Rigorous Software Development
CSCI-GA 3033-009

Instructor: Thomas Wies

Spring 2013

Lecture 10
What does this program print?

class A {
    public static int x = B.x + 1;
}

class B {
    public static int x = A.x + 1;
}

class C {
    public static void main(String[] p) {
        System.err.println("A: " + A.x + ", B: " + B.x);
    }
}
What does this program print?

If we run class C:
1) main-method of class C first accesses A.x.
2) Class A is initialized. The lock for A is taken.
3) Static initializer of A runs and accesses B.x.
4) Class B is initialized. The lock for B is taken.
5) Static initializer of B runs and accesses A.x.
6) Class A is still locked by current thread (recursive initialization). Therefore, initialization returns immediately.
7) The value of A.x is still 0 (section 12.3.2 and 4.12.5), so B.x is set to 1.
8) Initialization of B finishes.
9) The value of A.x is now set to 2.
10) The program prints “A: 2, B: 1”.
Formal Semantics of Java Programs

• The Java Language Specification (JLS) 3rd edition gives semantics to Java programs
  – The document has 684 pages.
  – 118 pages to define semantics of expression.
  – 42 pages to define semantics of method invocation.

• Semantics is only defined in prose.
  – How can we make the semantics formal?
  – We need a mathematical model of computation.
Semantics of Programming Languages

- **Denotational Semantics**
  - Meaning of a program is defined as the mathematical object it computes (e.g., partial functions).
  - Example: Abstract Interpretation

- **Axiomatic Semantics**
  - Meaning of a program is defined in terms of its effect on the truth of logical assertions.
  - Example: Hoare Logic

- **(Structural) Operational Semantics**
  - Meaning of a program is defined by formalizing the individual computation steps of the program.
  - Example: Labeled Transition Systems
IMP: A Simple Imperative Language

Before we move on to Java, we look at a simple imperative programming language IMP.

An IMP program:

\[
p := 0;
x := 1;
\textbf{while} \ x \leq n \ \textbf{do}
\begin{align*}
  & x := x + 1; \\
  & p := p + m;
\end{align*}
\]
IMP: Syntactic Entities

• $n \in \mathbb{Z}$ – integers
• true,false $\in \mathbb{B}$ – Booleans
• $x,y \in L$ – locations (program variables)
• $e \in Aexp$ – arithmetic expressions
• $b \in Bexp$ – Boolean expressions
• $c \in Com$ – commands
Syntax of Arithmetic Expressions

• Arithmetic expressions ($Aexp$)
  
  $$ e ::= n \text{ for } n \in \mathbb{Z} $$
  
  $$ | x \text{ for } x \in L $$
  
  $$ | e_1 + e_2 $$
  
  $$ | e_1 - e_2 $$
  
  $$ | e_1 \cdot e_2 $$

• Notes:
  
  – Variables are not declared before use.
  
  – All variables have integer type.
  
  – Expressions have no side-effects.
Syntax of Boolean Expressions

- Boolean expressions ($Bexp$)
  
  $b ::= \text{true}$
  
  $| \text{false}$
  
  $| e_1 = e_2 \text{ for } e_1, e_2 \in Aexp$
  
  $| e_1 \leq e_2 \text{ for } e_1, e_2 \in Aexp$
  
  $| \neg b \text{ for } b \in Bexp$
  
  $| b_1 \land b_2 \text{ for } b_1, b_2 \in Bexp$
  
  $| b_1 \lor b_2 \text{ for } b_1, b_2 \in Bexp$
Syntax of Commands

• Commands \((Com)\)

\[\begin{align*}
c & ::= \text{skip} \\
& \quad | \ x ::= e \\
& \quad | \ c_1; c_2 \\
& \quad | \ \text{if} \ b \ \text{then} \ c_1 \ \text{else} \ c_2 \\
& \quad | \ \text{while} \ b \ \text{do} \ c
\end{align*}\]

• Notes:
  – The typing rules have been embedded in the syntax definition.
  – Other parts are not context-free and need to be checked separately (e.g., all variables are declared).
  – Commands contain all the side-effects in the language.
  – Missing: references, function calls, ...
Meaning of IMP Programs

