Specification & Formal Analysis of Java Programs

Java Modelling Language

Prof. Dr. Bernhard Beckert | ADAPT 2010
Design by Contract

Idea

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

If caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation
- Contracts described in a mathematically precise language (JML)
  - 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 caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation
- Contracts described in a mathematically precise language (JML)
  - 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 caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation
- Contracts described in a mathematically precise language (JML)
  - 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 caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation
- Contracts described in a mathematically precise language (JML)
  - 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 caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation
- Contracts described in a mathematically precise language (JML)
  - 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 caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation
- Contracts described in a mathematically precise language (JML)
  - 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 caller guarantees precondition
then callee guarantees certain outcome

- Interface documentation
- Contracts described in a mathematically precise language (JML)
  - 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

```java
/*@ 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
- JML specifications may themselves contain comments
JML Annotations

```java
/*@ 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
- JML specifications may themselves contain comments
JML Annotations

```java
/*@ 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
- JML specifications may themselves contain comments
JML Annotations

```java
/*@ 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
- JML specifications may themselves contain comments
JML Annotations

```java
/*@ 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
- JML specifications may themselves contain comments
JML Annotations

```java
/*@ public normal_behavior //<hello!>
    @ 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
- JML specifications may themselves contain comments
Visibility Modifiers

```java
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.
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

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

Abbreviations

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

keyword 'also' separates the contracts of a method
Method Contracts

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

Abbreviations

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

Keyword ‘also’ separates the contracts of a method
Method Contracts

/*@ requires r; // what is the caller’s obligation? @ assignable a; // which locations may be assigned by m? @ diverges d; @ ensures post; @ signals_only E1,...,En; @ signals(E e) s; @*/

T m(...);

Abbreviations

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

keyword ‘also’ separates the contracts of a method
Method Contracts

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

Abbreviations

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

keyword 'also' separates the contracts of a method
Method Contracts

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

reducers r; //what is the caller’s obligation?
assignable a; //which locations may be assigned by m?
diverges d; //when may m non-terminate?
ensures post; //what must hold on normal termination?
signals_only E1, ..., En; //what exc-types may be thrown?
signals(E e) s;

T m(...);

Abbreviations

normal_behavior = signals(Exception) false;

exceptional_behavior = ensures false;

keyword ‘also’ separates the contracts of a method
Method Contracts

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

Abbreviations

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

keyword ‘also’ separates the contracts of a method
Method Contracts

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

Abbreviations

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

keyword ‘also’ separates the contracts of a method
Method Contracts

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

Abbreviations

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

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

Abbreviations

normal behavior = signals(Exception) false;
exceptional behavior = ensures false;

keyword ‘also’ separates the contracts of a method
Class Invariants

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

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)
- must hold “always”
  (cf. visible state semantics, observed state semantics)
- instance invariants can, static invariants cannot refer to this
- default: instance within classes, static within interfaces
Class Invariants

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

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)
- must hold “always”
  (cf. visible state semantics, observed state semantics)
- instance invariants can, static invariants cannot refer to this
- default: instance within classes, static within interfaces
Class Invariants

//@ invariant i;

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)
- must hold “always” (cf. visible state semantics, observed state semantics)
- instance invariants can, static invariants cannot refer to this
- default: instance within classes, static within interfaces
Class Invariants

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

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)
- must hold “always”
  (cf. visible state semantics, observed state semantics)
- instance invariants can, static invariants cannot refer to this
- default: instance within classes, static within interfaces
Class Invariants

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

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)
- must hold “always” (cf. visible state semantics, observed state semantics)
- `instance` invariants can, `static` invariants cannot refer to `this`
- default: `instance` within classes, `static` within interfaces
Class Invariants

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

- can be placed anywhere in a class (or interface)
- express global consistency properties (not specific to a particular method)
- must hold “always”
  (cf. visible state semantics, observed state semantics)
- `instance` invariants can, `static` invariants cannot refer to `this`
- default: `instance` within classes, `static` within interfaces
Pure Methods

Pure methods terminate and have no side effects.

After declaring

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

cardIsInserted() could replace

```java
insertedCard != null
```

in JML annotations.
Pure Methods

Pure methods terminate and have no side effects.

After declaring

```java
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.

