Chapter 3

Object Oriented Programming Concepts

3.1 Introduction

The use of Object Oriented (OO) design and Object Oriented Programming (OOP) are becoming increasingly popular. Thus, it is useful to have an introductory understanding of OOP and some of the programming features of OO languages. You can develop OO software in any high level language, like C or Pascal. However, newer languages such as Ada, C++, and F90 have enhanced features that make OOP much more natural, practical, and maintainable. C++ appeared before F90 and currently, is probably the most popular OOP language, yet F90 was clearly designed to have almost all of the abilities of C++. However, rather than study the new standards many authors simply refer to the two decades old F77 standard and declare that Fortran can not be used for OOP. Here we will overcome that misinformed point of view.

Modern OO languages provide the programmer with three capabilities that improve and simplify the design of such programs: *encapsulation*, *inheritance*, and *polymorphism* (or generic functionality). Related topics involve *objects*, *classes*, and *data hiding*. An *object* combines various classical data types into a set that defines a new variable type, or structure. A *class* unifies the new entity types and supporting data that represents its state with routines (functions and subroutines) that access and/or modify those data. Every object created from a class, by providing the necessary data, is called an *instance* of the class. In older languages like C and F77, the data and functions are separate entities. An OO language provides a way to couple or encapsulate the data and its functions into a unified entity. This is a more natural way to model real-world entities which have both data and functionality. The encapsulation is done with a "module" block in F90, and with a "class" block in C++. This encapsulation also includes a mechanism whereby some or all of the data and supporting routines can be hidden from the user. The accessibility of the specifications and routines of a class is usually controlled by optional "public" and "private" qualifiers. *Data hiding* allows one the means to protect information in one part of a program from access, and especially from being changed in other parts of the program. In C++ the default is that data and functions are "private" unless declared "public," while F90 makes the opposite choice for its default protection mode. In a F90 "module" it is the "contains" statement that, among other things, couples the data, specifications, and operators before it to the functions and subroutines that follow it.

Class hierarchies can be visualized when we realize that we can employ one or more previously defined classes (of data and functionality) to organize additional classes. Functionality programmed into the earlier classes may not need to be re-coded to be usable in the later classes. This mechanism is called *inheritance*. For example, if we have defined an `Employee_class`, then a `Manager_class` would inherit all of the data and functionality of an employee. We would then only be required to add only the totally new data and functions needed for a manager. We may also need a mechanism to re-define specific `Employee_class` functions that differ for a `Manager_class`. By using the concept of a class hierarchy, less programming effort is required to create the final enhanced program. In F90 the earlier class is brought into the later class hierarchy by the "use" statement followed by the name of the "module" statement block that defined the class.

Polymorphism allows different classes of objects that share some common functionality to be used in code that requires only that common functionality. In other words, routines having the same generic name are interpreted differently depending on the class of the objects presented as arguments to the routines. This is useful in class hierarchies where a small number of meaningful function names can be used to manipulate different, but related object classes. The above concepts are those essential to object oriented design and OOP. In the later sections we will demonstrate by example additional F90 implementations of these concepts.

3.2 Encapsulation, Inheritance, and Polymorphism

We often need to use existing classes to define new classes. The two ways to do this are called *composition* and *inheritance*. We will use both methods in a series of examples. Consider a geometry program that uses two different classes: `class_Circle` and `class_Rectangle`, as represented graphically in Figs. 3.1 and 3.2. and as partially implemented in F90 as shown in Fig. 3.3. Each class shown has the data types and specifications to define the object and the functionality to compute their respective areas (lines 3–22). The operator % is employed to select specific components of a defined type. Within the geometry (main) program a single routine, `compute_area`, is invoked (lines 38 and 44) to return the area for *any* of the defined geometry classes. That is, a generic function name is used for all classes of its arguments and it, in turn, branches to the corresponding functionality supplied with the argument class. To accomplish this branching the geometry program first brings in the functionality of the desired classes via a "use" statement for each class module (lines 25 and 26). Those "modules" are coupled to the generic function by an "interface" block which has the generic function name `compute_area` (lines 28, 29). There is included a "module procedure" list which gives one class routine name for each of the classes of argument(s) that the generic function is designed to accept. The ability of a function to respond differently when supplied with arguments that are objects of different types is called *polymorphism*. In this example we have employed different names, `rectangular_area` and `circle_area`, in their respective class modules, but that is not necessary. The "use" statement allows one to rename the class routines and/or to bring in only selected members of the functionality.

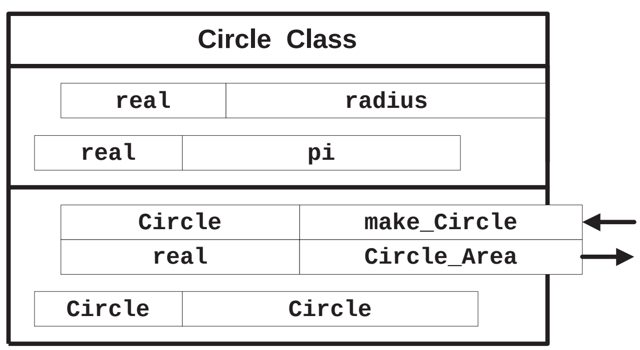


Figure 3.1: Representation of a Circle Class

Another terminology used in OOP is that of *constructors* and *destructors* for objects. An intrinsic constructor is a system function that is automatically invoked when an object is declared with all of its possible components in the defined order (see lines 37 and 43). In C++, and F90 the intrinsic constructor has the same name as the "type" of the object. One is illustrated in the statement

```
four_sides = Rectangle (2.1,4.3)
```

where previously we declared

```
type (Rectangle) :: four_sides
```

which, in turn, was coupled to the `class_Rectangle` which had two components, base and height, defined in that order, respectively. The intrinsic constructor in the example statement sets component

