Formale Entwicklung objektorientierter Software

Prof. P.H. Schmitt, C. Engel, F. Werner

Fakultät für Informatik
Universität Karlsruhe (TH)

Winter 2006/2007

The Java Modeling Language
JML

JML
By Example

A JML specification for `enterPIN`

General Structure

```
public class ATM {
  private
          BankCard insertedCard = null;
  private
          boolean customerAuthenticated = false;



  public void enterPIN (int pin) {
    // here the implementation follows
```

A JML specification for `enterPIN`

General Structure

```
public class ATM {
  private
          BankCard insertedCard = null;
  private
          boolean customerAuthenticated = false;
/*@ public normal_behavior // case 1 @*/
/*@ also public normal_behavior // case 2 @*/
/*@ also public normal_behavior // case 3 @*/
  public void enterPIN (int pin) {
    // here the implementation follows
```

A JML specification for `enterPIN`

General Structure

```
public class ATM {
  private/*@ spec_public @*/
        BankCard insertedCard = null;
  private/*@ spec_public @*/
        boolean customerAuthenticated = false;
/*@ public normal_behavior // case 1 @*/
/*@ also public normal_behavior // case 2 @*/
/*@ also public normal_behavior // case 3 @*/
  public void enterPIN (int pin) {
    // here the implementation follows
```

A JML specification for `enterPIN`

Visibility Modifiers

```
public class ATM {
  private /*@ spec_public @*/ BankCard insertedCard = null;
  private /*@ spec_public @*/
          boolean  customerAuthenticated = false;

  /*@ public normal_behavior
```

A JML specification for `enterPIN`

Visibility Modifiers

```
public class ATM {
  private /*@ spec_public @*/ BankCard insertedCard = null;
  private /*@ spec_public @*/
          boolean  customerAuthenticated = false;

  /*@ public normal_behavior
```

Modifiers to specification cases have no influence on their semantics.

A JML specification for `enterPIN`

Visibility Modifiers

```
public class ATM {
  private /*@ spec_public @*/ BankCard insertedCard = null;
  private /*@ spec_public @*/
          boolean  customerAuthenticated = false;

  /*@ public normal_behavior
```

Modifiers to specification cases have no influence on their semantics.

Specification items marked *public* cannot refer to *private* fields.

A JML specification for enterPIN

Visibility Modifiers

```
public class ATM {
  private /*@ spec_public @*/ BankCard insertedCard = null;
  private /*@ spec_public @*/
          boolean  customerAuthenticated = false;

  /*@ public normal_behavior
```

Modifiers to specification cases have no influence on their semantics.

Specification items marked *public* cannot refer to *private* fields.

Private fields can be declared public for specification purposes only.

The First Specification Case

```
/*@ public normal_behavior
    requires  insertedCard != null;
    requires  !customerAuthenticated;
    requires  pin == insertedCard.correctPIN;
    assignable customerAuthenticated;
    ensures    customerAuthenticated;
 @*/
public void enterPIN (int pin) {
  // here the implementation follows
```

The First Specification Case

```
/*@ public normal_behavior
    requires  insertedCard != null;
    requires  !customerAuthenticated;
    requires  pin == insertedCard.correctPIN;
    assignable customerAuthenticated;
    ensures    customerAuthenticated;
 @*/
public void enterPIN (int pin) {
  // here the implementation follows
```

The *normal_behavior* keyword specifies that the method may not throw an exception

The First Specification Case

```
/*@ public normal_behavior
    requires  insertedCard != null;
    requires  !customerAuthenticated;
    requires  pin == insertedCard.correctPIN;
    assignable customerAuthenticated;
    ensures    customerAuthenticated;
 @*/
public void enterPIN (int pin) {
  // here the implementation follows
```

The conjunction of all *requires* clauses is the precondition.

The First Specification Case

```
/*@ public normal_behavior
    requires  insertedCard != null;
    requires  !customerAuthenticated;
    requires  pin == insertedCard.correctPIN;
    assignable customerAuthenticated;
    ensures    customerAuthenticated;
 @*/
public void enterPIN (int pin) {
  // here the implementation follows
```

The conjunction of all *ensures* clauses is the postcondition.