After declaring

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

cardIsInserted() could replace

```java
insertedCard != null
```

in JML annotations.
Pure Methods

\texttt{\textquotesingle{}pure\textquotesingle{} \textasciitilde \textquotesingle{}diverges false;\textquotesingle{} + \textquotesingle{}assignable \textbackslash nothing;\textquotesingle{}}
Expressions

- All Java expressions without side-effects
- $\implies$, $\iff$: implication, equivalence
- $\forall$, $\exists$
- $\text{num of}$, $\text{sum}$, $\text{product}$, $\text{min}$, $\text{max}$
- $\text{old(...)}$: referring to pre-state in postconditions
- $\text{result}$: referring to return value in postconditions
Expressions

- All Java expressions without side-effects
- $\Rightarrow$, $\Leftrightarrow$: implication, equivalence
- $\forall$, $\exists$
- \texttt{num_of}, \texttt{sum}, \texttt{product}, \texttt{min}, \texttt{max}
- \texttt{old(...)}: referring to pre-state in postconditions
- \texttt{result}: referring to return value in postconditions
Expressions

- All Java expressions without side-effects
- $\Rightarrow$, $\Leftrightarrow$: implication, equivalence
- $\forall$, $\exists$
- $\text{num.of}$, $\text{sum}$, $\text{product}$, $\text{min}$, $\text{max}$
- $\text{old(...)}$: referring to pre-state in postconditions
- $\text{result}$: referring to return value in postconditions
Expressions

- All Java expressions without side-effects
- $\implies$, $\iff$: implication, equivalence
- \(\forall\), \(\exists\)
- \(\numof\), \(\sum\), \(\product\), \(\min\), \(\max\)
- \(\old(...)\): referring to pre-state in postconditions
- \(\result\): referring to return value in postconditions
Expressions

- All Java expressions without side-effects
- $\Rightarrow, \Leftrightarrow$: implication, equivalence
- $\forall, \exists$
- $\text{num\_of}, \text{sum}, \text{product}, \text{min}, \text{max}$
- $\text{old}(\ldots)$: referring to pre-state in postconditions
- $\text{result}$: referring to return value in postconditions
Expressions

- All Java expressions without side-effects
- $\implies$, $\iff$: implication, equivalence
- $\forall$, $\exists$
- $\text{num.of}$, $\text{sum}$, $\text{product}$, $\text{min}$, $\text{max}$
- $\text{old}(...)$: referring to pre-state in postconditions
- $\text{result}$: referring to return value in postconditions
Quantification in JML

\(\forall \text{int } i; 0 \leq i \land i < \text{result}.\text{length}; \ \text{result}[i] > 0\)
equivalent to
\(\forall \text{int } i; 0 \leq i \land i < \text{result}.\text{length} \implies \text{result}[i] > 0\)

\(\exists \text{int } i; 0 \leq i \land i < \text{result}.\text{length}; \ \text{result}[i] > 0\)
equivalent to
\(\exists \text{int } i; 0 \leq i \land i < \text{result}.\text{length} \land \text{result}[i] > 0\)

- 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.
Quantification in JML

\( (\forall \text{int } i; 0 \leq i \land i < \text{result.length}; \ \text{result}[i] > 0) \)

equivalent to

\( (\forall \text{int } i; 0 \leq i \land i < \text{result.length} \implies \text{result}[i] > 0) \)

\( (\exists \text{int } i; 0 \leq i \land i < \text{result.length}; \ \text{result}[i] > 0) \)

equivalent to

\( (\exists \text{int } i; 0 \leq i \land i < \text{result.length} \land \text{result}[i] > 0) \)

- 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.
Quantification in JML

\( \forall i \text{ int}; \ 0 \leq i < \text{result}.\text{length}; \ \text{result}[i] > 0 \) equivalent to
\( \forall i \text{ int}; \ 0 \leq i < \text{result}.\text{length} \implies \text{result}[i] > 0 \)

\( \exists i \text{ int}; \ 0 \leq i < \text{result}.\text{length}; \ \text{result}[i] > 0 \) equivalent to
\( \exists i \text{ int}; \ 0 \leq i < \text{result}.\text{length} \land \ \text{result}[i] > 0 \)

