Fortran 90 Subprograms

If Fortran is the lingua franca, then certainly it must be true that BASIC is the lingua playpen

Thomas E. Kurtz
Co-Designer of the BASIC language

Functions and Subroutines

- **Fortran 90 has two types of subprograms, functions and subroutines.**
- **A Fortran 90 function is a function like those in C/C++. Thus, a *function* returns a computed result via the function name.**
- **If a function does not have to return a function value, use *subroutine*.**

Function Syntax: 1/3

- A Fortran function, or function subprogram, has the following syntax:

```
type FUNCTION function-name (arg1, arg2, ..., argn)
   IMPLICIT NONE
   [specification part]
   [execution part]
   [subprogram part]
END FUNCTION function-name
```

- `type` is a Fortran 90 type (*e.g.*, `INTEGER`, `REAL`, `LOGICAL`, etc) with or without `KIND`.
- `function-name` is a Fortran 90 identifier
- `arg1`, …, `argn` are *formal arguments*.

Function Syntax: 2/3

- **A function is a self-contained unit that receives some "input" from the outside world via its *formal arguments*, does some computations, and returns the result with the name of the function.**
- **Somewhere in a function there has to be one or more assignment statements like this:**

  `function-name` = ***expression***

  where the result of *expression* is saved to the name of the function.
- **Note that `function-name` cannot appear in the right-hand side of any expression.**

Function Syntax: 3/3

- **In a type specification, formal arguments should have a new attribute `INTENT(IN)`.**
- **The meaning of `INTENT(IN)` is that the function only takes the value from a formal argument and does not change its content.**
- **Any statements that can be used in `PROGRAM` can also be used in a `FUNCTION`.**

Function Example

- Note that functions can have no formal argument.
- But, **()** is still required.

Factorial computation

```
INTEGER FUNCTION Factorial(n)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: n
   INTEGER :: i, Ans

   Ans = 1
   DO i = 1, n
      Ans = Ans * i
   END DO
   Factorial = Ans
END FUNCTION Factorial
```

Read and return a positive real number

```
REAL FUNCTION GetNumber()
   IMPLICIT NONE
   REAL :: Input_Value
   DO
      WRITE(*,*) 'A positive number: '
      READ(*,*) Input_Value
      IF (Input_Value > 0.0) EXIT
      WRITE(*,*) 'ERROR. try again.'
   END DO
   GetNumber = Input_Value
END FUNCTION GetNumber
```

Common Problems: 1/2

forget function type

```
FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   DoSomthing = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

forget `INTENT(IN)` – not an error

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER :: a, b
   DoSomthing = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

change `INTENT(IN)` argument

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
      IF (a > b) THEN
        a = a - b
      ELSE
        a = a + b
   END IF
   DoSomthing = SQRT(a*a+b*b)
END FUNCTION DoSomething
```

forget to return a value

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   INTEGER :: c
   c = SQRT(a*a + b*b)
END FUNCTION DoSomething
```

Common Problems: 2/2

incorrect use of function name

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   DoSomething = a*a + b*b
   DoSomething = SQRT(DoSomething)
END FUNCTION DoSomething
```

only the most recent value is returned

```
REAL FUNCTION DoSomething(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: a, b
   DoSomething = a*a + b*b
   DoSomething = SQRT(a*a - b*b)
END FUNCTION DoSomething
```

Using Functions

- **The use of a user-defined function is similar to the use of a Fortran 90 intrinsic function.**
- **The following uses function `Factorial(n)` to compute the combinatorial coefficient *C(m,n)* , where `m` and `n` are *actual arguments*:**

```
Cmn = Factorial(m)/(Factorial(n)*Factorial(m-n))
```

- **Note that the combinatorial coefficient is defined as follows, although it is *not* the most efficient way:**

$$C(m,n) = \frac{m!}{n! \times (m-n)!}$$

Argument Association : 1/5

- ***Argument association*** **is a way of passing values from actual arguments to formal arguments.**
- **If an actual argument is an *expression*, it is evaluated and *stored in a temporary location* from which the value is passed to the corresponding formal argument.**
- **If an actual argument is a *variable*, its value is passed to the corresponding formal argument.**
- **Constant and `(A)`, where `A` is variable, are considered expressions.**

