# Introduction to Modern Fortran

## *Modules and Interfaces*

Nick Maclaren

**nmm1@cam.ac.uk**

March 2014

# Module Summary

- Similar to same term in other languages

As usual, modules fulfil multiple purposes

- For shared declarations (i.e. "headers")

- Defining global data (old COMMON)

- Defining procedure interfaces

- Semantic extension (described later)

And more ...

# Use Of Modules

- Think of a module as a high–level interface
Collects <whatevers> into a coherent unit

- Design your modules carefully
As the ultimate top–level program structure
Perhaps only a few, perhaps dozens

- Good place for high–level comments
Please document purpose and interfaces

# Module Interactions

Modules can USE other modules
Dependency graph shows visibility / usage

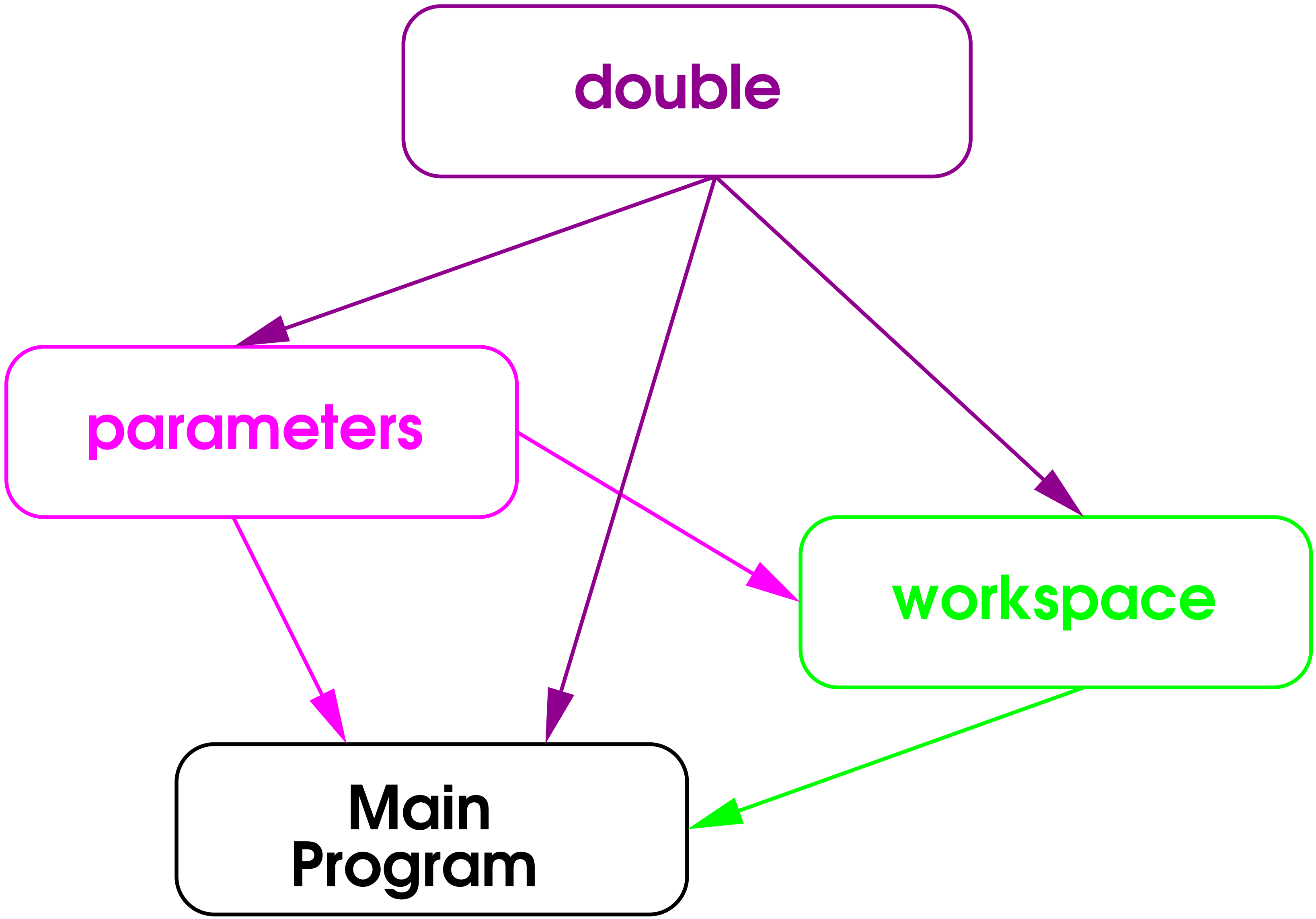
- Modules may not depend on themselves

Languages that allow that are very confusing

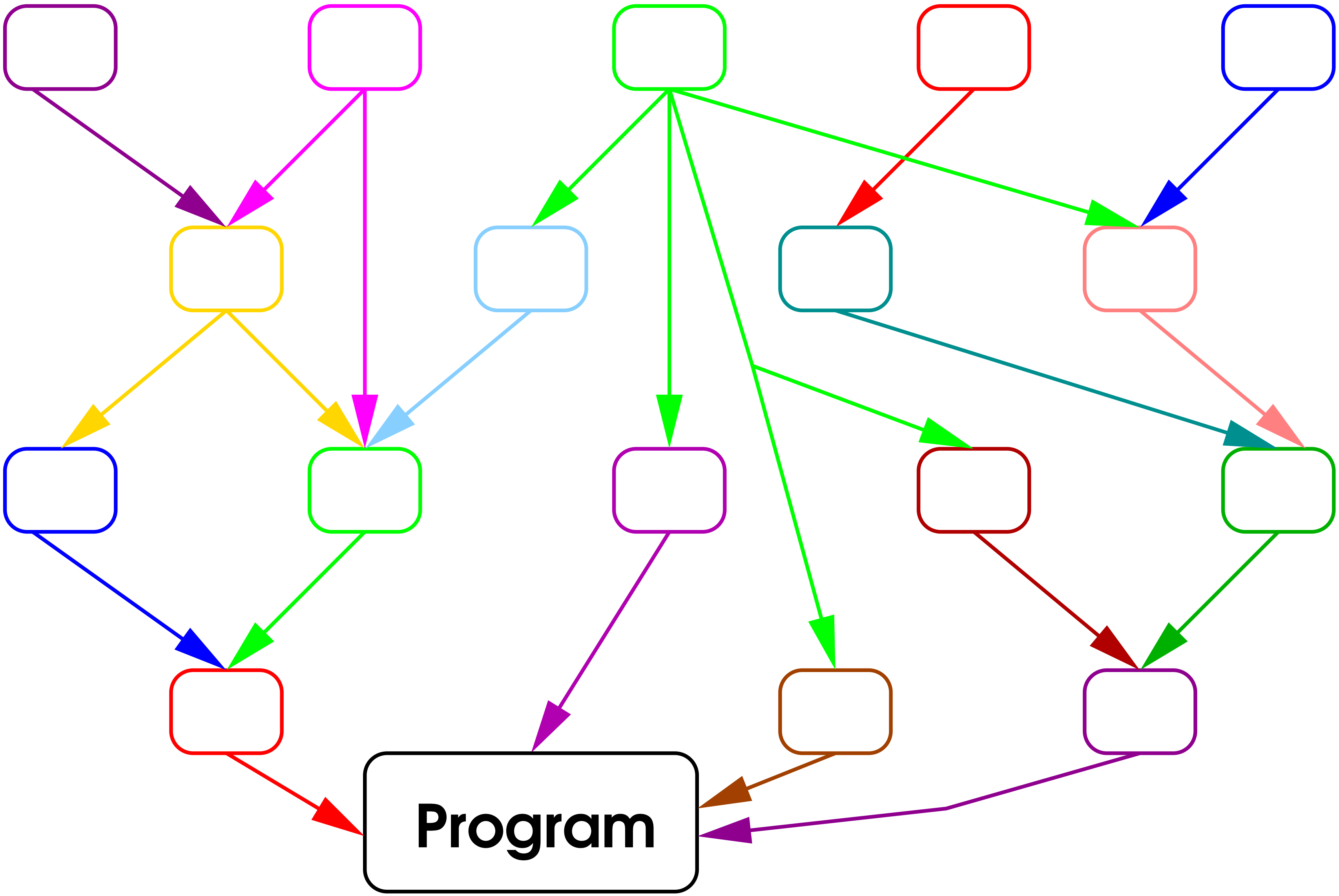
Can do anything you are likely to get to work