- Note that quantifiers bind two expressions, the range predicate and the body expression.
- A missing range predicate is by default \text{true}.
- JML excludes \text{null} from the range of quantification.
Quantification in JML

\[ (\forall \texttt{int } i; \ 0 \leq i \land i < \texttt{result}.length; \ \texttt{result}[i] > 0) \]
equivalent to

\[ (\forall \texttt{int } i; \ 0 \leq i \land i < \texttt{result}.length \implies \texttt{result}[i] > 0) \]

\[ (\exists \texttt{int } i; \ 0 \leq i \land i < \texttt{result}.length; \ \texttt{result}[i] > 0) \]
equivalent to

\[ (\exists \texttt{int } i; \ 0 \leq i \land i < \texttt{result}.length \land \texttt{result}[i] > 0) \]

- 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.
Quantification in JML

(\forall \text{int } i; \ 0 \leq i \land i < \text{result}.\text{length}; \ \text{result}[i] > 0)
equivalent to
(\forall \text{int } i; \ 0 \leq i \land i < \text{result}.\text{length} \implies \text{result}[i] > 0)

(\exists \text{int } i; \ 0 \leq i \land i < \text{result}.\text{length}; \ \text{result}[i] > 0)
equivalent to
(\exists \text{int } i; \ 0 \leq i \land i < \text{result}.\text{length} \land \text{result}[i] > 0)

- 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.
Quantification in JML

\(\forall i; 0 \leq i < \text{result}.\text{length}; \ \text{result}[i] > 0\) equivalent to \\
\(\forall i; 0 \leq i < \text{result}.\text{length} \implies \text{result}[i] > 0\)

\(\exists i; 0 \leq i < \text{result}.\text{length}; \ \text{result}[i] > 0\) equivalent to \\
\(\exists i; 0 \leq i < \text{result}.\text{length} \land \text{result}[i] > 0\)

- 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.
Quantification in JML

\( (\forall \text{int } i; \ 0 \leq i \land i < \text{result}.\text{length}; \ \text{result}[i] > 0) \)

equivalent to

\( (\forall \text{int } i; \ 0 \leq i \land i < \text{result}.\text{length} \implies \text{result}[i] > 0) \)

\( (\exists \text{int } i; \ 0 \leq i \land i < \text{result}.\text{length}; \ \text{result}[i] > 0) \)

equivalent to

\( (\exists \text{int } i; \ 0 \leq i \land i < \text{result}.\text{length} \land \text{result}[i] > 0) \)

- 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.
Quantification in JML

\( \forall i; 0 \leq i < \text{result}.\text{length}; \ \text{result}[i] > 0 \)
equivalent to
\( \forall i; 0 \leq i < \text{result}.\text{length} \implies \text{result}[i] > 0 \)

\( \exists i; 0 \leq i < \text{result}.\text{length}; \ \text{result}[i] > 0 \)
equivalent to
\( \exists i; 0 \leq i < \text{result}.\text{length} \land \text{result}[i] > 0 \)

- 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

\(\num_of C c; e\)

\(\sum C c; p; t\)

\(\product C c; p; t\)

\(\min C c; p; t\)

\(\max C c; p; t\)

\(\#\{c|e\}\), number of elements of class \(C\) with property \(e\)

\(\sum [t]_{c:[p]}\)

\(\prod [t]_{c:[p]}\)

\(min\{[t]\} \quad c:[p]\)

\(max\{[t]\} \quad c:[p]\)
Generalised and Numerical Quantifiers

\( \text{num\_of } C \ c; \ e \)  \#\{c\|[e]\},\text{number of elements of class } C \text{ with property } e

\( \sum C \ c; \ p; \ t \)  \( \sum_{c:[p]} [t] \)

\( \text{product } C \ c; \ p; \ t \)  \( \prod_{c:[p]} [t] \)

\( \text{min } C \ c; \ p; \ t \)  \( \min_{c:[p]} \{[t]\} \)

\( \text{max } C \ c; \ p; \ t \)  \( \max_{c:[p]} \{[t]\} \)
Generalised and Numerical Quantifiers

(\texttt{\num_of C c; e}) \quad \#\{c|e\}, number of elements of class \( C \) with property \( e \)

(\texttt{\sum C c; p; t}) \quad \sum_{c:[p]} [t]

(\texttt{\product C c; p; t}) \quad \prod_{c:[p]} [t]