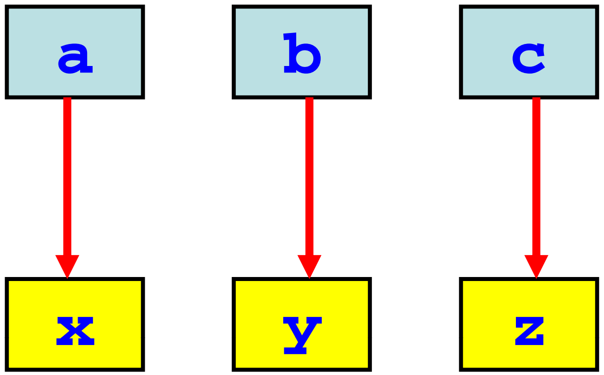
Argument Association : 2/5

- **Actual arguments are variables:**

```
WRITE(*,*) Sum(a,b,c)



INTEGER FUNCTION Sum(x,y,z)
   IMPLICIT NONE
   INTEGER,INTENT(IN)::x,y,z
      ……..
END FUNCTION  Sum
```



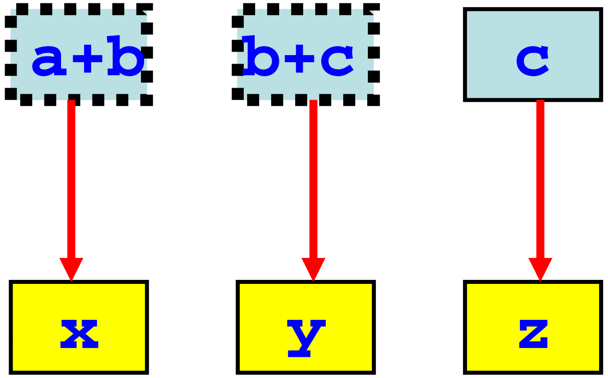
Argument Association : 3/5

- **Expressions as actual arguments. Dashed line boxes are temporary locations.**

```
WRITE(*,*) Sum(a+b,b+c,c)



INTEGER FUNCTION Sum(x,y,z)
   IMPLICIT NONE
   INTEGER,INTENT(IN)::x,y,z
      ……..
END FUNCTION  Sum
```



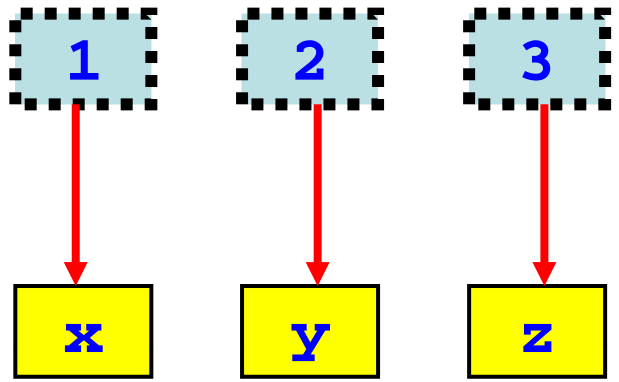
Argument Association : 4/5

- **Constants as actual arguments. Dashed line boxes are temporary locations.**

```
WRITE(*,*) Sum(1, 2, 3)



INTEGER FUNCTION Sum(x,y,z)
   IMPLICIT NONE
   INTEGER,INTENT(IN)::x,y,z
      ……..
END FUNCTION  Sum
```



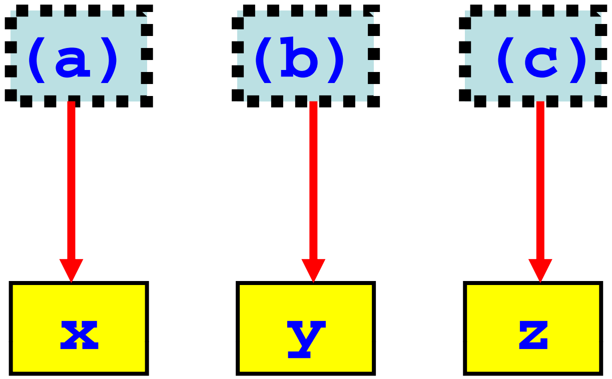
Argument Association : 5/5

- **A variable in `( )` is considered as an expression. Dashed line boxes are temporary locations.**

```
WRITE(*,*) Sum((a), (b), (c))



INTEGER FUNCTION Sum(x,y,z)
   IMPLICIT NONE
   INTEGER,INTENT(IN)::x,y,z
      ……..
END FUNCTION  Sum
```



Where Do Functions Go: 1/2

- **Fortran 90 functions can be internal or external.**
- ***Internal* functions are inside of a `PROGRAM`, the *main program*:**

```
PROGRAM program-name
   IMPLICIT NONE
   [specification part]
   [execution part]
CONTAINS
   [functions]
END PROGRAM program-name
```

- **Although a function can contain other functions, internal functions *cannot* have internal functions.**

