Syntax of a Java class

$\langle$`public`$\rangle$ $\langle$`abstract`$\rangle$ $\langle$`final`$\rangle$
`class` A $\langle$`extends` $B\rangle$ $\langle$`implements` $I_1, \ldots, I_n\rangle$ $\{$

$\vdots$

constructor declarations

$\vdots$

field declarations

$\vdots$

method declarations

$\vdots$

static initializers

$\vdots$

$\}$

$A \prec_{\mathrm{d}} B$: A is a direct subclass of B or B is a direct superclass of A.

$A \prec_{\mathrm{d}} I_j$: $I_1, \ldots, I_n$ are direct superinterfaces of A.

Syntax of a Java interface

$$\langle\texttt{public}\rangle\ \texttt{interface}\ I\ \langle\texttt{extends}\ J_1, \ldots, J_n\rangle\ \{$$

⋮

constant declarations

⋮

abstract method declarations

⋮

$\}$

$I \prec_{\mathrm{d}} J_i$: I is a direct subinterface of $J_1, \ldots, J_n$.

Definition. Let $\prec_{\mathrm{h}}$ be the transitive closure of $\prec_{\mathrm{d}}$.

$A \prec_{\mathrm{h}} B$: A is a subclass of B or B is a superclass of A.

$A \prec_{\mathrm{h}} I$: A implements I or I is superinterface of A.

$I \prec_{\mathrm{h}} J$: I is a subinterface of J or J is a superinterface of I.

Definition. $A \preceq_{\mathrm{h}} B \ :\Longleftrightarrow\ A \prec_{\mathrm{h}} B$ or $A = B$.

The inheritance relation

Constraint. $\prec_{\mathrm{h}}$ must be acyclic.

Consequence. $\neg\,(A \prec_{\mathrm{h}} A)$

Lemma. The relation $\preceq_{\mathrm{h}}$ is a partial ordering:

1. $A \preceq_{\mathrm{h}} A$.
2. If $A \preceq_{\mathrm{h}} B$ and $B \preceq_{\mathrm{h}} C$, then $A \preceq_{\mathrm{h}} C$.
3. If $A \preceq_{\mathrm{h}} B$ and $B \preceq_{\mathrm{h}} A$, then $A = B$.

The relation $\preceq_{\mathrm{h}}$ restricted to classes is a finite tree.

Lemma. Let A, B, C be classes. Then we have:

- $A \preceq_{\mathrm{h}}$ `Object`
- If $A \prec_{\mathrm{d}} B$ and $A \prec_{\mathrm{d}} C$, then $B = C$.
- If $A \preceq_{\mathrm{h}} B$ and $A \preceq_{\mathrm{h}} C$, then $B \preceq_{\mathrm{h}} C$ or $C \preceq_{\mathrm{h}} B$.

Packages

Definition. A package is a set of classes and interfaces.

Definition. A $\text{Java}_{\mathcal{C}}$ program is a set of packages.

Package statement:

```
package ch.ethz.inf.staerk;
```

Fully qualified names:

```
ch.ethz.inf.staerk.Point3D
```

Definition. A type B is accessible from A, if one of the following conditions is true:

- B is a primitive type, or
- B is in the same package as A, or
- B is `public`.

Constraint. If $A \prec_{\mathrm{d}} B$, then B must be acessible from A.

Syntax of Java$_{\mathcal{C}}$

Class .... (fully qualified) class and interface names,
Field ..... field names (identifiers),
Meth .... method names (identifiers),
Invk ..... method invocations.

$$
\begin{aligned}
\mathit{Exp} &:= \ldots \mid \mathit{Field} \mid \mathit{Class}.\mathit{Field} \mid \mathit{Invk} \\
\mathit{Asgn} &:= \ldots \mid \mathit{Field} = \mathit{Exp} \mid \mathit{Class}.\mathit{Field} = \mathit{Exp} \\
\mathit{Exps} &:= \mathit{Exp}_1, \ldots, \mathit{Exp}_n \\
\mathit{Invk} &:= \mathit{Meth}(\mathit{Exps}) \mid \mathit{Class}.\mathit{Meth}(\mathit{Exps}) \\
\mathit{Stm} &:= \ldots \mid \mathit{Invk}; \mid \texttt{return}\ \mathit{Exp}; \mid \texttt{return}; \\
\mathit{Phrase} &:= \ldots \mid \texttt{static}\ \mathit{Block}
\end{aligned}
$$