Questions to answer:
• What is the “meaning” of a given IMP expression/command?
• How would we evaluate IMP expressions and commands?
• How are the evaluator and the meaning related?
• How can we reason about the effect of a command?
Semantics of IMP

• The meaning of IMP expressions depends on the values of variables, i.e. the current state.
• A state at a given moment is represented as a function from $L$ to $\mathbb{Z}$
• The set of all states is $Q = L \rightarrow \mathbb{Z}$
• We shall use $q$ to range over $Q$
Judgments

• We write $<e, q> \Downarrow n$ to mean that $e$ evaluates to $n$ in state $q$.
  – The formula $<e, q> \Downarrow n$ is a judgment (a statement about a relation between $e$, $q$ and $n$)
  – In this case, we can view $\Downarrow$ as a function of two arguments $e$ and $q$

• This formulation is called natural operational semantics
  – or big-step operational semantics
  – the judgment relates the expression and its “meaning”

• How can we define $<e_1 + e_2, q> \Downarrow ...$?
Inference Rules

• We express the evaluation rules as inference rules for our judgments.
• The rules are also called evaluation rules.

An inference rule

\[
\frac{F_1 \ldots F_n}{G} \quad \text{where } H
\]

defines a relation between judgments \( F_1, \ldots, F_n \) and \( G \).
• The judgments \( F_1, \ldots, F_n \) are the premises of the rule;
• The judgments \( G \) is the conclusion of the rule;
• The formula \( H \) is called the side condition of the rule.

If \( n=0 \) the rule is called an axiom. In this case, the line separating premises and conclusion may be omitted.
Inference Rules for $Aexp$

- In general, we have one rule per language construct:

  $<n, q> \downarrow n$ ← Axiom → $<x, q> \downarrow q(x)$

  $<e_1, q> \downarrow n_1$  $<e_2, q> \downarrow n_2$  $<e_1 + e_2, q> \downarrow (n_1 + n_2)$

  $<e_1 - e_2, q> \downarrow (n_1 - n_2)$

  $<e_1, q> \downarrow n_1$  $<e_2, q> \downarrow n_2$  $<e_1 \cdot e_2, q> \downarrow (n_1 \cdot n_2)$

- This is called **structural operational semantics**.
  - rules are defined based on the structure of the expressions.
Inference Rules for $\text{Bexp}$

\[
\begin{align*}
<\text{true}, q> & \downarrow \text{true} & <\text{false}, q> & \downarrow \text{false} \\
\frac{<e_1, q> \downarrow n_1 \quad <e_2, q> \downarrow n_2}{<e_1 = e_2, q> \downarrow (n_1 = n_2)} & \quad \frac{<e_1, q> \downarrow n_1 \quad <e_2, q> \downarrow n_2}{<e_1 \leq e_2, q> \downarrow (n_1 \leq n_2)} \\
\frac{<b_1, q> \downarrow t_1 \quad <e_2, q> \downarrow t_2}{<b_1 \land b_2, q> \downarrow (t_1 \land t_2)}
\end{align*}
\]
How to Read Inference Rules?

• Forward, as derivation rules of judgments
  – if we know that the judgments in the premise hold then we can infer that the conclusion judgment also holds.
  – Example:
    \[
    \langle 2, q \rangle \downarrow 2 \quad \langle 3, q \rangle \downarrow 3
    \]
    \[
    \hline
    \langle 2 \times 3, q \rangle \downarrow 6
    \]
How to Read Inference Rules?

• Backward, as evaluation rules:
  – Suppose we want to evaluate $e_1 + e_2$, i.e., find $n$ s.t. $e_1 + e_2 \downarrow n$ is derivable using the previous rules.
  – By inspection of the rules we notice that the last step in the derivation of $e_1 + e_2 \downarrow n$ must be the addition rule.
  – The other rules have conclusions that would not match $e_1 + e_2 \downarrow n$.
• This is called reasoning by inversion on the derivation rules.
  – Thus we must find $n_1$ and $n_2$ such that $e_1 \downarrow n_1$ and $e_2 \downarrow n_2$ are derivable.
  – This is done recursively.
• Since there is exactly one rule for each kind of expression, we say that the rules are syntax-directed.
  – At each step at most one rule applies.
  – This allows a simple evaluation procedure as above.
How to Read Inference Rules?