Where Do Functions Go: 2/2

- The right shows two internal functions, **`ArithMean()`** and **`GeoMean()`**.
- They take two **`REAL`** actual arguments and compute and return a **`REAL`** function value.

```
PROGRAM TwoFunctions
   IMPLICIT NONE
   REAL :: a, b, A_Mean, G_Mean
   READ(*,*) a, b
   A_Mean = ArithMean(a, b)
   G_Mean = GeoMean(a,b)
   WRITE(*,*) a, b, A_Mean, G_Mean
CONTAINS
   REAL FUNCTION ArithMean(a, b)
      IMPLICIT NONE
      REAL, INTENT(IN) :: a, b
      ArithMean = (a+b)/2.0
   END FUNCTION ArithMean
   REAL FUNCTION GeoMean(a, b)
      IMPLICIT NONE
      REAL, INTENT(IN) :: a, b
      GeoMean = SQRT(a*b)
   END FUNCTION GeoMean
END PROGRAM TwoFunctions
```

Scope Rules: 1/5

- ***Scope rules*** **tell us if an entity (*i.e.*, variable, parameter and function) is *visible* or *accessible* at certain places.**
- **Places where an entity can be accessed or visible is referred as the *scope* of that entity.**

Scope Rules: 2/5

- ***Scope Rule #1:*** **The scope of an entity is the program or function in which it is declared.**

```
PROGRAM Scope_1
   IMPLICIT NONE
   REAL, PARAMETER :: PI = 3.1415926
   INTEGER :: m, n
      ...................
   CONTAINS
      INTEGER FUNCTION Funct1(k)
         IMPLICIT NONE
         INTEGER, INTENT(IN) :: k
         REAL :: f, g
            ..........
      END FUNCTION Funct1
      REAL FUNCTION Funct2(u, v)
         IMPLICIT NONE
         REAL, INTENT(IN) :: u, v
            ..........
         END FUNCTION Funct2
END PROGRAM Scope_1
```

Scope of `PI`, `m` and `n`

Scope of `k`, `f` and `g` local to `Funct1()`

Scope of `u` and `v` local to Funct2()

Scope Rules: 3/5

- ***Scope Rule #2*** **:A global entity is *visible* to all contained functions.**

```
PROGRAM Scope_2
   IMPLICIT NONE
   INTEGER :: a = 1, b = 2, c = 3
   WRITE(*,*) Add(a)
   c = 4
   WRITE(*,*) Add(a)
   WRITE(*,*) Mul(b,c)
CONTAINS
   INTEGER FUNCTION Add(q)
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: q
      Add = q + c
   END FUNCTION Add
   INTEGER FUNCTION Mul(x, y)
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: x, y
      Mul = x * y
   END FUNCTION Mul
END PROGRAM Scope_2
```

- **`a`, `b` and `c` are global**
- **The first `Add(a)` returns 4**
- **The second `Add(a)` returns 5**
- **`Mul(b,c)` returns 8**

Thus, the two `Add(a)`'s produce different results, even though the formal arguments are the same! This is usually referred to as *side effect*.

Avoid using global entities!

Scope Rules: 4/5

- ***Scope Rule #2*** **:A global entity is *visible* to all contained functions.**

```
PROGRAM Global
  IMPLICIT NONE
  INTEGER ::  a = 10, b = 20
  WRITE(*,*)  Add(a,b)
  WRITE(*,*)  b
  WRITE(*,*)  Add(a,b)
CONTAINS
  INTEGER FUNCTION Add(x,y)
    IMPLICIT NONE
    INTEGER, INTENT(IN)::x, y
    b    = x+y
    Add = b
  END FUNCTION Add
END PROGRAM Global
```

- **The first `Add(a,b)` returns `30`**
- **It also changes `b` to `30`**
- **The 2nd `WRITE(*,*)` shows `30`**
- **The 2nd `Add(a,b)` returns `40`**
- **This is a bad side effect**
- **Avoid using global entities!**

Scope Rules: 5/5

- ***Scope Rule #3*** **:An entity declared in the scope of another entity is always a different one even if their names are identical.**

```
PROGRAM Scope_3
  IMPLICIT NONE
  INTEGER :: i, Max = 5
  DO i = 1, Max
    Write(*,*) Sum(i)
  END DO
CONTAINS
  INTEGER FUNCTION Sum(n)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n
    INTEGER :: i, s
    s = 0
    …… other computation ……
    Sum = s
  END FUNCTION Sum
END PROGRAM Scope_3
```

Although `PROGRAM` and `FUNCTION` `Sum()` both have `INTEGER` variable `i`, They are *TWO* different entities.

