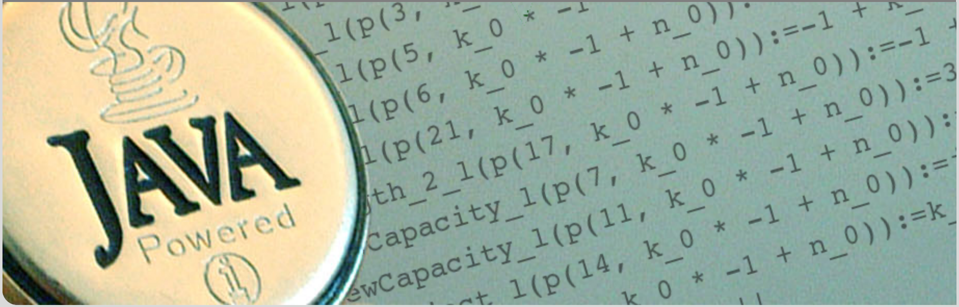
Applications of Formal Verification

Functional Verification of Java Programs: Java Modeling Language

Shmuel Tyszberowicz
based on materials by Bernhard Beckert · Vladimir Klebanov | 2014

MTA



Hoare Logic

Hoare triple

Has the form

$$\{\textit{Precondition}\}\ \text{Prog-Segment}\ \{\textit{Postcondition}\}$$

Expresses partial correctness

If the Prog-Segment starts with a state satisfying the precondition
and it terminates
then the final state of the Prog-Segment satisfies the postscondition

- Example
  - $\{X \geq 0\}$ X:=X+1; $\{X \geq 1\}$

Hoare Logic

Hoare triple

Has the form

$$\{\textit{Precondition}\}\ \text{Prog-Segment}\ \{\textit{Postcondition}\}$$

Expresses partial correctness

If the Prog-Segment starts with a state satisfying the precondition
and it terminates
then the final state of the Prog-Segment satisfies the postscondition

- Example
  - $\{X \geq 0\}$ X:=X+1; $\{X \geq 1\}$

Hoare Logic

Hoare triple

Has the form

$$\{\textit{Precondition}\}\ \text{Prog-Segment}\ \{\textit{Postcondition}\}$$

Expresses partial correctness

If the Prog-Segment starts with a state satisfying the precondition
and it terminates
then the final state of the Prog-Segment satisfies the postscondition

- Example
  - $\{X \geq 0\}$ X:=X+1; $\{X \geq 1\}$

Hoare Logic

Hoare triple

Has the form

$$\{\mathit{Precondition}\}\ \text{Prog-Segment}\ \{\mathit{Postcondition}\}$$

Expresses partial correctness

If the Prog-Segment starts with a state satisfying the precondition
and it terminates
then the final state of the Prog-Segment satisfies the postscondition

- Example
  - $\{X \geq 0\}$ X:=X+1; $\{X \geq 1\}$

Hoare Logic

Hoare triple

Has the form

$$\{\textit{Precondition}\}\ \text{Prog-Segment}\ \{\textit{Postcondition}\}$$

Expresses partial correctness

If the Prog-Segment starts with a state satisfying the precondition
and it terminates
then the final state of the Prog-Segment satisfies the postscondition

- Example
  - $\{X \geq 0\}$ X:=X+1; $\{X \geq 1\}$

Hoare Logic

Hoare triple

Has the form

{*Precondition*} Prog-Segment {*Postcondition*}

Expresses partial correctness

If the Prog-Segment starts with a state satisfying the precondition
and it terminates
then the final state of the Prog-Segment satisfies the postscondition

- Example
  - $\{X \geq 0\}$ X:=X+1; $\{X \geq 1\}$

Design by Contract

Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module)

If the caller guarantees precondition
then the callee guarantees postcondition

- Interface documentation
- Contracts described in a mathematically precise language
  - Higher degree of precision
  - *Automation* of program analysis of various kinds
    - Runtime assertion checking
    - Static verification
- Note: Errors in specifications are at least as common as errors in code

Design by Contract

Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module)

If the caller guarantees precondition
then the callee guarantees postcondition

- Interface documentation
- Contracts described in a mathematically precise language
  - Higher degree of precision
  - *Automation* of program analysis of various kinds
    - Runtime assertion checking
    - Static verification
- Note: Errors in specifications are at least as common as errors in code

