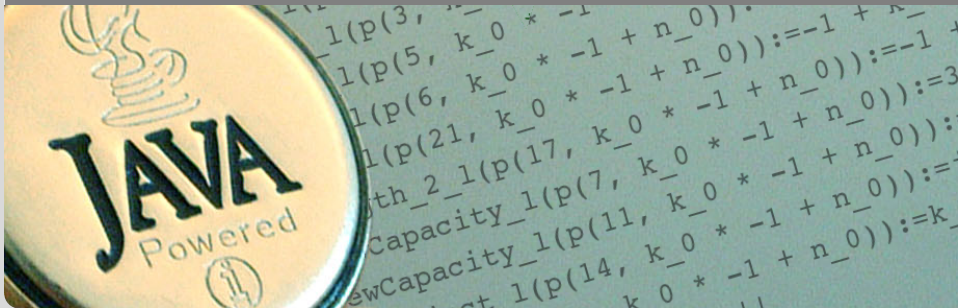


# Applications of Formal Verification

## Functional Verification of Java Programs: Java Modeling Language

Bernhard Beckert · Mattias Ulbrich | SS 2019

KIT – INSTITUT FÜR THEORETISCHE INFORMATIK



short for

Behavioural Interface Specification Language

- used to describe formally input/output-behaviour of operations
- abstraction from implementation details
  - code structure, algorithms and
  - data structures
- tailored for a particular programming language

Example BISLs:

Java Modeling Language	for	Java
Spec#	for	C#
ACSL	for	C (tool: Frama-C)
VCC	for	C (concurrency)
LARCH/C/C++	for	C/C++ (discontinued)
JAZZ	for	Java (discontinued)

short for

Behavioural Interface Specification Language

- used to describe formally input/output-behaviour of operations
- abstraction from implementation details
  - code structure, algorithms and
  - data structures
- tailored for a particular programming language

Example BISLs:

Java Modeling Language	for	Java
Spec#	for	C#
ACSL	for	C (tool: Frama-C)
VCC	for	C (concurrency)
LARCH/C/C++	for	C/C++ (discontinued)
JAZZ	for	Java (discontinued)

short for

Behavioural Interface Specification Language

- used to describe formally input/output-behaviour of operations
- abstraction from implementation details
  - code structure, algorithms and
  - data structures
- tailored for a particular programming language

Example BISLs:

Java Modeling Language	for	Java
Spec#	for	C#
ACSL	for	C (tool: Frama-C)
VCC	for	C (concurrency)
LARCH/C/C++	for	C/C++ (discontinued)
JAZZ	for	Java (discontinued)

## 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:  
“Behavioural Interface Specification Language”
- 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

## 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:  
“Behavioural Interface Specification Language”
- 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

## 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:  
“Behavioural Interface Specification Language”
- 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

## 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:  
“**B**ehavioural **I**nterface **S**pecification **L**anguage”
- 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



## 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:  
“**B**ehavioural **I**nterface **S**pecification **L**anguage”
- 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

## 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:  
“**B**ehavioural **I**nterface **S**pecification **L**anguage”
- 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

## 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:  
“**B**ehavioural **I**nterface **S**pecification **L**anguage”
- 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

```
/*@ 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's a *convention* to use them.
- JML specifications may themselves contain comments

```
/*@ 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's a *convention* to use them.
- JML specifications may themselves contain comments

```
/*@ 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's a *convention* to use them.
- JML specifications may themselves contain comments

```
/*@ 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's a *convention* to use them.
- JML specifications may themselves contain comments

```
/*@ 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's a *convention* to use them.
- JML specifications may themselves contain comments



```
/*@ 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:
    - 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’s a *convention* to use them.
- JML specifications may themselves contain comments

```
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.

```
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.

```
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.

```
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.

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

## Abbreviations

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

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

```
/*@ 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 e) 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;  
   @ ensures post;  
   @ signals_only E1, ..., En;  
   @ signals(E e) s;  
   @*/  
T m(...);
```

## Abbreviations

```
normal behavior = signals(Exception e) 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;  
   @ signals_only E1, ..., En;  
   @ signals(E e) s;  
   @*/  
T m(...);
```

## Abbreviations

```
normal behavior = signals(Exception e) 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;  
   @ signals(E e) s;  
   @*/  
T m(...);
```

## Abbreviations

```
normal behavior = signals(Exception e) 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;  
  @*/  
T m(...);
```

## Abbreviations

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

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

```
//@ 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*, *ownership*, *dynamic frames*)
- **instance** invariants *can*, **static** invariants *cannot* refer to **this**
- default: **instance** within classes, **static** within interfaces