Hence, any changes to `i` in `Sum()` will not affect the `i` in `PROGRAM`.

Example: 1/4

- **If a triangle has side lengths *a*, *b* and *c*, the Heron formula computes the triangle area as follows, where *s* = (*a*+*b*+*c*)/2:**

$$Area = \sqrt{s \times (s-a) \times (s-b) \times (s-c)}$$

- **To form a triangle, *a*, *b* and *c* must fulfill the following two conditions:**
  - ***a* > 0, *b* > 0 and *c* > 0**
  - ***a*+*b* > *c*, *a*+*c* > *b* and *b*+*c* > *a***

Example: 2/4

- **LOGICAL Function TriangleTest() makes sure all sides are positive, and the sum of any two is larger than the third.**

```
LOGICAL FUNCTION TriangleTest(a, b, c)
   IMPLICIT NONE
   REAL, INTENT(IN) :: a, b, c
   LOGICAL          :: test1, test2
   test1 = (a > 0.0) .AND. (b > 0.0) .AND. (c > 0.0)
   test2 = (a + b > c) .AND. (a + c > b) .AND. (b + c > a)
   TriangleTest = test1 .AND. test2    ! both must be .TRUE.
END FUNCTION TriangleTest
```

Example: 3/4

- **This function implements the Heron formula.**
- **Note that *a*, *b* and *c* must form a triangle.**

```
REAL FUNCTION Area(a, b, c)
   IMPLICIT NONE
   REAL, INTENT(IN) :: a, b, c
   REAL             :: s
   s = (a + b + c) / 2.0
   Area = SQRT(s*(s-a)*(s-b)*(s-c))
END FUNCTION Area
```

Example: 4/4

- **Here is the main program!**

```
PROGRAM HeronFormula
   IMPLICIT NONE
   REAL :: a, b, c, TriangleArea
   DO
      WRITE(*,*) 'Three sides of a triangle please --> '
      READ(*,*) a, b, c
      WRITE(*,*) 'Input sides are ', a, b, c
      IF (TriangleTest(a, b, c)) EXIT ! exit if they form a triangle
      WRITE(*,*) 'Your input CANNOT form a triangle. Try again'
   END DO
   TriangleArea = Area(a, b, c)
   WRITE(*,*) 'Triangle area is ', TriangleArea
CONTAINS
   LOGICAL FUNCTION TriangleTest(a, b, c)
      ……
   END FUNCTION TriangleTest
   REAL FUNCTION Area(a, b, c)
      ……
   END FUNCTION Area
END PROGRAM HeronFormula
```

Subroutines: 1/2

- **A Fortran 90 function takes values from its formal arguments, and returns a *single value* with the function name.**
- **A Fortran 90 subroutine takes values from its formal arguments, and *returns some computed results with its formal arguments*.**
- **A Fortran 90 subroutine does not return any value with its name.**

Subroutines: 2/2

- The following is Fortran 90 subroutine syntax:

```
SUBROUTINE subroutine-name(arg1,arg2,...,argn)
   IMPLICIT NONE
   [specification part]
   [execution part]
   [subprogram part]
END SUBROUTINE subroutine-name
```

- If a subroutine does not require any formal arguments, "`arg1,arg2,...,argn`" can be removed; however, `()` must be there.
- Subroutines are similar to functions.

The **`INTENT()`** Attribute: 1/2

- **Since subroutines use formal arguments to receive values and to pass results back, in addition to `INTENT(IN)`, there are `INTENT(OUT)` and `INTENT(INOUT)`.**
- **`INTENT(OUT)` means a formal argument does not receive a value; but, it will return a value to its corresponding actual argument.**
- **`INTENT(INOUT)` means a formal argument receives a value from and returns a value to its corresponding actual argument.**

The INTENT() Attribute: 2/2

- Two simple examples:

Am, Gm and Hm are used to return the results

```
SUBROUTINE Means(a, b, c, Am, Gm, Hm)
   IMPLICIT NONE
   REAL, INTENT(IN)  :: a, b, c
   REAL, INTENT(OUT) :: Am, Gm, Hm
   Am = (a+b+c)/3.0
   Gm = (a*b*c)**(1.0/3.0)
   Hm = 3.0/(1.0/a + 1.0/b + 1.0/c)
END SUBROUTINE Means
```

values of a and b are swapped

```
SUBROUTINE Swap(a, b)
   IMPLICIT NONE
   INTEGER, INTENT(INOUT) :: a, b
   INTEGER :: c
   c = a
   a = b
   b = c
END SUBROUTINE Swap
```

