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;
while x ≤ n do
        x := x + 1;
        p := p + m;
```

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 for $n \in \mathbb{Z}$
  | x for $x \in L$
  | $e_1 + e_2$
  | $e_1 - e_2$
  | $e_1 * e_2$
- 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 ::= true
  | false
  | $e_1 = e_2$ for $e_1, e_2 \in Aexp$
  | $e_1 \leq e_2$ for $e_1, e_2 \in Aexp$
  | $\neg\, b$ for $b \in Bexp$
  | $b_1 \wedge b_2$ for $b_1, b_2 \in Bexp$
  | $b_1 \vee b_2$ for $b_1, b_2 \in Bexp$

Syntax of Commands

- Commands (*Com*)
  c ::= `skip`
  | $x := e$
  | $c_1 ; c_2$
  | `if` b `then` c_1 `else` c_2
  | `while` b `do` c
- 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 $Q = L \to \mathbb{Z}$
- We shall use q to range over Q

Judgments

- We write <*e*, *q*> ⇓ *n* to mean that *e* evaluates to *n* in state *q*.
  - The formula <*e*, *q*> ⇓ *n* is a judgment (a statement about a relation between *e*, *q* and *n*)
  - In this case, we can view ⇓ 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 <e1 + e2, q> ⇓ … ?

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:

$$<n, q> \Downarrow n \longleftarrow \boxed{\text{Axiom}} \longrightarrow <x, q> \Downarrow q(x)$$

$$\frac{<e_1, q> \Downarrow n_1 \quad <e_2, q> \Downarrow n_2}{<e_1 + e_2, q> \Downarrow (n_1 + n_2)} \qquad \frac{<e_1, q> \Downarrow n_1 \quad <e_2, q> \Downarrow n_2}{<e_1 - e_2, q> \Downarrow (n_1 - n_2)}$$

$$\frac{<e_1, q> \Downarrow n_1 \quad <e_2, q> \Downarrow n_2}{<e_1 * e_2, q> \Downarrow (n_1 \cdot n_2)}$$

- This is called structural operational semantics.
  - rules are defined based on the structure of the expressions.

Inference Rules for *Bexp*

$$<true, q> \Downarrow \text{true} \qquad\qquad <\text{false}, q> \Downarrow \text{false}$$

$$\frac{<e_1, q> \Downarrow n_1 \quad <e_2, q> \Downarrow n_2}{<e_1 = e_2, q> \Downarrow (n_1 = n_2)} \qquad \frac{<e_1, q> \Downarrow n_1 \quad <e_2, q> \Downarrow n_2}{<e_1 \leq e_2, q> \Downarrow (n_1 \leq n_2)}$$

$$\frac{<b_1, q> \Downarrow t_1 \quad <e_2, q> \Downarrow t_2}{<b_1 \wedge b_2, q> \Downarrow (t_1 \wedge t_2)}$$

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:

$$\frac{<2, q> \Downarrow 2 \qquad <3, q> \Downarrow 3}{<2 * 3, q> \Downarrow 6}$$

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:

$$\frac{<x, \{x \mapsto 3, y \mapsto 2> \Downarrow 3 \quad \dfrac{\begin{array}{c}<y, \{x \mapsto 3, y \mapsto 2> \Downarrow 2 \\ <2, \{x \mapsto 3, y \mapsto 2> \Downarrow 2\end{array}}{<2 * y, \{x \mapsto 3, y \mapsto 2> \Downarrow 4}}{<x + (2 * y), \{x \mapsto 3, y \mapsto 2> \Downarrow 7}$$

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 transition from *q* to a post-state *q'*:
  $q \xrightarrow{c} q'$
- We can formalize this in terms of 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

$$q \xrightarrow{\texttt{skip}} q$$

$$\frac{<e, q> \Downarrow n}{q \xrightarrow{x := e} q \mathbin{++} \{x \mapsto n\}}$$

$$\frac{q \xrightarrow{c_1} q' \qquad q' \xrightarrow{c_2} q''}{q \xrightarrow{c_1\,;\,c_2} q''}$$

$$\frac{<b, q> \Downarrow \text{true} \qquad q \xrightarrow{c_1} q'}{q \xrightarrow{\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2} q'}$$

$$\frac{<b, q> \Downarrow \text{false} \qquad q \xrightarrow{c_2} q'}{q \xrightarrow{\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2} q'}$$

$$\frac{<b, q> \Downarrow \text{false}}{q \xrightarrow{\texttt{while}\ b\ \texttt{do}\ c} q}$$

$$\frac{<b, q> \Downarrow \text{true} \qquad q \xrightarrow{c} q' \qquad q' \xrightarrow{\texttt{while}\ b\ \texttt{do}\ c} q''}{q \xrightarrow{\texttt{while}\ b\ \texttt{do}\ c} q''}$$

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?