- If you need to do more, ask for advice

# Module Dependencies



# Module Dependencies



# Module Structure

MODULE <name>

    Static (often exported) data definitions

CONTAINS

    Procedure definitions (i.e. their code)

END MODULE <name>

Files may contain several modules

Modules may be split across many files

- For simplest use, keep them 1≡1

# IMPLICIT NONE

Add MODULE to the places where you use this

```
MODULE double
    IMPLICIT NONE
    INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double

MODULE parameters
    USE double
    IMPLICIT NONE
    REAL(KIND=DP), PARAMETER :: one = 1.0_DP
END MODULE parameters
```

# Reminder

I do not always do it, because of space

# Example (1)

```
MODULE double
    INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double

MODULE parameters
    USE double
    REAL(KIND=DP), PARAMETER :: one = 1.0_DP
    INTEGER, PARAMETER :: NX = 10, NY = 20
END MODULE parameters

MODULE workspace
    USE double ;    USE parameters
  REAL(KIND=DP), DIMENSION(NX, NY) :: now, then
END MODULE workspace
```

# Example (2)

The main program might use them like this

```
PROGRAM main
    USE double
    USE parameters
    USE workspace
    . . .
END PROGRAM main
```

- Could omit the USE double and USE parameters

They would be inherited through USE workspace

# Shared Constants

We have already seen and used this:

```
MODULE double
    INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double
```

You can do a great deal of that sort of thing

- Greatly improves clarity and maintainability

The larger the program, the more it helps

# Example

```
MODULE hotchpotch
    INTEGER, PARAMETER :: DP = KIND(0.0D0)
    REAL(KIND=DP), PARAMETER ::    &
        pi = 3.141592653589793_DP,    &
        e = 2.718281828459045_DP
    CHARACTER(LEN=*), PARAMETER ::    &
        messages(3) =    &
            (\ "Hello", "Goodbye", "Oh, no!" \)
    INTEGER, PARAMETER :: stdin = 5, stdout = 6
    REAL(KIND=DP), PARAMETER,    &
       DIMENSION(0:100, -1:25, 1:4) :: table =    &
        RESHAPE( (/ . . . /), (/ 101, 27, 4 /) )
END MODULE hotchpotch
```

# Global Data

Variables in modules define global data
These can be fixed–size or allocatable arrays

- You need to specify the SAVE attribute

Set automatically for initialised variables
But it is good practice to do it explicitly

A simple SAVE statement saves everything

- That isn't always the best thing to do

# Example (1)

```
MODULE state_variables
    INTEGER, PARAMETER :: nx=100, ny=100
    REAL, DIMENSION(NX, NY), SAVE ::    &
        current, increment, values
    REAL, SAVE :: time = 0.0
END MODULE state_variables

USE state_variables
IMPLICIT NONE
DO
    current = current + increment
    CALL next_step(current, values)
END DO
```

# Example (2)

This is equivalent to the previous example

```
MODULE state_variables
    IMPLICIT NONE
    SAVE
    INTEGER, PARAMETER :: nx=100, ny=100
    REAL, DIMENSION(NX, NY) ::    &
        current, increment, values
    REAL :: time = 0.0
END MODULE state_variables
```

# Example (3)

The sizes do not have to be fixed

```
MODULE state_variables
    REAL, DIMENSION(:, :), ALLOCATABLE,   &
        SAVE :: current, increment, values
END MODULE state_variables

USE state_variables
IMPLICIT NONE
INTEGER :: NX, NY
READ *, NX, NY
ALLOCATE (current(NX, NY), increment(NX, NY),   &
    values(NX, NY))
```