The CALL Statement: 1/2

- **Unlike C/C++ and Java, to use a Fortran 90 subroutine, the `CALL` statement is needed.**
- **The `CALL` statement may have one of the three forms:**
  - `CALL sub-name(arg1,arg2,…,argn)`
  - `CALL sub-name( )`
  - `CALL sub-name`
- **The last two forms are equivalent and are for calling a subroutine without formal arguments.**

The CALL Statement: 2/2

```
PROGRAM Test
   IMPLICIT NONE
   REAL :: a, b
   READ(*,*) a, b
   CALL Swap(a,b)
   WRITE(*,*) a, b
CONTAINS
   SUBROUTINE Swap(x,y)
      IMPLICIT NONE
      REAL, INTENT(INOUT) :: x,y
      REAL :: z
      z = x
      x = y
      y = z
   END SUBROUTINE  Swap
END PROGRAM Test
```

```
PROGRAM SecondDegree
   IMPLICIT NONE
   REAL :: a, b, c, r1, r2
   LOGICAL :: OK
   READ(*,*) a, b, c
   CALL Solver(a,b,c,r1,r2,OK)
   IF (.NOT. OK) THEN
      WRITE(*,*) "No root"
   ELSE
      WRITE(*,*) a, b, c, r1, r2
   END IF
CONTAINS
   SUBROUTINE Solver(a,b,c,x,y,L)
      IMPLICIT NONE
      REAL, INTENT(IN) :: a,b,c
      REAL, INTENT(OUT) :: x, y
      LOGICAL, INTENT(OUT) :: L
      ………
   END SUBROUTINE Solver
END PROGRAM SecondDegree
```

More Argument Association: 1/2

- **Since a formal argument with the `INTENT(OUT)` or `INTENT(INOUT)` attribute will pass a value back to the corresponding actual argument, the *actual argument must be a variable*.**

```
PROGRAM Errors                        SUBROUTINE Sub(u,v,w,p,q)
  IMPLICIT NONE                         IMPLICIT NONE
  INTEGER :: a, b, c                    INTEGER, INTENT(OUT) :: u
    ..........                          INTEGER, INTENT(INOUT) :: v
  CALL Sub(1,a,b+c,(c),1+a)             INTEGER, INTENT(IN) :: w
    ..........                          INTEGER, INTENT(OUT) :: p
END PROGRAM Errors                      INTEGER, INTENT(IN) :: q
                                          ..........
                                      END SUBROUTINE Sub
```

these two are incorrect!

More Argument Association: 2/2

- **The number of arguments and their types must match properly.**
- **There is no type-conversion between arguments!**

```
PROGRAM Error
  IMPLICIT NONE
  INTEGER :: a, b
  CALL ABC(a, b)
  CALL ABC(a)
CONTAINS
  .........
END PROGRAM Error
```

```
SUBROUTINE  ABC(p, q)
   IMPLICIT NONE
   INTEGER, INTENT(IN)  :: p
   REAL, INTENT(OUT)    :: q
      .........
END SUBROUTINE ABC
```

wrong # of arguments

type mismatch

Fortran 90 Modules: 1/4

- **One may collect all relevant functions and subroutines together into a module.**
- **A module, in OO's language, is perhaps close to a static class that has public/private information and methods.**
- **So, in some sense, Fortran 90's module provides a sort of object-based rather than object-oriented programming paradigm.**

Fortran 90 Modules: 2/4

- **A Fortran 90 module has the following syntax:**

```
MODULE module-name
   IMPLICIT NONE
   [specification part]
CONTAINS
   [internal functions/subroutines]
END MODULE module-name
```

- **The specification part and internal functions and subroutines are optional.**
- **A module looks like a `PROGRAM`, except that it does not have the executable part. Hence, a main program must be there to use modules.**

Fortran 90 Modules: 3/4

- **Examples:**

Module `SomeConstants` does not have the subprogram part

```
MODULE SomeConstants
  IMPLICIT NONE
  REAL, PARAMETER :: PI=3.1415926
  REAL, PARAMETER :: g = 980
  INTEGER :: Counter
END MODULE SomeConstants
```

Module `SumAverage` does not have the specification part

```
MODULE SumAverage

CONTAINS
  REAL FUNCTION Sum(a, b, c)
    IMPLICIT NONE
    REAL, INTENT(IN) :: a, b, c
    Sum = a + b + c
  END FUNCTION Sum
  REAL FUNCTION Average(a, b, c)
    IMPLICIT NONE
    REAL, INTENT(IN) :: a, b, c
    Average = Sum(a,b,c)/2.0
  END FUNCTION Average
END MODULE SumAverage
```