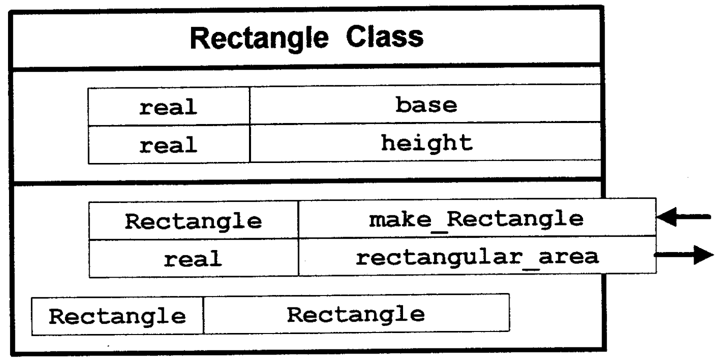


Figure 3.2: Representation of a Rectangle Class

```
!   Areas of shapes of different classes, using different
!             function names in each class
module class_Rectangle    ! define the first object class
implicit none
  type Rectangle
    real :: base, height ; end  type Rectangle
contains !  Computation of area for rectangles.
  function rectangle_area ( r ) result ( area )
    type ( Rectangle ), intent(in) :: r
    real                           :: area
      area = r%base * r%height ; end function rectangle_area
end module class_Rectangle

module class_Circle    ! define the second object class
  real :: pi = 3.1415926535897931d0 ! a circle constant
  type Circle
    real :: radius ; end  type Circle
contains !      Computation of area for circles.
  function circle_area ( c ) result ( area )
    type ( Circle ), intent(in) :: c
    real                        :: area
      area = pi * c%radius**2 ; end function circle_area
end module class_Circle

program geometry  ! for both types in a single function
 use class_Circle
 implicit none
 use class_Rectangle
!     Interface to generic routine to compute area for any type
 interface compute_area
   module procedure rectangle_area, circle_area ; end interface

!     Declare a set geometric objects.
 type ( Rectangle ) :: four_sides
 type ( Circle    ) :: two_sides       ! inside, outside
 real               :: area = 0.0      ! the result

!     Initialize a rectangle and compute its area.
   four_sides = Rectangle ( 2.1, 4.3 )    ! implicit constructor
   area = compute_area ( four_sides )    ! generic function
   write ( 6,100 ) four_sides, area       ! implicit components list
   100 format ("Area of ",f3.1," by ",f3.1," rectangle is ",f5.2)

!     Initialize a circle and compute its area.
   two_sides = Circle ( 5.4 )             ! implicit constructor
   area = compute_area ( two_sides )     ! generic function
   write ( 6,200 ) two_sides, area
   200 format ("Area of circle with ",f3.1," radius is ",f9.5 )
end program geometry                    ! Running gives:
! Area of 2.1 by 4.3 rectangle is  9.03
! Area of circle with 5.4 radius is  91.60885
```

Figure 3.3: Multiple Geometric Shape Classes

`base = 2.1` and component `height = 4.3` for that instance, `four_sides`, of the type `Rectangle`. This intrinsic construction is possible because all the expected components of the type were supplied. If all the components are not supplied, then the object cannot be constructed unless the functionality of the

```
[ 1]   function  make_Rectangle (bottom, side) result (name)
[ 2]  !          Constructor for a Rectangle type
[ 3]  implicit none
[ 4]     real, optional, intent(in) :: bottom, side
[ 5]     type (Rectangle)          :: name
[ 6]      name = Rectangle (1.,1.)      ! default to unit square
[ 7]      if ( present(bottom) ) then   ! default to square
[ 8]        name = Rectangle (bottom, bottom) ; end if
[ 9]      if ( present(side) ) name = Rectangle (bottom, side) ! intrinsic
[10]   end function  make_Rectangle
[11]   . . .
[12]   type ( Rectangle ) :: four_sides, square, unit_sq
[13]  !     Test manual constructors
[14]    four_sides = make_Rectangle (2.1,4.3)  ! manual constructor, 1
[15]    area = compute_area ( four_sides)      ! generic function
[16]    write ( 6,100 ) four_sides, area
[17]  !     Make a square
[18]    square = make_Rectangle (2.1)          ! manual constructor, 2
[19]    area = compute_area ( square)          ! generic function
[20]    write ( 6,100 ) square, area
[21]  !     "Default constructor", here a unit square
[22]    unit_sq = make_Rectangle ()            ! manual constructor, 3
[23]    area = compute_area (unit_sq)          ! generic function
[24]    write ( 6,100 ) unit_sq, area
[25]    . . .
[26]  ! Running gives:
[27]  ! Area of 2.1 by 4.3 rectangle is  9.03
[28]  ! Area of 2.1 by 2.1 rectangle is  4.41
[29]  ! Area of 1.0 by 1.0 rectangle is  1.00
```

Figure 3.4: A Manual Constructor for Rectangles

class is expanded by the programmer to accept a different number of arguments.

Assume that we want a special member of the `Rectangle` class, a square, to be constructed if the height is omitted. That is, we would use `height = base` in that case. Or, we may want to construct a unit square if both are omitted so that the constructor defaults to `base = height = 1`. Such a manual constructor, named `make_Rectangle`, is illustrated in Fig. 3.4 (see lines 5, 6). It illustrates some additional features of F90. Note that the last two arguments were declared to have the additional type attributes of "optional" (line 3), and that an associated logical function "present" is utilized (lines 6 and 8) to determine if the calling program supplied the argument in question. That figure also shows the results of the area computations for the corresponding variables "square" and "unit_sq" defined if the manual constructor is called with one or no optional arguments (line 5), respectively.

In the next section we will illustrate the concept of data hiding by using the `private` attribute. The reader is warned that the intrinsic constructor can not be employed if any of its arguments have been hidden. In that case a manual constructor must be provided to deal with any hidden components. Since data hiding is so common it is probably best to plan on prividing a manual constructor.

3.2.1 Example Date, Person, and Student Classes

Before moving to some mathematical examples we will introduce the concept of data hiding and combine a series of classes to illustrate composition and inheritance[†]. First, consider a simple class to define dates and to print them in a pretty fashion, as shown in Figs. 3.5 and 3.6. While other modules will have access to the Date class they will not be given access to the number of components it contains (3), nor their names (month, day, year), nor their types (integers) because they are declared "private" in the defining module (lines 5 and 6). The compiler will not allow external access to data and/or routines declared as private. The module, `class_Date`, is presented as a source "include" file in Fig. 3.6, and in the future will be reference by the file name `class_Date.f90`. Since we have chosen to hide all the user defined components we must decide what functionality we will provide to the users, who may have only executable access. The supporting documentation would have to name the public routines and describe their arguments and return results. The default intrinsic constructor would be available only to those that know full details about the components of the data type, and if those components are "public."