```
//@ 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*, *ownership*, *dynamic frames*)
- **instance** invariants *can*, **static** invariants *cannot* refer to **this**
- default: **instance** within classes, **static** within interfaces

```
//@ 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*, *ownership*, *dynamic frames*)
- **instance** invariants *can*, **static** invariants *cannot* refer to **this**
- default: **instance** within classes, **static** within interfaces

```
//@ 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*, *ownership*, *dynamic frames*)
- `instance` invariants *can*, `static` invariants *cannot* refer to `this`
- default: `instance` within classes, `static` within interfaces

```
//@ 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*, *ownership*, *dynamic frames*)
- **instance** invariants *can*, **static** invariants *cannot* refer to **this**
- default: **instance** within classes, **static** within interfaces

```
//@ 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*, *ownership*, *dynamic frames*)
- **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.

Hence, they can be used in JML specifications.

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.

Hence, they can be used in JML specifications.

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.

Hence, they can be used in JML specifications.

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.

Hence, they can be used in JML specifications.

After declaring

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

cardIsInserted()

could replace

insertedCard != null

in JML annotations.

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

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

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

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

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

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

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



```
(\forall int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\forall int i; 0<=i && i<\result.length
```

```
(\exists int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\exists int i; 0<=i && i<\result.length
```

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

*equivalent to*

```
(\forall int i; 0<=i && i<\result.length
```

```
(\forall int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\exists int i; 0<=i && i<\result.length
```

```
(\exists int i; 0<=i && i<\result.length
```

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

*equivalent to*

```
(\forall int i; 0<=i && i<\result.length
```

```
(\forall int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\exists int i; 0<=i && i<\result.length
```

```
(\exists int i; 0<=i && i<\result.length
```

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

```
(\forall int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\forall int i; 0<=i && i<\result.length
```

```
(\exists int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\exists int i; 0<=i && i<\result.length
```

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

*equivalent to*

```
(\forall int i; 0<=i && i<\result.length
```

```
(\forall int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\exists int i; 0<=i && i<\result.length
```

```
(\exists int i; 0<=i && i<\result.length
```

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

```
(\forall int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\forall int i; 0<=i && i<\result.length
```

```
(\exists int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\exists int i; 0<=i && i<\result.length
```

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

```
(\forall int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\forall int i; 0<=i && i<\result.length
```

```
(\exists int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\exists int i; 0<=i && i<\result.length
```

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

```
(\forall int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\forall int i; 0<=i && i<\result.length
```

```
(\exists int i; 0<=i && i<\result.length
```

*equivalent to*

```
(\exists int i; 0<=i && i<\result.length
```

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



# Generalised and Numerical Quantifiers

$(\text{\backslash num\_of } T \ i; \ e)$	$\#\{i [e]\}$ , number of elements of type $T$ with property $e$
$(\text{\backslash sum } T \ i; \ p; \ t)$	$\sum_{i:[p]} [t]$
$(\text{\backslash product } T \ i; \ p; \ t)$	$\prod_{i:[p]} [t]$
$(\text{\backslash min } T \ i; \ p; \ t)$	$\min_{i:[p]} \{[t]\}$
$(\text{\backslash max } T \ i; \ p; \ t)$	$\max_{i:[p]} \{[t]\}$

# The assignable Clauses

Comma-separated list of:

- $e.f$  (where  $f$  a field)

# The assignable Clauses

Comma-separated list of:

- $e.f$  (where  $f$  a field)
- $a[*]$ ,  $a[x..y]$  (where  $a$  an array expression)

# 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)

# 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

```
C x, y; int i;
//@ 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

```
C x, y; int i;
//@ 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

```
C x, y; int i;
//@ 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

```
C x, y; int i;
//@ 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

```
C x, y; int i;
//@ 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

```
C x, y; int i;
//@ 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

```
C x, y; int i;
//@ 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)
}
```

**assignable** clauses are always evaluated in the pre-state!

# “Nothing” is more than you think it is

assignable \nothing means that no memory location **existing at method invocation** must be changed.

## Valid specification

```
//@ assignable \nothing;  
void n() {  
  C c = new C();  
  c.i = 42;  
}
```

**strictly pure:** no memory location must be changed

**weakly pure:** only freshly created locations may be changed

Strict purity is too restrictive for realistic use cases.

JML allows weakly pure method calls in specifications.

# “Nothing” is more than you think it is

assignable \nothing means that no memory location **existing at method invocation** must be changed.