(\texttt{\min C c; p; t}) \quad min\{[t]\}

(\texttt{\max C c; p; t}) \quad max\{[t]\}
Generalised and Numerical Quantifiers

\( \text{num}_\text{of} \ C \ c; \ e \) #\{c\mid e\}, number of elements of class \( C \) with property \( e \)

\( \text{sum} \ C \ c; \ p; \ t \) \( \sum_{c:p} [t] \)

\( \text{product} \ C \ c; \ p; \ t \) \( \prod_{c:p} [t] \)

\( \text{min} \ C \ c; \ p; \ t \) \( \min_{c:p} \{[t]\} \)

\( \text{max} \ C \ c; \ p; \ t \) \( \max_{c:p} \{[t]\} \)
Generalised and Numerical Quantifiers

\( (\text{num}_\text{of }\{c|e\}) \), number of elements of class \( C \) with property \( e \)

\( (\text{sum }\{c|p|t\}) \)

\( (\text{product }\{c|p|t\}) \)

\( (\text{min }\{c|p|t\}) \)

\( (\text{max }\{c|p|t\}) \)
The assignable Clauses

Comma-separated list of:
- e.f (where f a field)
- a[*], a[x..y] (where a an array expression)
- nothing, everything (default)

Example

```java
C x, y;
//@ assignable x, x.i;
void m() {
    C tmp = x;   //allowed (local variable)
    tmp.i = 27;  //allowed (in assignable clause)
    x = y;       //allowed (in assignable clause)
    x.i = 27;    //forbidden (not local, not in assignable)
}
```
The assignable Clauses

Comma-separated list of:

- e.f (where f a field)
- a[ ], a[x..y] (where a an array expression)
- nothing, everything (default)

Example

```java
C x, y;
//@ assignable x, x.i;
void m() {
    C tmp = x;  //allowed (local variable)
    tmp.i = 27;  //allowed (in assignable clause)
    x = y;       //allowed (in assignable clause)
    x.i = 27;    //forbidden (not local, not in assignable)
}
```
The assignable Clauses

Comma-separated list of:
- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where `a` an array expression)
- `\nothing`, `\everything` (default)

Example

```java
C x, y;
//@ assignable x, x.i;
void m() {
    C tmp = x;  //allowed (local variable)
    tmp.i = 27; //allowed (in assignable clause)
    x = y;      //allowed (in assignable clause)
    x.i = 27;   //forbidden (not local, not in assignable)
}
```
The assignable Clauses

Comma-separated list of:
- e.f (where f a field)
- a[*], a[x..y] (where a an array expression)
- \nothing, \everything (default)

Example

```java
C x, y;
//@ assignable x, x.i;
void m() {
}
```
The assignable Clauses

Comma-separated list of:
- e.f (where f a field)
- a[∗], a[x..y] (where a an array expression)
- \nothing, \everything (default)

Example

```java
C x, y;
//@ assignable x, x.i;

void m() {
    C tmp = x;
    tmp.i = 27;
    x = y;
    x.i = 27;
}
```
The assignable Clauses

Comma-separated list of:

- e.f (where f a field)
- a[*], a[x..y] (where a an array expression)
- \nothing, \everything (default)

Example

```java
C x, y;
//@ assignable x, x.i;
void m() {
    C tmp = x;  //allowed (local variable)
    tmp.i = 27;
    x = y;
    x.i = 27;
}
```
The assignable Clauses

Comma-separated list of:
- e.f (where f a field)
- a[*], a[x..y] (where a an array expression)
- \nothing, \everything (default)

Example

```java
C x, y;
//@ assignable x, x.i;
void m() {
    C tmp = x;  //allowed (local variable)
    tmp.i = 27; //allowed (in assignable clause)
    x = y;
    x.i = 27;
}
```
The assignable Clauses

Comma-separated list of:

- e.f (where f a field)
- a[()], a[x..y] (where a an array expression)
- \nothing, \everything (default)

Example

```java
C x, y;
//@ assignable x, x.i;

void m() {
    C tmp = x;   //allowed (local variable)
    tmp.i = 27;  //allowed (in assignable clause)
    x = y;       //allowed (in assignable clause)
    x.i = 27;
}
```
The assignable Clauses

Comma-separated list of:
- e.f (where f a field)
- a[*], a[x..y] (where a an array expression)
- \nothing, \everything (default)

