Modules and Interfaces

Carlos Cruz
Jules Kouatchou
Bruce Van Aartsen

NASA GSFC Code 606 (ASTG)
Greenbelt Maryland 20771

October24, 2018

Modules

A MODULE is a program unit whose internal data and subroutines can be easily accessed by other program units via the USE statement.

A module can contain:

- Procedure declarations – Several related procedures can be encapsulated into a module, and made visible to any program through the USE statement
- Global object declarations – Useful to cut down argument passing between routines. Data objects can be used by attaching the module – values retained between uses.
- Interface declarations – Can be packaged into a module, and then made accessible by USE-ing the module
- Controlled object accessibility – Variables, procedures and operator declarations can have their visibility controlled by access statements

Modules – General Form

A MODULE uses the following syntax:

```
MODULE ModuleName
    declarations
    global data
    ...
    CONTAINS
        module procedure definitions
        ...
END MODULE ModuleName
```

Where ***declarations*** may include:

- USE statements to inherit other modules
- TYPE definitions
- Object definitions
- PRIVATE/PUBLIC accessibility statements
- INTERFACE declarations

Modules – Simple Example

```
MODULE StationObservations
   use CalendarMod
   implicit none

   private
   real, allocatable  :: precip(:)
   real, allocatable  :: temperature(:)
   integer, parameter :: secPerDay = 86400
   public  readObs
   public  calcAvgPrecip
   ...
   data stationLocation / 37.2709, -79.9414 /


   CONTAINS
      subroutine readObs(station, startDate, endDate)
      ...
      end subroutine readObs

      function calcAvgPrecip(station, startDate, endDate)
      ...
      end function calcAvgPrecip
END MODULE StationObservations
```

Interfaces

An INTERFACE block can be used for a few different purposes.

1. It can allow external procedures to be declared, making them “visible” to the program
2. A named interface can enable a set of similar module procedures to be referenced via a single generic name, (aka an overloaded procedure) using polymorphic typing.
3. It can extend the meaning of an intrinsic operator to apply to additional data types (aka Operator Overloading).
4. It is sometimes used to organize the interfaces to all the procedures in a large program, becoming a handy reference for coding

Interface

This INTERFACE module can be used to access external procedures:

```
MODULE MyInterfaces
   implicit none
   INTERFACE
      subroutine mySub1(A, B)
         real, intent(in) :: A
         integer, intent(in) :: B
      end subroutine mySub1

      subroutine mySub2(C, D, E)
      ...
      end subroutine mySub2
   END INTERFACE
END MODULE MyInterfaces
```

```
PROGRAM MyProgram
   use MyInterfaces
   implicit none
   call mySub1(273.15, 12)
END PROGRAM MyProgram


!External procedure
subroutine mySub1(A, B)
   implicit none
   ...
   print*, A, B
end subroutine mySub1
```

Generic Interface

A Generic INTERFACE declaration allows procedures which perform the same function to be called via the same generic name. The specific procedure invoked depends on the number and/or type of arguments.

For example:

```
INTERFACE mySub
   subroutine mySub1(A)          !use: CALL mySub(int)
      integer :: A
   end subroutine mySub1

   subroutine mySub2(A)          !use: CALL mySub(real)
      real :: A
   end subroutine mySub2

   subroutine mySub3(A, B)       !use: CALL mySub(real,int)
      real :: A
      integer :: B
   end subroutine mySub3
END INTERFACE
```

Operator Interface

The INTERFACE OPERATOR declaration can extend the capabilities of intrinsic operators. For instance, the "+" character could be extended for character variables in order to concatenate two strings:

```
MODULE OperatorOverloading
   implicit none
   ...
   INTERFACE OPERATOR (+)
      MODULE PROCEDURE concat
   END INTERFACE
   ...
CONTAINS
   function concat(cha,chb)
      implicit none
      character (LEN=*), INTENT(IN) :: cha, chb
      character (LEN = (LEN_TRIM(cha) + LEN_TRIM(chb))) :: concat
      concat = TRIM(cha)//TRIM(chb)
   end function concat
   ...
END MODULE OperatorOverloading
```

Example

```
module CircleMod
   implicit none

   private
   public setRadius, areaCircle
   real, parameter :: PI = 3.1415927
   real :: radius

CONTAINS
   subroutine setRadius(r)
      implicit none
      real, intent(in) :: r
      radius = r
   end function

   real function areaCircle()
      implicit none
      areaCircle = PI * radius**2
   end function areaCircle
end module CircleMod
```

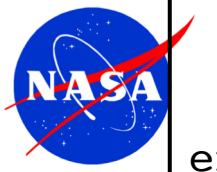
```
program CircleOperations
   use CircleMod
   implicit none
   real :: rad

   print *, “Enter radius of circle:”
   read *, rad

   call setRadius(rad)

   print *, "Area of circle is", areaCircle()

end program CircleOperations
```

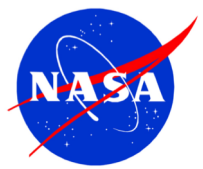
- Variable “radius” is hidden from the main program
- Must use module procedures to access it
- ***Information hiding*** is an important feature of Object Oriented Programming (discussed tomorrow)

Exercise

```
program GenericSwap
   implicit none
   integer         :: i, j
   character       :: c, d

   ! Add the proper generic interface
   ! block below to enable this code
   interface swap



   end interface

   i = 1
   j = 2
   c = 'a'
   d = 'b'

   print *, i, j, c, d
   call swap(i, j)
   call swap(c, d)
   print *, i, j, c, d

end program
```

```
subroutine swap_i(a, b)
  implicit none
  integer, intent (inout) :: a, b
  integer                 :: temp
  temp = a
  a = b
  b = temp
end subroutine swap_i


subroutine swap_c(a, b)
  implicit none
  character, intent (inout) :: a, b
  character                 :: temp
  temp = a
  a = b
  b = temp
end subroutine swap_c
```




NASA


SSAI