[†]These examples mimic those given in Chapter 11 and 8 of the J.R. Hubbard book "Programming with C++," McGraw-Hill, 1994, and usually use the same data for verification.

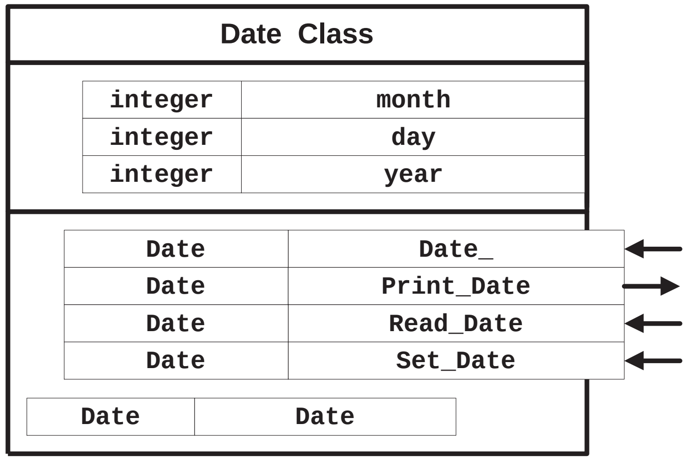


Figure 3.5: Graphical Representation of a Date Class

The intrinsic constructor, Date (lines 14 and 34), requires all the components be supplied, but it does no error or consistency checks. My practice is to also define a "public constructor" whose name is the same as the intrinsic constructor except for an appended underscore, that is, Date_. Its sole purpose is to do data checking and invoke the intrinsic constructor, Date. If the function Date_ (line 10) is declared "public" it can be used outside the module class_Date to invoke the intrinsic constructor, even if the components of the data type being constructed are all "private." In this example we have provided another manual constructor to set a date, set_Date (line 31), with a variable number of optional arguments. Also supplied are two subroutines to read and print dates, read_Date (line 27) and print_Date (line 16), respectively.

A sample main program that employs this class is given in Fig. 3.7, which contains sample outputs as comments. This program uses the default constructor as well as all three programs in the public class functionality. Note that the definition of the class was copied in via an "include" (line 1) statement and activated with the "use" statement (line 4).

Now we will employ the class_Date within a class_Person which will use it to set the date of birth (DOB) and date of death (DOD) in addition to the other Person components of name, nationality, and sex. As shown in Fig. 3.8, we have made all the type components "private," but make all the supporting functionality public, as represented graphically in Fig. 3.8. The functionality shown provides a manual constructor, make_Person, routines to set the DOB or DOD, and those for the printing of most components. The source code for the new Person class is given in Fig. 3.9. Note that the manual constructor (line 12) utilizes "optional" arguments and initializes all components in case they are not supplied to the constructor. The Date_ public function from the class_Date is "inherited" to initialize the DOB and DOD (lines 18, 57, and 62). That function member from the previous module was activated with the combination of the "include" and "use" statements. Of course, the include could have been omitted if the compile statement included the path name to that source. A sample main program for testing the class_Person is in Fig. 3.10 along with comments containing its output. It utilizes the constructors Date_ (line 7), Person_ (line10), and make_Person (line 24).

Next, we want to use the previous two classes to define a class_Student which adds something else special to the general class_Person. The student person will have additional "private" components for an identification number, the expected date of matriculation (DOM), the total course credit hours earned (credits), and the overall grade point average (GPA), as represented in Fig. 3.11. The source lines for the type definition and selected public functionality are given in Fig. 3.12. There the constructors are make_Student (line 19) and Student_ (line 47). A testing main program with sample output is illustrated in Fig. 3.13. Since there are various ways to utilize the various constructors three alternate methods have been included as comments to indicate some of the programmers options. The first two include statements (lines 1, 2) are actually redundant because the third include automatically brings in those first two classes.


```
module class_Date        ! filename: class_Date.f90
implicit none
 public  :: Date ! and everything not "private"

 type Date
   private
   integer :: month, day, year ; end type Date

contains ! encapsulated functionality

 function Date_ (m, d, y) result (x) ! public constructor
   integer, intent(in) :: m, d, y    ! month, day, year
   type (Date)         :: x          ! from intrinsic constructor
     if ( m < 1 .or. d < 1 ) stop 'Invalid components, Date_'
       x = Date (m, d, y) ; end function Date_

subroutine  print_Date (x)    ! check and pretty print a date
  type (Date), intent(in)     :: x
  character (len=*),parameter :: month_Name(12) = &
    (/ "January  ", "February ", "March    ", "April    ",&
       "May      ", "June     ", "July     ", "August   ",&
       "September", "October  ", "November ", "December "/)
    if ( x%month < 1 .or. x%month > 12 ) print *, "Invalid month"
    if ( x%day   < 1 .or. x%day   > 31 ) print *, "Invalid day  "
    print *, trim(month_Name(x%month)),' ', x%day, ", ", x%year;
end subroutine  print_Date

 subroutine  read_Date (x)       ! read month, day, and year
   type (Date), intent(out) :: x  ! into intrinsic constructor
     read *, x ; end subroutine  read_Date

 function set_Date (m, d, y) result (x)    ! manual constructor
   integer, optional, intent(in) :: m, d, y ! month, day, year
   type (Date)                   :: x
     x = Date (1,1,1997)           ! default, (or use current date)
     if ( present(m) ) x%month = m ; if ( present(d) ) x%day   = d
     if ( present(y) ) x%year  = y ; end function set_Date

end module class_Date
```