# Use of SAVE

If a variable is set in one procedure
and then it is used in another

- You must specify the SAVE attribute

- If not, very strange things may happen

If will usually "work", under most compilers
A new version will appear, and then it won't

- Applies if the association is via the module

Not when it is passed as an argument

# Example (1)

```
MODULE status
      REAL :: state
END MODULE status

SUBROUTINE joe
      USE status
      state = 0.0
END SUBROUTINE joe

SUBROUTINE alf (arg)
      REAL :: arg
      arg = 0.0
END SUBROUTINE alf
```

# Example (2)

```
SUBROUTINE fred
    USE status

    CALL joe
    PRINT *, state    ! this is UNDEFINED

    CALL alf(state)
    PRINT *, state    ! this is defined to be 0.0

END SUBROUTINE fred
```

# Shared Workspace

Shared scratch space can be useful for HPC
It can avoid excessive memory fragmentation

You can omit SAVE for simple scratch space
This can be significantly more efficient

- Design your data use carefully

Separate global scratch space from storage
And use them consistently and correctly

- This is good practice in any case

# Module Procedures (1)

Procedures now need explicit interfaces
E.g. for assumed shape or keywords
Without them, must use Fortran 77 interfaces

- Modules are the primary way of doing this

We will come to the secondary one later

Simplest to include the procedures in modules
The procedure code goes after CONTAINS
This is what we described earlier

# Example

```
MODULE mymod
CONTAINS
    FUNCTION Variance (Array)
        REAL :: Variance, X
        REAL, INTENT(IN), DIMENSION(:) :: Array
        X = SUM(Array)/SIZE(Array)
        Variance = SUM((Array-X)**2)/SIZE(Array)
    END FUNCTION Variance
END MODULE mymod

PROGRAM main
    USE mymod
    . . .
    PRINT *, 'Variance = ', Variance(array)
```

# Module Procedures (2)

- Modules can contain any number of procedures
- You can use any number of modules

```
PROGRAM main
      USE mymod
      REAL, DIMENSION(10) :: array
      PRINT *, 'Type 10 values'
      READ *, array
      PRINT *, 'Variance = ', Variance(array)
END PROGRAM main
```

# Using Procedures

Internal procedures or module procedures?
Use either technique for solving test problems

- They are the best techniques for real code
Simplest, and give full access to functionality
We will cover some other ones later

- Note that, if a procedure is in a module
  it may still have internal procedures

# Example

```
MODULE mymod
CONTAINS
    SUBROUTINE Sorter (array, opts)
        . . .
    CONTAINS
        FUNCTION Compare (value1, value2, flags)
            . . .
        END FUNCTION Compare
        SUBROUTINE Swap (loc1, loc2)
            . . .
        END FUNCTION Swap
    END SUBROUTINE Sorter
END MODULE mymod
```

# Procedures in Modules (1)

That is including all procedures in modules
Works very well in almost all programs

- There really isn't much more to it

It doesn't handle very large modules well
Try to avoid designing those, if possible

It also doesn't handle procedure arguments
Unfortunately, doing that has had to be omitted

# Procedures in Modules (2)

They are very like internal procedures

Everything accessible in the module
can also be used in the procedure

Again, a local name takes precedence
But reusing the same name is very confusing

# Procedures in Modules (3)

```
MODULE thing
    INTEGER, PARAMETER :: temp = 123
CONTAINS
    SUBROUTINE pete ()
        INTEGER, PARAMETER :: temp = 456
        PRINT *, temp
    END SUBROUTINE pete
END MODULE thing
```

Will print 456, not 123

Avoid doing this – it's very confusing

# Derived Type Definitions

We shall cover these later:

```
MODULE Bicycle
    TYPE Wheel
        INTEGER :: spokes
        REAL  :: diameter, width
        CHARACTER(LEN=15) :: material
    END TYPE Wheel
END MODULE Bicycle

USE Bicycle
TYPE(Wheel) :: w1
```