Example

```java
C x, y;
//@ assignable x, x.i;
void m() {
    C tmp = x;  //allowed (local variable)
    tmp.i = 27; //allowed (in assignable clause)
    x = y;      //allowed (in assignable clause)
    x.i = 27;   //forbidden (not local, not in assignable)
}
```
The assignable Clauses

Comma-separated list of:

- `e.f` (where `f` a field)
- `a[*]`, `a[x..y]` (where `a` an array expression)
- `\nothing`, `\everything` (default)

Example

```java
C x, y;
//@ assignable x, x.i;
void m() {
    C tmp = x;  //allowed (local variable)
    tmp.i = 27; //allowed (in assignable clause)
    x = y;     //allowed (in assignable clause)
    x.i = 27;  //forbidden (not local, not in assignable)
}
```
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 $n! = 0$. 

Prof. Dr. Bernhard Beckert – Specification & Formal Analysis of Java Programs
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

```java
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**

```java
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 \( n \neq 0 \).
```
The diverges Clause

\[ \text{diverges } e; \]

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

Examples

\[ \text{diverges false}; \]
The method must always terminate.

\[ \text{diverges true}; \]
The method may terminate or not.

\[ \text{diverges } n == 0; \]
The method must terminate, when called in a state with \( n \neq 0 \).
The signals Clauses

\begin{itemize}
  \item \textbf{ensures} \ p;
  \item \textbf{signals\_only} ET_1, \ldots, ET_m;
  \item \textbf{signals} (E_1 e_1) s_1;
  \item \ldots
  \item \textbf{signals} (E_n e_n) s_n;
\end{itemize}

\begin{itemize}
  \item normal termination $\Rightarrow$ \ p must hold (in post-state)
  \item exception thrown $\Rightarrow$ must be of type ET_1, \ldots, or ET_m
  \item exception of type E_1 thrown $\Rightarrow$ s_1 must hold (in post-state)
  \item \ldots
  \item exception of type E_n thrown $\Rightarrow$ s_n must hold (in post-state)
\end{itemize}
The signals Clauses

\textbf{ensures} p;
\textbf{signals\_only} ET1, \ldots, ETm;
\textbf{signals} (E1 e1) s1;
\ldots
\textbf{signals} (En en) sn;

- normal termination \implies p \text{ must hold (in post-state)}
- exception thrown \implies \text{must be of type } ET1, \ldots, \text{ or } ETm
- exception of type E1 thrown \implies s1 \text{ must hold (in post-state)}
  \ldots
- exception of type En thrown \implies sn \text{ must hold (in post-state)}
The signals Clauses

\begin{enumerate}
  \item \textbf{normal termination} \implies \textit{p} must hold (in post-state)
  \item \textbf{exception thrown} \implies \textbf{must be of type} \textit{ET1}, \ldots, \textbf{or} \textit{ETm}
  \item \textbf{exception of type} \textit{E1} \textbf{thrown} \implies \textit{s1} \textbf{must hold (in post-state)}
  \item \ldots
  \item \textbf{exception of type} \textit{En} \textbf{thrown} \implies \textit{sn} \textbf{must hold (in post-state)}
\end{enumerate}

\textbf{ensures} \textit{p};
\textbf{signals\_only} \textit{ET1}, \ldots, \textit{ETm};
\textbf{signals} \ (\textit{E1 \ e1}) \textit{s1};
\ldots
\textbf{signals} \ (\textit{En \ en}) \textit{sn};
The signals Clauses

\[\text{ensures } p;\]
\[\text{signals\_only } \text{ET}_1, \ldots, \text{ET}_m;\]
\[\text{signals } (\text{E}_1 \ e_1) \ s_1;\]

\[\ldots\]
\[\text{signals } (\text{E}_n \ e_n) \ s_n;\]

- normal termination \(\Rightarrow\ p\) must hold (in post-state)
- exception thrown \(\Rightarrow\) must be of type \(\text{ET}_1, \ldots, \text{ET}_m\)
- exception of type \(\text{E}_1\) thrown \(\Rightarrow\ s_1\) must hold (in post-state)

\[\ldots\]
- exception of type \(\text{E}_n\) thrown \(\Rightarrow\ s_n\) must hold (in post-state)
The signals Clauses

\[
\text{ensures } p; \\
\text{signals\_only } ET_1, \ldots, ET_m; \\
\text{signals } (E_1 e_1) s_1; \\
\ldots \\
\text{signals } (E_n e_n) s_n;
\]

- normal termination \( \Rightarrow \) \( p \) must hold (in post-state)
- exception thrown \( \Rightarrow \) must be of type \( ET_1, \ldots, \) or \( ET_m \)
- exception of type \( E_1 \) thrown \( \Rightarrow \) \( s_1 \) must hold (in post-state)
- \ldots
- exception of type \( E_n \) thrown \( \Rightarrow \) \( s_n \) must hold (in post-state)
How to add contracts to abstract methods in interfaces? 
Remember: There are no attributes in interfaces. 
More precisely: Only static final fields.
public interface IBonusCard {

    public void addBonus(int newBonusPoints);

}
Model Fields