Design by Contract

Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module)

If the caller guarantees precondition
then the callee guarantees postcondition

- Interface documentation
- Contracts described in a mathematically precise language
  - Higher degree of precision
  - *Automation* of program analysis of various kinds
    - Runtime assertion checking
    - Static verification
- Note: Errors in specifications are at least as common as errors in code

Design by Contract

Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module)

If the caller guarantees precondition
then the callee guarantees postcondition

- Interface documentation
- Contracts described in a mathematically precise language
  - Higher degree of precision
  - *Automation* of program analysis of various kinds
    - Runtime assertion checking
    - Static verification
- Note: Errors in specifications are at least as common as errors in code

Design by Contract

Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module)

If the caller guarantees precondition
then the callee guarantees postcondition

- Interface documentation
- Contracts described in a mathematically precise language
  - Higher degree of precision
  - *Automation* of program analysis of various kinds
    - Runtime assertion checking
    - Static verification
- Note: Errors in specifications are at least as common as errors in code

Design by Contract

TEL AVIV-YAFFO ACADEMIC COLLEGE

Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module)

If the caller guarantees precondition
then the callee guarantees postcondition

- Interface documentation
- Contracts described in a mathematically precise language
  - Higher degree of precision
  - *Automation* of program analysis of various kinds
    - Runtime assertion checking
    - Static verification
- Note: Errors in specifications are at least as common as errors in code

Design by Contract

Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module)

If the caller guarantees precondition
then the callee guarantees postcondition

- Interface documentation
- Contracts described in a mathematically precise language
  - Higher degree of precision
  - *Automation* of program analysis of various kinds
    - Runtime assertion checking
    - Static verification
- Note: Errors in specifications are at least as common as errors in code

Design by Contract

Idea

Specifications fix a contract between caller and callee of a method (between client and implementor of a module)

If the caller guarantees precondition
then the callee guarantees postcondition

- Interface documentation
- Contracts described in a mathematically precise language
  - Higher degree of precision
  - *Automation* of program analysis of various kinds
    - Runtime assertion checking
    - Static verification
- Note: Errors in specifications are at least as common as errors in code

JML Annotations

```
/*@ public normal_behavior
  @   requires pin == correctPin;
  @   ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with '@' as first character are JML specifications
- Within a JML annotation, an '@' is ignored:
  - If it is the first (non-white) character in the line
  - If it is the last character before '*/'

  ⇒ The blue '@'s are not required, but it is a *convention* to use them
- JML specifications may themselves contain comments

JML Annotations

```
/*@ public normal_behavior
  @   requires pin == correctPin;
  @   ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with '@' as first character are JML specifications
- Within a JML annotation, an '@' is ignored:
  - If it is the first (non-white) character in the line
  - If it is the last character before '*/'

  ⇒ The blue '@'s are not required, but it is a *convention* to use them
- JML specifications may themselves contain comments

JML Annotations

```
/*@ public normal_behavior
  @    requires pin == correctPin;
  @    ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with '@' as first character are JML specifications
- Within a JML annotation, an '@' is ignored:
  - If it is the first (non-white) character in the line
  - If it is the last character before '*/'

$\Rightarrow$ The blue '@'s are not required, but it is a *convention* to use them

- JML specifications may themselves contain comments

JML Annotations

```
/*@ public normal_behavior
  @   requires pin == correctPin;
  @   ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with '@' as first character are JML specifications
- Within a JML annotation, an '@' is ignored:
  - If it is the first (non-white) character in the line
  - If it is the last character before '*/'

  ⇒ The blue '@'s are not required, but it is a *convention* to use them
- JML specifications may themselves contain comments

JML Annotations

```
/*@ public normal_behavior
  @   requires pin == correctPin;
  @   ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with '@' as first character are JML specifications
- Within a JML annotation, an '@' is ignored:
  - If it is the first (non-white) character in the line
  - If it is the last character before '*/'

  ⇒ The blue '@'s are not required, but it is a *convention* to use them
- JML specifications may themselves contain comments

JML Annotations

