6.4.2 Scope, Visibility and Inheritance

The scope of a method or field is the class and the scope of a local variable is the method it occurs in. Within a method we have the following rules of visibility.

- local variables < fields of the class < fields of its superclass < fields of the superclass of its superclass ...
- methods of the class < methods of its superclass < methods of the superclass of its superclass ...
- in Java: classes in the same package < imported classes

Hidden fields or methods can be made visible by using a prefix notation like `A.y` or `super.m()` in constructors (in Java).

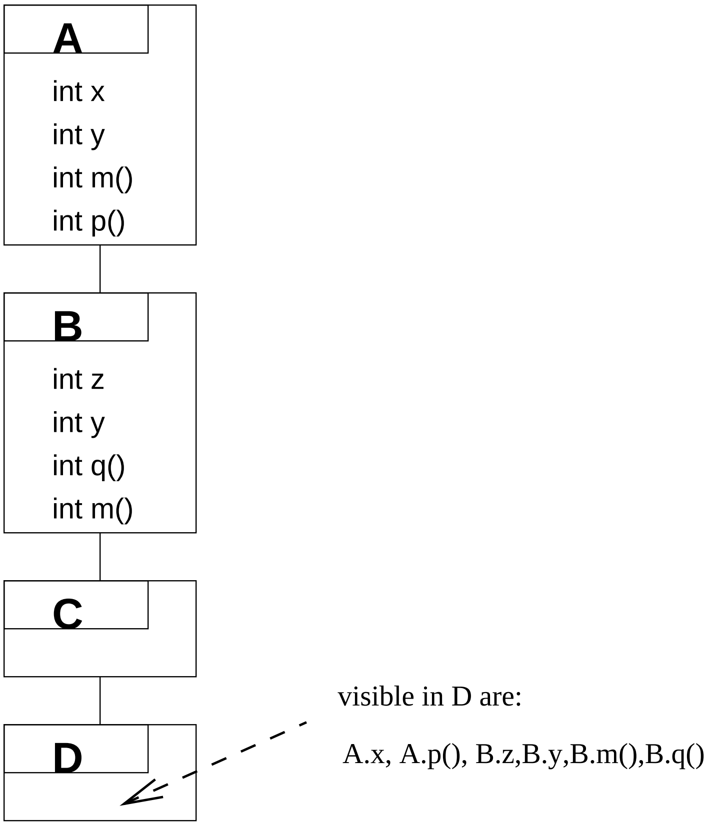

Figure 3: Visibility and Inheritance

In JVM instructions like `invokevirtual`, `putstatic` or `getfield` one uses the signatures of the visibly, defining occurrences as an argument. For example if we consider the inheritance hierarchy in Figure 3 and define `C` as

```
class D extends C { int v=C.m(); }
```

Then the above call to m is translated into (in JASMIN syntax):

```
invokestatic B/m()V
```

As a consequence the compiler has to compute the visible, defining occurrence for each using occurrence of field or method. First we introduce some notation for the inheritance relation on classes:

- $c \downarrow d$ iff c inherits directly from b
- $c \downarrow^0 c$
- $c \downarrow^1 d$ iff $c \downarrow b$
- $c \downarrow^n e$ iff $c \downarrow^{n-1} d$ and $d \downarrow e$
- $c \downarrow^* d$ if $c \downarrow^n d$
- $c \downarrow^+ d$ if $c \downarrow^n d$ and $n > 0$
- $methods(c) = \{m(t_1 \ldots t_k)t \mid$ class c contains a definition of a method with signature $m(t_1 \ldots t_k)t\}$
- $fields(c) = \{(v, t) \mid$ class c contains a definition of a field v of type $t\}$

Overriding of methods is restricted. A method overrides an inherited method if it has the same name and the same number and types of arguments. In this case it must also have the same result type:

- $\forall m(t_1 \ldots t_k)t \in methods(c) : \forall d : c \downarrow^+ d : \forall m(t_1 \ldots t_k)t' \in methods(d) : t = t'$

Note, that there is no similar restriction on fields, i.e. a field needn't have the type of the field it overrides.

With the above notation we can now define the visible, defining occurrences within a class:

- $vis\_meths(c) = \{d/m(t_1 \ldots t_k)t \mid m(t_1 \ldots t_k)t \in methods(d),\ c \downarrow^n d$ and n is minimal $\}$
- $vis\_fields(c) = \{(d/v, t) \mid (v, t) \in fields(d),\ c \downarrow^n d$ and n is minimal $\}$

6.4.3 Type Inference

When generating code for a method call, we have to infer the types of its argument expressions first. Then we can use the method name and these types to identify the method and look up its result type in the environment. Type inference computes the type of every expression in the program. It proceeds bottom-up, i.e. the types of variables and constants are known. If we know the types of its subexpressions we can compute the type of a compound expression.

