Radia Perlman (1951)

- Experta en comunicacións
- Deseñadora de Internet
- Creadora do *Spanning Tree Protocol* (STP), fundamental para as pontes entre redes de ordenadores
- Realizou grandes avances nos métodos de transmisión de datos en rede
- Pertencente á *National Academy of Engineering* de EEUU

Módulo

- Bloque de código que contén datos e subprogramas: ***use*** *modulo*

Sintaxe:

```
module nome
    tipo :: var
    interface
        tipo subprograma(...)
        ...
        end tipo
    end interface
contains:
    tipo subprograma(...)
    ...
    end tipo
end module
```

Exemplo:

```
module proba
      integer :: x
contains
      subroutine sub(y)
            integer,intent(in) :: y
            print *,y
      end subroutine sub
end module
```

```
program modulo
use proba
call sub(5)
end program modulo
```

- Pode estar en arquivo distinto do programa principal
- Se está en arquivo distinto: *gfortran modulo.f90 principal.f90*

antes do arquivo que o usa!

Clase e obxecto

- Un dato pode ser:

1) Atómico (indivisible): *integer*, *real*, etc.
2) Agregado: colección de datos do mesmo tipo (vectores e matrices)
3) Heteroxéneo: colección de datos de distintos tipos (libro= título, autor, ISBN,editorial,ano,...)

- Unha *clase* é unha agrupación de:

1) Variables, en xeral de distintos tipos (dato heteroxéneo)

2) Subprogramas (funcións ou subrutinas) que operan con estas variables

- Un *obxecto* é unha variable dunha clase. Exemplo: defino a clase *persoa* e dous obxectos (variábeis) *p* e *q* de tipo *persoa*

Clase e obxecto en Fortran

- É un bloque *type*: inclúe variábeis e nomes de subprogramas que operan con estas variábeis
- Os subprogramas da clase van no mesmo módulo que o *type*
- Declaración dun obxecto *persoa*:

  type(*persoa*) :: *p=persoa*(*nome*,idade)
- Acceso a variables ou subprogramas: *p%nome*, *p%idade*, *p%mostra*()
- *persoa* é unha clase, *p* é un obxecto desa clase

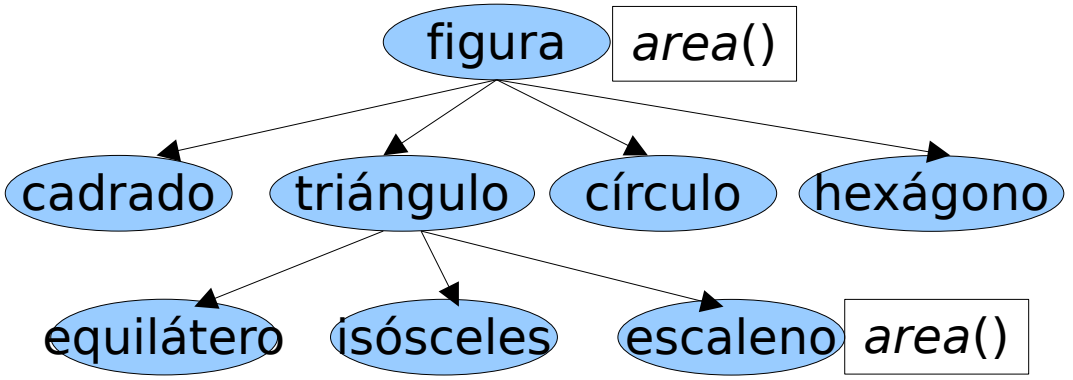
```
module modulo_persoa
   type persoa
      character(10) :: nome
      integer :: idade
   contains
      procedure :: mostra
   end type persoa
contains
   subroutine mostra(a)
      class(persoa),intent(in) :: a
      print *,'persoa:',a%nome
   end subroutine mostra
end module modulo_persoa
```

Alternativa a *p%mostra*()

```
program exemplo_tipos
use modulo_persoa
type(persoa) :: p=persoa('carlos',14)
call p%mostra()
print *,'nome=',p%nome,'idade=',p%idade
end program exemplo_tipos
```