```
/*@ public normal_behavior              // Comment
  @    requires pin == correctPin;
  @    ensures customerAuthenticated;
  @*/
public void enterPIN (int pin) {
    ...
```

- Java comments with '@' as first character are JML specifications
- Within a JML annotation, an '@' is ignored:
  - If it is the first (non-white) character in the line
  - If it is the last character before '*/'

  ⇒ The blue '@'s are not required, but it is a *convention* to use them
- JML specifications may themselves contain comments

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
- *public* specification items cannot refer to *private* fields
- Private fields can be declared public for specification purposes only

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
- *public* specification items cannot refer to *private* fields
- Private fields can be declared public for specification purposes only

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
- *public* specification items cannot refer to *private* fields
- Private fields can be declared public for specification purposes only

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
- *public* specification items cannot refer to *private* fields
- Private fields can be declared public for specification purposes only

Method Contracts

```
/*@ requires pre;
  @ ensures post;
  @ assignable a;
  @ diverges d;
  @ signals_only E1,...,En;
  @ signals(E e) s;
  @*/
T m(...);
```

Implicitly

```
      normal_behavior  =  signals(Exception e) false;
exceptional_behavior  =  ensures false;
```

The keyword `also` separates the contracts of a method

Method Contracts

```
/*@ requires pre; //what is the caller's obligation?
  @ ensures post;
  @ assignable a;
  @ diverges d;
  @ signals_only E1,...,En;
  @ signals(E e) s;
  @*/
T m(...);
```

Implicitly

```
    normal_behavior  =  signals(Exception e) false;
exceptional_behavior  =  ensures false;
```

The keyword `also` separates the contracts of a method

Method Contracts

```
/*@ requires pre; //what is the caller's obligation?
  @ ensures post; //what must hold on normal termination?
  @ assignable a;
  @ diverges d;
  @ signals_only E1,...,En;
  @ signals(E e) s;
  @*/
T m(...);
```

Implicitly

```
    normal_behavior  =  signals(Exception e) false;
exceptional_behavior =  ensures false;
```

The keyword `also` separates the contracts of a method

Method Contracts

```
/*@ requires pre; //what is the caller's obligation?
  @ ensures post; //what must hold on normal termination?
  @ assignable a; //which locations may be assigned by m?
  @ diverges d;
  @ signals_only E1,...,En;
  @ signals(E e) s;
  @*/
T m(...);
```

Implicitly

```
   normal_behavior  =  signals(Exception e) false;
exceptional_behavior  =  ensures false;
```

The keyword `also` separates the contracts of a method

Method Contracts

```
/*@ requires pre;   //what is the caller's obligation?
  @ ensures post;   //what must hold on normal termination?
  @ assignable a;   //which locations may be assigned by m?
  @ diverges d;     //when may m non-terminate?
  @ signals_only E1,...,En;
  @ signals(E e) s;
  @*/
T m(...);
```

Implicitly

```
   normal_behavior  =  signals(Exception e) false;
exceptional_behavior  =  ensures false;
```

The keyword also separates the contracts of a method

Method Contracts

```
/*@ requires r;   //what is the caller's obligation?
  @ ensures post; //what must hold on normal termination?
  @ assignable a; //which locations may be assigned by m?
  @ diverges d;   //when may m non-terminate?
  @ signals_only E1,...,En; //what exc-types may be thrown?
  @ signals(E e) s;
  @*/
T m(...);
```

Implicitly

```
      normal_behavior  =  signals(Exception e) false;
exceptional_behavior  =  ensures false;
```

The keyword `also` separates the contracts of a method

Method Contracts

```
/*@ requires pre; //what is the caller's obligation?
  @ ensures post;  //what must hold on normal termination?
  @ assignable a; //which locations may be assigned by m?
  @ diverges d;   //when may m non-terminate?
  @ signals_only E1,...,En; //what exc-types may be thrown?
  @ signals(E e) s; //what must hold when an E is thrown?
  @*/
T m(...);
```

Implicitly

```
      normal_behavior  =  signals(Exception e) false;
exceptional_behavior  =  ensures false;
```

The keyword `also` separates the contracts of a method

Method Contracts

```
/*@ requires pre; //what is the caller's obligation?
  @ ensures post;  //what must hold on normal termination?
  @ assignable a; //which locations may be assigned by m?
  @ diverges d;   //when may m non-terminate?
  @ signals_only E1,...,En; //what exc-types may be thrown?
  @ signals(E e) s; //what must hold when an E is thrown?
  @*/
T m(...);
```

Implicitly