# Compiling Modules (1)

This is a FAQ – Frequently Asked Question
The problem is the answer isn't simple

- That is why I give some of the advice that I do

The following advice will not always work
OK for most compilers, but not necessarily all

- This is only the Fortran module information

# Compiling Modules (2)

The module name need not be the file name
Doing that is strongly recommended, though

- You can include any number of whatevers

You now compile it, but don't link it

```
nagfor –C=all –c mymod.f90
```

It will create files like mymod.mod and mymod.o
They contain the interface and the code

Will describe the process in more detail later

# Using Compiled Modules

All the program needs is the USE statements

- Compile all of the modules in a dependency order

If A contains USE B, compile B first

- Then add a *.o for every module when linking

```
nagfor −C=all −o main main.f90 mymod.o

nagfor −C=all −o main main.f90  \
     mod_a.o mod_b.o mod_c.o
```

# Take a Breather

That is most of the basics of modules
Except for interfaces and access control

The first question covers the material so far

The remainder is important and useful
But it is unfortunately rather more complicated

# What Are Interfaces?

The FUNCTION or SUBROUTINE statement
And everything directly connected to that
USE if needed for argument declarations
- And don't forget a function result declaration

Strictly, the argument names are not part of it
You are strongly advised to keep them the same
Which keywords if the interface and code differ?

Actually, it's the ones in the interface

# Interface Blocks

These start with an INTERFACE statement
Include any number of procedure interfaces
And end with an END INTERFACE statement

```
INTERFACE
    SUBROUTINE Fred (arg)
      REAL :: arg
    END FUNCTION Fred
    FUNCTION Joe ()
      LOGICAL :: Joe
    END FUNCTION Joe
END INTERFACE
```

# Example

```
SUBROUTINE CHOLESKY (A)    ! this is part of it
    USE errors    ! this ISN'T part of it
    USE double    ! this is, because of A
    IMPLICIT NONE    ! this ISN'T part of it
    INTEGER :: J, N    ! this ISN'T part of it
    REAL(KIND=dp) :: A(:, :), X    ! A is but not X
    . . .
END SUBROUTINE CHOLESKY

INTERFACE
    SUBROUTINE CHOLESKY (A)
        USE double
        REAL(KIND=dp) :: A(:, :)
    END SUBROUTINE CHOLESKY
END INTERFACE
```

# Interfaces In Procedures

Can use an interface block as a declaration
Provides an explicit interface for a procedure

Can be used for ordinary procedure calls
But using modules is almost always better

- It is essential for procedure arguments

Can't put a dummy argument name in a module!

More on this in the Make and Linking lecture

# Example (1)

Assume this is in module application

```
FUNCTION apply (arr, func)
    REAL :: apply, arr(:)
    INTERFACE
        FUNCTION func (val)
            REAL :: func, val
        END FUNCTION
    END INTERFACE
    apply = 0.0
    DO I = 1,UBOUND(arr, 1)
        apply = apply + func(val = arr(i))
    END DO
END FUNCTION apply
```

# Example (2)

And these are in module functions

```
FUNCTION square (arg)
    REAL :: square, arg
    square = arg**2
END FUNCTION square

FUNCTION cube (arg)
    REAL :: cube, arg
    cube = arg**3
END FUNCTION cube
```

# Example (3)

```
PROGRAM main
    USE application
    USE functions
   REAL, DIMENSION(5) :: A = (/ 1.0, 2.0, 3.0, 4.0, 5.0 /)
    PRINT *, apply(A,square)
    PRINT *, apply(A,cube)
END PROGRAM main
```

Will produce something like:

```
 55.0000000
  2.2500000E+02
```

# Interface Bodies and Names (1)

An interface body does not import names
The reason is that you can't undeclare names

For example, this does not work as expected:

```
USE double        ! This doesn't help
INTERFACE
    FUNCTION square (arg)
        REAL(KIND=dp) :: square, arg
    END FUNCTION square
END INTERFACE
```