Fortran 90 Modules: 4/4

- The right module has both the **specification part** and **internal functions**.
- Normally, this is the case.

```
MODULE DegreeRadianConversion
  IMPLICIT NONE
  REAL, PARAMETER :: PI = 3.1415926
  REAL, PARAMETER :: Degree180 = 180.0

CONTAINS
  REAL FUNCTION DegreeToRadian(Degree)
    IMPLICIT NONE
    REAL, INTENT(IN) :: Degree
    DegreeToRadian = Degree*PI/Degree180
  END FUNCTION DegreeToRadian
  REAL FUNCTION RadianToDegree(radian)
    IMPLICIT NONE
    REAL, INTENT(IN) :: Radian
    RadianToDegree = Radian*Degree180/PI
  END FUNCTION RadianToDegree
END MODULE DegreeRadianConversion
```

Some Privacy: 1/2

- **Fortran 90 allows a module to have *private* and *public* items. However, *all global entities of a module, by default, are public* (*i.e.*, visible in all other programs and modules).**
- **To specify public and private, do the following:**

```
PUBLIC  :: name-1, name-2, …, name-n
PRIVATE :: name-1, name-2, …, name-n
```

- **The `PRIVATE` statement without a name makes all entities in a module *private*. To make some entities visible, use `PUBLIC`.**
- **`PUBLIC` and `PRIVATE` may also be used in type specification:**

```
INTEGER, PRIVATE :: Sum, Phone_Number
```

Some Privacy: 2/2

- Any global entity (*e.g.*, **PARAMETER**, variable, function, subroutine, etc) can be in **PUBLIC** or **PRIVATE** statements.

```
MODULE TheForce
  IMPLICIT NONE
  INTEGER :: SkyWalker, Princess
  REAL, PRIVATE :: BlackKnight
  LOGICAL :: DeathStar
  REAL, PARAMETER :: SecretConstant = 0.123456
  PUBLIC   :: SkyWalker, Princess
  PRIVATE  :: VolumeOfDeathStar
  PRIVATE  :: SecretConstant
CONTAINS
  INTEGER FUNCTION VolumeOfDeathStar()
    ..........
  END FUNCTION WolumeOfDeathStar
  REAL FUNCTION WeaponPower(SomeWeapon)
    ..........
  END FUNCTION ..........
END MODULE TheForce
```

Is this public?

By default, this **PUBLIC** statement does not make much sense

Using a Module: 1/5

- **A `PROGRAM` or `MODULE` can use `PUBLIC` entities in any other modules. However, one must declare this intention (of use).**
- **There are two forms of the `USE` statement for this task:**

```
USE module-name

USE module-name, ONLY: name-1, name-2, ..., name-n
```

- **The first `USE` indicates all `PUBLIC` entities of `MODULE` `module-name` will be used.**
- **The second makes use only the names listed after the `ONLY` keyword.**

Using a Module: 2/5

- Two simple examples:

```
PROGRAM Main
   USE SomeConstants
   IMPLICIT NONE
      ..........
END PROGRAM Main
```

```
MODULE SomeConstants
   IMPLICIT NONE
   REAL, PARAMETER :: PI = 3.1415926
   REAL, PARAMETER :: g = 980
   INTEGER         :: Counter
END MODULE SomeConstants
```

```
MODULE DoSomething
   USE SomeConstants, ONLY : g, Counter
   IMPLICIT NONE
CONTAINS
   SUBROUTINE Something(…)
      ……
   END SUBROUTINE Something
END MODULE DoSomething
```

PI is not available

Using a Module: 3/5

- **Sometimes, the "*imported*" entities from a `MODULE` may have identical names with names in the "*importing*" `PROGRAM` or `MODULE`.**
- **If this happens, one may use the "*renaming*" feature of `USE`.**
- **For each identifier in `USE` to be renamed, use the following syntax:**

```
name-in-this-PROGRAM => name-in-module
```

- **In this program, the use of `name-in-this-PROGRAM` is equivalent to the use of `name-in-module` in the "*imported*" `MODULE`.**

Using a Module: 4/5

- The following uses module **MyModule**.
- Identifiers **Counter** and **Test** in module **MyModule** are renamed as **MyCounter** and **MyTest** in *this* module, respectively:

```
USE MyModule, MyCounter => Counter &
              MyTest    => Test
```

- The following only uses identifiers **Ans**, **Condition** and **X** from module **Package** with **Condition** renamed as **Status**:

```
USE Package, ONLY : Ans, Status => Condition, X
```