```
     normal_behavior  =  signals(Exception e) false;
exceptional_behavior  =  ensures false;
```

The keyword `also` separates the contracts of a method

Method Contracts

```
/*@ requires pre; //what is the caller's obligation?
  @ ensures post;  //what must hold on normal termination?
  @ assignable a; //which locations may be assigned by m?
  @ diverges d;   //when may m non-terminate?
  @ signals_only E1,...,En; //what exc-types may be thrown?
  @ signals(E e) s; //what must hold when an E is thrown?
  @*/
T m(...);
```

Implicitly

```
     normal_behavior  =  signals(Exception e) false;
exceptional_behavior  =  ensures false;
```

The keyword **also** separates the contracts of a method

Method Contracts

```
/*@ requires pre; //what is the caller's obligation?
  @ ensures post;  //what must hold on normal termination?
  @ assignable a; //which locations may be assigned by m?
  @ diverges d;   //when may m non-terminate?
  @ signals_only E1,...,En; //what exc-types may be thrown?
  @ signals(E e) s; //what must hold when an E is thrown?
  @*/
T m(...);
```

Implicitly

```
     normal_behavior  =  signals(Exception e) false;
exceptional_behavior  =  ensures false;
```

The keyword **also** separates the contracts of a method

Class Invariants

```
//@ invariant i;
```

- Express global consistency properties
  - Not specific to a particular method
- Must always hold

Class Invariants

```
//@ invariant i;
```

- Express global consistency properties
  - Not specific to a particular method
- Must always hold

Class Invariants

```
//@ invariant i;
```

- Express global consistency properties
  - Not specific to a particular method
- Must always hold

Pure Methods

Pure methods terminate and have no side effects

'**pure**' ≈ '**diverges false;**' + '**assignable \nothing;**'

After declaring

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

`cardIsInserted()`

could replace

`insertedCard != null`

in JML annotations

Pure Methods

Pure methods terminate and have no side effects

'`pure`' ≈ '`diverges false;`' + '`assignable \nothing;`'

After declaring

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

`cardIsInserted()`

could replace

`insertedCard != null`

in JML annotations

Pure Methods

Pure methods terminate and have no side effects

‘**pure**’ ≈ ‘**diverges false;**’ + ‘**assignable \nothing;**’

After declaring

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

`cardIsInserted()`

could replace

`insertedCard != null`

in JML annotations

Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- `\forall`, `\exists`
- `\num_of`, `\sum`, `\product`, `\min`, `\max`
- `\old`(...): referring to pre-state in postconditions
- `\result`: referring to return value in postconditions

Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- `\forall`, `\exists`
- `\num_of`, `\sum`, `\product`, `\min`, `\max`
- `\old`(...): referring to pre-state in postconditions
- `\result`: referring to return value in postconditions

Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- **`\forall`**, **`\exists`**
- **`\num_of`**, **`\sum`**, **`\product`**, **`\min`**, **`\max`**
- **`\old`**(...): referring to pre-state in postconditions
- **`\result`**: referring to return value in postconditions

Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- **`\forall`**, **`\exists`**
- **`\num_of`**, **`\sum`**, **`\product`**, **`\min`**, **`\max`**
- **`\old`**(...): referring to pre-state in postconditions
- **`\result`**: referring to return value in postconditions

Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- `\forall`, `\exists`
- `\num_of`, `\sum`, `\product`, `\min`, `\max`
- `\old`(...): referring to pre-state in postconditions
- `\result`: referring to return value in postconditions

Expressions

- All Java expressions without side-effects
- `==>`, `<==>`: implication, equivalence
- **`\forall`**, **`\exists`**
- **`\num_of`**, **`\sum`**, **`\product`**, **`\min`**, **`\max`**
- **`\old`**(...): referring to pre-state in postconditions
- **`\result`**: referring to return value in postconditions

Quantification in JML

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)
```

```
(\exists int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\exists int i; 0<=i && i<\result.length && \result[i]>0)
```

- Quantifiers bind two expressions: the range predicate and the body expression
- A missing range predicate is by default `true`