The First Specification Case

```
/*@ public normal_behavior
    requires  insertedCard != null;
    requires  !customerAuthenticated;
    requires  pin == insertedCard.correctPIN;
    assignable customerAuthenticated;
    ensures    customerAuthenticated;
 @*/
public void enterPIN (int pin) {
  // here the implementation follows
```

The *assignable* clause list all expressions that may be assigned to.

The Second Specification Case

```
public normal_behavior
requires  insertedCard != null;
requires  !customerAuthenticated;
requires  pin != insertedCard.correctPIN;
requires  wrongPINCounter < 2;
assignable wrongPINCounter;
ensures   wrongPINCounter
                == \old(wrongPINCounter) + 1;
ensures   !customerAuthenticated;
```

The Second Specification Case

```
public normal_behavior
requires  insertedCard != null;
requires  !customerAuthenticated;
requires  pin != insertedCard.correctPIN;
requires  wrongPINCounter < 2;
assignable wrongPINCounter;
ensures   wrongPINCounter
                == \old(wrongPINCounter) + 1;
ensures   !customerAuthenticated;
```

Special JML keywords start with \

The Second Specification Case

```
public normal_behavior
requires  insertedCard != null;
requires  !customerAuthenticated;
requires  pin != insertedCard.correctPIN;
requires  wrongPINCounter < 2;
assignable wrongPINCounter;
ensures   wrongPINCounter
                == \old(wrongPINCounter) + 1;
ensures   !customerAuthenticated;
```

\old(`wrongPINCounter`) refers to the value of the field `wrongPINCounter` before method invocation.

The Third Specification Case

```
public normal_behavior
requires  insertedCard != null;
requires  !customerAuthenticated;
requires  pin != insertedCard.correctPIN;
requires  wrongPINCounter >= 2;
assignable insertedCard, wrongPINCounter,
           insertedCard.invalid;
ensures   insertedCard == null;
ensures   \old(insertedCard).invalid;
ensures   !customerAuthenticated;
```

Exceptional Behaviour

```
/*@
  @ (* the contracts as defined above *)
  @ also public exceptional_behavior
  @   requires insertedCard==null;
  @   signals_only ATMException;
  @   signals (ATMException) !customerAuthenticated;
  @*/
public void enterPIN (int pin) {
 // here the implementation follows
```

Exceptional Behaviour

```
/*@
  @ (* the contracts as defined above *)
  @ also public exceptional_behavior
  @   requires insertedCard==null;
  @   signals_only ATMException;
  @   signals (ATMException) !customerAuthenticated;
  @*/
public void enterPIN (int pin) {
 // here the implementation follows
```

If an exception of type `ATMException` is thrown then `!customerAuthenticated` has to be true.

Exceptional Behaviour

```
/*@
  @ (* the contracts as defined above *)
  @ also public exceptional_behavior
  @   requires insertedCard==null;
  @   signals_only ATMException;
  @   signals (ATMException) !customerAuthenticated;
  @*/
public void enterPIN (int pin) {
 // here the implementation follows
```

`signals_only` says that only exceptions of type `ATMException` may be thrown.

Pure Methods

Pure methods terminate and have no side effects.

Pure Methods

Pure methods terminate and have no side effects.

After declaring

```
public /*@ pure @*/ boolean cardIsInserted() {
  return insertedCard!=null;
}
```

Pure Methods

Pure methods terminate and have no side effects.

After declaring

```
public /*@ pure @*/ boolean cardIsInserted() {
  return insertedCard!=null;
}
```

`cardIsInserted()`

could replace

`insertedCard != null`

in the above JML annotations.

A Static Invariant

```
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}
```

A Static Invariant

```
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}
```

Compare to OCL version:

A Static Invariant

```
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}
```

Compare to OCL version:

```
context BankCard
inv:    BankCard::allInstances() ->  forall(p1,p2|
        p1<>p2 implies p1.cardNumber<>p2.cardNumber)
```

An Instance Invariant

```
public class BankCard {
 /*@ public instance invariant
   @    (\forall BankCard p; this!=p ==>
                     this.cardNumber!=p.cardNumber)
   @*/
   private /*@ spec_public @*/ int cardNumber;
   // rest of class follows}
```

An Instance Invariant

```
public class BankCard {
 /*@ public instance invariant
   @    (\forall BankCard p; this!=p ==>
                   this.cardNumber!=p.cardNumber)
   @*/
   private /*@ spec_public @*/ int cardNumber;
   // rest of class follows}
```

Instance invariants must evaluate to true for all created objects of their class.