Class members

Field declaration in class C:

$\langle \texttt{public} \mid \texttt{protected} \mid \texttt{private} \rangle\ \langle \texttt{final} \rangle\ \langle \texttt{static} \rangle\ A\ \mathit{field}\ \langle = \mathit{exp} \rangle;$

C/field static $\Longrightarrow$ class field

C/field not static $\Longrightarrow$ instance field

Method declaration in class C:

$\langle \texttt{public} \mid \texttt{protected} \mid \texttt{private} \rangle$

$\langle \texttt{abstract} \rangle\ \langle \texttt{final} \rangle\ \langle \texttt{static} \rangle\ \langle \texttt{native} \rangle$

$A\ \mathit{meth}(B_1\ \mathit{loc}_1, \ldots, B_n\ \mathit{loc}_n)\ \mathit{body}$

$\mathit{body} ::= \text{`;'} \mid \mathit{block}$

Method signature $\mathit{msig} = \mathit{meth}(B_1, \ldots, B_n)$

C/msig static $\Longrightarrow$ class method

C/msig not static $\Longrightarrow$ instance method

Static initializer in class C:

$\texttt{static}\ \mathit{block}$

Interface members

Constant declaration in interface I:

$$\boxed{A\; field = exp;}$$

$I/field$ is `public`, `static`, `final`.

Abstract method declaration in interface I:

$$\boxed{A\; meth(B_1\; loc_1, \ldots, B_n\; loc_n);}$$

Signature $msig = meth(B_1, \ldots, B_n)$

$I/msig$ is `public` and `abstract` (and not `static`).

Accessibility and visibility

Definition. An element C/x is accessible from A (with respect to B) iff

- x is `private` in C and $A = C$, or
- x is not `private` in C and C is in the same package as A, or
- x is `public` in C, or
- x is `protected` in C and $A \prec_{\mathrm{h}} C$ (and $B \preceq_{\mathrm{h}} A$).

Definition. The visibility of members is defined inductively:

- If x is declared in A, then A/x is visible in A.
- If $A \prec_{\mathrm{d}} B$, C/x is visible in B, x is not declared in A and C/x is accessible from A, then C/x is visible in A.

Examples (visibility)

Example 1:

```
class A {
  public  static int i = 2;
  private static int j = 3;
}
class B extends A {
   public static int i = 4;
}
```

Example 2:

```
interface I {
  int MAX = 100;
}
class A implements I {}
class B extends A implements I {}
```

Example: The meaning of private?

```
class A {
  private int i = 7;

  public static void main(String[] argv) {
    B x = new B();
    System.out.println(x.i);
  }
}

class B extends A { }
```

```
JDK version "1.1.7"
tomis> javac Test.java
tomis> java A
7
```

```
JDK version "1.3.0"
tomis> javac Test.java
Test.java:6: i has private access in A
    System.out.println(x.i);
                        ^
```

Method overriding

Definition. A method $A/msig$ directly overrides a method $C/msig$, if there is a class or interface B such that

- $A \prec_{\mathrm{d}} B$,
- $C/msig$ is visible in B,
- $C/msig$ is accessible from A.

Definition. The relation 'overrides' is the reflexive, transitive closure of 'direct overrides'

Method overriding (continued)

Constraint. If $A/msig$ directly overrides $C/msig$, then the following constraints must be satisfied:

- The return type of $msig$ in A is the same as in C.
- Method $msig$ is not `final` in C.
- Method $msig$ is `static` in A if, and only if, it is `static` in C.
- Method $msig$ is not `private` in A.
- If $msig$ is `public` in C, then $msig$ is `public` in A.
- If $msig$ is `protected` in C, then $msig$ is `public` or `protected` in A.

The access may not decrease according to the following ordering:

$$\texttt{private} < \texttt{default} < \texttt{protected} < \texttt{public}$$

Implementing

Constraint. If two methods $B/msig$ and $C/msig$ with the same signature are both visible in A, then the following constraints must be satisfied:

- $msig$ has the same return type in B and C,
- If $msig$ is `public` in B, then $msig$ is `public` in C.
- If $msig$ is not `static` in B, then $msig$ is not `static` in C.

