# 10.4 Scope rules

Objects may have attributes in the form of data-items, methods and/or classes. An attribute of an object has a name and the *scope* of such a name is the part of the program where the attribute may be *directly* accessed using the name. The name cannot be referred outside scope.

We have seen several examples of attributes being directly accessed in the code. For the `withdraw` method in class `Account`, it accesses the parameter `amount` and data-item `balance` in the enclosing `Account`-object.

For the flight-example in section 6.2, we have seen that the method `delay` accesses the data-item `actualArrivalTime` in the enclosing `Flight`-class and `scheduledArrivalTime` in the `Route`-class enclosing the `Flight`-class.

The *direct scope* of a name of an attribute declaration is:

- the main-part of the object-descriptor (OD) containing the declaration
- all object-descriptors nested within OD
- all object-descriptors that directly or indirectly have OD as a super-descriptor.

The following figure illustrates this definition:

```
anObj: obj
   x: var integer
   class C:
      y: var integer
      M1:
         z: var integer
         -- x, y, and z may be accessed
         -- anObj, C, D, and M1 may be accessed
         -- a, b, and M2 cannot be accessed
   class D:
      a: var integer
      M2:
         b var: integer
         -- x, x, and b may be accessed
         -- anObj, C, D, and M2 may be accessed
         -- y, z, and M1 cannot be accessed
```

***Double declaration.*** There are a number of additional rules to be considered. One of these is that *double declaration* of the same name is forbidden within an object-descriptor:

```
anObj: obj
   x: var integer
   ...
   x: ref Person
```

The above code fragment is illegal since it contains two declarations of `x`. This is independent of the

kind of attribute being declared. The first x is the name of an integer variable, the second x is a reference to a Person-object, but still it is illegal.

***Redeclaration in a subclass***. A subclass may not declare am attribute that has the same name as an attribute in a possible superclass:

```
class Super:
   ...
   x: var integer
   ...
class Sub: Super
   ...
   x: ref T
   ...
```

The declaration `x: ref T` in class Sub is illegal since the superclass Super has an x.

***Name shadowing.*** An object-descriptor may declare an attribute, which has the same name as an attribute declared in an enclosing object-descriptor:

```
anObj: obj
   x: var integer
   class C:
      y: var integer
      M1:
         x: var integer
         -- x here is the x declared in M1, not the one in anObj
```

The declaration of x in M1 shadows for the x declared in the enclosing object-descriptor of anObj. Name shadowing is often considered problematic in the sense that this can lead to confusion, as it may be unclear which name subsequent uses of the shadowed name refer to, especially in large programs. Name shadowing shall thus be used with care.

***Local and global names.*** A name declared in an object-descriptor is a *local name* in the object-descriptor. Names declared in enclosing object-descriptors are called *global names*.

***Formal parameters.*** The name of a formal parameter of a class or method is considered a local name of the associated object-descriptor.

***Static/lexical scoping.*** The kind of scoping described here is called *static* scoping or *lexical* scoping. This refers to the fact that the scope of a name is a property of the program text since it is possible to figure out the scope of a name by looking at the program text only.

***Dynamic scoping.*** There is also a form of scoping called *dynamic* scoping where the scoping of a name depends on the execution. It dates back to the first version of the Lisp-language, but few or no modern languages have dynamic scoping.

# Remote scope

So far we have described the direct scope of a name, which is how a name may be accessed directly using just the name. If x is a name, then the direct scope of x is the object-descriptors where x may be accessed by just writing x.

A name `x` may also be accessed by a *remote name* such as `r.x`. Any object-descriptor that potentially may have a reference to an object described by the object-descriptor where `x` is declared is in the *remote scope* of `x`.

An example of this is attribute accessors like `account_1010.deposit(100)` and `account_1022.withdraw(200)`.

Here is another example of using attribute accessors :

```
class T:
   x: var integer
   ...
r: obj T
s: ref T
s := r         -- now s and r refer to the same object
r.x := 117     -- assign 117 to the x attribute of r
s.x := s.x + 1 -- increments the x attribute of s
               -- which is the same as the x attribute of r
```

`r.x` and `s.x` are both remote names and in this case they refer to the same `x` because `r` and `s` refer to the same `T` object.

# Virtual methods and classes and dynamic dispatch

A virtual method or class may be further specified in subclasses/sub-descriptors. It is important to point out that such further specifications are not redeclaration/redefinition of the virtual attribute.

The actual virtual method invoked or actual virtual class instantiated by a virtual name is, however, first determined at run-time during program execution. Consider the example:

```
class Super:
   V:<
     ...
   class T:<
      ...
class SubA: Super
   V::<
      ...
   class T::<
      ...
class SubB: Super
   V::<
      ...
   class T::<
      ...
R: ref Super  -- the static type of R is Super
...
R := SubA     -- the dynamic type of R is now SubA
...
R := SubB     -- the dynamic type of R is now SubB
...
R.V
```

Here we have a class `Super` with two virtual attributes `V` and `T.` We have two subclasses of `Super`, `SuperA` and `superB` which both contains bindings of `V` and `T`.

We have a reference variable `R` that may refer to instances of `Super` and to instances of subclasses of `Super`. As the example indicates `R` may at different places in the program refer to instances of `SubA` or instances of `SubB`.

A remote invocation `R.V` instantiates the `V` being further specified in `SubA` or `SubB` depending on whether `R` refers to a `SubA`– or `SubB`-object. This is sometimes called *dynamic dispatch*.

The type `Super` of `R` from the declaration of `R` is sometimes called the *static type* of `R` since it can be read from the program text. Since `R` may refer to instance of subtypes of `Super`, the type of the object being referred to at given point during the program execution is sometimes called the *dynamic type* of `R`. In the above example, the dynamic type may be either `SubA` or `SubB` as indicated in the above code sketch.

## Restricting the scope of names

Sometime it may be desirable to restrict the scope (direct as well as remote) of a name. In chapter , we introduce access modifiers as a mechanism for doing this.