Variation on the Static Invariant

```
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}
```

Variation on the Static Invariant

```
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}
```

```
public class BankCard {
  /*@ public instance invariant
    @    (\forall BankCard p; this!=p ==>
                   this.cardNumber!=p.cardNumber)
    @*/
    private /*@ spec_public @*/ int cardNumber;
    // rest of class follows}
```

Another Example

OCL constraint:

```
context CentralHost
inv: validCardsCount =
     BankCard::allInstances() ->
        select(not invalid) -> size()
```

Another Example

OCL constraint:

```
context CentralHost
inv: validCardsCount =
     BankCard::allInstances() ->
        select(not invalid) -> size()
```

JML annotation:

```
public class CentralHost {
  /*@ public instance invariant this.validCardsCount
    @                 ==(\num_of BankCard p; !p.invalid)
    @*/}
```

JML
Expressions

Definition

Every JAVA expression according to the language specification which does not include

Definition

Every JAVA expression according to the language specification which does not include

1. operators with side-effect like `e++`, `e--`, `++e`, or `--e`,

Definition

Every JAVA expression according to the language specification which does not include

1. operators with side-effect like `e++`, `e--`, `++e`, or `--e`,
2. non-pure method invocation expressions,

Definition

Every JAVA expression according to the language specification which does not include

1. operators with side-effect like `e++`, `e--`, `++e`, or `--e`,
2. non-pure method invocation expressions,
3. assignment operators

Definition

Every JAVA expression according to the language specification which does not include

1. operators with side-effect like `e++`, `e--`, `++e`, or `--e`,
2. non-pure method invocation expressions,
3. assignment operators

Definition

Every JAVA expression according to the language specification which does not include

1. operators with side-effect like `e++`, `e--`, `++e`, or `--e`,
2. non-pure method invocation expressions,
3. assignment operators

is a JML expression.

Definition

Every JAVA expression according to the language specification which does not include

1. operators with side-effect like `e++`, `e--`, `++e`, or `--e`,
2. non-pure method invocation expressions,
3. assignment operators

is a JML expression.

Any such expression e has a natural representation in KeY's first-order logic, which we will denote by $[e]$.

Mapping from JML plus JAVA *to FOL*

Selected Items

JML Expression	first-order logic formula
`!e_0`	$![e_0]$
`e_0 && e_1`	$[e_0]$ & $[e_1]$
`e_0 || e_1`	$[e_0] \mid [e_1]$
`e_0 ? e_1 : e_2`	$\mathrm{if}[e_0]\,\mathrm{then}[e_1]\,\mathrm{else}[e_2]$
`e_0 != e_1`	$!([e_0] = [e_1])$
`e_0 >= e_1`	$[e_0] >= [e_1]$
`e_0 ==> e_1`	$[e_0] -> [e_1]$
`e_0 <==> e_1`	$[e_0] <-> [e_1]$
`(\forall T e; e_0; e_1)`	$\backslash\mathit{forall}\ T\ e(![e] = \mathtt{null}\ \&\ [e_0] -> [e_1])$
`(\exists T e; e_0; e_1)`	$\backslash\mathit{exists}\ T\ e(![e] = \mathtt{null}\ \&\ [e_0]\ \&\ [e_1])$

Quantification in JML

Note that quantifiers bind two expressions, the range predicate and the body expression.

Quantification in JML

Note that quantifiers bind two expressions, the range predicate and the body expression.

A missing range predicate is by default `true`.

Quantification in JML

Note that quantifiers bind two expressions, the range predicate and the body expression.

A missing range predicate is by default `true`.

JML excludes `null` from the range of quantification.