Definition. A class A implements a method $msig$, if there exists a class B such that

- $A \preceq_{\mathrm{h}} B$ and $msig$ is declared in B,
- $B/msig$ is visible in A,
- $msig$ is not `abstract` in B.

Constraint. If the `abstract` method $C/msig$ is visible in class A and A does not implement $msig$, then A is `abstract`.

Examples (implementing abstract methods)

```
interface I { int m(int i); }

class B {
  public int m(int i) { return i * i; }
}
class A extends B implements I { }

class B {
  private int m(int i) { return i * i; }
}
abstract class A extends B implements I { }

class B {
  int m(int i) { return i * i; }
}
abstract class A extends B implements I { }
```

Class field access expressions in class A

$$\left.\begin{array}{l} B.\mathit{field} \\ \mathit{field} \end{array}\right\} \Longrightarrow C.\mathit{field}$$

[Compiler]

Case 1. If C is unique with

- field is declared in C
- C/field is visible in B and accessible from A

Syntaxerror, if field is not `static` in C.

Case 2. If field is not in scope of a local variable declaration of field and C is unique with

- field is declared in C
- C/field is visible in A

and if field is `static` in C.

Class method invocation expressions in class A

$$\left.\begin{array}{l} {}^{\alpha}meth^{\beta}(exps) \\ {}^{\alpha}C.meth^{\beta}(exps) \end{array}\right\} \Longrightarrow {}^{\alpha}D.m^{\beta}(exps)$$ [Compiler]

Let $msig = meth(\mathcal{T}(\beta))$.

Case 1. Let $app(\alpha)$ be the set of all methods D/m such that

1. $A/msig$ is more specific than D/m and
2. D/m is visible in A.

Case 2. Let $app(\alpha)$ be the set of all methods D/m such that

1. $C/msig$ is more specific than D/m and
2. D/m is visible in C and accessible from A with respect to C.

Definition. $C/meth(A_1, \ldots, A_n)$ is more specific than $D/meth(B_1, \ldots, B_n)$, iff $C \preceq_{\mathrm{h}} D$ and $A_i \preceq B_i$ for $i = 1, \ldots, n$.

Method invocation expressions in class A (continued)

Method resolution:

Assume that $app(\alpha)$ contains a most specific element D/m, i.e.,

- $D/m \in app(\alpha)$
- If $E/k \in app(\alpha)$, then D/m is more specific than E/k

Assume that m is `static` in D.

Then D/m is the method chosen by the compiler.

Examples (overloaded methods)

Example 1:

```
class A {
  static void m(double d) {}
  static void m(long l) {}
  static void test(int i) {
    m(i); // Method m(long) is chosen.
  }
}
```

Example 2:

```
class A {
  static void m(int x,long y) {}
  static void m(long x,int y) {
    m(0,0); // Reference to m is ambiguous.
  }
}
```

Examples (overloaded methods)

Example 3:

```
class A {
  static void m(int x) {}
}

class B extends A {
  static void m(long x) {
    m(0); // Reference to m is ambiguous.
  }
}
```

Type constraints for Java$_\mathcal{C}$

${}^{\alpha}C.\mathit{field}$	$\mathcal{T}(\alpha)$ is the declared type of field in C.
${}^{\alpha}(C.\mathit{field} = {}^{\beta}\mathit{exp})$	$\mathcal{T}(\alpha)$ is the declared type of field in C, field is not `final` in C, $\mathcal{T}(\beta) \preceq \mathcal{T}(\alpha)$.
${}^{\alpha}C.\mathit{msig}(\mathit{exps})$	$\mathcal{T}(\alpha)$ is the declared return type of msig in class C.
`return` ${}^{\alpha}\mathit{exp};$	If the position α is in the body of a method with return type A, then $\mathcal{T}(\alpha) \preceq A$.