```java
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.
public interface IBonusCard {

    public void addBonus(int newBonusPoints);

}
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.
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.
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.
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

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

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

Implementation

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

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

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

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

Implementation

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

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

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

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

/*@ represents \[ x \text{ such that } A(x); */
```
Other Representations

/*@ private represents bonusPoints
   = bankCardPoints; */

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

/*@ represents x \such_that A(x); */
Other Representations

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

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

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

- assertions ‘//@ assert e;’
- loop invariants ‘//@ maintaining p;’
- data groups
- refines
- many more...
Other JML Features

- assertions `//@ assert e;`
- loop invariants `//@ maintaining p;`
- data groups
- refines
- many more...
Other JML Features

- assertions ‘//@ assert e;’
- loop invariants ‘//@ maintaining p;’
- data groups
  - refines
  - many more...
Other JML Features

- assertions `//@ assert e;`
- loop invariants `//@ maintaining p;`
- data groups
- refines
- many more...
Other JML Features

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

JML has modifiers `non_null` and `nullable`

```java
private /*@spec_public non_null@*/ Object x;
⇝ implicit invariant added to class: ‘invariant x != null;’

void m(/*@non_null@*/ Object p);
⇝ implicit precondition added to all contracts: ‘requires p != null;’

/*@non_null@*/ Object m();
⇝ implicit postcondition added to all contracts: ‘ensures \result != null;’
```

`non_null` is the default!
If something may be `null`, you have to declare it `nullable`
Nullity

JML has modifiers **non_null** and **nullable**

```java
private /*@spec_public non_null@*/ Object x;
⇝ implicit invariant added to class: ‘invariant x != null;’
```

```java
void m(/*@non_null@*/ Object p);
⇝ implicit precondition added to all contracts: ‘requires p != null;’
```

```java
/*@non_null@*/ Object m();
⇝ implicit postcondition added to all contracts: ‘ensures \result != null;’
```

**non_null** is the default!
If something may be null, you have to declare it **nullable**
Nullity

JML has modifiers non_null and nullable

```java
private /*@spec_public non_null@*/ Object x;
⇝ implicit invariant added to class: ‘invariant x != null;’
```

```java
void m(/*@non_null@*/ Object p);
⇝ implicit precondition added to all contracts: ‘requires p != null;’
```

```java
/*@non_null@*/ Object m();
⇝ implicit postcondition added to all contracts: ‘ensures \result != null;’
```

**non_null** is the default!
If something may be null, you have to declare it nullable
Nullity

JML has modifiers non_null and nullable

```java
private /*@spec_public non_null@*/ Object x;
⇝ implicit invariant added to class: ‘invariant x != null;’
```

```java
void m(/*@non_null@*/ Object p);
⇝ implicit precondition added to all contracts:
‘requires p != null;’
```

```java
/*@non_null@*/ Object m();
⇝ implicit postcondition added to all contracts:
‘ensures \result != null;’
```

non_null is the default!
If something may be null, you have to declare it nullable
Nullity

JML has modifiers `non_null` and `nullable`

```java
private /*@spec_public non_null@*/ Object x;
⇝ implicit invariant added to class: `invariant x != null;`
```

```java
void m(/*@non_null@*/ Object p);
⇝ implicit precondition added to all contracts: `requires p != null;`
```

```java
/*@non_null@*/ Object m();
⇝ implicit postcondition added to all contracts: `ensures \result != null;`
```

non_null is the default!
If something may be null, you have to declare it `nullable`
Nullity

JML has modifiers **non_null** and **nullable**