• Example: evaluation of an arithmetic expression via reasoning by inversion:

\[
\begin{align*}
<y, \{x \mapsto 3, \ y \mapsto 2\} & \Downarrow 2 \\
<2, \{x \mapsto 3, \ y \mapsto 2\} & \Downarrow 2 \\
<x, \{x \mapsto 3, \ y \mapsto 2\} & \Downarrow 3 \\
<2 \times y, \{x \mapsto 3, \ y \mapsto 2\} & \Downarrow 4 \\
x + (2 \times y), \{x \mapsto 3, \ y \mapsto 2\} & \Downarrow 7
\end{align*}
\]
Semantics of Commands

• The evaluation of a command in $Com$ has side-effects, but no direct result.
  – What is the result of evaluating a command?
• The “result” of a command $c$ in a pre-state $q$ is a $\text{transition}$ from $q$ to a post-state $q'$:
  $$q \xrightarrow{c} q'$$
• We can formalize this in terms of $\text{transition systems}$. 
Labeled Transition Systems

A labeled transition system (LTS) is a structure $LTS = (Q, Act, \rightarrow)$ where

- $Q$ is a set of states,
- $Act$ is a set of actions,
- $\rightarrow \subseteq Q \times Act \times Q$ is a transition relation.

We write $q \xrightarrow{a} q'$ for $(q, a, q') \in \rightarrow$. 
### Inference Rules for Transitions

\[
\begin{align*}
    q \xrightarrow{\text{skip}} q \\
    q \xrightarrow{x := e} q +\{x \mapsto n\} \\
\end{align*}
\]

\[
\begin{align*}
    \langle e, q \rangle \downarrow n \quad q \xrightarrow{c_1} q' \\
    q \xrightarrow{c_1 \cdot c_2} q'' \\
\end{align*}
\]

\[
\begin{align*}
    \langle b, q \rangle \downarrow \text{true} \quad q \xrightarrow{c_1} q' \\
    q \xrightarrow{\text{if } b \text{ then } c_1 \text{ else } c_2} q' \\
\end{align*}
\]

\[
\begin{align*}
    \langle b, q \rangle \downarrow \text{false} \quad q \xrightarrow{c_2} q' \\
    q \xrightarrow{\text{if } b \text{ then } c_1 \text{ else } c_2} q' \\
\end{align*}
\]

\[
\begin{align*}
    \langle b, q \rangle \downarrow \text{false} \\
    q \xrightarrow{\text{while } b \text{ do } c} q \\
\end{align*}
\]

\[
\begin{align*}
    \langle b, q \rangle \downarrow \text{true} \quad q \xrightarrow{c} q' \\
    q' \xrightarrow{\text{while } b \text{ do } c} q'' \\
\end{align*}
\]

\[
\begin{align*}
    q \xrightarrow{\text{while } b \text{ do } c} q'' \\
\end{align*}
\]
Operational Semantics of Java (and JML)

• Can we give an operational semantics of Java programs and JML specifications?
• What is the state of a Java program?
  – We have to take into account the state of the heap.
• How can we deal with side-effects in expressions?
• How can we handle exceptions?
A (labeled) transition system (LTS) is a structure
$LTS = (Q, Act, \rightarrow)$ where
- $Q$ is a set of states,
- $Act$ is a set of actions,
- $\rightarrow \subseteq Q \times Act \times Q$ is a transition relation.

- $Q$ reflects the current dynamic state of the program (heap and local variables).
- $Act$ is the executed code.
- Based on: D. v. Oheimb, T. Nipkow, Machine-checking the Java specification: Proving type-safety, 1999
What is the state after executing this code?

```java
List mylist = new LinkedList();
mylist.add(new Integer(1));
```

<table>
<thead>
<tr>
<th>Heap</th>
<th>Lcl</th>
</tr>
</thead>
<tbody>
<tr>
<td>6: LinkedList</td>
<td>1 7 1</td>
</tr>
<tr>
<td>7: LinkedList.Node</td>
<td>0 8 8</td>
</tr>
<tr>
<td>8: LinkedList.Node</td>
<td>9 7 7</td>
</tr>
<tr>
<td>9: Integer</td>
<td>1</td>
</tr>
</tbody>
</table>

mylist: 6
A state of a Java program gives valuations to local and global (heap) variables.