Figure 3.6: Defining a Date Class

```
include 'class_Date.f90'   ! see previous figure
program  main
 use class_Date
 implicit none
   type (Date) :: today, peace

   ! peace = Date (11,11,1918) ! NOT allowed for private components
     peace = Date_ (11,11,1918)             ! public constructor
     print *, "World War I ended on "  ; call print_Date (peace)
     peace = set_Date (8, 14, 1945)         ! optional constructor
     print *, "World War II ended on " ; call print_Date (peace)
     print *, "Enter today as integer month, day, and year: "
     call read_Date(today)                  ! create today's date

     print *, "The date is ";  call print_Date (today)
end program  main                       ! Running produces:
! World War I ended on November 11, 1918
! World War II ended on August 14, 1945
! Enter today as integer month, day, and year: 7 10 1997
! The date is July 10, 1997
```

Figure 3.7: Testing a Date Class

3.3 Object Oriented Numerical Calculations

OOP is often used for numerical computation, especially when the standard storage mode for arrays is not practical or efficient. Often one will find specialized storage modes like linked lists, or tree structures used for dynamic data structures. Here we should note that many matrix operators are intrinsic to F90, so one is more likely to define a `class_sparse_matrix` than a `class_matrix`. However, either class would allow us to encapsulate several matrix functions and subroutines into a module that could be reused easily in other software. Here, we will illustrate OOP applied to rational numbers and introduce

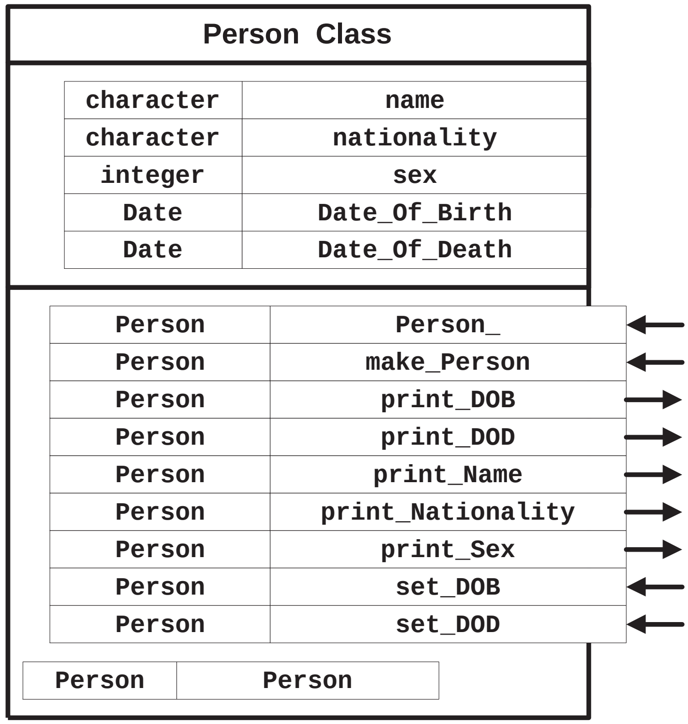


Figure 3.8: Graphical Representation of a Person Class

the important topic of operator overloading. Additional numerical applications of OOP will be illustrated in later chapters.

3.3.1 A Rational Number Class and Operator Overloading

To illustrate an OOP approach to simple numerical operations we will introduce a fairly complete rational number class, called `class_Rational` which is represented graphically in Fig. 3.14. The defining F90 module is given in Fig. 3.15. The type components have been made private (line 5), but not the type itself, so we can illustrate the intrinsic constructor (lines 38 and 102), but extra functionality has been provided to allow users to get either of the two components (lines 52 and 57). The provided routines shown in that figure are:

add_Rational	convert	copy_Rational	delete_Rational
equal_integer	gcd	get_Denominator	get_Numerator
invert	is_equal_to	list	make_Rational
mult_Rational	Rational_	reduce	

Procedures with only one return argument are usually implemented as functions instead of subroutines.

Note that we would form a new rational number, z, as the product of two other rational numbers, x and y, by invoking the `mult_Rational` function (line 90),

```
z = mult_Rational (x, y)
```

which returns z as its result. A natural tendency at this point would be to simply write this as $z = x * y$. However, before we could do that we would have to have to tell the operator, "*", how to act when provided with this new data type. This is known as *overloading* an intrinsic operator. We had the foresight to do this when we set up the module by declaring which of the "module procedures" were equivalent to this operator symbol. Thus, from the "interface operator (*)" statement block (line 14) the system now knows that the left and right operands of the "*" symbol correspond to the first and second arguments in the function `mult_Rational`. Here it is not necessary to overload the assignment operator, "=", when both of its operands are of the same intrinsic or defined type. However, to convert

```
module class_Person         ! filename: class_Person.f90
use class_Date
implicit none
  public  :: Person
    type Person
      private
      character (len=20) :: name
      character (len=20) :: nationality
      integer            :: sex
      type (Date)        :: dob, dod    ! birth, death
    end type Person
contains
  function make_Person (nam, nation, s, b, d) result (who)
  !        Optional Constructor for a Person type
    character (len=*), optional, intent(in) :: nam, nation
    integer,           optional, intent(in) :: s     ! sex
    type (Date),       optional, intent(in) :: b, d  ! birth, death
    type (Person)                           :: who
      who = Person (" ","USA",1,Date_(1,1,0),Date_(1,1,0)) ! defaults
      if ( present(nam)    ) who % name        = nam
      if ( present(nation) ) who % nationality = nation
      if ( present(s)      ) who % sex         = s
      if ( present(b)      ) who % dob         = b
      if ( present(d)      ) who % dod         = d ; end function

  function Person_ (nam, nation, s, b, d) result (who)
  !        Public Constructor for a Person type
    character (len=*), intent(in) :: nam, nation
    integer,           intent(in) :: s     ! sex
    type (Date),       intent(in) :: b, d  ! birth, death
    type (Person)                 :: who
      who = Person (nam, nation, s, b, d) ; end function Person_

  subroutine print_DOB (who)
    type (Person), intent(in) :: who
      call  print_Date (who % dob) ; end subroutine  print_DOB

  subroutine print_DOD (who)
    type (Person), intent(in) :: who
      call print_Date (who % dod) ; end subroutine  print_DOD

  subroutine print_Name (who)
    type (Person), intent(in) :: who
      print *, who % name ; end subroutine print_Name

  subroutine print_Nationality (who)
    type (Person), intent(in) :: who
      print *, who % nationality ; end subroutine  print_Nationality

  subroutine print_Sex (who)
    type (Person), intent(in) :: who
      if ( who % sex == 1 ) then ; print *, "male"
      else ; print *, "female" ; end if ; end subroutine print_Sex

  subroutine set_DOB (who, m, d, y)
    type (Person), intent(inout) :: who
    integer, intent(in)          :: m, d, y ! month, day, year
      who % dob = Date_ (m, d, y) ;  end subroutine set_DOB

  subroutine set_DOD(who, m, d, y)
    type (Person), intent(inout) :: who
    integer, intent(in)          :: m, d, y ! month, day, year
      who % dod = Date_ (m, d, y) ;  end subroutine set_DOD
end module class_Person
```