Example:

```
class Test {
  static long m(int i) {
    return i;
  }
}
```

Type constraints after introduction of primitive type casts

$^{\alpha}(C.field = {}^{\beta}exp)$	Let D be the declared type of $field$ in C. If D is primitive, then $\mathcal{T}(\beta) = D = \mathcal{T}(\alpha)$.
$^{\alpha}C.msig({}^{\beta_1}exp_1, \ldots, {}^{\beta_n}exp_n)$	If $msig = meth(B_1, \ldots, B_n)$ and B_i is a primitive type, then $\mathcal{T}(\beta_i) = B_i$.
`return` $^{\alpha}exp;$	If the position α is in the body of a method with a primitive return type A, then $\mathcal{T}(\alpha) = A$.

Vocabulary of the ASM for Java$_\mathcal{C}$

Universes:

MSig ...... method signatures
$\mathit{ClassState}$ .. $\mathit{Linked} \mid \mathit{InProgress} \mid \mathit{Initialized} \mid \mathit{Unusable}$
Frame ...... $(\mathit{Class}/\mathit{MSig}, \mathit{Phrase}, \mathit{Pos}, \mathit{Locals})$
Abr .... $\mathit{Break}(\mathit{Lab}) \mid \mathit{Continue}(\mathit{Lab}) \mid \mathit{Return} \mid \mathit{Return}(\mathit{Val})$

Static functions:

$super: \mathit{Class} \to \mathit{Class}$
$body\ : \mathit{Class}/\mathit{MSig} \to \mathit{Block}$

Dynamic functions and constants:

$classState: \mathit{Class} \to \mathit{ClassState}$
$globals\ : \mathit{Class}/\mathit{Field} \to \mathit{Val}$
$meth\ : \mathit{Class}/\mathit{MSig}$
$frames\ : \mathit{Frame}^*$

Transition rules for Java$_{\mathcal{C}}$

Initial state of Java$_{\mathcal{C}}$:

$$
\begin{aligned}
meth &= \texttt{Main/main}()\\
restbody &= body(meth)\\
pos &= \mathit{firstPos}\\
locals &= \emptyset\\
frames &= []\\
globals &= \emptyset\\
classState(c) &= Linked, \quad \text{for all classes } c
\end{aligned}
$$

Main transition rule for Java$_{\mathcal{C}}$:

$$
\begin{aligned}
&execJava_C =\\
&\quad execJavaExp_C\\
&\quad execJavaStm_C
\end{aligned}
$$

Transition rules for Java$_{\mathcal{C}}$ (continued)

Derived predicates:

$$initialized(c) =$$
$$\quad classState(c) = Initialized \lor classState(c) = InProgress$$

$$propagatesAbr(phrase) =$$
$$\quad phrase \neq lab : s \land$$
$$\quad phrase \neq \texttt{static}\ s$$

Rule macro:

$$initialize(c) =$$
$$\quad \textbf{if } classState(c) = Linked \textbf{ then}$$
$$\qquad classState(c) := InProgress$$
$$\qquad \textbf{forall } f \in staticFields(c)$$
$$\qquad\quad globals(f) := defaultVal(type(f))$$
$$\qquad invokeMethod(pos, c/\texttt{<clinit>}, [\,])$$

Transition rules for Java$_{\mathcal{C}}$ (Expressions)

$$
\begin{array}{l}
execJavaExp_C = \textbf{case } context(pos) \textbf{ of} \\
\quad c.f \qquad\qquad \rightarrow \textbf{if } initialized(c) \textbf{ then } yield(globals(c/f)) \textbf{ else } initialize(c) \\
\quad c.f = {}^{\alpha}exp \rightarrow pos := \alpha \\
\quad c.f = {}^{\blacktriangleright}val \rightarrow \textbf{if } initialized(c) \textbf{ then} \\
\qquad\qquad\qquad\qquad globals(c/f) := val \\
\qquad\qquad\qquad\qquad yieldUp(val) \\
\qquad\qquad\qquad\quad \textbf{else } initialize(c)
\end{array}
$$

$$
\begin{array}{l}
\quad c.m^{\alpha}(exps) \rightarrow pos := \alpha \\
\quad c.m^{\blacktriangleright}(vals) \rightarrow \textbf{if } initialized(c) \textbf{ then } invokeMethod(up(pos), c/m, vals) \\
\qquad\qquad\qquad\qquad \textbf{else } initialize(c)
\end{array}
$$

$$
\begin{array}{ll}
\quad (\,) & \rightarrow yield([\,]) \\
\quad ({}^{\alpha_1}exp_1, \ldots, {}^{\alpha_n}exp_n) & \rightarrow pos := \alpha_1 \\
\quad ({}^{\alpha_1}val_1, \ldots, {}^{\blacktriangleright}val_n) & \rightarrow yieldUp([val_1, \ldots, val_n]) \\
\quad ({}^{\alpha_1}val_1, \ldots, {}^{\blacktriangleright}val_i, {}^{\alpha_{i+1}}exp_{i+1} \ldots {}^{\alpha_n}exp_n) & \rightarrow pos := \alpha_{i+1}
\end{array}
$$