Operational Semantics of Java

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

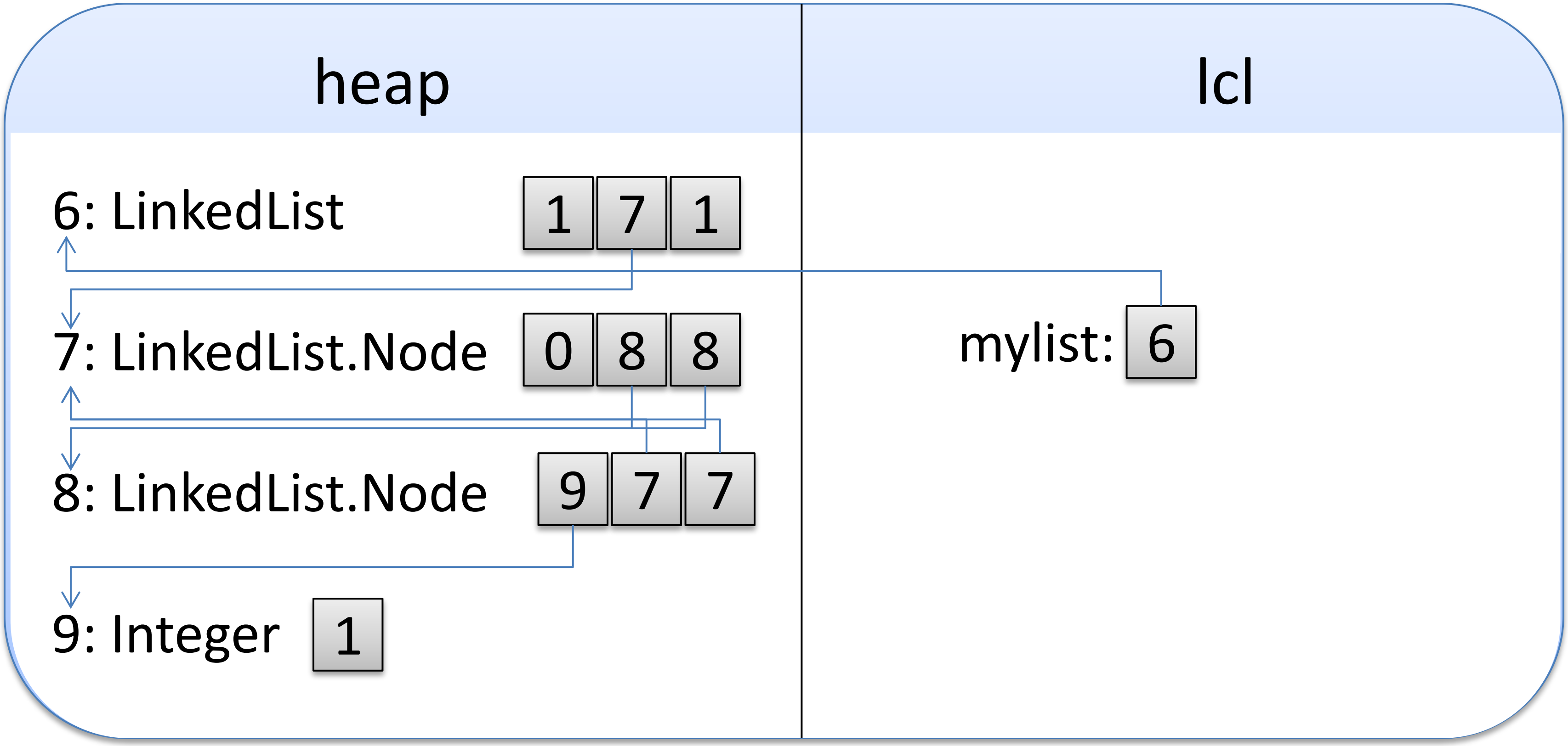
- Q is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq Q \times Act \times Q$ is a transition relation.