Figure 3.9: Definition of a Typical Person Class

an integer to a rational we could, and have, defined an overloaded assignment operator procedure (line 10). Here we have provided the procedure, `equal_Integer`, which is automatically invoked when we write: `type(Rational)y; y = 4`. That would be simpler than invoking the constructor called `make_rational`. Before moving on note that the system does not yet know how to multiply an integer times a rational number, or visa versa. To do that one would have to add more functionality, such as a function, say `int_mult_rn`, and add it to the "module procedure" list associated with the "*" operator. A typical main program which exercises most of the rational number functionality is given in Fig. 3.16, along with typical numerical output. It tests the constructors `Rational_` (line 8), `make_Rational`

```
include 'class_Date.f90'
include 'class_Person.f90'                    ! see previous figure
program main
 use class_Date ; use class_Person           ! inherit class members
 implicit none
   type (Person) :: author, creator
   type (Date)   :: b, d                      ! birth, death
     b = Date_(4,13,1743) ; d = Date_(7, 4,1826) ! OPTIONAL
   !                       Method 1
   ! author =  Person ("Thomas Jefferson", "USA", 1, b, d) ! NOT if private
     author = Person_ ("Thomas Jefferson", "USA", 1, b, d) ! constructor
     print *, "The author of the Declaration of Independence was ";
     call  print_Name (author);
     print *, ". He was born on "; call print_DOB (author);
     print *, " and died on ";     call print_DOD (author); print *, ".";
   !                       Method 2
     author = make_Person ("Thomas Jefferson", "USA") ! alternate
     call  set_DOB (author, 4, 13, 1743)              ! add DOB
     call  set_DOD (author, 7,  4, 1826)              ! add DOD
     print *, "The author of the Declaration of Independence was ";
     call  print_Name (author)
     print *, ". He was born on "; call print_DOB (author);
     print *, " and died on ";     call print_DOD (author); print *, ".";
   !                       Another Person
     creator = make_Person ("John Backus", "USA")     ! alternate
     print *, "The creator of Fortran was "; call print_Name (creator);
     print *, " who was born in ";    call print_Nationality (creator);
     print *, ".";
end program main                                      ! Running gives:
! The author of the Declaration of Independence was Thomas Jefferson.
! He was born on April 13, 1743 and died on July 4, 1826.
! The author of the Declaration of Independence was Thomas Jefferson.
! He was born on April 13, 1743 and died on July 4, 1826.
! The creator of Fortran was John Backus who was born in the USA.
```

Figure 3.10: Testing the Date and Person Classes

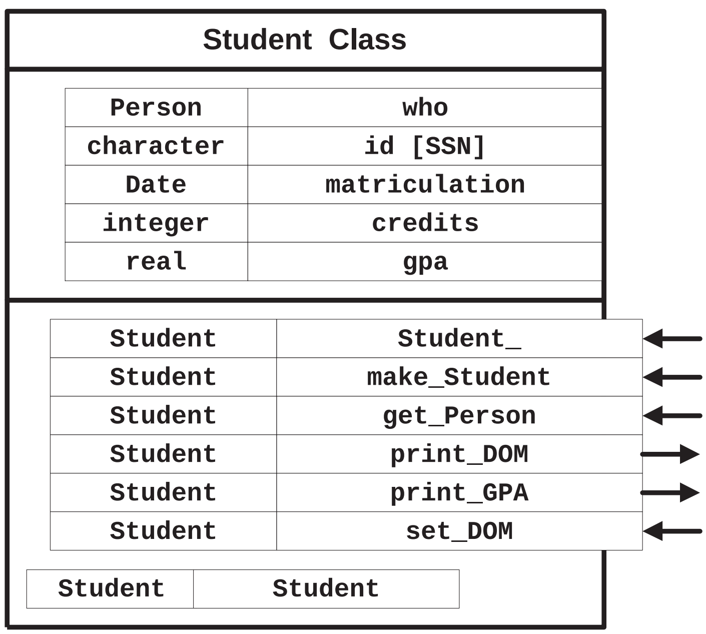


Figure 3.11: Graphical Representation of a Student Class

(lines 14, 18, 25), and a simple destructor `delete_Rational` (line 38). The intrinsic constructor (line 6) could have been used only if all the attributes were public, and that is considered an undesirable practice in OOP. The simple destructor actually just sets the “deleted” number to have a set of default components. Later we will see that constructors and destructors often must dynamically `allocate` and `deallocate`, respectively, memory associated with a specific instance of some object.

```
module class_Student             ! filename class_Student.f90
use class_Person                 ! inherits class_Date
implicit none
 public :: Student, set_DOM, print_DOM
   type Student
     private
     type (Person)     :: who    ! name and sex
     character (len=9) :: id     ! ssn digits
     type (Date)       :: dom    ! matriculation
     integer           :: credits
     real              :: gpa    ! grade point average
   end type Student
contains  ! coupled functionality

  function get_person (s) result (p)
    type (Student), intent(in) :: s
    type (Person)              :: p      ! name and sex
      p = s % who ; end function get_person

 function make_Student (w, n, d, c, g) result (x) ! constructor
 !        Optional Constructor for a Student type
   type (Person),                intent(in) :: w ! who
   character (len=*), optional, intent(in) :: n ! ssn
   type (Date),       optional, intent(in) :: d ! matriculation
   integer,           optional, intent(in) :: c ! credits
   real,              optional, intent(in) :: g ! grade point ave
   type (Student)                           :: x ! new student
     x = Student_(w, " ", Date_(1,1,1), 0, 0.)   ! defaults
     if ( present(n) ) x % id      = n           ! optional values
     if ( present(d) ) x % dom     = d
     if ( present(c) ) x % credits = c
     if ( present(g) ) x % gpa     = g ; end function make_Student

  subroutine print_DOM (who)
    type (Student), intent(in) :: who
      call print_Date(who%dom) ; end subroutine print_DOM

  subroutine print_GPA (x)
    type (Student), intent(in) :: x
      print *, "My name is "; call print_Name (x % who)
      print *, ", and my G.P.A. is ", x % gpa, "." ; end subroutine

  subroutine set_DOM (who, m, d, y)
    type (Student), intent(inout) :: who
    integer,        intent(in)    :: m, d, y
      who % dom = Date_( m, d, y) ; end subroutine set_DOM

  function Student_ (w, n, d, c, g) result (x)
  !         Public Constructor for a Student type
    type (Person),     intent(in) :: w ! who
    character (len=*), intent(in) :: n ! ssn
    type (Date),       intent(in) :: d ! matriculation
    integer,           intent(in) :: c ! credits
    real,              intent(in) :: g ! grade point ave
    type (Student)                :: x ! new student
      x = Student (w, n, d, c, g) ; end function Student_
end module class_Student
```