Quantification in JML

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)
```

```
(\exists int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\exists int i; 0<=i && i<\result.length && \result[i]>0)
```

- Quantifiers bind two expressions: the range predicate and the body expression
- A missing range predicate is by default `true`

Quantification in JML

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

equivalent to

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

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

equivalent to

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

- Quantifiers bind two expressions: the range predicate and the body expression
- A missing range predicate is by default `true`

Quantification in JML

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)
```

```
(\exists int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\exists int i; 0<=i && i<\result.length && \result[i]>0)
```

- Quantifiers bind two expressions: the range predicate and the body expression
- A missing range predicate is by default `true`

Quantification in JML

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)
```

```
(\exists int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\exists int i; 0<=i && i<\result.length && \result[i]>0)
```

- Quantifiers bind two expressions: the range predicate and the body expression
- A missing range predicate is by default `true`

Quantification in JML

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)
```

```
(\exists int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\exists int i; 0<=i && i<\result.length && \result[i]>0)
```

- Quantifiers bind two expressions: the range predicate and the body expression
- A missing range predicate is by default `true`

Quantification in JML

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)
```

```
(\exists int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\exists int i; 0<=i && i<\result.length && \result[i]>0)
```

- Quantifiers bind two expressions: the range predicate and the body expression
- A missing range predicate is by default `true`

Quantification in JML

```
(\forall int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\forall int i; 0<=i && i<\result.length ==> \result[i]>0)
```

```
(\exists int i; 0<=i && i<\result.length; \result[i]>0)
```
equivalent to
```
(\exists int i; 0<=i && i<\result.length && \result[i]>0)
```

- Quantifiers bind two expressions: the range predicate and the body expression
- A missing range predicate is by default `true`

Generalized and Numerical Quantifiers

`(\num_of T i; e)`	$\#\{i \mid [e]\}$, number of elements of type `T` with property `e`
`(\sum T i; p; t)`	$\sum_{i:[p]} [t]$
`(\product T i; p; t)`	$\prod_{i:[p]} [t]$
`(\min T i; p; t)`	$\mathit{min}_{i:[p]}\{[t]\}$
`(\max T i; p; t)`	$\mathit{max}_{i:[p]}\{[t]\}$

e.g.: ensures \result==(\num_of int j; 0$\leq$ j && j <n; a[j] == i);

The `assignable` Clause

- Specifies frame conditions
- Locations in this (comma-separated) may be assigned to
- List of:
  - `e.f` (where `f` a field)

The `assignable` Clause

- Specifies frame conditions
- Locations in this (comma-separated) may be assigned to
- List of:
  - `e.f` (where `f` a field)
  - `a[*]`, `a[x..y]` (where `a` an array expression)

The `assignable` Clause

- Specifies frame conditions
- Locations in this (comma-separated) may be assigned to
- List of:
  - `e.f` (where `f` a field)
  - `a[*]`, `a[x..y]` (where `a` an array expression)
  - **`\nothing`**, **`\everything`** (default)

The `diverges` Clause

```
diverges e;
```

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

Examples

```
diverges false;
```

The method must always terminate

```
diverges true;
```

The method may terminate or not

```
diverges n == 0;
```

The method must terminate, when called in a state with `n!=0`

The `diverges` Clause

```
    diverges e;
```

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

Examples

```
    diverges false;
```

The method must always terminate

```
    diverges true;
```

The method may terminate or not

```
    diverges n == 0;
```

The method must terminate, when called in a state with `n!=0`

The `diverges` Clause

```
diverges e;
```

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

Examples

```
diverges false;
```

The method must always terminate

```
diverges true;
```

The method may terminate or not

```
diverges n == 0;
```

The method must terminate, when called in a state with `n!=0`

The `diverges` Clause

```
diverges e;
```

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

Examples

```
diverges false;
```

The method must always terminate

```
diverges true;
```

The method may terminate or not

```
diverges n == 0;
```

The method must terminate, when called in a state with `n!=0`

The `signals` Clauses

```
ensures p;
signals_only ET1, ..., ETm;
signals (E1 e1) s1;
...
signals (En en) sn;
```

- Normal termination $\Rightarrow$ `p` must hold (in post-state)
- Exception thrown $\Rightarrow$ must be of type `ET1`, ..., or `ETm`
- Exception of type `E1` thrown $\Rightarrow$ `s1` must hold (in post-state)

  ...
- Exception of type `En` thrown $\Rightarrow$ `sn` must hold (in post-state)

The `signals` Clauses