## Valid specification

```
//@ assignable \nothing;  
void n() {  
  C c = new C();  
  c.i = 42; // allowed: fresh objects can be modified  
}
```

**strictly pure:** no memory location must be changed

**weakly pure:** only freshly created locations may be changed

Strict purity is too restrictive for realistic use cases.

JML allows weakly pure method calls in specifications.

# “Nothing” is more than you think it is

assignable \nothing means that no memory location **existing at method invocation** must be changed.

## Valid specification

```
//@ assignable \nothing;  
void n() {  
    C c = new C();  
    c.i = 42;    // allowed: fresh objects can be modified  
}
```

**strictly pure:** no memory location must be changed

**weakly pure:** only freshly created locations may be changed

Strict purity is too restrictive for realistic use cases.

JML allows weakly pure method calls in specifications.

# “Nothing” is more than you think it is

`assignable \nothing` means that no memory location **existing at method invocation** must be changed.

## Valid specification

```
//@ assignable \nothing;  
void n() {  
    C c = new C();  
    c.i = 42;    // allowed: fresh objects can be modified  
}
```

**strictly pure:** no memory location must be changed

**weakly pure:** only freshly created locations may be changed

Strict purity is too restrictive for realistic use cases.

JML allows weakly pure method calls in specifications.

```
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  $n \neq 0$ .



```
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  $n \neq 0$ .

```
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  $n \neq 0$ .

```
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  $n \neq 0$ .

```
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)

```
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)

```
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)

```
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)

```
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, ..., 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)





```
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);  
  
}
```

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);  
  
}
```

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);  
  
}
```

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; }  
}
```



```
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; }  
}
```

```
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; }  
}
```

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

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

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

```
/*@ 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); @*/
```

## Behavioral Subtyping, (Lizkov-Leavens Substitution Principle)

If  $D$  is a subclass of  $C$ , then objects of type  $C$  may be replaced with objects of type  $D$  without altering the desirable properties of the program.

- A class invariant is inherited by all subclasses.
- An operation contract is inherited by all overridden methods.
- Subclass may add invariants and contracts

## Behavioral Subtyping, (Lizkov-Leavens Substitution Principle)

If  $D$  is a subclass of  $C$ , then objects of type  $C$  may be replaced with objects of type  $D$  without altering the desirable properties of the program.

- A class invariant is inherited by all subclasses.
- An operation contract is inherited by all overridden methods.
- Subclass may add invariants and contracts

## Behavioral Subtyping, (Lizkov-Leavens Substitution Principle)

If  $D$  is a subclass of  $C$ , then objects of type  $C$  may be replaced with objects of type  $D$  without altering the desirable properties of the program.

- A class invariant is inherited by all subclasses.
- An operation contract is inherited by all overridden methods.
- Subclass may add invariants and contracts



## Dealing with loops

- Loops are a challenge for reasoning about programs
- Loop specifications to guide program proof systems

```
/*@ loop_invariant linv;  
   @ decreases variant;  
   @ assignable A;  
   @*/  
while(...) { ... }
```

- Loop invariant *linv* needs to hold for all iterations  
(every time the loop condition is checked)
- The *variant* must be decreasing non-negative integer  
(termination – there is no infinite decreasing sequence)
- assignable, cf. assignable for methods

## Dealing with loops

- Loops are a challenge for reasoning about programs
- Loop specifications to guide program proof systems

```
/*@ loop_invariant linv;  
   @ decreases variant;  
   @ assignable A;  
   @*/  
while(...) { ... }
```

- Loop invariant *linv* needs to hold for all iterations (every time the loop condition is checked)
- The *variant* must be decreasing non-negative integer (termination – there is no infinite decreasing sequence)
- assignable, cf. assignable for methods

## Dealing with loops

- Loops are a challenge for reasoning about programs
- Loop specifications to guide program proof systems

```
/*@ loop_invariant linv;  
   @ decreases variant;  
   @ assignable A;  
   @*/  
while(...) { ... }
```

- Loop invariant *linv* needs to hold for all iterations (every time the loop condition is checked)
- The *variant* must be decreasing non-negative integer (termination – there is no infinite decreasing sequence)
- assignable, *cf.* assignable for methods

## Dealing with loops

- Loops are a challenge for reasoning about programs
- Loop specifications to guide program proof systems

```
/*@ loop_invariant linv;  
   @ decreases variant;  
   @ assignable A;  
   @*/  
while(...) { ... }
```

- Loop invariant *linv* needs to hold for all iterations (every time the loop condition is checked)
- The *variant* must be decreasing non-negative integer (termination – there is no infinite decreasing sequence)
- assignable, *cf.* assignable for methods

- assertions `'//@ assert e;'`
- assumptions `'//@ assume e;'`
- data groups
- **refines**
- many more...