Figure 3.12: Defining a Typical Student Class

When considering which operators to overload for a newly defined object one should consider those that are used in `sorting` operations, such as the greater-than, $>$, and less-than, $<$, operators. They are often useful because of the need to sort various types of objects. If those symbols have been correctly overloaded then a generic object sorting routine might be used, or require few changes.

3.4 Discussion

The previous sections have only briefly touched on some important OOP concepts. More details will be covered later after a general overview of the features of the Fortran language. There are more than one hundred OOP languages. Persons involved in software development need to be aware that F90 can meet almost all of their needs for a OOP language. At the same time it includes the F77 standard as a subset and thus allows efficient use of the many millions of Fortran functions and subroutines developed in the past. The newer F95 standard is designed to make efficient use of super computers and massively parallel

```
include 'class_Date.f90'
include 'class_Person.f90'
include 'class_Student.f90' ! see previous figure
program main                ! create or correct a student
 use class_Student          ! inherits class_Person, class_Date also
 implicit none
  type (Person) :: p  ; type (Student) :: x
!               Method 1
    p = make_Person ("Ann Jones","",0)  ! optional person constructor
    call set_DOB (p, 5, 13, 1977)       ! add birth to person data
    x = Student_(p, "219360061", Date_(8,29,1955), 9, 3.1) ! public
    call print_Name (p)                                ! list name
    print *, "Born        :"; call print_DOB (p)      ! list dob
    print *, "Sex         :"; call print_Sex (p)      ! list sex
    print *, "Matriculated:"; call print_DOM (x)      ! list dom
    call print_GPA (x)                                ! list gpa
 !               Method 2
    x = make_Student (p, "219360061")    ! optional student constructor
    call set_DOM (x, 8, 29, 1995)        ! correct matriculation
    call print_Name (p)                                ! list name
    print *, "was born on :"; call print_DOB (p)      ! list dob
    print *, "Matriculated:"; call print_DOM (x)      ! list dom
 !               Method 3
    x = make_Student (make_Person("Ann Jones"), "219360061") ! optional
    p = get_Person (x)                   ! get defaulted person data
    call set_DOM (x, 8, 29, 1995)        ! add matriculation
    call set_DOB (p, 5, 13, 1977)        ! add birth
    call print_Name (p)                                ! list name
    print *, "Matriculated:"; call print_DOM (x)      ! list dom
    print *, "was born on :"; call print_DOB (p)      ! list dob
end program main                                     ! Running gives:
! Ann Jones
! Born        : May 13, 1977
! Sex         : female
! Matriculated: August 29, 1955
! My name is Ann Jones, and my G.P.A. is 3.0999999.
! Ann Jones was born on: May 13, 1977 , Matriculated: August 29, 1995
! Ann Jones Matriculated: August 29, 1995 , was born on: May 13, 1977
```

Figure 3.13: Testing the Student, Person, and Date Classes

machines. It includes most of the High Performance Fortran features that are in wide use. Thus, efficient use of OOP on parallel machines is available through F90 and F95.

None of the OOP languages have all the features one might desire. For example, the useful concept of a "template" which is standard in C++ is not in the F90 standard. Yet the author has found that a few dozen lines of F90 code will define a preprocessor that allows templates to be defined in F90 and expanded in line at compile time. The real challenge in OOP is the actual OOA and OOD that must be completed before programming can begin, regardless of the language employed. For example, several authors have described widely different approaches for defining classes to be used in constructing OO finite element systems. Additional example applications of OOP in F90 will be given in the following chapters.