- $Q = Heap \times Local$
- $Heap = Address \rightarrow Class \times seq Value$
- $Local = Identifier \rightarrow Value$
- $Value = \mathbb{Z}, Address \subseteq \mathbb{Z}$

A state is denoted as $(heap, lcl)$, where $heap: Heap$ and $lcl: Local$. 
Actions of a Java Program

An action of a Java program is either

• the evaluation of an expression $e$ to a value $v$, denoted as $e \Rightarrow v$, or

• a Java statement, or

• a Java code block.

Note that expressions with side effects can modify the current state.
Example: Actions of a Java Program

• Post-increment expression
  \[(heap, lcl \cup \{x \mapsto 5\}) \xrightarrow{x++ \gg 5} (heap, lcl \cup \{x \mapsto 6\})\]

• Pre-increment expression
  \[(heap, lcl \cup \{x \mapsto 5\}) \xrightarrow{++x \gg 6} (heap, lcl \cup \{x \mapsto 6\})\]

• Assignment expression
  \[(heap, lcl \cup \{x \mapsto 5\}) \xrightarrow{x=2*x \gg 10} (heap, lcl \cup \{x \mapsto 10\})\]

• Assignment statement
  \[(heap, lcl \cup \{x \mapsto 5\}) \xrightarrow{x=2*x;} (heap, lcl \cup \{x \mapsto 10\})\]
Rules for Java Expressions (1)

• axiom for evaluating local variables
  \[(heap, lcl) \xrightarrow{x \mapsto lcl(x)} (heap, lcl)\]

• rule for assignment to local
  \[(heap, lcl) \xrightarrow{e \mapsto v} (heap', lcl')\]
  \[(heap, lcl) \xrightarrow{x=e \mapsto v} (heap', lcl' \{x \mapsto v\})\]

• rule for field access
  \[(heap, lcl) \xrightarrow{e \mapsto v} (heap', lcl')\]
  \[(heap, lcl) \xrightarrow{e.fld \mapsto \text{heap'}(v)(idx)} (heap', lcl')\]
  where \(idx\) is the index of the field \(fld\) in the object \(heap'(v)\)
Rules for Java Expressions (2)

• axiom for evaluating a constant expression $c$

\[(heap, lcl) \xrightarrow{c \gg c} (heap, lcl)\]

• rule for multiplication

\[(heap, lcl) \xrightarrow{e_1 \gg v_1} (heap', lcl')\]

\[(heap', lcl') \xrightarrow{e_2 \gg v_2} (heap'', lcl'')\]

\[(heap, lcl) \xrightarrow{e_1 \ast e_2 \gg v_1 \cdot v_2 \mod 2^{32}} (heap'', lcl'')\]

• similarly for other binary operators
Example: Derivation for $x = 2 \cdot x$

\[
\begin{align*}
(\text{heap}, \ lcl \cup \{x \mapsto 5\}) \quad &\xrightarrow{x \mapsto 5} \quad (\text{heap}, \ lcl \cup \{x \mapsto 5\}) \\
(\text{heap}, \ lcl \cup \{x \mapsto 5\}) \quad &\xrightarrow{2 \mapsto 2} \quad (\text{heap}, \ lcl \cup \{x \mapsto 5\}) \\
\quad &\xrightarrow{2 \cdot x \mapsto 10} \quad (\text{heap}, \ lcl \cup \{x \mapsto 5\}) \\
(\text{heap}, \ lcl \cup \{x \mapsto 5\}) \quad &\xrightarrow{x = 2 \cdot x \mapsto 10} \quad (\text{heap}, \ lcl \cup \{x \mapsto 10\})
\end{align*}
\]
Rules for Java Statements (1)

• expression statement (assignment or method call)

\[(heap, lcl) \xrightarrow{e \Rightarrow v} (heap', lcl')\]

\[(heap, lcl) \xrightarrow{e;} (heap', lcl')\]