Generalised and Numerical Quantifiers

`\num_of C  c; e` number of elements of class `C` with property `e`

Generalised and Numerical Quantifiers

`\num_of C c; e` number of elements of class `C` with property `e`

`\sum`

Generalised and Numerical Quantifiers

`\num_of C c; e` number of elements of class `C` with property `e`

`\sum`

`\product`

Generalised and Numerical Quantifiers

`\num_of C c; e` number of elements of class `C` with property `e`

`\sum`

`\product`

`\min`

Generalised and Numerical Quantifiers

`\num_of C c; e` number of elements of class `C` with property `e`

`\sum`

`\product`

`\min`

`\max`

Generalised and Numerical Quantifiers

`\num_of C c; e` number of elements of class `C` with property `e`

`\sum`

`\product`

`\min`

`\max`

JML
Operation Contracts

Clauses in Operation Contracts

Clause	Lightweight default	Heavyweight default
`requires`	`\not_specified`	`true`
`assignable`	`\not_specified`	`\everything`
`ensures`	`\not_specified`	`true`
`diverges`	`false`	`false`
`signals`	`\not_specified`	`(Exception)true`
`signals_only`	All exception types declared in the JAVA method declaration	

Signals Clauses

JML

```
ensures   E;
signals  (ET_1)  S_1;
⋮
signals  (ET_n)  S_n;
signals_only  OT_1,...,OT_m;
```

$$
\begin{array}{l}
\texttt{ensures} \quad E\,; \\
\texttt{signals} \;\; (ET_1) \;\; S_1\,; \\
\vdots \\
\texttt{signals} \;\; (ET_n) \;\; S_n\,; \\
\texttt{signals\_only} \;\; OT_1,\ldots,OT_m\,;
\end{array}
$$

Signals Clauses

JML

```
ensures   E;
signals  (ET_1)  S_1;
⋮
signals  (ET_n)  S_n;
signals_only  OT_1, ..., OT_m;
```

FOL Translation

$$
\begin{array}{ll}
(& \texttt{e} = \texttt{null} \mathrel{-\!\!>} [E]) \\
\&\ (& [\mathrm{ET_1}]\text{::instance(e)} = \mathrm{TRUE} \mathrel{-\!\!>} [S_1]) \ldots \\
\&\ (& [\mathrm{ET_n}]\text{::instance(e)} = \mathrm{TRUE} \mathrel{-\!\!>} [S_n]) \\
\&\ (& [\mathrm{OT_1}]\text{::instance(e)} = \mathrm{TRUE} \mid \ldots \mid [\mathrm{OT_m}]\text{::instance(e)} = \mathrm{TRUE})
\end{array}
$$

Signals Clauses

JML

```
ensures E;
signals (ET_1) S_1;
⋮
signals (ET_n) S_n;
signals_only OT_1,...,OT_m;
```

FOL Translation

$$
\begin{array}{ll}
(& \mathtt{e} = \mathtt{null} \texttt{->} [E]) \\
\&\ (& [\mathrm{ET_1}]\text{::instance}(\mathtt{e}) = \mathrm{TRUE} \texttt{->} [S_1]) \ldots \\
\&\ (& [\mathrm{ET_n}]\text{::instance}(\mathtt{e}) = \mathrm{TRUE} \texttt{->} [S_n]) \\
\&\ (& [\mathrm{OT_1}]\text{::instance}(\mathtt{e}) = \mathrm{TRUE} \mid \ldots \mid [\mathrm{OT_m}]\text{::instance}(\mathtt{e}) = \mathrm{TRUE})
\end{array}
$$

The variable e stores a thrown exception. If the operation terminates normally then e equals `null`.

The `diverges` *Clause*

`diverges e`

with a boolean JML expression `e` specifies that the methode may not terminate only when `e` is true in the pre-state.

The `diverges` *Clause*

`diverges e`

with a boolean JML expression `e` specifies that the methode may not terminate only when `e` is true in the pre-state.

If

`diverges false`

is part of the operation contract for `m` then `m` must always terminate.

The `diverges` *Clause*

`diverges e`

with a boolean JML expression `e` specifies that the methode may not terminate only when `e` is true in the pre-state.

If

`diverges false`

is part of the operation contract for `m` then `m` must always terminate.

If

`diverges true`

is part of the operation contract for `m` then `m` may terminate or not.