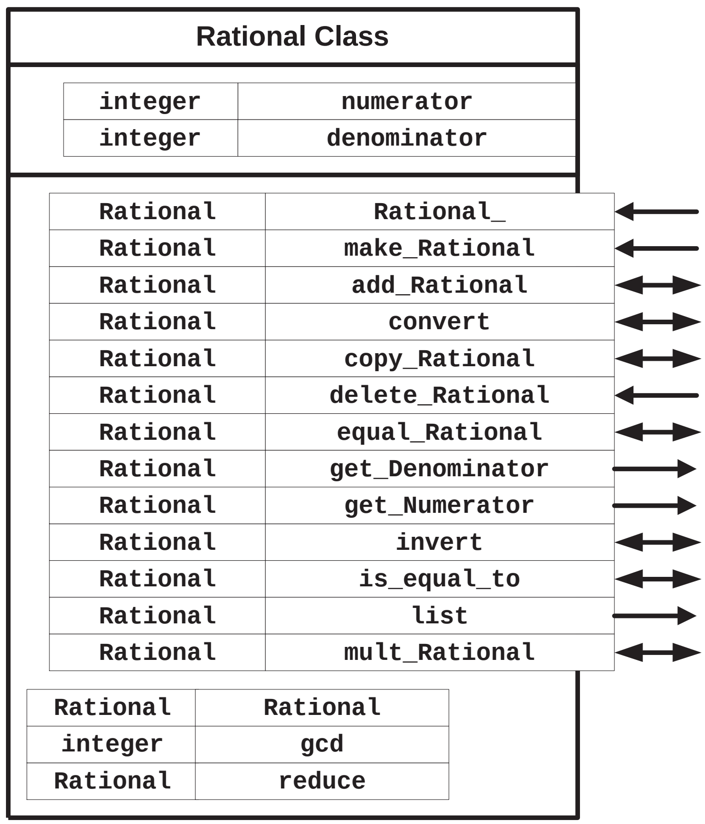


Figure 3.14: Representation of a Rational Number Class

```
module class_Rational                ! filename: class_Rational.f90
implicit none
 ! public, everything but following private routines
 private :: gcd, reduce
   type Rational
     private ! numerator and denominator
     integer :: num, den ; end type Rational

          ! overloaded operators interfaces
   interface assignment (=)
     module procedure equal_Integer ; end interface
   interface operator (+)          ! add unary versions & (-) later
     module procedure add_Rational ; end interface
   interface operator (*)          ! add integer_mult_Rational, etc
     module procedure mult_Rational ; end interface
   interface operator (==)
     module procedure is_equal_to ; end interface
contains                      ! inherited operational functionality
 function add_Rational (a, b) result (c)       ! to overload +
   type (Rational), intent(in) :: a, b          ! left + right
   type (Rational)             :: c
     c % num = a % num*b % den + a % den*b % num
     c % den = a % den*b % den
     call reduce (c) ; end function add_Rational

 function convert (name) result (value) ! rational to real
   type (Rational), intent(in) :: name
   real                        :: value  ! decimal form
     value = float(name % num)/name % den ; end function convert

 function copy_Rational (name) result (new)
   type (Rational), intent(in) :: name
   type (Rational)             :: new
     new % num = name % num
     new % den = name % den ; end function copy_Rational

 subroutine delete_Rational (name)     ! deallocate allocated items
   type (Rational), intent(inout) :: name      ! simply zero it here
     name = Rational (0, 1) ; end subroutine delete_Rational

 subroutine equal_Integer (new, I) ! overload =, with integer
   type (Rational), intent(out) :: new  ! left  side of operator
   integer,         intent(in)  :: I    ! right side of operator
     new % num = I ; new % den = 1 ; end subroutine equal_Integer

 recursive function gcd (j, k) result (g) ! Greatest Common Divisor
   integer, intent(in) :: j, k ! numerator, denominator
   integer             :: g
     if ( k == 0 ) then ; g = j
     else ; g = gcd ( k, modulo(j,k) )           ! recursive call
     end if ; end function gcd

 function  get_Denominator (name) result (n)   ! an access function
   type (Rational), intent(in) :: name
   integer                     :: n             ! denominator
     n = name % den ; end function  get_Denominator
```

(Fig. 3.15, A Fairly Complete Rational Number Class (continued))

```
  function  get_Numerator (name) result (n)     ! an access function
    type (Rational), intent(in) :: name
    integer                     :: n             ! numerator
      n = name % num ; end function  get_Numerator

  subroutine  invert (name)         ! rational to rational inversion
    type (Rational), intent(inout) :: name
    integer                        :: temp
      temp       = name % num
      name % num = name % den
      name % den = temp ; end subroutine invert

   function is_equal_to (a_given, b_given) result (t_f)
    type (Rational), intent(in) :: a_given, b_given  ! left == right
    type (Rational)             :: a, b              ! reduced copies
    logical                     :: t_f
      a = copy_Rational (a_given) ; b = copy_Rational (b_given)
      call reduce(a) ; call reduce(b)        ! reduced to lowest terms
      t_f = (a%num == b%num) .and. (a%den == b%den) ; end function

   subroutine list(name)                 ! as a pretty print fraction
    type (Rational), intent(in) :: name
      print *, name % num, "/", name % den ; end subroutine list

   function make_Rational (numerator, denominator) result (name)
   !        Optional Constructor for a rational type
    integer, optional, intent(in) :: numerator, denominator
    type (Rational)               :: name
      name = Rational(0, 1)                             ! set defaults
      if ( present(numerator)  ) name % num = numerator
      if ( present(denominator)) name % den = denominator
      if ( name % den == 0     ) name % den = 1        ! now simplify
      call reduce (name) ; end function make_Rational

  function  mult_Rational (a, b) result (c)          ! to overload *
    type (Rational), intent(in) :: a, b
    type (Rational)             :: c
      c % num = a % num * b % num
      c % den = a % den * b % den
      call reduce (c) ; end function mult_Rational

  function Rational_ (numerator, denominator) result (name)
  !        Public Constructor for a rational type
    integer, optional, intent(in) :: numerator, denominator
    type (Rational)               :: name
      if ( denominator == 0 ) then ; name = Rational (numerator, 1)
      else ; name = Rational (numerator, denominator) ; end if
  end function Rational_

  subroutine reduce (name)               ! to simplest rational form
    type (Rational), intent(inout) :: name
    integer                        :: g     ! greatest common divisor
      g          = gcd (name % num, name % den)
      name % num = name % num/g
      name % den = name % den/g ; end subroutine reduce
end module class_Rational
```