```java
private /*@spec_public non_null@*/ Object x;
⇝ implicit invariant added to class: ‘invariant x != null;’
```

```java
void m(/*@non_null@*/ Object p);
⇝ implicit precondition added to all contracts:
‘requires p != null;’
```

```java
/*@non_null@*/ Object m();
⇝ implicit postcondition added to all contracts:
‘ensures \result != null;’
```

**non_null** is the default!
If something may be **null**, you have to declare it **nullable**
Nullity

JML has modifiers \texttt{non\_null} and \texttt{nullable}

\begin{verbatim}
private /*@spec_public non_null@*/ Object x;
\end{verbatim}
\implies implicit invariant added to class: ‘\texttt{invariant x \neq null;}’

\begin{verbatim}
void m(/*@non_null@*/ Object p);
\end{verbatim}
\implies implicit precondition added to all contracts: ‘\texttt{requires p \neq null;}’

\begin{verbatim}
/*@non_null@*/ Object m();
\end{verbatim}
\implies implicit postcondition added to all contracts: ‘\texttt{ensures \result \neq null;}’

\texttt{non\_null} is the default!
If something may be \texttt{null}, you have to declare it \texttt{nullable}
Nullity

JML has modifiers non_null and nullable

```java
private /*@spec_public non_null@*/ Object x;
⇝ implicit invariant added to class: 'invariant x != null';
```

```java
void m(/*@non_null@*/ Object p);
⇝ implicit precondition added to all contracts:
‘requires p != null;’
```

```java
/*@non_null@*/ Object m();
⇝ implicit postcondition added to all contracts:
‘ensures \result != null;’
```

**non_null** is the default!

If something may be null, you have to declare it nullable
Problems with Specifications Using Integers

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

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

JML uses the Java semantics of integers:

- $1073741821 \times 1073741821 = −2147483639$
- $1073741822 \times 1073741822 = 4$

The JML type \texttt{\textbackslash bigint} provides arbitrary precision integers.
Problems with Specifications Using Integers

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

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

JML uses the Java semantics of integers:

\[
1073741821 \times 1073741821 = −2147483639 \\
1073741822 \times 1073741822 = 4
\]

The JML type `\texttt{bigint}` provides arbitrary precision integers.
Problems with Specifications Using Integers

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

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

JML uses the Java semantics of integers:

$$1073741821 \times 1073741821 = -2147483639$$

$$1073741822 \times 1073741822 = 4$$

The JML type `bigint` provides arbitrary precision integers.
Problems with Specifications Using Integers

```java
/*@ requires y >= 0; @
@ ensures @
@ result * result <= y &&
@ y < (abs(result) + 1) * (abs(result) + 1); @ */

public static int isqrt(int y)
```

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

JML uses the Java semantics of integers:

- $1073741821 \times 1073741821 = -2147483639$
- $1073741822 \times 1073741822 = 4$

The JML type \texttt{\textbackslash 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**: lightweight static verification
- **KeY**: full static verification
- **OpenJML**: tool suite, under development

The tools do not yet support the new features of Java 5! 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**: lightweight static verification
- **KeY**: full static verification
- **OpenJML**: tool suite, under development

The tools do not yet support the new features of Java 5! 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**: lightweight static verification
- **KeY**: full static verification
- **OpenJML**: tool suite, under development

The tools do not yet support the new features of Java 5!
- 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**: lightweight static verification
- **KeY**: full static verification
- **OpenJML**: tool suite, under development

The tools do not yet support the new features of Java 5!
  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**: lightweight static verification
- **KeY**: full static verification
- **OpenJML**: tool suite, under development

The tools do not yet support the new features of Java 5!
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**: lightweight static verification
- **KeY**: full static verification
- **OpenJML**: tool suite, under development

The tools do not yet support the new features of Java 5! e.g.: no generics, no enums, no enhanced for-loops, no autoboxing
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**: lightweight static verification
- **KeY**: full static verification
- **OpenJML**: tool suite, under development

The tools do not yet support the new features of Java 5!

- 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**: lightweight static verification
- **KeY**: full static verification
- **OpenJML**: tool suite, under development

The tools do not yet support the new features of Java 5!

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