• sequence of statements

\[(heap, lcl) \xrightarrow{s_1} (heap', lcl') \quad (heap', lcl') \xrightarrow{s_2} (heap'', lcl'')\]

\[(heap, lcl) \xrightarrow{s_1 s_2} (heap'', lcl'')\]
Rules for Java Statements (2)

• rules for `if` statement

\[
\begin{align*}
(\text{heap, lcl}) & \xrightarrow{e \gg v} (\text{heap}', \text{lcl}') \\
(\text{heap}', \text{lcl}') & \xrightarrow{bl_1} (\text{heap}'', \text{lcl}'') \\
(\text{heap, lcl}) & \xrightarrow{\text{if}\ (e) \{bl_1\} \text{ else } \{bl_2\}} (\text{heap}'', \text{lcl}'')
\end{align*}
\]

where \( v \neq 0 \)

\[
\begin{align*}
(\text{heap, lcl}) & \xrightarrow{e \gg v} (\text{heap}', \text{lcl}') \\
(\text{heap}', \text{lcl}') & \xrightarrow{bl_2} (\text{heap}'', \text{lcl}'') \\
(\text{heap, lcl}) & \xrightarrow{\text{if}\ (e) \{bl_1\} \text{ else } \{bl_2\}} (\text{heap}'', \text{lcl}'')
\end{align*}
\]

where \( v = 0 \)
Rules for Java Statements (3)

• rules for `while` statement

\[
\begin{align*}
(\text{heap}, \text{lcl}) \xrightarrow{e \gg v} (\text{heap}', \text{lcl}') & \quad \text{where } v = 0 \\
(\text{heap}, \text{lcl}) \xrightarrow{\text{while}(e)\{bl\}} (\text{heap}', \text{lcl}')
\end{align*}
\]

\[
\begin{align*}
(\text{heap}, \text{lcl}) \xrightarrow{e \gg v} (\text{heap}', \text{lcl}') & \\
(\text{heap}', \text{lcl}') \xrightarrow{bl} (\text{heap}'', \text{lcl}'') & \\
(\text{heap}'', \text{lcl}'') \xrightarrow{\text{while}(e)\{bl\}} (\text{heap''}', \text{lcl''}') & \quad \text{where } v \neq 0 \\
(\text{heap}, \text{lcl}) \xrightarrow{\text{while}(e)\{bl\}} (\text{heap''}', \text{lcl''}')
\end{align*}
\]
Rule for Java Method Calls

\[
(\text{heap}, \text{lcl}) \xrightarrow{e \gg v} (\text{heap}_0, \text{lcl}_0)
\]

\[
(\text{heap}_0, \text{lcl}_0) \xrightarrow{e_1 \gg v_1} (\text{heap}_1, \text{lcl}_1)
\]

\[
\vdots
\]

\[
(\text{heap}_{n-1}, \text{lcl}_{n-1}) \xrightarrow{e_n \gg v_n} (\text{heap}_n, \text{lcl}_n)
\]

\[
(\text{heap}_n, \text{mlcl}) \xrightarrow{\text{body}} (\text{heap}_{n+1}, \text{mlcl}')
\]

\[
(\text{heap}, \text{lcl}) \xrightarrow{e.m(e_1, \ldots, e_n) \gg \text{mlcl}'(\text{\textbackslash result})} (\text{heap}_{n+1}, \text{lcl}_n)
\]

where \text{body} is the body of method \text{m} in the object \text{heap}_{n+1}(v), and \text{mlcl} = \{\text{this} \mapsto v, \text{param}_1 \mapsto v_1, \ldots \text{param}_n \mapsto v_n\} where \text{param}_1, \ldots, \text{param}_n are the names of the parameters of \text{m}. 
Rule for Object Creation