$\mathit{infer\_type}\ c\ \rho = t$	where c is a constant of type t
$\mathit{infer\_type}\ x\ \rho = t$	where x is a local variable and $(t, a) = \rho(x)$
$\mathit{infer\_type}\ (e_1\ op\ e_2)\ \rho = t_1 \tilde{op} t_2$	where $t_1 = \mathit{infer\_type}\ e_1\ \rho$, $t_2 = \mathit{infer\_type}\ e_2\ \rho$ and $\tilde{op}$ is the abstract operation on types induced by *op*.
$\mathit{infer\_type}\ o.x\ \rho = t$	where x is field, $\mathit{infer\_type}\ o\ \rho = t_0$ and $(d/x, t) \in \mathit{vis\_fields}(t_0)$
$\mathit{infer\_type}\ x\ \rho = t$	as above, but $t_0 = c$ where c is the current class
$\mathit{infer\_type}\ c.x\ \rho = t$	as above, but $t_0 = c$
$\mathit{infer\_type}\ o.m(e_1, \ldots, e_k)\ \rho = t$	where $t_0 = \mathit{infer\_type}\ o\ \rho$, $t_1 = \mathit{infer\_type}\ e_1\ \rho$, ..., $e_k = \mathit{infer\_type}\ e_k\ \rho$ and $d/m(t_1 \ldots t_k)t \in \mathit{vis\_meths}(t_0)$
$\mathit{infer\_type}\ m(e_1, \ldots, e_k)\ \rho = t$	as above, but $t_0 = c$ where c is the current class
$\mathit{infer\_type}\ c.m(e_1, \ldots, e_k)\ \rho = t$	as above, but $t_0 = c$

So far we didn't consider casting, but assumed that the types of the arguments in a method call exactly match those of a defined method. In Java primitive datatypes and objects can be explicitly casted into other datatypes or objects:

```
double pi=3.14;
int    x = (int) pi;

class A { }
class B extends A
 { int x=3;
   int m() { A obj=new B();
             int z=obj.x;        <------ leads to an error
             int z=((B)obj).x;   <------ ok
           }
 }
```

In the above example we applied a narrowing conversion. Java performs implicit casting, when the actual arguments of a method call do not exactly match those of parameters of the defined method. In this case a widening conversion is applied to the actual parameters. Let d be a subclass of c, then

widening: an instance of d can be used where an instance of c is expected. An explicit cast is not necessary.

narrowing: an instance of c can be used as an instance of d if it is explicitly casted.

Method Overloading In TassKaf methods can be overloaded. It is legal for a class to have several methods with the same method name but different argument types and result types. The only restriction is, that there are not two methods with the same name and the same argument types but different result types (In Ada this is legal that two methods only differ in their result types, as a consequence overloading resolution is slightly more difficult (an additional top-down pass in the type inference). In fact overloading resolution in Java is more difficult than that. If there is no method with exactly those argument types infered, but the arguments in the method call can be converted into the types of one or more existing methods, then one of these methods has to be chosen.

In a preliminary version of the "Java 1.0 Language Specification" this choice was based on conversion costs. This was a very problematic approach to overloading resolution, because it is very difficult to handle by the Java programmer. In the final version of the "Java 1.0 Language Specification" this approach was replaced by one based on maximally specific methods.

Although maximally specific methods are easier to identify by the programmer, many Java programmers use casts, i.e. explicit type conversion, to make sure that the right method is called. For TassKaf we decided to use a simpler form of overloading, i.e. we only allowed method calls, for which the infered argument types exactly match those of a defined method, see the above definition of $\mathit{infer\_type}\ o.m(e_1, \ldots, e_k)\ \rho$.

6.4.4 Overloading Resolution in Java

If the parameter types of the defined method do not exactly match those of the arguments in a call, we have to compare these types. Do do this we define the relation $\sqsubseteq$ as follows:

- $t \sqsubseteq t'$ iff t, t' are classes and $t \downarrow^* t'$
- $t \sqsubseteq t'$ iff t, t' are primitive types and entities of type t can be converted into entities of type t' without data loss.
- $m(t_1 \ldots t_k) \sqsubseteq m(t'_1 \ldots t'_k)$ iff $t_i \sqsubseteq t'_i$ $(1 \leq i \leq k)$

Now we define the set of applicable methods for a given class and method:

- $\mathit{applicable}(c/m(t_1 \ldots t_k)) = \{c'/m(t'_1 \ldots t'_k)t' \in \mathit{vis\_meths}(c) \mid m(t_1 \ldots t_k) \sqsubseteq m(t'_1 \ldots t'_k)\}$

Given a set of applicable methods A we can now define the most specific method

- $\mathit{most\_specific}(A) = c/m(t_1 \ldots t_k)t \in A$ such that $\forall c'/m(t'_1 \ldots t'_k)t' \in A : m(t_1 \ldots t_k) \sqsubseteq m(t'_1 \ldots t'_k)$

With these definitions the type of a method call in Java is computed as:

$\mathit{infer\_type}\ o.m(e_1, \ldots, e_k)\ \rho = t'$ where $t_0 = \mathit{infer\_type}\ o\ \rho$, $t_1 = \mathit{infer\_type}\ e_1\ \rho$, …, $e_k = \mathit{infer\_type}\ e_k\ \rho$ and $d/m(t'_1 \ldots t'_k)t' = \mathit{most\_specific}(\mathit{applicable}(t_0, m(t_1 \ldots t_k)))$

If $applicable(t_0, m(t_1 \ldots t_k)))$ is not empty but there exists no most specific method, we have an ambiguous method call. The following example shows such a case:

```
class A {}
class B extends A {}
class C extends B
 { void m(A x, C y) { }
   void m(B x, A y)
       { m(new B(), new C());
       }
 }
```

For the arguments of the method call we infer types B and C and thus get $applicable(C, m(B, C)) = \{m(A, C), m(B, A)\}$ because $m(B, C) \sqsubseteq m(A, C)$ and $m(B, C) \sqsubseteq m(B, A)$. Unfortunately neither is $m(A, C) \not\sqsubseteq m(B, A)$ nor $m(B, A) \not\sqsubseteq m(A, C)$ and thus the call is ambiguous.