Herdanza

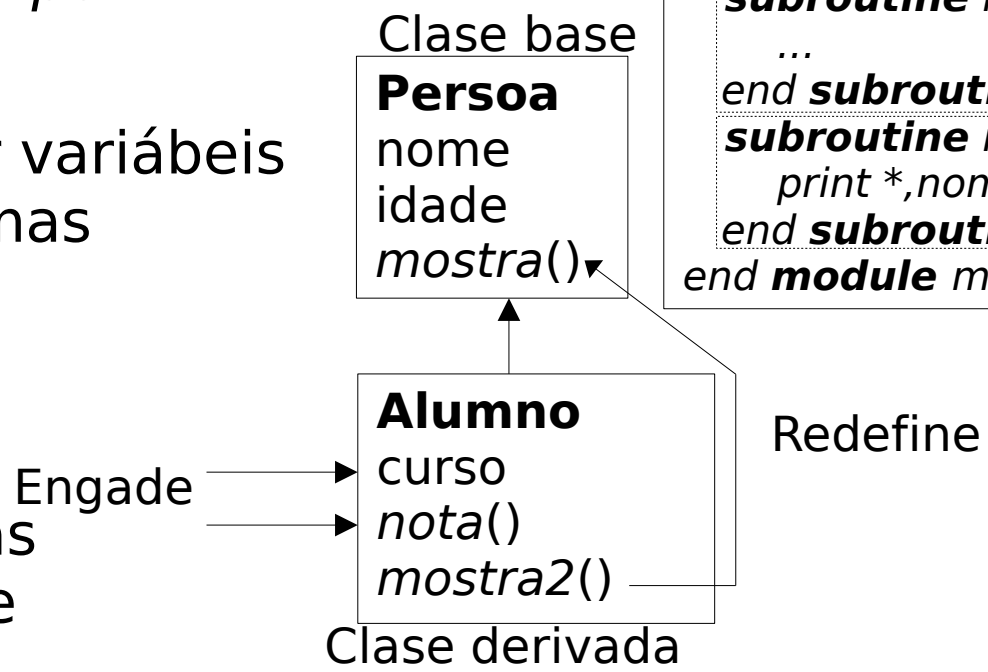
- Consiste en definir unha clase (derivada) que herede de outra (base) as variables e subprogramas.
- Isto evita ter que redefinir estas variables e subprogramas e permite reutilizar o código
- A clase derivada pode engadir variables e subprogramas, e tamén redefinir subprogramas heredados
- Pode haber varias clases derivadas da mesma clase base, definindo unha xerarquía de clases con varios niveis
- Cada clase derivada pode redefinir un subprograma da clase base ou heredalo
- Isto chámase *sobrecarga* (*overload*) do subprograma



Herdanza en Fortran

- Palabra clave *extends*: define unha clase derivada que extende a outra clase base
- A clase *alumno* hereda de *persoa*, engade a variábel *curso* e o subprograma *nota*(), e redefine o subprograma *mostra*() de *persoa substituíndoo por mostra2()*
- Pode engadir variábeis e subprogramas
- Tamén pode redefinir subprogramas da clase base

```
module modulo_persoa
   type,extends(persoa) :: alumno
      integer :: curso
   contains
      procedure :: mostra=>mostra2
      procedure :: nota
   end type alumno
contains
   subroutine nota()
      ...
   end subroutine nota
   subroutine mostra2()
      print *,nome,idade,curso
   end subroutine mostra2
end module modulo_persoa
```



Polimorfismo

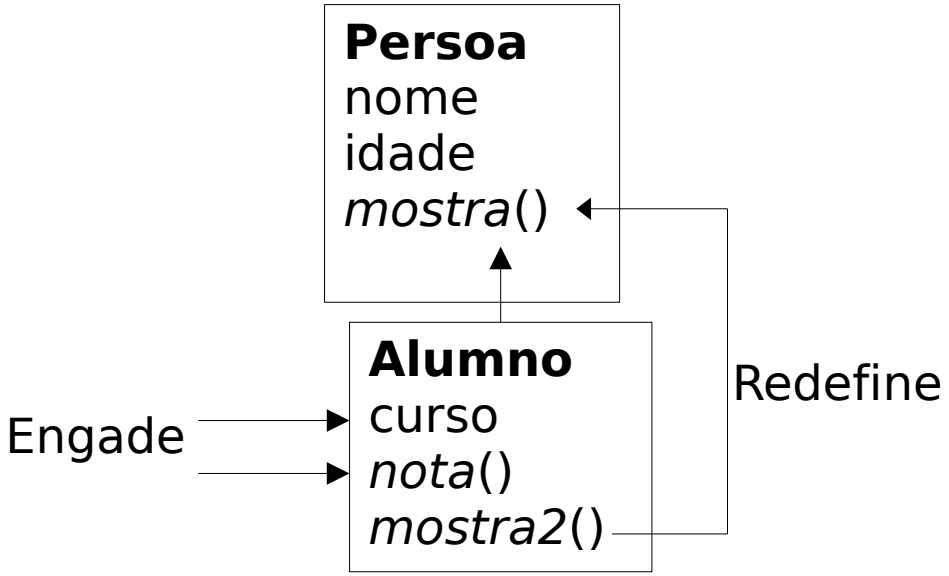
- Se hai varias clases derivadas e chamamos a un método redefinido por elas, será distinto para cada unha
- Deste modo, podes chamar a subprogramas distintos co mesmo nome sen ter que comprobar que subprograma executar
- Isto chámase *polimorfismo*: chamar da mesma forma a varios subprogramas distintos
- Vantaxe: cando chamas a un subprograma dun obxecto derivado, non hai que comprobar de que tipo é cada obxecto
- Polo tipo de obxecto, Fortran xa sabe a que subprograma chamar
- Para conseguir polimorfismo en Fortran declaras un obxecto dinámico da clase base e inicializas este obxecto coa clase derivada