- Object creation is always combined with a call of a constructor

\[
\begin{align*}
(h_{\text{heap}_1}, l_{\text{lcl}}) & \xrightarrow{\text{na.<init>}(e_1,\ldots,e_n) \Rightarrow v} (h_{\text{heap}’}, l_{\text{lcl}’}) \\
(h_{\text{heap}}, l_{\text{lcl}}) & \xrightarrow{\text{new Type}(e_1,\ldots,e_n) \Rightarrow na} (h_{\text{heap}’}, l_{\text{lcl}’})
\end{align*}
\]

where
\[
\begin{align*}
\text{na} & \notin \text{dom}(\text{heap}), \\
\text{heap}_1 & = \text{heap} \cup \{\text{na} \mapsto (\text{Type}, \langle 0,\ldots,0 \rangle)\}, \text{ and} \\
\text{<init>} & \text{ stands for the internal name of the constructor}
\end{align*}
\]
Formalizing Exceptions

In order to handle exceptions, a few changes in the semantics are necessary:

• We extend states by a flow component
  \[ Q = \text{Flow} \times \text{Heap} \times \text{Local} \]

• \( \text{Flow} ::= \text{Norm} \mid \text{Ret} \mid \text{Exc}(\langle \text{Address} \rangle) \)

We use the identifiers \( \text{flow} \in \text{Flow}, \text{heap} \in \text{Heap} \) and \( \text{lcl} \in \text{Local} \) in the rules. Also \( q \in Q \) stands for an arbitrary state.

In an abnormal state, statements are not executed:

\[
\begin{align*}
\text{(flow, heap, lcl)} & \xrightarrow{e \gg v} \text{(flow, heap, lcl)} & \text{where flow} \neq \text{Norm} \\
\text{(flow, heap, lcl)} & \xrightarrow{s} \text{(flow, heap, lcl)} & \text{where flow} \neq \text{Norm}
\end{align*}
\]
Rules for Expressions with Exceptions

The previously defined rules are valid only if the left-hand-state is not an abnormal state.

\[
\begin{align*}
(Norm, heap, lcl) \quad & e_1 \gg v_1 \quad q \quad q \quad e_2 \gg v_2 \quad q' \\
(Norm, heap, lcl) \quad & e_1 \ast e_2 \gg v_1 \cdot v_2 \bmod 2^{32} \quad q'
\end{align*}
\]

\[
\begin{align*}
(Norm, heap, lcl) \quad & s_1 \rightarrow q \quad q \quad s_2 \rightarrow q' \\
(Norm, heap, lcl) \quad & s_1 s_2 \rightarrow q'
\end{align*}
\]

Note that exceptions are propagated using the axioms from the previous slide

\[
(flow, heap, lcl) \quad e \gg v \quad (flow, heap, lcl) \quad \text{where } flow \neq Norm
\]
Rules for Throwing Exceptions

\[
\begin{align*}
(Norm, heap, lcl) \xrightarrow{e \gg v} (Norm, heap', lcl') \\
(Norm, heap, lcl) \xrightarrow{\text{throw } e;} (Exc(v), heap', lcl')
\end{align*}
\]

What happens if the object in a field access is \textbf{null}?

\[
\begin{align*}
(Norm, heap, lcl) \xrightarrow{e \gg 0} (Norm, heap', lcl') \\
(Norm, heap', lcl') \xrightarrow{\text{throw new } \text{NullPointerException}()} q' \\
(Norm, heap, lcl) \xrightarrow{e.fld \gg v} q'
\end{align*}
\]

where \( v \) is some arbitrary value
Complete Rules for `throw`

\[
\begin{align*}
(Norm, heap, lcl) & \xrightarrow{e \gg v} (Norm, heap', lcl') & \text{where } v \neq 0 \\
(Norm, heap, lcl) & \xrightarrow{\text{throw } e;} (Exc(v), heap', lcl') \\
(Norm, heap, lcl) & \xrightarrow{e \gg 0} (Norm, heap', lcl') \\
(Norm, heap', lcl') & \xrightarrow{\text{throw new } \text{NullPointerException}()} q' \\
(Norm, heap, lcl) & \xrightarrow{e.fld \gg v} q' & \text{where } v \text{ is some arbitrary value} \\
(Norm, heap, lcl) & \xrightarrow{e \gg v} (flow', heap', lcl') & \text{where } flow' \neq Norm
\end{align*}
\]
Rules for Catching Exceptions