The diverges *Clause*

`diverges e`

with a boolean JML expression `e` specifies that the methode may not terminate only when `e` is true in the pre-state.

If

`diverges false`

is part of the operation contract for `m` then `m` must always terminate.

If

`diverges true`

is part of the operation contract for `m` then `m` may terminate or not.

If

`diverges n == 0`

is part of the operation contract for `m` then `m` must terminate, when called in a state with `n != 0`.

De-Sugaring

```
normal_behavior
  requires R;
  assignable A;
  ensures E;
  diverges D;
```

$\Longrightarrow$

```
behavior
  requires R;
  assignable A;
  ensures E;
  diverges D;
  signals (Exception) false;
```

De-Sugaring

normal_behavior
  requires R;
  assignable A;
  ensures E;
  diverges D;
	$\Longrightarrow$	behavior
		  requires R;
		  assignable A;
		  ensures E;
		  diverges D;
		  signals (Exception) false;

exceptional_behavior
  requires R;
  assignable A;
  diverges D;
  signals (ET) S;
  signals_only (OT);
	$\Longrightarrow$	behavior
		  requires R;
		  assignable A;
		  ensures false;
		  diverges D;
		  signals (ET) S;
		  signals_only (OT);

Inheritance of Specifications in JML

An invariant to a class is inherited by all its subclasses.

Inheritance of Specifications in JML

An invariant to a class is inherited by all its subclasses.

An operation contract is inherited by all overridden methods.

JML
Invariants

An Instance Invariant

JML

```
public class BankCard {
  /*@ public instance invariant
    @   (\forall BankCard p; this!=p ==>
                    this.cardNumber!=p.cardNumber)
    @*/
    private /*@ spec_public @*/ int cardNumber;
    // rest of class follows}
```

An Instance Invariant

JML

```
public class BankCard {
  /*@ public instance invariant
    @   (\forall BankCard p; this!=p ==>
                    this.cardNumber!=p.cardNumber)
    @*/
    private /*@ spec_public @*/ int cardNumber;
    // rest of class follows}
```

FOL

```
\forall BankCard o; o.<created> = TRUE ->
  \forall BankCard p; p.<created> = TRUE ->
     !o = p -> !o.cardNumber = p.cardNumber
```

Visible State Semantics

According to the JML reference manual instance invariants defined in a class C must hold at any visible state for any object o of C.

Visible State Semantics

According to the JML reference manual instance invariants defined in a class C must hold at any visible state for any object o of C.

A state is visible for an object o if it is reached at one of the following moments during the execution of a program; we leave out finalizers and JML's helper methods for simplicity:

- at the end of a constructor invocation which is initialising o,

Visible State Semantics

According to the JML reference manual instance invariants defined in a class C must hold at any visible state for any object o of C.

A state is visible for an object o if it is reached at one of the following moments during the execution of a program; we leave out finalizers and JML's helper methods for simplicity:

- at the end of a constructor invocation which is initialising o,
- at the beginning and end of a non-static method invocation with o as receiver,

Visible State Semantics

According to the JML reference manual instance invariants defined in a class C must hold at any visible state for any object o of C.

A state is visible for an object o if it is reached at one of the following moments during the execution of a program; we leave out finalizers and JML's helper methods for simplicity:

- at the end of a constructor invocation which is initialising o,
- at the beginning and end of a non-static method invocation with o as receiver,
- at the beginning and end of a static method which is declared in the class of o or a superclass.

Visible State Semantics

According to the JML reference manual instance invariants defined in a class C must hold at any visible state for any object o of C.

A state is visible for an object o if it is reached at one of the following moments during the execution of a program; we leave out finalizers and JML's helper methods for simplicity:

- at the end of a constructor invocation which is initialising o,
- at the beginning and end of a non-static method invocation with o as receiver,
- at the beginning and end of a static method which is declared in the class of o or a superclass.
- when no constructor, non-static method invocation with o as receiver, or static method invocation for a method in o's class or a superclass is in progress.

