Software Engineering using Formal Methods

Java Modeling Language, Part II

Wolfgang Ahrendt

1 October 2015

Recap: Specifying LimitedIntegerSet

```
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
}
```

Result Values in Postcondition

```
/*@ 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 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;

  /*@ 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/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;
    @     p1 != p2 ==> p1.cardNumber != p2.cardNumber)
    @*/

  private /*@ spec_public @*/ int cardNumber;

  // rest of class follows

}
```

Class Invariants: Intuition, Notions & Scope

Basic Intention/Intuition: Class invariants must be

- established by
  - the constructor (instance invariants) and
  - static initialisation (static invariants)
- preserved by all (non-helper) methods
  - assumed in prestate (i.e., invariants are implicit preconditions)
  - ensured in poststate (i.e., invariants are implicit postconditions)
  - they can be violated during method execution

Scope of invariants

Invariants are written local, but potentially are system wide properties.
This depends on the visibility (`private` vs. `public`) of local state.
Accordingly: Invariants 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 almost always **`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 almost always **`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 when 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 when all invariants of o are true, otherwise false

Pragmatics:

- \invariant_for(**o**), where **o** $\neq$ this:
  assume/guarantee or maintain invariant of **o**, by putting \invariant_for(**o**) into **requires**/**ensures** clause or **invariant**

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` when all invariants of `o` are `true`, otherwise `false`

Pragmatics:

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

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 pre-conditions>;
    @ ensures <post-condition>;
    @*/
  public int withdraw (int amount) { ... }
  ...
```

Examples of Referring to Invariants

```
public class Database {
  ...
  /*@ private normal_behavior
    @ ensures \invariant_for(this);
    @*/
  private /*@ helper @*/ void cleanUp () { ... }
  ...
```

Examples of Referring to Invariants

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

Notes on \invariant_for

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

Notes on `\invariant_for`

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

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

where $\mathtt{E_1,\ldots,E_n}$ are exception types

`signals_only` Clause: General Case

an exceptional specification case can have one clause of the form

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

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

Meaning:

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

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.

$\Rightarrow$ 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 a few years.

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

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) post-state: All of the following items must hold
  - only heap locations (static/instance fields, array elements) that did not exist in the pre-state 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) post-state: All of the following items must hold
  - only heap locations ...
  - if *m* terminated normally then in its post-state, 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) post-state: All of the following items must hold
  - only heap locations ...
  - if *m* terminated normally then ...
  - if *m* terminated abruptly then with
    - one of the exception listed in **`signals_only`** (default: all exceptions of *m*'s throws declaration + `RuntimeException` and `Error`) and
    - for matching **`signals`**, 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) post-state: 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

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

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

Many thanks to Bart!

Literature for this Lecture

essential reading:

New JML Tutorial M. Huisman, W. Ahrendt, D. Grahl, M. Hentschel: Formal Specification with the Java Modeling Language. Chapter in the new KeY book, to appear (see "JML" on literature/tools page)

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*