# Interface Bodies and Names (2)

So there is another statement to import names:

```
USE double
INTERFACE
    FUNCTION square (arg)
        IMPORT :: dp        ! This solves it
        REAL(KIND=dp) :: square, arg
    END FUNCTION square
END INTERFACE
```

It is available only in interface bodies

# Accessibility (1)

Can separate exported from hidden definitions

Fairly easy to use in simple cases

- Worth considering when designing modules

PRIVATE names accessible only in module
I.e. in module procedures after CONTAINS

PUBLIC names are accessible by USE
This is commonly called exporting them

# Accessibility (2)

They are just another attribute of declarations

```
MODULE fred
    REAL, PRIVATE :: array(100)
    REAL, PUBLIC :: total
    INTEGER, PRIVATE :: error_count
    CHARACTER(LEN=50), PUBLIC :: excuse
CONTAINS
    ...
END MODULE fred
```

# Accessibility (3)

PUBLIC/PRIVATE statement sets the default
The default default is PUBLIC

```
MODULE fred
    PRIVATE
    REAL :: array(100)
    REAL, PUBLIC :: total
CONTAINS
    . . .
END MODULE fred
```

Only TOTAL is accessible by USE

# Accessibility (4)

You can specify names in the statement
Especially useful for included names

```
MODULE workspace
    USE double
    PRIVATE :: DP
    REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace
```

DP is no longer exported via workspace

# Partial Inclusion (1)

You can include only some names in USE

```
USE bigmodule, ONLY : errors, invert
```

Makes only errors and invert visible
However many names bigmodule exports

Using ONLY is good practice
Makes it easier to keep track of uses

Can find out what is used where with grep

# Partial Inclusion (2)

- One case when it is strongly recommended
When using USE in modules

- All included names are exported
Unless you explicitly mark them PRIVATE

- Ideally, use both ONLY and PRIVATE
Almost always, use at least one of them

- Another case when it is almost essential
Is if you don’t use IMPLICIT NONE religiously

# Partial Inclusion (3)

If you don't restrict exporting and importing:

A typing error could trash a module variable

Or forget that you had already used the name
In another file far, far away ...

- The resulting chaos is almost unfindable

From bitter experience – in Fortran and C!

# Example (1)

```
MODULE settings
    INTEGER, PARAMETER :: DP = KIND(0.0D0)
    REAL(KIND=DP) :: Z = 1.0_DP
END MODULE settings

MODULE workspace
    USE settings
    REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace
```

# Example (2)

```
PROGRAM main
    IMPLICIT NONE
    USE workspace
    Z = 123
    . . .
END PROGRAM main
```

- DP is inherited, which is OK
- Did you mean to update Z in settings?

No problem if workspace had used ONLY : DP

# Example (3)

The following are better and best

```
MODULE workspace
    USE settings, ONLY : DP
    REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace

MODULE workspace
    USE settings, ONLY : DP
    PRIVATE :: DP
    REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace
```

# Renaming Inclusion (1)

You can rename a name when you include it

WARNING: this is footgun territory
[ i.e. point gun at foot; pull trigger ]

This technique is sometimes incredibly useful

- But is always incredibly dangerous

Use it only when you really need to
And even then as little as possible

# Renaming Inclusion (2)

```
MODULE corner
    REAL, DIMENSION(100) :: pooh
END MODULE corner

PROGRAM house
    USE corner, sanders => pooh
    INTEGER, DIMENSION(20) :: pooh
    . . .
END PROGRAM house
```

pooh is accessible under the name sanders
The name pooh is the local array

# Why Is This Lethal?

```
MODULE one
    REAL :: X
END MODULE one

MODULE two
    USE one, Y => X
    REAL :: Z
END MODULE two

PROGRAM three
    USE one ;   USE two
    ! Both X and Y refer to the same variable
END PROGRAM three
```

# Interfaces and Access Control

These are things that have been omitted
They're there in the notes, if you are interested

They are extremely important for large programs
But time is too tight to teach them now

- Do only the first practical and skip the rest