Transition rules for Java$_{\mathcal{C}}$ (Method Invocation)

$$
\begin{array}{l}
invokeMethod(nextPos, c/m, values) \\
\quad | \; Native \in modifiers(c/m) = \\
\quad\quad invokeNative(c/m, values) \\
\quad | \; \textbf{otherwise} = \\
\quad\quad \begin{array}{ll}
frames & := push(frames, (meth, restbody, nextPos, locals)) \\
meth & := c/m \\
restbody & := body(c/m) \\
pos & := firstPos \\
locals & := zip(argNames(c/m), values)
\end{array}
\end{array}
$$

Transition rules for Java$_{\mathcal{C}}$ (Statements)

$$
\begin{array}{l}
\mathit{execJavaStm}_C = \textbf{case } \mathit{context}(\mathit{pos}) \textbf{ of} \\
\quad \texttt{static}\ {}^{\alpha}\mathit{stm} \rightarrow \textbf{let } c = \mathit{classNm}(\mathit{meth}) \\
\qquad \textbf{if } c = \texttt{Object} \lor \mathit{initialized}(\mathit{super}(c)) \textbf{ then } \mathit{pos} := \alpha \\
\qquad \textbf{else } \mathit{initialize}(\mathit{super}(c)) \\
\quad \texttt{static}\ {}^{\alpha}\mathit{Return} \rightarrow \mathit{yieldUp}(\mathit{Return})
\end{array}
$$

$$
\begin{array}{ll}
\texttt{return}\ {}^{\alpha}\mathit{exp}; & \rightarrow \mathit{pos} := \alpha \\
\texttt{return}\ {}^{\blacktriangleright}\mathit{val}; & \rightarrow \mathit{yieldUp}(\mathit{Return}(\mathit{val})) \\
\texttt{return}; & \rightarrow \mathit{yield}(\mathit{Return}) \\
\mathit{lab} : {}^{\blacktriangleright}\mathit{Return} & \rightarrow \mathit{yieldUp}(\mathit{Return}) \\
\mathit{lab} : {}^{\blacktriangleright}\mathit{Return}(\mathit{val}) & \rightarrow \mathit{yieldUp}(\mathit{Return}(\mathit{val})) \\
\mathit{Return} & \rightarrow \textbf{if } \mathit{pos} = \mathit{firstPos} \land \neg \mathit{null}(\mathit{frames}) \textbf{ then} \\
 & \qquad \mathit{exitMethod}(\mathit{Norm}) \\
\mathit{Return}(\mathit{val}) & \rightarrow \textbf{if } \mathit{pos} = \mathit{firstPos} \land \neg \mathit{null}(\mathit{frames}) \textbf{ then} \\
 & \qquad \mathit{exitMethod}(\mathit{val})
\end{array}
$$

$$
{}^{\blacktriangleright}\mathit{Norm}; \rightarrow \mathit{yieldUp}(\mathit{Norm})
$$

Transition rules for Java$_{\mathcal{C}}$ (continued)

$$
\begin{array}{l}
exitMethod(result) = \\
\quad \textbf{let } (oldMeth, oldPgm, oldPos, oldLocals) = top(frames) \\
\quad meth := oldMeth \\
\quad pos := oldPos \\
\quad locals := oldLocals \\
\quad frames := pop(frames) \\
\quad \textbf{if } methNm(meth) = \texttt{"<clinit>"} \land result = Norm \textbf{ then} \\
\qquad restbody := oldPgm \\
\qquad classState(classNm(meth)) := Initialized \\
\quad \textbf{else} \\
\qquad restbody := oldPgm[result/oldPos]
\end{array}
$$