Object-oriented programming

Arjen Markus

Deltares

June 6, 2018

Objects and classes

Fortran 2003 follows (more or less) the C++ model for OOP:

- Modules and derived types are used to define a class
- Variables of these derived types are the objects
- But there are differences! See below

Syntax: Extensible types

In principle a derived type can be extended:

```
type mytype
    integer :: value
end type mytype

type, extends(mytype) :: mynewtype
    real :: extra
end type mynewtype
```

The new type inherits the old components

Syntax: Extensible types (2)

The parent component:

```
type(mynewtype) :: v

write(*,*) Value: , v%value
write(*,*) Parent: , v%mytype%value
```

In this case they refer to the same component
The new type inherits the old components

Syntax: Extensible types (3)

Methods:

```
module mytypes
    type mytype
        integer :: value
    contains
        procedure :: write => write_mytype !<= Alias for actual routine
    end type mytype

contains
subroutine write_mytype( v, lun )
    class(mytype) :: v                  ! <= Note: "class", not "type",
                                        !    this allows extension
    integer :: lun
end subroutine
end module
```

Syntax: Extensible types (4)

Usage:

```
use mytypes
type(mytype) :: v

call v%write( 10 )          ! <= The first argument is implicit!
```

Syntax: Extensible types (5)

Overriding in extending type:

```
module mytypes
    type, extends(mytype) :: mynewtype
        real :: extra
    contains
        procedure :: write => write_mynewtype
    end type mynewtype

contains
subroutine write_mynewtype( v, lun )
    class(mynewtype) :: v
    integer :: lun
    call v%mytype%write( lun )    ! <= Invoke the parents routine
    write( lun, * ) Extra: , v%extra
end subroutine
end module
```

More syntax: pass and nopass

Control how the object is passed:

```
module mytypes
    type, extends(mytype) :: mynewtype
        real :: extra
    contains
        procedure, pass(value) :: write => write_mynewtype
    end type mytype

contains
subroutine write_mynewtype( lun, value )
    class(mynewtype) :: value
    integer :: lun
    call value%mytype%write( lun )
    write( lun, * ) 'Extra: ', value%extra
end subroutine
end module
```

More syntax: pass and nopass (2)

Invoking it:

```
use mytypes
type(mynewtype) :: v
call v%write(lun)     ! <= Not different than before!

call write_mynewtype( lun, v ) ! Alternative
```

More syntax: pass and nopass (2)

Do not pass it implicitly:

```
procedure, nopass :: write => write_mynewtype

type(mynewtype) :: v
call v%write( lun, v )    ! <= Now we need to pass it
                          !    ourselves
```

Useful if it works on the class, for instance

Types and classes

Dummy arguments:

```
subroutine calculate( data )
   type(mydata) :: data    ! <= Exactly that type

   ! Or:
   class(mydata) :: data   ! <= This or an extension
```

A class is a "polymorphic argument"

Types and classes (2)

Difference: declared type and dynamic type
Use the "select type" construct:

```
select type (data)
   type is (mydata) :
       ! Variable data must be exactly of type mydata
   type is (mynewdata) :
       ! Extended from mydata, access to new components
   class is (mydata) :
       ! Dynamic type is mydata or an extension
end select
```

Or: `extends_type_of(a, mold)` and `same_type_as(a, b)`

Unlimited polymorphic types

Declare as follows:

```
class(*), pointer     :: p_any
class(*), allocatable :: p_value

select type (p_any)
   type is (integer) :
       ! Variable p_any points to an integer
   type is (mynewdata) :
       ! Variable p_any points to a derived type mydata
   class is (mydata) :
       ! Points to a dynamic type mydata or an
       ! extension
end select
```

Unlimited polymorphic types (2)

You can not use unlimited polymorphic types directly:

- Pass them to a routine
- Use `select type` to "transform" them to a specific type
- They are either pointers or allocatables (also for scalars!)

Abstract classes and deferred procedures

Types can be made "abstract" - they serve as a template
One or more procedures can be "deferred" - define them in an extended type

```
type, abstract :: mytemplate
    ...
contains
    procedure(calculate_template), deferred :: calc
end type mytemplate
```

Abstract classes and deferred procedures (2)

Extending this abstract type:

```
type, extends(mytemplate) :: mydata
    ...
contains
    procedure :: calc => calc_mydata  ! <= Concrete procedure
end type mydata
```

Abstract classes and deferred procedures (3)

Abstract interfaces:

```
abstract interface
    subroutine calculate_template( t, x, y )
        import :: mytemplate
        class(mytemplate) :: t
        real :: x, y
    end subroutine calculate_template
end interface
```

Abstract classes and deferred procedures (4)

- Abstract interfaces are applicable to ordinary types too.
- And for procedure pointers see below

Procedure pointers

New class of pointers:

```
procedure(calculate_template), pointer :: p

p => my_calculation

call p( t, x, y )
call my_calculation( t, x, y )
```

Note: *no syntactical difference!*
The signature must match

Procedure pointers (2)

Each object can have its own routine:

```
type :: mydata
    real :: x
contains
    procedure(calculate), pointer :: calc
end type

type(mydata), dimension(10) :: data

data%calc => calculation_1
data(2)%calc => calculation_2   ! <= slightly different,
                                !    same interface

do i = 1,10
    call data(i)%calc( y ) ! Object data(2) does it differently
enddo
```

Finalizers

- You may need to do something special if the object ceases to exist.
- For instance: close a file, release memory that was accessed via a pointer.
- This happens if the object is local to a routine or is explicitly deallocated.
- That is what finalizers are for.

```
module mytypes
    type :: mydata
        real, dimension(:), pointer :: px
    contains
        final :: mydata_release_px
    end type
contains
subroutine mydata_release_px( data )
    type(mydata) :: data

    deallocate( data%px )
end subroutine my_data_release_px
```

Generic procedures

Generic type-bound procedures work in a similar way as ordinary generic procedures. But they require a different syntax.

```
module mytypes
    type :: mycollection
        integer :: ivalue
        real    :: rvalue
    contains
        procedure :: get_r :: get_real
        procedure :: get_i :: get_int

        generic   :: get => get_real, get_int
    end type
contains
subroutine get_real( c, x )  ! Similar for integer
    type(mycollection) :: c
    real               :: x

end subroutine get_real
end module mytypes
```

Differences with C++

Some differences with c++:

- F2003 does not define constructor routines.
- It does define finalizer routines
- F2003 does not allow multiple inheritance
  - You can extend the interface
  - Or use "aggregation" to handle a lot of the things you would want to do
- In F2003 no difference between calling routines via pointers or via fixed names! This is therefore transparent in the users code.