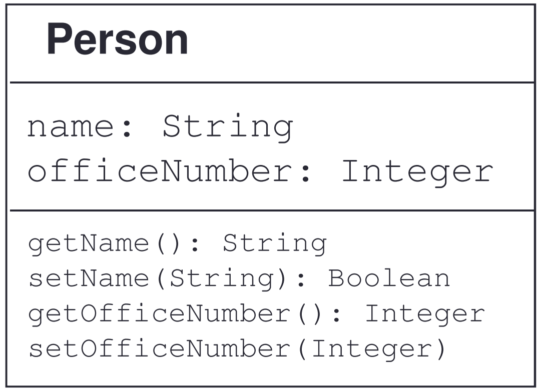
Fortran classes and data visibility

Reusing this material

Classes

- Extends derived types
  - Introduces concept of type-bound procedures
  - Class methods

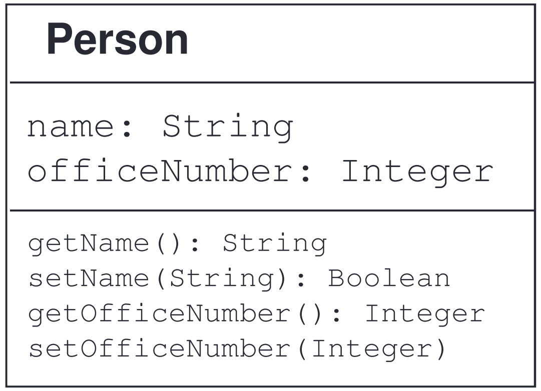
```
module building
   implicit none
   integer, parameter :: MAXLEN = 100
   type person
         character(MAXLEN) :: name
         integer :: officeNumber
   contains
         procedure, nopass :: getName
         procedure :: setName
         procedure :: getOfficeNumber
         procedure :: setOfficeNumber
   end type person
end module building
```



Classes

- Extends derived types
  - Introduces concept of type-bound procedures
  - Class methods

```
module building
  implicit none
  integer, parameter :: MAXLEN = 100
  type person
......
      procedure :: setOfficeNumber => newOfficeNumber
  end type person
contains
  subroutine newOfficeNumber(this, officeNumber)
     type(person) :: this
      integer :: officeNumber
      this%officeNumber = officeNumber
  end subroutine
end module building
```



Type bound procedure

```
PROCEDURE [(interface-name)] [[,binding-
attr-list ]::] binding-name[=> procedure-
name]
```

`binding-attr-list:`

- `PASS, NOPASS`
- `NON_OVERRIDABLE`
- `DEFERRED`
- `PUBLIC, PRIVATE`

Visibility

- Recall, derived type by default public
- Can make data and procedures default private using the `private` keyword
  - For procedures keyword comes after `contains`
- Explicitly can set procedures:
  - `private`
  - `public`

Visibility example

```
module building
  implicit none
  private
  integer, parameter :: MAXLEN = 100
  type person
    private
      character(MAXLEN) :: name
      integer :: officeNumber
  contains
    private
      procedure, public :: getName
      procedure, public :: setName
      procedure, public :: getOfficeNumber
      procedure, public :: setOfficeNumber
  end type person
end module building
```

Class variable

- Type bound procedures must take a class variable
  - Variable name is not prescribed (self is not a keyword)
  - Automatically passed
  - Allows for data polymorphism

```
…
contains
function getName(self)
class(person), intent(inout):: self
character(MAXLEN) :: getName
  getName = self%name
end function
…
end module building
```

- Could then be used:

```
type(person) :: bob
…
write(*,*) bob%getName()
…
```

Unlimited type

- Allowed unlimited polymorphic type

```
class(*)
```

- Pass in any type of variable or object
- Enables truly polymorphic routines
  - Combine with type-guarding for useful functionality
- If allocatable
  - Either type needs specified:

```
class(*),allocatable :: fred
allocate(person::fred)
```

  - Or source type needs specified:

```
person :: bob
class(*),allocatable :: fred
allocate(fred, source=bob)
```

    - In this case the allocation is made and the values copies into the new object

Select type

- Type inquiry/type guarding is possible
- `type is`
  - Type of object is the specified type
- `class is`
  - Class of the object is the same as the specified class or an extension of that class

```
select type (bob)
type is (manager)
  print *, ‘This is a manager’
class is (person)
  print *, ‘This could be a manager or person’
class default
  print *, ‘Unknown type used'
end select
```

Type comparison functions

- Two new intrinsic functions to inquire about types:

`EXTENDS_TYPE_OF(X,Y)`

  - Returns true if the type of X is the same as, or extends the type of Y
  - Some subtleties if Y is unallocated unlimited polymorphic type

`SAME_TYPE_AS(X,Y)`

  - Returns true if the type of Y is the same as the type of X

Class constructor

- Can specify a constructor
  - Using interface with same name as the derived type

```
…
  public :: person
  type person
    character(MAXLEN) :: name
    integer :: officeNumber
  contains
    procedure, public :: getName
    procedure, public :: setName
    procedure, public :: getOfficeNumber
    procedure, public :: setOfficeNumber
  end type person
  interface person
    module procedure initialise_person
  end interface
```

- Can be overloaded
- Not mandatory

Class destructor

- `final` keyword can be used to define procedure(s) to be called on object destruction

```
public :: person
 type person
   character(MAXLEN) :: name
   integer :: officeNumber
 contains
   procedure, public :: getName
   procedure, public :: setName
   procedure, public :: getOfficeNumber
   procedure, public :: setOfficeNumber
   final :: cleanUp
 end type person
 interface person
   module procedure initialise_person
 end interface
```

Class destructor

- Final routines must take a single argument of the same type as the derived type, i.e.:

```
subroutine cleanUp(object)
  type(person) :: object
   …..
end subroutine cleanUp
```

- Final routines are not called at the end of a program:
  - Termination of the program by **error**, by a **stop** statement or by execution of the **end** statement in the main program does not invoke any final subroutines (Modern Fortran Explained)
- If you want them to run at the end of a program wrap the main functionality in a subroutine

Summary

- F2003 allows tying procedures to derived types
  - Creates true classes
- Class procedures, by default, pass the class as an argument
- Default visibility of data and procedures public
  - Can easily restrict to make object safer and more object like
- Constructors and destructors available

Exercise

- Convert your basic derived types into classes by adding type bound procedures
- Explore unlimited polymorphism to build procedures that can work on different data types
- Do the same with percolate