Using a Module: 5/5

- Two **USE** and **=>** examples

```
MODULE SomeConstants
  IMPLICIT NONE
  REAL, PARAMETER :: PI = 3.1415926
  REAL, PARAMETER :: g = 980
  INTEGER         :: Counter
END MODULE SomeConstants
```

GravityG is the **g** in the module; however, **g** is the "**g**" in **Test**

```
PROGRAM Test
  USE SomeConstants, &
      GravityG => g
  IMPLICIT NONE
  INTEGER :: g
    ......
END PROGRAM Test
```

```
MODULE Compute
  USE SomeConstants, ONLY : PI, g
  IMPLICIT NONE
  REAL :: Counter
CONTAINS
  ......
END MODULE Compute
```

without **ONLY**, **Counter** would appear in **MODULE Compute** causing a name conflict!

Compile Your Program: 1/4

- **Suppose a program consists of the main program `main.f90` and 2 modules `Test.f90` and `Compute.f90`. In general, they can be compiled in the following way:**

```
f90 main.f90 Test.f90 Compute.f90 -o main
```

- **However, some compilers may be a little more restrictive. *List those modules that do not use any other modules first, followed by those modules that only use those listed modules, followed by your main program.***

Compile Your Program: 2/4

- Suppose we have modules **A**, **B**, **C**, **D** and **E**, and **C** uses **A**, **D** uses **B**, and **E** uses **A**, **C** and **D**, then a safest way to compile your program is the following command:

```
f90 A.f90 B.f90 C.f90 D.f90 E.f90 main.f90 -o main
```

- Since modules are supposed to be designed and developed separately, they can also be compiled separately to object codes:

```
f90 -c test.f90
```

- The above compiles a module/program in file `test.f90` to its object code `test.o`

This means compile only

Compile Your Program: 3/4

- **Suppose we have modules `A`, `B`, `C`, `D` and `E`, and `C` uses `A`, `D` uses `B`, and `E` uses `A`, `C` and `D`.**
- **Since modules are developed separately with some specific functionality in mind, one may compile each module to object code as follows:**

```
f90 -c A.f90
f90 -c B.f90
f90 -c C.f90
f90 -c D.f90
f90 -c E.f90
```

If your compiler is picky, some modules may have to compiled together!

- **Note that the order is still important. The above generates object files `A.o, B.o, C.o, D.o` and `E.o`**

Compile Your Program: 4/4

- If a main program in file **`prog2.f90`** uses modules in **`A.f90`** and **`B.f90`**, one may compile and generate executable code for **`prog2`** as follows:

```
f90 A.o B.o prog2.f90 -o prog2
```

- If **`prog2.f90`** uses module **`E.f90`** only, the following must be used since **`E.f90`** uses **`A.f90`**, **`C.f90`** and **`D.f90`**:

```
f90 A.o C.o D.o E.o prog2.f90 -o prog2
```

- Note the order of the object files.

Example 1

- The combinatorial coefficient of *m* and *n* ($m \geq n$) is $C_{m,n} = m!/(n! \times (m-n)!)$.

```
MODULE FactorialModule
  IMPLICIT NONE
CONTAINS
  INTEGER FUNCTION Factorial(n)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n
      … other statements …
  END FUNCTION Factorial
  INTEGER FUNCTION Combinatorial(n, r)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n, r
      … other statements …
  END FUNCTION Combinatorial
END MODULE FactorialModule
```

```
PROGRAM ComputeFactorial
  USE FactorialModule
  IMPLICIT NONE
  INTEGER :: N, R
  READ(*,*) N, R
  WRITE(*,*) Factorial(N)
  WRITE(*,*) Combinatorial(N,R)
END PROGRAM ComputeFactorial
```

`Combinatorial(n,r)` uses `Factorial(n)`

Example 2

- Trigonometric functions use degree.

```
MODULE MyTrigonometricFunctions
  IMPLICIT NONE
  REAL, PARAMETER :: PI = 3.1415926
  REAL, PARAMETER :: Degree180 = 180.0
  REAL, PARAMETER :: R_to_D=Degree180/PI
  REAL, PARAMETER :: D_to_R=PI/Degree180
CONTAINS
  REAL FUNCTION DegreeToRadian(Degree)
    IMPLICIT NONE
    REAL, INTENT(IN) :: Degree
    DegreeToRadian = Degree * D_to_R
  END FUNCTION DegreeToRadian
  REAL FUNCTION MySIN(x)
    IMPLICIT NONE
    REAL, INTENT(IN) :: x
    MySIN = SIN(DegreeToRadian(x))
  END FUNCTION MySIN
    … other functions …
END MODULE MyTrigonometricFunctions
```

```
PROGRAM TrigonFunctTest
 USE MyTrigonometricFunctions
  IMPLICIT NONE
  REAL :: Begin = -180.0
  REAL :: Final = 180.0
  REAL :: Step = 10.0
  REAL :: x
  x = Begin
  DO
    IF (x > Final) EXIT
    WRITE(*,*) MySIN(x)
    x = x + Step
  END DO
END PROGRAM TrigonFunctTest
```