- assertions `'//@ assert e;'`
- assumptions `'//@ assume e;'`
- data groups
- **refines**
- many more...

- assertions `'//@ assert e;'`
- assumptions `'//@ assume e;'`
- data groups
- `refines`
- many more...

- assertions `'//@ assert e;'`
- assumptions `'//@ assume e;'`
- data groups
- **refines**
- many more...



- assertions `'//@ assert e;'`
- assumptions `'//@ assume e;'`
- data groups
- **refines**
- many more...

JML has modifiers `non_null` and `nullable`

```
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**

JML has modifiers `non_null` and `nullable`

```
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**

JML has modifiers `non_null` and `nullable`

```
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**

JML has modifiers `non_null` and `nullable`

```
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`

JML has modifiers `non_null` and `nullable`

```
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`

JML has modifiers `non_null` and `nullable`

```
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`

JML has modifiers `non_null` and `nullable`

```
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`



JML has modifiers `non_null` and `nullable`

```
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**

# 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  $\text{\result} = 1073741821 = \frac{1}{2}(\text{MAX\_INT} - 5)$  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:

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

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  $\text{\result} = 1073741821 = \frac{1}{2}(\text{MAX\_INT} - 5)$  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:

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

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  $\text{\result} = 1073741821 = \frac{1}{2}(\text{MAX\_INT} - 5)$  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:

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

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  $\text{\result} = 1073741821 = \frac{1}{2}(\text{MAX\_INT} - 5)$  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:

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

The JML type `\bigint` provides arbitrary precision integers.

Many tools support JML (see JML homepage). Among them:

- **KeY**: full static verification
- `OpenJML`: tool suite, under development
- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlunit`: unit testing (like JUnit)
- `JMLUnitNG`: unit test generation
- `ESC/Java2`: lightweight static verification

Many 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:

- KeY: full static verification
- OpenJML: tool suite, under development
- `jml`: JML syntax checker
- `jmldoc`: code documentation (like Javadoc)
- `jmlunit`: unit testing (like JUnit)
- `JMLUnitNG`: unit test generation
- ESC/Java2: lightweight static verification

Many 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:

- KeY: full static verification
- OpenJML: tool suite, under development
- jml: JML syntax checker
- jml`doc`: code documentation (like Javadoc)
- jml`unit`: unit testing (like JUnit)
- JMLUnitNG: unit test generation
- ESC/Java2: lightweight static verification

Many 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:

- KeY: full static verification
- OpenJML: tool suite, under development
- jml: JML syntax checker
- jmldoc: code documentation (like Javadoc)
- jmlunit: unit testing (like JUnit)
- JMLUnitNG: unit test generation
- ESC/Java2: lightweight static verification

Many 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:

- KeY: full static verification
- OpenJML: tool suite, under development
- jml: JML syntax checker
- jml doc: code documentation (like Javadoc)
- jmlunit: unit testing (like JUnit)
- JMLUnitNG: unit test generation
- ESC/Java2: lightweight static verification

Many 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:

- KeY: full static verification
- OpenJML: tool suite, under development
- jml: JML syntax checker
- jmldoc: code documentation (like Javadoc)
- jmlunit: unit testing (like JUnit)
- JMLUnitNG: unit test generation
- ESC/Java2: lightweight static verification

Many 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:

- KeY: full static verification
- OpenJML: tool suite, under development
- jml: JML syntax checker
- jmldoc: code documentation (like Javadoc)
- jmlunit: unit testing (like JUnit)
- JMLUnitNG: unit test generation
- ESC/Java2: lightweight static verification

Many 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:

- KeY: full static verification
- OpenJML: tool suite, under development
- jml: JML syntax checker
- jmldoc: code documentation (like Javadoc)
- jmlunit: unit testing (like JUnit)
- JMLUnitNG: unit test generation
- ESC/Java2: lightweight static verification

Many 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:

- KeY: full static verification
- OpenJML: tool suite, under development
- jml: JML syntax checker
- jmldoc: code documentation (like Javadoc)
- jmlunit: unit testing (like JUnit)
- JMLUnitNG: unit test generation
- ESC/Java2: lightweight static verification

Many tools do not yet support the new features of Java 5!  
e.g.: no generics, no enums, no enhanced for-loops, no  
autoboxing