• Catching an exception

\[
(Norm, heap, lcl) \xrightarrow{s_1} (Exc(v), heap', lcl')
\]

\[
(Norm, heap', lcl' \cup \{ex \mapsto v\}) \xrightarrow{s_2} q''
\]

\[
(Norm, heap, lcl) \xrightarrow{\text{try } s_1 \text{ catch } (Type \ ex) s_2} q''
\]

where \(v\) is an instance of \(Type\)

• No exception caught

\[
(Norm, heap, lcl) \xrightarrow{s_1} (flow', heap', lcl')
\]

\[
(Norm, heap, lcl) \xrightarrow{\text{try } s_1 \text{ catch } (Type \ ex) s_2} (flow', heap', lcl')
\]

where \(flow' \neq \text{Exc}(v)\) or \(v\) is not an instance of \(Type\)
Rules for **return** statements

- The **return** statement stores the return value in $\backslash \text{result}$ and signals $\text{Ret}$ in the flow component:

  $$
  (\text{Norm}, heap, lcl) \xrightarrow{e \gg v} (\text{Norm}, heap', lcl')
  $$

  $$
  (\text{Norm}, heap, lcl) \xrightarrow{\text{return } e;} (\text{Ret}, heap', lcl' + \{\backslash \text{result} \mapsto v\})
  $$

- But evaluating $e$ can also throw an exception

  $$
  (\text{Norm}, heap, lcl) \xrightarrow{e \gg v} (\text{flow'}, heap', lcl')
  $$

  $$
  (\text{Norm}, heap, lcl) \xrightarrow{\text{return } e;} (\text{flow'}, heap', lcl')
  $$

  where $\text{flow'} \neq \text{Norm}$
Method Call (Normal Case)

\[(Norm, heap, lcl) \xrightarrow{e \gg v} q_0\]
\[q_0 \xrightarrow{e_1 \gg v_1} q_1\]
\[\vdots\]
\[q_{n-1} \xrightarrow{e_n \gg v_n} (flow_n, heap_n, lcl_n)\]
\[(flow_n, heap_n, mlcl) \xrightarrow{\text{body}} (\text{Ret}, heap_{n+1}, mlcl')\]
\[(Norm, heap, lcl) \xrightarrow{e.m(e_1, \ldots, e_n) \gg mlcl'\text{(result)}} (Norm, heap_{n+1}, lcl_n)\]

where \textit{body} is the body of method \(m\) in the object \(heap_{n+1}(v)\), and \(mlcl = \{\text{\textbf{this}} \mapsto v, \text{param}_1 \mapsto v_1, \ldots \text{param}_n \mapsto v_n\}\) where \textit{param}_1, \ldots, \textit{param}_n are the names of the parameters of \(m\).
Method Call (Exception Case)

\[
(Norm, \text{heap}, lcl) \xrightarrow{e \gg v} q_0
\]

\[
q_0 \xrightarrow{e_1 \gg v_1} q_1
\]

\[
\vdots
\]

\[
q_{n-1} \xrightarrow{e_n \gg v_n} (\text{flow}_n, \text{heap}_n, lcl_n)
\]

\[
(\text{flow}_n, \text{heap}_n, mlcl) \xrightarrow{\text{body}} (\text{Exc}(v_e), \text{heap}_{n+1}, mlcl')
\]

\[
(Norm, \text{heap}, lcl) \xrightarrow{e.m(e_1, \ldots, e_n) \gg mlcl'(\text{\textbackslash result})} (\text{Exc}(v_e), \text{heap}_{n+1}, lcl_n)
\]

where \(body\) is the body of method \(m\) in the object \(heap_{n+1}(v)\), and \(mlcl = \{\textbf{this} \mapsto v, \, \text{param}_1 \mapsto v_1, \ldots \, \text{param}_n \mapsto v_n\}\) where \(\text{param}_1, \ldots, \text{param}_n\) are the names of the parameters of \(m\).
Semantics of Specifications

/*@ requires x >= 0;
@ ensures \result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
@*/

public static int isqrt(int x) {
    body
}

Whenever the method is called with values that satisfy the requires clause and the method terminates normally then the ensures clause holds.