- 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

Example: State of a Java Program

What is the state after executing this code?

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



State of a Java Program

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

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

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 $e \gg v$, 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

$$(\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\}) \xrightarrow{\texttt{x++} \gg 5} (\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 6\})$$

- Pre-increment expression

$$(\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\}) \xrightarrow{\texttt{++x} \gg 6} (\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 6\})$$

- Assignment expression

$$(\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\}) \xrightarrow{\texttt{x=2*x} \gg 10} (\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 10\})$$

- Assignment statement

$$(\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\}) \xrightarrow{\texttt{x=2*x;}} (\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 10\})$$

Rules for Java Expressions (1)

- axiom for evaluating local variables

$$(heap, lcl) \xrightarrow{x \gg lcl(x)} (heap, lcl)$$

- rule for assignment to local

$$\frac{(heap, lcl) \xrightarrow{e \gg v} (heap', lcl')}{(heap, lcl) \xrightarrow{x=e \gg v} (heap', lcl' \mathbin{++} \{x \mapsto v\})}$$

- rule for field access

$$\frac{(heap, lcl) \xrightarrow{e \gg v} (heap', lcl')}{(heap, lcl) \xrightarrow{\text{e.fld} \gg \text{heap'(v)(idx)}} (heap', lcl')}$$

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

$$\frac{(heap, lcl) \xrightarrow{e_1 \gg v_1} (heap', lcl') \qquad (heap', lcl') \xrightarrow{e_2 \gg v_2} (heap'', lcl'')}{(heap, lcl) \xrightarrow{e_1 * e_2 \gg v_1 \cdot v_2 \bmod 2^{32}} (heap'', lcl'')}$$

- similarly for other binary operators

Example: Derivation for x=2*x

$$\dfrac{\dfrac{\begin{array}{c}(\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\}) \xrightarrow{\mathrm{x} \gg 5} (\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\}) \\ (\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\}) \xrightarrow{2 \gg 2} (\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\})\end{array}}{(\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\}) \xrightarrow{2*\mathrm{x} \gg 10} (\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\})}}{(\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 5\}) \xrightarrow{\mathrm{x}=2*\mathrm{x} \gg 10} (\mathit{heap}, \mathit{lcl} \cup \{\mathrm{x} \mapsto 10\})}$$

Rules for Java Statements (1)

- expression statement (assignment or method call)

$$\frac{(\mathit{heap}, \mathit{lcl}) \xrightarrow{e \gg v} (\mathit{heap}', \mathit{lcl}')}{(\mathit{heap}, \mathit{lcl}) \xrightarrow{e;} (\mathit{heap}', \mathit{lcl}')}$$

- sequence of statements

$$\frac{(\mathit{heap}, \mathit{lcl}) \xrightarrow{s_1} (\mathit{heap}', \mathit{lcl}') \qquad (\mathit{heap}', \mathit{lcl}') \xrightarrow{s_2} (\mathit{heap}'', \mathit{lcl}'')}{(\mathit{heap}, \mathit{lcl}) \xrightarrow{s_1\, s_2} (\mathit{heap}'', \mathit{lcl}'')}$$

Rules for Java Statements (2)

- rules for **`if`** statement

$$\frac{(heap, lcl) \xrightarrow{e \gg v} (heap', lcl') \qquad (heap', lcl') \xrightarrow{bl_1} (heap'', lcl'')}{(heap, lcl) \xrightarrow{\texttt{if}\ (e)\ \{bl_1\}\ \texttt{else}\ \{bl_2\}} (heap'', lcl'')} \quad \text{where } v \neq 0$$