Figure 3.15: A Fairly Complete Rational Number Class

```
include 'class_Rational.f90'
program main
use class_Rational
implicit none
  type (Rational) :: x, y, z
  ! ------- only if Rational is NOT private ----------
  ! x = Rational(22,7)    ! intrinsic constructor if public components

    x = Rational_(22,7)   ! public constructor if private components
    write (*,'("public    x  = ")',advance='no'); call list(x)
    write (*,'("converted x  = ", g9.4)') convert(x)
    call invert(x)
    write (*,'("inverted 1/x = ")',advance='no');  call list(x)

    x = make_Rational ()               ! default constructor
    write (*,'("made null x  = ")',advance='no'); call list(x)
    y = 4                              ! rational = integer overload
    write (*,'("integer y    = ")',advance='no'); call list(y)
    z = make_Rational (22,7)           ! manual constructor
    write (*,'("made full z  = ")',advance='no'); call list(z)
  !                Test Accessors
    write (*,'("top of z     = ", g4.0)') get_numerator(z)
    write (*,'("bottom of z  = ", g4.0)') get_denominator(z)
  !                Misc. Function Tests
    write (*,'("making x = 100/360, ")',advance='no')
    x = make_Rational (100,360)
    write (*,'("reduced x = ")',advance='no'); call list(x)
    write (*,'("copying x to y gives ")',advance='no')
    y = copy_Rational (x)
    write (*,'("a new y = ")',advance='no'); call list(y)
  !               Test Overloaded Operators
    write (*,'("z * x gives ")',advance='no'); call list(z*x) ! times
    write (*,'("z + x gives ")',advance='no'); call list(z+x) ! add
    y = z                                     ! overloaded assignment
    write (*,'("y = z gives y as ")',advance='no'); call list(y)
    write (*,'("logic y == x gives ")',advance='no'); print *, y==x
    write (*,'("logic y == z gives ")',advance='no'); print *, y==z
  !                Destruct
    call delete_Rational (y)    ! actually only null it here
    write (*,'("deleting y gives y = ")',advance='no'); call list(y)
end program main                                   ! Running gives:
!  public    x  =  22 / 7       !  converted x  = 3.143
!  inverted 1/x =  7 / 22       !  made null x  =  0 / 1
!  integer y    =  4 / 1        !  made full z  =  22 / 7
!  top of z     =   22          !  bottom of z  =    7
!  making x = 100/360, reduced x =  5 / 18
!  copying x to y gives a new y =  5 / 18
!  z * x gives  55 / 63         !  z + x gives  431 / 126
!  y = z gives y as  22 / 7     !  logic y == x gives  F
!  logic y == z gives  T        !  deleting y gives y =  0 / 1
```

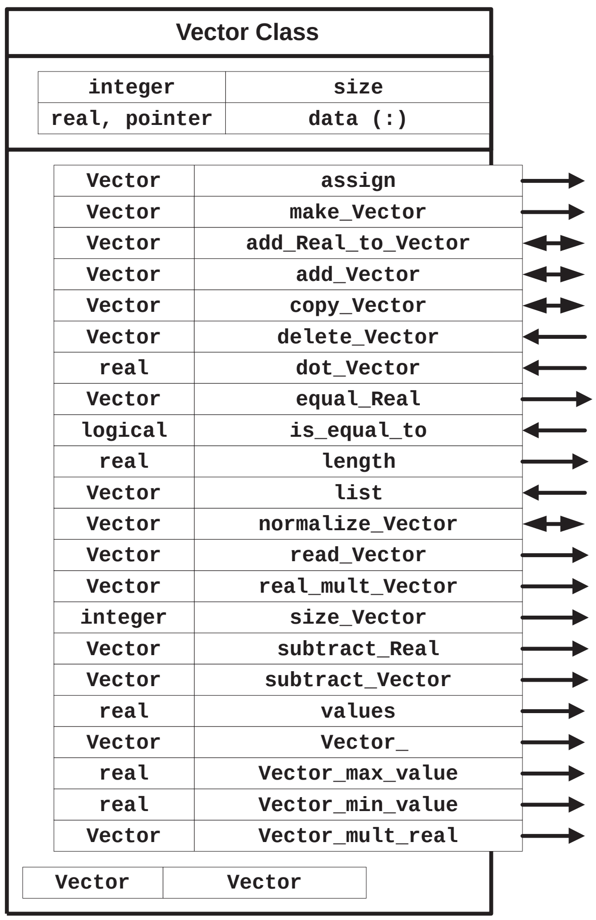
Figure 3.16: Testing the Rational Number Class

3.5 Exercises

1. Use the `class_Circle` to create a `class_Sphere` that computes the volume of a sphere. Have a method that accepts an argument of a `Circle`. Use the `radius` of the `Circle` via a new member `get_Circle_radius` to be added to the `class_Circle`.

2. Use the `class_Circle` and `class_Rectangle` to create a `class_Cylinder` that computes the volume of a right circular cylinder. Have a method that accepts arguments of a `Circle` and a `height`, and a second method that accepts arguments of a `Rectangle` and a `radius`. In the latter member use the `height` of the `Rectangle` via a new member `get_Rectangle_height` to be added to the `class_Rectangle`.

3. Create a vector class to treat vectors with an arbitrary number of real coefficients. Assume that the `class_Vector` is defined as follows:



Overload the common operators of (+) with `add_Vector` and `add_Real_to_Vector`, (–) with `subtract_Vector` and `subtract_Real`, (*) with `dot_Vector`, `real_mult_Vector` and `Vector_mult_real`, (=) with `equal_Real` to set all coefficients to a single real number, and (==) with routine `is_equal_to`.

Include two constructors *assign* and `make_Vector`. Let *assign* convert a real array into an instance of a Vector. Provide a destructor, means to read and write a Vector, normalize a Vector, and determine its extreme values.

4. Modify the above Vector class to extend it to a `Sparse_Vector_Class` where the vast majority of the coefficients are zero. Store and operate only on the non-zero entries.

Sparse_Vector Class

integer	non_zeros
integer, pointer	rows (:)
real, pointer	values (:)

Sparse_Vector	make_Sparse_Vector
Sparse_Vector	add_Real_to_Sparse_Vector
Sparse_Vector	add_Sparse_Vector
Sparse_Vector	delete_Sparse_Vector
real	dot_Vector
Sparse_Vector	el_by_el_Mult
Sparse_Vector	equal_Vector
real	get_element
logical	is_equal_to
integer	largest_index
real	length
real	norm
Sparse_Vector	normalize_Vector
Sparse_Vector	pretty
Sparse_Vector	read_Vector
Sparse_Vector	real_mult_Sparse
integer	rows_of
Sparse_Vector	set_element
Sparse_Vector	show
Sparse_Vector	show_r_v
integer	size_of
Sparse_Vector	Sparse_mult_real
Sparse_vector	sub_Sparse_Vector
Sparse_Vector	sum_Sparse_Vector
real	Vector_max_value
real	Vector_min_value
Sparse_Vector	Vector_to_Sparse
Sparse_Vector	zero_Sparse

Sparse_Vector	Sparse_Vector