If \( lcl(x) \geq x \) and

\[(Norm, heap, lcl) \xrightarrow{body}(Ret, heap', lcl')\]

then \( lcl'(\result) \leq Math.sqrt(lcl(x)) < lcl'(\result) + 1 \)

What about Exceptions?

/*@ ensures result <= Math.sqrt(x) && Math.sqrt(x) < result + 1;
  @ signals (IllegalArgumentException) x < 0;
  @ signals_only IllegalArgumentException;
  @*/
public static int isqrt(int x) { body }

For all transitions

\[(Norm, heap, lcl) \xrightarrow{\text{body}} (Exc(v), heap', lcl')\]

where \( lcl \) satisfies the precondition and \( v \) is an exception, \( v \) must be of type \texttt{IllegalArgumentException}. Furthermore, \( lcl \) must satisfy \( x < 0 \).
Side Effects

A method can change the heap in an unpredictable way.

The **assignable** clause restricts changes:

```java
/*@ requires x >= 0;
  assignable \nothing;
  ensures \result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
@*/
public static int isqrt(int x) {
  body

  If \( lcl(x) \geq x \) and
  \[ (Norm, heap, lcl) \xrightarrow{\text{body}} (Ret, heap', lcl') \]
  then \( lcl'(\result) \leq Math.sqrt(lcl(x)) < lcl'(\result) + 1 \)
  and \( heap = heap' \)
```
What Is the Meaning of a JML Formula?

A formula like \( x \geq 0 \) is a Boolean Java expression. It can be evaluated with the operational semantics.

\[ x \geq 0 \] holds in state \((\text{Norm}, \text{heap}, lcl)\) iff

\[ (\text{Norm}, \text{heap}, lcl) \xrightarrow{x \geq 0 \, \triangleright \, v} (\text{flow'}, \text{heap'}, lcl') \] where \( v \neq 0 \)

A formula may not have side effects but it can throw an exception.

For the ensures formula both the pre-state and the post-state are necessary to evaluate the formula.
Semantics of Specifications (formally)

A method satisfies the specification

\[
\text{requires } e_1; \\
\text{ensures } e_2;
\]

iff for all executions

\[(Norm, heap, lcl) \xrightarrow{body} (Ret, heap', lcl')\]

with

\[(Norm, heap, lcl) \xrightarrow{e_1 \gg v_1} q_1 \quad \text{where } v_1 \neq 0\]

the post-condition holds, i.e., there exist \(v_2, q_2\) s.t.

\[(Norm, heap', lcl') \xrightarrow{e_2 \gg v_2} q_2 \quad \text{where } v_2 \neq 0\]

- However, we need a new rule for evaluating \(\text{old}\)

\[
(Norm, heap, lcl) \xrightarrow{e \gg v} q
\]

\[
(Norm, heap', lcl') \xrightarrow{\text{old}(e) \gg v} q
\]

where \(heap, lcl\) refer to the state before \(body\) was executed.
Method Parameters in **ensures** Clauses

```java
/*@ requires x >= 0;
   @ assignable \nothing;
   @ ensures \result <= Math.sqrt(x) && Math.sqrt(x) < \result + 1;
   @*/

public static int isqrt(int x) {
    x = 0;
    return 0;
}
```

Is this code a correct implementation of the specification?

**No**, because method parameters are always evaluated in the pre-state, so

\[
\text{\result} <= \text{Math.sqrt}(x) && \text{Math.sqrt}(x) < \text{\result} + 1;
\]

has the same meaning as

\[
\text{\result} <= \text{Math.sqrt}(\text{old}(x)) && \text{Math.sqrt}(\text{old}(x)) < \text{\result} + 1;
\]
Side Effects in Specifications

In JML side effects in specifications are forbidden: If $e$ is an expression in a specification and

$$(Norm, heap, lcl) \xrightarrow{e \gg v} (flow', heap', lcl')$$

then $heap = heap'$ and $lcl = lcl'$. To be more precise, $heap \subseteq heap'$ since the new heap may contain new (unreachable) objects.

Also $flow' \neq Norm$ is allowed. In that case the value of $v$ may be unpredictable and the tools should assume the worst case, i.e., report that code is buggy.