$$\frac{(heap, lcl) \xrightarrow{e \gg v} (heap', lcl') \qquad (heap', lcl') \xrightarrow{bl_2} (heap'', lcl'')}{(heap, lcl) \xrightarrow{\texttt{if}\ (e)\ \{bl_1\}\ \texttt{else}\ \{bl_2\}} (heap'', lcl'')} \quad \text{where } v = 0$$

Rules for Java Statements (3)

- rules for **`while`** statement

$$\frac{(heap, lcl) \xrightarrow{e \gg v} (heap', lcl')}{(heap, lcl) \xrightarrow{\texttt{while}\ (e)\ \{bl\}} (heap', lcl')} \quad \text{where } v = 0$$

$$\frac{\begin{array}{l}(heap, lcl) \xrightarrow{e \gg v} (heap', lcl') \\ (heap', lcl') \xrightarrow{bl} (heap'', lcl'') \\ (heap'', lcl'') \xrightarrow{\texttt{while}\ (e)\ \{bl\}} (heap''', lcl''')\end{array}}{(heap, lcl) \xrightarrow{\texttt{while}\ (e)\ \{bl\}} (heap''', lcl''')} \quad \text{where } v \neq 0$$

Rule for Java Method Calls

$$\frac{\begin{array}{c} (heap, lcl) \xrightarrow{e \gg v} (heap_0, lcl_0) \\ (heap_0, lcl_0) \xrightarrow{e_1 \gg v_1} (heap_1, lcl_1) \\ \vdots \\ (heap_{n\text{-}1}, lcl_{n\text{-}1}) \xrightarrow{e_n \gg v_n} (heap_n, lcl_n) \\ (heap_n, mlcl) \xrightarrow{body} (heap_{n+1}, mlcl') \end{array}}{(heap, lcl) \xrightarrow{e.m(e_1,\ldots,e_n) \gg mlcl'(\backslash result)} (heap_{n+1}, lcl_n)}$$

where *body* is the body of method *m* in the object $heap_{n+1}(v)$, and $mlcl = \{\mathtt{\textbf{this}} \mapsto v,\ param_1 \mapsto v_1,\ \ldots\ param_n \mapsto v_n\}$ where $param_1, \ldots, param_n$ are the names of the parameters of *m.*

Rule for Object Creation

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

$$\frac{(heap_1, lcl) \xrightarrow{na.\texttt{<init>}(e_1,\ldots,e_n) \gg v} (heap', lcl')}{(heap, lcl) \xrightarrow{\textbf{new}\ Type(e_1,\ldots,e_n) \gg na} (heap', lcl')}$$

where

$na \notin \mathrm{dom}(heap)$,

$heap_1 = heap \cup \{na \mapsto (Type, \langle 0,\ldots,0 \rangle\}$, and

`<init>` stands for the internal name of the constructor

Formalizing Exceptions

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

- We extend states by a flow component
  $Q = Flow \times Heap \times Local$
- $Flow ::= Norm \mid Ret \mid Exc\langle\langle Address\rangle\rangle$

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

In an abnormal state, statements are not executed:

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

$$(flow, heap, lcl) \xrightarrow{s} (flow, heap, lcl) \quad \text{where } flow \neq Norm$$

Rules for Expressions with Exceptions

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

$$\frac{(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{e_1 \gg v_1} q \qquad q \xrightarrow{e_2 \gg v_2} q'}{(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{e_1 * e_2 \gg v_1 \cdot v_2 \bmod 2^{32}} q'}$$

$$\frac{(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{s_1} q \qquad q \xrightarrow{s_2} q'}{(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{s_1\, s_2} q'}$$

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

$$(\mathit{flow}, \mathit{heap}, \mathit{lcl}) \xrightarrow{e \gg v} (\mathrm{flow}, \mathit{heap}, \mathit{lcl}) \quad \text{where } \mathit{flow} \neq \mathit{Norm}$$

Rules for Throwing Exceptions

$$\frac{(Norm, heap, lcl) \xrightarrow{e \gg v} (Norm, heap', lcl')}{(Norm, heap, lcl) \xrightarrow{\texttt{throw}\ e;} (Exc(v), heap', lcl')}$$

What happens if the object in a field access is **`null`**?

