Object-oriented programming – introduction

Arjen Markus

Deltares

July 4, 2020

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!

Syntax: Extensible types

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 (2)

Usage:

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

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

Syntax: Extensible types (3)

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 and gets new components

Syntax: Extensible types (4)

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, including the original methods.

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 parent's routine
    write( lun, * ) 'Extra: ', v%extra
end subroutine
end module
```

Simple example – no extension

Quasi-random numbers:

```
type quasirandom_generator
    integer :: dimin    = -1
    integer :: step     = 1
    integer :: stepsize = 1
    real(kind=dp), dimension(:), allocatable :: factor
contains
    procedure :: init => init_quasi
    procedure :: restart => restart_quasi
    procedure :: single_next => single_next_quasi
    procedure :: double_next => double_next_quasi
    generic   :: next => single_next, double_next
end type quasirandom_generator
```

Simple example – explanation

Quasi-random numbers select points in an n-dimensional space:

```
type(quasirandom_generator)           :: q
real(kind=kind(1.0d0)), dimension(3) :: coords
real(kind=kind(1.0d0))                :: sum
integer                               :: i

call q%init( size(coords) )  ! Points in three-dimensional space
!
! "Monte-Carlo" evaluation of an integral:
! f(x,y,z) = x**2 + y**2 + z**2
!
sum = 0.0d0
do i = 1,100
    call q%next( coords )

    sum = sum + sum(coords**2)
enddo
write(*,*) 'Approximate integral: ', sum / 100
```

Simple example – initial values of components

In the type definition default values are used:

```
type quasirandom_generator
    integer :: dimin    = -1
    integer :: step     = 1
    integer :: stepsize = 1
    real(kind=dp), dimension(:), allocatable :: factor
contains
    ...
end type quasirandom_generator
```

You need not do that, but you could check in the implementation that the object has been properly filled, 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": you can also use *extended* types.

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 cannot 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!)