Polimorfismo en Fortran

- A sentenza *procedure* indica o subprograma da clase derivada que sobrescribe o subprograma da clase base

procedure subprograma=>subprograma2

- *subprograma*: nome do subprograma na clase base
- *subprograma2*: redefinido pola clase derivada




```
module modulo_persoa
   type,extends(persoa) :: alumno
      integer :: curso
   contains
      procedure :: mostra=>mostra2
      procedure :: nota
   end type alumno
contains
   subroutine nota(a)
      class(alumno),intent(in) :: a
      ...
   end subroutine nota
   subroutine mostra2(a)
      class(alumno),intent(in) :: a
      print *,a%nome,a%idade,a%curso
   end subroutine mostra2
end module modulo_persoa
!-------------------------------------------------------
program principal
use modulo_persoa
class(persoa),allocatable :: p
p=alumno('Carlos',16,2)
call p%mostra()  ! chama a mostra2() de alumno
end program principal
```

p é de tipo *persoa* pero inicialízase a *alumno*

Sobrecarga de operadores aritméticos, relacionais ou lóxicos

- Define un operador a medida para unha clase
- Bloque *interface operator,* indicando o operador a sobrecargar e a función que o sobrecarga:

```
interface operator (+)
    procedure suma
end interface operator(+)
```

- Chámase á función *suma* cando se executa *p*+*q* sendo *p* e *q* obxectos da clase *persoa*. A función *suma* substitúe a +
- O operador pode ser aritmético, relacional ou lóxico.
- Se o operador é aritmético, a función debe retornar un obxecto do mesmo tipo que os operandos.

Sobrecarga de operador aritmético (+)

```
module modulo_persoa
    type persoa
        character(10) :: nome
        real :: peso,altura
    end type persoa
    interface operator (+)
        procedure suma
    end interface operator(+)
    contains
    type(persoa) function suma(x,y)
        class(persoa),intent(in) :: x,y
        character(10) :: nome
        real :: peso,altura
        nome=concat(x%nome,y%nome)
        peso=x%peso+y%peso
        altura=x%altura+y%altura
        suma=persoa(nome,peso,altura)
    end function suma
    character(10) function concat(x,y) result(s)
        character(10),intent(in) :: x,y
        do i=1,len_trim(x)
            s(i:i)=x(i:i)
        end do
        do j=1,len_trim(y)
            s(i:i)=y(j:j);i=i+1
        end do
    end function concat
end module modulo_persoa
```

```
program principal
use modulo_persoa
type(persoa) :: x,y,s
x=persoa('alba',65.2,1.84)
y=persoa('clara',70.1,1.98)
s=x+y
print *,'s=',s%nome,s%peso,x%altura
stop
end program principal
```

Función *concat*: concatena os nomes de *x* e *y*

Sobrecarga de operador relacional (>)

Se o operador que se sobrecarga é relacional ou lóxico, o subprograma que sobrecarga ao operador debe retornar un dato de tipo *logical*

```
program principal
use modulo_persoa
type(persoa) :: x=persoa('alba',65.2,1.84)
type(persoa) :: y=persoa('clara',70.1,1.98)
if(x>y) then
      print *,x%nome,'>',y%nome
else
      print *,x%nome,'<=',y%nome
end if
stop
end program principal
```

```
module modulo_persoa
      type persoa
            character(10) :: nome
            real :: peso,altura
      end type persoa
contains
      interface operator (>)
            procedure maior
      end interface operator(>)
      logical function maior(x,y)
            type(persoa),intent(in) :: x,y
            if(x%peso/x%altura > y%peso/y%altura) then
                  maior=.true.
            else
                  maior=.false.
            end if
      end function maior
end module modulo_persoa
```