INTERFACE Blocks: 1/5

- **Legacy Fortran programs do not have internal subprograms in `PROGRAM`s or `MODULE`s.**
- **These subprograms are in separate files. These are *external* subprograms that may cause some compilation problems in Fortran 90.**
- **Therefore, Fortran 90 has the `INTERFACE` block for a program or a module to know the type of the subprograms, the intent and type of each argument, etc.**

INTERFACE Blocks: 2/5

- **Consider the following triangle area program.**
- **How does the main program know the type and number of arguments of the two functions?**

```
LOGICAL FUNCTION Test(a, b, c)
  IMPLICIT NONE
  REAL, INTENT(IN) :: a, b, c
  LOGICAL :: test1, test2
  test1 = (a>0.0) .AND. (b>0.0) .AND. (c>0.0)
  test2 = (a+b>c) .AND. (a+c>b) .AND. (b+c>a)
  Test = test1 .AND. test2
END FUNCTION Test

REAL FUNCTION Area(a, b, c)
   IMPLICIT NONE
   REAL, INTENT(IN) :: a, b, c
   REAL :: s = (a + b + c) / 2.0
   Area = SQRT(s*(s-a)*(s-b)*(s-c))
END FUNCTION Area
```

file **area.f90**

```
PROGRAM HeronFormula
  IMPLICIT NONE
    … some important here …
  REAL :: a, b, c
  REAL :: TriangleArea
  DO
    READ(*,*) a, b, c
    IF (Test(a,b,c)) EXIT
  END DO
  TriangleArea = Area(a, b, c)
  WRITE(*,*) TriangleArea
END PROGRAM HeronFormula
```

file **main.f90**

INTERFACE Blocks: 3/5

- An **INTERFACE** block has the following syntax:

```
INTERFACE
   type FUNCTION name(arg-1, arg-2, ..., arg-n)
      type, INTENT(IN) :: arg-1
      type, INTENT(IN) :: arg-2
      ..........
      type, INTENT(IN) :: arg-n
   END FUNCTION name
   SUBROUTINE name(arg-1, arg-2, …, arg-n)
      type, INTENT(IN or OUT or INOUT) :: arg-1
      type, INTENT(IN or OUT or INOUT) :: arg-2
      ..........
      type, INTENT(IN or OUT or INOUT) :: arg-n
   END SUBROUTINE name
   ....... other functions/subroutines .......
END INTERFACE
```

INTERFACE Blocks: 4/5

- **All external subprograms should be listed between `INTERFACE` and `END INTERFACE`.**
- **However, only the `FUNCTION` and `SUBROUTINE` headings, argument types and `INTENT`s are needed. *No executable statements should be included*.**
- **The argument names do not have to be identical to those of the formal arguments, because they are "*place-holders*" in an `INTERFACE` block.**
- **Thus, a main program or subprogram will be able to know exactly how to use a subprogram.**

INTERFACE Blocks: 5/5

- Return to Heron's formula for triangle area.
- The following shows the INTERFACE block in a main program.

```
LOGICAL FUNCTION Test(a, b, c)
  IMPLICIT NONE
  REAL, INTENT(IN) :: a, b, c
  LOGICAL :: test1, test2
  test1 = (a>0.0) .AND. (b>0.0) .AND. (c>0.0)
  test2 = (a+b>c) .AND. (a+c>b) .AND. (b+c>a)
  Test = test1 .AND. test2
END FUNCTION Test

REAL FUNCTION Area(a, b, c)
   IMPLICIT NONE
   REAL, INTENT(IN) :: a, b, c
   REAL :: s
   s = (a + b + c) / 2.0
   Area = SQRT(s*(s-a)*(s-b)*(s-c))
END FUNCTION Area
```

file area.f90

```
PROGRAM HeronFormula
 IMPLICIT NONE
 INTERFACE
  LOGICAL FUNCTION Test(x,y,z)
    REAL, INTENT(IN)::x,y,z
  END FUNCTION Test
  REAL FUNCTION Area(l,m,n)
    REAL, INTENT(IN)::l,m,n
  END FUNCTION Area
 END INTERFACE
  …… other statements …
END PROGRAM HeronFormula
```

file main.f90

The End