Observed State Semantics

A program P is observed-state correct w.r.t. a specification S, if

1. all operations *op* fulfil all operation contracts of S for *op*,

Observed State Semantics

A program P is observed-state correct w.r.t. a specification S, if

1. all operations *op* fulfil all operation contracts of S for *op*,
2. all invariants Inv_S of S are preserved by all operations of P, and

Observed State Semantics

A program P is observed-state correct w.r.t. a specification S, if

1. all operations *op* fulfil all operation contracts of S for *op*,
2. all invariants Inv_S of S are preserved by all operations of P, and
3. all invariants are valid in the initial state of P.

Example Program

```
public class A {
  private int i = 1;
  /*@ instance invariant i>0 */
  public int getI() { return i; }
  /*@ requires p>0;
    @ ensures i==p; @*/
  public void setI(int p) { i=p; }
  public void m1() {setI(0);i=1; }
  public void m2() {i=0;setI(1); }
  public int m3() {i=0;i=(new B()).m5(this);}
  /*@ ensures \result>0 @*/
  public int m4() {return 42/i;}  }
public class B {
  /*@ ensures \result>0 @*/
  public int m5(A a){if(a.getI()<=0) a.setI(1);
    return a.m4();}  }
```

Visible vs Observable States

```
public class A {
   private int i = 1;
   /*@ instance invariant i>0 */

   /*@ requires p>0;
     @ ensures i==p; @*/
   public void setI(int p) { i=p; }
   public void m1() {setI(0);
```

visible, but not observable state

```
   i=1; }
}
```

Visible vs Observable States

```
public class A {
  private int i = 1;
  /*@ instance invariant i>0 */

  /*@ requires p>0;
    @ ensures i==p; @*/
  public void setI(int p) { i=p; }
  public void m1() {setI(0);
  visible, but not observable state
  i=1; }
}
```

Invariant `i > 0` not satisfied in visible state semantics.

Visible vs Observable States

```
public class A {
  private int i = 1;
  /*@ instance invariant i>0 */

  /*@ requires p>0;
    @ ensures i==p; @*/
  public void setI(int p) { i=p; }
  public void m1() {setI(0);
  visible, but not observable state
  i=1; }
}
```

Invariant `i > 0` not satisfied in visible state semantics.

Invariant `i > 0` satisfied in observable state semantics.

A Static Invariant

```
public class CentralHost {
  /*@ public static invariant maxAccountNumber>=0 @*/
//...
```

must hold already after the static initialisation of CentralHost is completed.

JML
Model Fields and Methods

Java Interfaces

```
public interface IBonusCard {

    public void addBonus(int newBonusPoints);
}
```

Java Interfaces

```
public interface IBonusCard {





    public void addBonus(int newBonusPoints);

}
```

How to add contracts to abstract methods in interfaces?

Java Interfaces

```
public interface IBonusCard {

    public void addBonus(int newBonusPoints);

}
```

How to add contracts to abstract methods in interfaces?
Remember: There are no attributes in interfaces.

Java Interfaces

```
public interface IBonusCard {




    public void addBonus(int newBonusPoints);

}
```

How to add contracts to abstract methods in interfaces?
Remember: There are no attributes in interfaces.
More precisely: Only static final fields.

Java Interfaces

Model Fields

```
public interface IBonusCard {

/*@ public instance model int bonusPoints; @*/



    public void addBonus(int newBonusPoints);

}
```

How to add contracts to abstract methods in interfaces?
Remember: There are no attributes in interfaces.
More precisely: Only static final fields.

Java Interfaces

Model Fields

```
public interface IBonusCard {

/*@ public instance model int bonusPoints; @*/

/*@ ensures bonusPoints == \old(bonusPoints)+newBonusPoints;


    public void addBonus(int newBonusPoints);

}
```

How to add contracts to abstract methods in interfaces?
Remember: There are no attributes in interfaces.
More precisely: Only static final fields.

Java Interfaces
Model Fields

```
public interface IBonusCard {

/*@ public instance model int bonusPoints; @*/

/*@ ensures bonusPoints == \old(bonusPoints)+newBonusPoints;
  @ assignable bonusPoints;
  @ */
    public void addBonus(int newBonusPoints);

}
```

How to add contracts to abstract methods in interfaces?
Remember: There are no attributes in interfaces.
More precisely: Only static final fields.