```
ensures p;
signals_only ET1, ..., ETm;
signals (E1 e1) s1;
...
signals (En en) sn;
```

- Normal termination ⇒ `p` must hold (in post-state)
- Exception thrown ⇒ must be of type `ET1`, ..., or `ETm`
- Exception of type `E1` thrown ⇒ `s1` must hold (in post-state)

  ...
- Exception of type `En` thrown ⇒ `sn` must hold (in post-state)

The `signals` Clauses

```
ensures p;
signals_only ET1, ..., ETm;
signals (E1 e1) s1;
...
signals (En en) sn;
```

- Normal termination ⇒ `p` must hold (in post-state)
- Exception thrown ⇒ must be of type `ET1`, ..., or `ETm`
- Exception of type `E1` thrown ⇒ `s1` must hold (in post-state)

  ...
- Exception of type `En` thrown ⇒ `sn` must hold (in post-state)

The `signals` Clauses

```
ensures p;
signals_only ET1, ..., ETm;
signals (E1 e1) s1;
...
signals (En en) sn;
```

- Normal termination $\Rightarrow$ `p` must hold (in post-state)
- Exception thrown $\Rightarrow$ must be of type `ET1`, ..., or `ETm`
- Exception of type `E1` thrown $\Rightarrow$ `s1` must hold (in post-state)

  ...
- Exception of type `En` thrown $\Rightarrow$ `sn` must hold (in post-state)

The `signals` Clauses

```
ensures p;
signals_only ET1, ..., ETm;
signals (E1 e1) s1;
...
signals (En en) sn;
```

- Normal termination ⇒ `p` must hold (in post-state)
- Exception thrown ⇒ must be of type `ET1`, . . . , or `ETm`
- Exception of type `E1` thrown ⇒ `s1` must hold (in post-state)

  . . .
- Exception of type `En` thrown ⇒ `sn` must hold (in post-state)

Model Fields

```
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

Model Fields

```
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

Model Fields

```
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

Model Fields

```
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

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

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

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

```
public interface IBonusCard {
   /*@ public instance model int bonusPoints; @*/

   /*@ ... @*/
   public void addBonus(int newBonusPoints);
```

Implementation

```
public class BankCard implements IBonusCard{
      public int bankCardPoints;
/*@   private represents bonusPoints = bankCardPoints; @*/

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

Implementing Interfaces

```
public interface IBonusCard {
   /*@ public instance model int bonusPoints; @*/

   /*@ ... @*/
   public void addBonus(int newBonusPoints);

```

Implementation

```
public class BankCard implements IBonusCard{
     public int bankCardPoints;


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

Implementing Interfaces

```
public interface IBonusCard {
   /*@ public instance model int bonusPoints; @*/

   /*@ ... @*/
   public void addBonus(int newBonusPoints);

```

Implementation

```
public class BankCard implements IBonusCard{
     public int bankCardPoints;
/*@  private represents bonusPoints = bankCardPoints; @*/

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

Other Representations

```
/*@ private represents bonusPoints
            = bankCardPoints; @*/
```

```
/*@ private represents bonusPoints
            = bankCardPoints * 100; @*/
```

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

Other Representations

```
/*@ private represents bonusPoints
              = bankCardPoints; @*/
```

```
/*@ private represents bonusPoints
              = bankCardPoints * 100; @*/
```

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

Other Representations

```
/*@ private represents bonusPoints
            = bankCardPoints; @*/
```

```
/*@ private represents bonusPoints
            = bankCardPoints * 100; @*/
```

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

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
    - It can be extended there

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
  - It can be extended there

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
    - It can be extended there

Other JML Features

- loop invariants `//@ loop_invariant p;`
- data groups
- assertions `//@ assert e;`
- many more…

Other JML Features

- loop invariants `//@ loop_invariant p;`
- data groups
- assertions `//@ assert e;`
- many more…

Other JML Features

- loop invariants `//@ loop_invariant p;`
- data groups
- assertions `//@ assert e;`
- many more. . .

Other JML Features

- loop invariants `//@ loop_invariant p;`
- data groups
- assertions `//@ assert e;`
- many more...