$$\frac{\begin{array}{c}(Norm, heap, lcl) \xrightarrow{e \gg 0} (Norm, heap', lcl') \\ (Norm, heap', lcl') \xrightarrow{\texttt{throw new}\ \text{NullPointerException();}} q'\end{array}}{(Norm, heap, lcl) \xrightarrow{e.fld \gg v} q'}$$

where *v* is some arbitrary value

Complete Rules for `throw`

$$\frac{(Norm, heap, lcl) \xrightarrow{e \gg v} (Norm, heap', lcl')}{(Norm, heap, lcl) \xrightarrow{\texttt{throw}\ e;} (Exc(v), heap', lcl')} \quad \text{where } v \neq 0$$

$$\frac{\begin{array}{c}(Norm, heap, lcl) \xrightarrow{e \gg 0} (Norm, heap', lcl') \\ (Norm, heap', lcl') \xrightarrow{\texttt{throw new}\ \text{NullPointerException();}} q'\end{array}}{(Norm, heap, lcl) \xrightarrow{e.fld \gg v} q'}$$

where *v* is some arbitrary value

$$\frac{(Norm, heap, lcl) \xrightarrow{e \gg v} (flow', heap', lcl')}{(Norm, heap, lcl) \xrightarrow{\texttt{throw}\ e;} (flow', heap', lcl')} \quad \text{where } flow' \neq Norm$$

Rules for Catching Exceptions

- Catching an exception

$$\frac{\begin{array}{c}(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{s_1} (\mathit{Exc}(v), \mathit{heap}', \mathit{lcl}') \\ (\mathit{Norm}, \mathit{heap}', \mathit{lcl}' \cup \{\mathit{ex} \mapsto v\}) \xrightarrow{s_2} \mathrm{q}''\end{array}}{(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{\mathbf{try}\ s_1\ \mathbf{catch}\ (\mathit{Type}\ \mathit{ex})\ s_2} \mathrm{q}''}$$

where v is an instance of *Type*

- No exception caught

$$\frac{(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{s_1} (\mathit{flow}', \mathit{heap}', \mathit{lcl}')}{(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{\mathbf{try}\ s_1\ \mathbf{catch}\ (\mathit{Type}\ \mathit{ex})\ s_2} (\mathit{flow}', \mathit{heap}', \mathit{lcl}')}$$

where $\mathit{flow}' \neq \mathit{Exc}(v)$ or v is not an instance of *Type*

Rules for **`return`** statements

- The **`return`** statement stores the return value in \result and signals *Ret* in the flow component:

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

- But evaluating *e* can also throw an exception

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

where $flow' \neq Norm$

Method Call (Normal Case)

$$\frac{\begin{array}{c}(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{e \gg v} q_0 \\ q_0 \xrightarrow{e_1 \gg v_1} q_1 \\ \vdots \\ q_{n\text{-}1} \xrightarrow{e_n \gg v_n} (\mathit{flow}_n, \mathit{heap}_n, \mathit{lcl}_n) \\ (\mathit{flow}_n, \mathit{heap}_n, \mathit{mlcl}) \xrightarrow{\mathit{body}} (\mathrm{Ret}, \mathit{heap}_{n+1}, \mathit{mlcl}')\end{array}}{(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{e.m(e_1,\ldots,e_n) \gg \mathit{mlcl}'(\backslash\mathit{result})} (\mathit{Norm}, \mathit{heap}_{n+1}, \mathit{lcl}_n)}$$

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

Method Call (Exception Case)