Implementing Interfaces

Interface

```
public interface IBonusCard {
   /*@ public instance model int bonusPoints; @*/
   public void addBonus(int newBonusPoints);
```

Implementing Interfaces

Interface

```
public interface IBonusCard {
   /*@ public instance model int bonusPoints; @*/
   public void addBonus(int newBonusPoints);
```

Implementation

```
public class BankCard implements IBonusCard{
   public int bankCardPoints;
  /*@ public instance model int bonusPoints; @*/



  public void addBonus(int newBonusPoints) {
    bankCardPoints+=newBonusPoints;
  }}
```

Implementing Interfaces

Interface

```
public interface IBonusCard {
   /*@ public instance model int bonusPoints; @*/
   public void addBonus(int newBonusPoints);
```

Implementation

```
public class BankCard implements IBonusCard{
    public int bankCardPoints;
   /*@ public instance model int bonusPoints; @*/

/*@ private represents bonusPoints <-bankCardPoints; @*/

   public void addBonus(int newBonusPoints) {
     bankCardPoints+=newBonusPoints;
   }}
```

Other Representations

```
/*@ private represents bonusPoints
              <- bankCardPoints; @*/
```

Other Representations

```
/*@ private represents bonusPoints
            <- bankCardPoints; @*/
```

```
/*@ private represents bonusPoints
            <- bankCardPoints * 100; @*/
```

Other Representations

```
/*@ private represents bonusPoints
              <- bankCardPoints; @*/
```

```
/*@ private represents bonusPoints
              <- bankCardPoints * 100; @*/
```

```
/*@ represents x \such_that A(x); @*/
```

Problems with Specifications Using Integers

```
/*@ requires y >= 0;
  @ ensures
  @  \result * \result <= y &&
  @ y < (abs(\result)+1) * (abs(\result)+1);
  @ */
  public static int isqrt(int y)
```

Problems with Specifications Using Integers

```
/*@ requires y >= 0;
  @ ensures
  @  \result * \result <= y &&
  @ y < (abs(\result)+1) * (abs(\result)+1);
  @ */
  public static int isqrt(int y)
```

For $y = 1$ and $\backslash result = 1073741821 = \frac{1}{2}(max\_int - 5)$ the above postcondition is true, though we do not want 1073741821 to be a square root of 1.

Problems with Specifications Using Integers

```
/*@ requires y >= 0;
  @ ensures
  @  \result * \result <= y &&
  @ y < (abs(\result)+1) * (abs(\result)+1);
  @ */
  public static int isqrt(int y)
```

For $y = 1$ and $\backslash result = 1073741821 = \frac{1}{2}(max\_int - 5)$ the above postcondition is true, though we do not want 1073741821 to be a square root of 1.

The problem arises since JML uses the JAVA semantics of integers which yields

$$\begin{aligned} 1073741821 * 1073741821 &= -2147483639 \\ 1073741822 * 1073741822 &= 4 \end{aligned}$$

Advantages of OCL over JML

1. It lives on a higher level of abstraction. A UML diagram can be annotated with OCL constraints before code is developed.

Advantages of OCL over JML

1. It lives on a higher level of abstraction. A UML diagram can be annotated with OCL constraints before code is developed.
2. As a consequence of the previous item OCL is not committed to a particular programming language and better suited for model driven system development.

Advantages of OCL over JML

1. It lives on a higher level of abstraction. A UML diagram can be annotated with OCL constraints before code is developed.
2. As a consequence of the previous item OCL is not committed to a particular programming language and better suited for model driven system development.
3. OCL is an OMG standard, though one has to admit that the official standard draft still contains serious inconsistencies and many unfinished items.

Advantages of JML over OCL

1. JML is closer to JAVA code, which encourages its use by programmers and developers. In fact, today JML specifications are much more widespread than OCL specifications.

Advantages of JML over OCL

1. JML is closer to JAVA code, which encourages its use by programmers and developers. In fact, today JML specifications are much more widespread than OCL specifications.
2. JML offers a greater variety of concepts on the implementation level, like exceptional behavior, modifies (assignable) clauses and loop invariants.

THE

THE

END