Problems with Specifications Using Integers

```
/*@ requires y >= 0;
  @ ensures \result >= 0;
  @ ensures \result * \result <= y;
  @ ensures (\result+1) * (\result+1) > y;
  @ */
 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 1073741821 is not a square root of 1

JML uses the Java semantics of integers:

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

The JML type `\bigint` provides arbitrary precision integers

Problems with Specifications Using Integers

```
/*@ requires y >= 0;
  @ ensures \result >= 0;
  @ ensures \result * \result <= y;
  @ ensures (\result+1) * (\result+1) > y;
  @ */
 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 1073741821 is not a square root of 1

JML uses the Java semantics of integers:

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

The JML type `\bigint` provides arbitrary precision integers

Problems with Specifications Using Integers

```
/*@ requires y >= 0;
  @ ensures \result >= 0;
  @ ensures \result * \result <= y;
  @ ensures (\result+1) * (\result+1) > y;
  @ */
 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 1073741821 is not a square root of 1

JML uses the Java semantics of integers:

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

The JML type `\bigint` provides arbitrary precision integers

Problems with Specifications Using Integers

```
/*@ requires y >= 0;
  @ ensures \result >= 0;
  @ ensures \result * \result <= y;
  @ ensures (\result+1) * (\result+1) > y;
  @ */
 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 1073741821 is not a square root of 1

JML uses the Java semantics of integers:

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

The JML type `\bigint` provides arbitrary precision integers

JML Tools

Many tools support JML (see JML homepage); among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- `rac`: runtime assertion checker
- ESC/Java2: leightweight static verification
- KeY: full static verification
- ...

Tools do not yet support the new features of Java 5 and higher
e.g.: no generics, no enums, no enhanced for-loops, no autoboxing

JML Tools

Many tools support JML (see JML homepage); among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- `rac`: runtime assertion checker
- ESC/Java2: leightweight static verification
- KeY: full static verification
- ...

Tools do not yet support the new features of Java 5 and higher

e.g.: no generics, no enums, no enhanced for-loops, no autoboxing

JML Tools

Many tools support JML (see JML homepage); among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- `rac`: runtime assertion checker
- ESC/Java2: leightweight static verification
- KeY: full static verification
- ...

Tools do not yet support the new features of Java 5 and higher
e.g.: no generics, no enums, no enhanced for-loops, no autoboxing

JML Tools

Many tools support JML (see JML homepage); among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- `rac`: runtime assertion checker
- ESC/Java2: leightweight static verification
- KeY: full static verification
- ...

Tools do not yet support the new features of Java 5 and higher
e.g.: no generics, no enums, no enhanced for-loops, no autoboxing

JML Tools

Many tools support JML (see JML homepage); among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- `rac`: runtime assertion checker
- ESC/Java2: leightweight static verification
- KeY: full static verification
- ...

Tools do not yet support the new features of Java 5 and higher
e.g.: no generics, no enums, no enhanced for-loops, no autoboxing

JML Tools

Many tools support JML (see JML homepage); among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- `rac`: runtime assertion checker
- ESC/Java2: leightweight static verification
- KeY: full static verification
- ...

Tools do not yet support the new features of Java 5 and higher
e.g.: no generics, no enums, no enhanced for-loops, no autoboxing

JML Tools

Many tools support JML (see JML homepage); among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- `rac`: runtime assertion checker
- ESC/Java2: leightweight static verification
- KeY: full static verification
- ...

Tools do not yet support the new features of Java 5 and higher

e.g.: no generics, no enums, no enhanced for-loops, no autoboxing

JML Tools

Many tools support JML (see JML homepage); among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- `rac`: runtime assertion checker
- ESC/Java2: leightweight static verification
- KeY: full static verification
- …

Tools do not yet support the new features of Java 5 and higher
e.g.: no generics, no enums, no enhanced for-loops, no autoboxing

JML Tools

Many tools support JML (see JML homepage); among them:

- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlc`: compiles Java+JML into bytecode with assertion checks
- `jmlunit`: unit testing (like JUnit)
- `rac`: runtime assertion checker
- ESC/Java2: leightweight static verification
- KeY: full static verification
- …

Tools do not yet support the new features of Java 5 and higher
e.g.: no generics, no enums, no enhanced for-loops, no autoboxing