- defaults to '`ensures false`'
- forbids overwriting of `ensures`

Both default to '`diverge false`', but allow it to be overwritten.

Tools

Several tools support JML
(see `www.eecs.ucf.edu/~leavens/JML//index.shtml`).

On the course website:
web interface, implemented by Bart van Delft, to OpenJML.

Many thanks to Bart!

Literature for this Lecture

KeYbook *W. Ahrendt*, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich, editors.
Deductive Software Verification - The KeY Book
Vol 10001 of *LNCS*, Springer, 2016
(E-book at `link.springer.com`)

Essential reading:

JML Tutorial M. Huisman, *W. Ahrendt*, D. Grahl, M. Hentschel.
Formal Specification with the Java Modeling Language
Chapter 7 in [KeYbook]

Further reading available at
`www.eecs.ucf.edu/~leavens/JML//index.shtml`