$$\frac{\begin{array}{c}(Norm, heap, lcl) \xrightarrow{e \gg v} q_0 \\ q_0 \xrightarrow{e_1 \gg v_1} q_1 \\ \vdots \\ q_{n\text{-}1} \xrightarrow{e_n \gg v_n} (flow_n, heap_n, lcl_n) \\ (flow_n, heap_n, mlcl) \xrightarrow{body} (Exc(v_e), heap_{n+1}, mlcl')\end{array}}{(Norm, heap, lcl) \xrightarrow{e.m(e_1,\ldots,e_n) \gg mlcl'(\backslash result)} (Exc(v_e), heap_{n+1}, lcl_n)}$$

where *body* is the body of method *m* in the object $heap_{n+1}(v)$, and $mlcl = \{\texttt{this} \mapsto v, param_1 \mapsto v_1, \ldots\ param_n \mapsto v_n\}$ where $param_1, \ldots, param_n$ 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 $lcl(\mathrm{x}) >= \mathrm{x}$ and

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

then $lcl'(\backslash\mathrm{result}) <= \mathrm{Math.sqrt}(lcl(\mathrm{x})) < lcl'(\backslash\mathrm{result}) + 1$

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

$$(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{\mathit{body}} (\mathit{Exc}(v), \mathit{heap}', \mathit{lcl}')$$

where *lcl* satisfies the precondition and *v* is an exception, *v* must be of type `IllegalArgumentException`. Furthermore, *lcl* must satisfy `x < 0`.

Side Effects

A method can change the heap in an unpredictable way.

The `assignable` clause restricts changes:

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

If $lcl(x) >= x$ and

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

then $lcl'(\backslash result) <= Math.sqrt(lcl(x)) < lcl'(\backslash result) + 1$

and $heap = heap'$

What Is the Meaning of a JML Formula?

A formula like `x >= 0` is a Boolean Java expression. It can be evaluated with the operational semantics.

`x >= 0` holds in state (*Norm*, *heap*, *lcl*) iff

$$(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{\texttt{x >= 0} \gg v} (\mathit{flow}', \mathit{heap}', \mathit{lcl}') \quad \text{where } v \neq 0$$

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

requires e_1;
ensures e_2;

iff for all executions

$$(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{\mathit{body}} (\mathit{Ret}, \mathit{heap}', \mathit{lcl}')$$

with $(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{e_1 \gg v_1} q_1$ where $v_1 \neq 0$

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

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

- However, we need a new rule for evaluating **`\old`**

$$\frac{(\mathit{Norm}, \mathit{heap}, \mathit{lcl}) \xrightarrow{e \gg v} q}{(\mathit{Norm}, \mathit{heap}', \mathit{lcl}') \xrightarrow{\texttt{\textbackslash old(e)} \gg v} q}$$

where *heap, lcl* refer to the state before *body* was executed.

Method Parameters in **ensures** Clauses

```
/*@ 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

\result <= Math.sqrt(x) && Math.sqrt(x) < **\result** + 1;

has the same meaning as

`\result <= Math.sqrt(\old(x)) && Math.sqrt(\old(x)) < \result + 1;`

Side Effects in Specifications

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

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

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

Also flow’ ≠ 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.