Chapter 2

Data Types

Any computer program is going to have to operate on the available data. The valid data types that are available will vary from one language to another. Here we will examine the intrinsic or built-in data types, user-defined data types or structures and, finally, introduce the concept of the abstract data type which is the basic foundation of object-oriented methods. We will also consider the precision associated with numerical data types. The Fortran data types are listed in Table 2–1. Such data can be used as constants, variables, pointers and targets.

Table 2–1. F90/95 Data Types and Pointer Attributes

```
                               Data Option
                          _____________________
                          |                    |
                      Intrinsic                Derived
                          |          [Components of intrinsic type and/or
                          |          previously declared derived types.]
                          |
 _________________________
 |             |          |
Character    Logical    Numeric
                           |
              ___________________
              |                  |
       Floating Point         Integer
              |            (Default Precision)
              |             Selected-Int-Kind
              |
 _______________  _  _  _  _  _  _
 |                    |                  |
Complex              Real               Double Precision
(Default Precision)  (Default Precision)     [Obsolete]
Selected-Real-Kind's  Selected-Real-Kind
```

2.1 Intrinsic Types

The simplest data type is the `LOGICAL` type which has the Boolean values of either `.true.` or `.false.` and is used for relational operations. The other non-numeric data type is the `CHARACTER`. The sets of valid character values will be defined by the hardware system on which the compiler is installed. Character sets may be available in multiple languages, such as English and Japanese. There are international standards for computer character sets. The two most common ones are the English character sets defined in the ASCII and EBCDIC standards that have been adapted by the International Standards Organization (ISO). Both of these standards for defining single characters include the digits (0 to 9), the 26 upper case letters (A to Z), the 26 lower case letters (a to z), common mathematical symbols, and many non-printable codes known as control characters. We will see later that strings of characters are still referred to as being of the `CHARACTER` type, but they have a length that is greater than one. In other languages such a data type is often called a *string*. [While not part of the F95 standard, the ISO Committee created a user-defined type known as the `ISO_VARIABLE_LENGTH_STRING` which is available as a F95 source module.]

For numerical computations, numbers are represented as integers or decimal values known as *floating point numbers* or *floats*. The former is called an `INTEGER` type. The decimal values supported in Fortran are the `REAL` and `COMPLEX` types. The range and precision of these three types depends on the hardware being employed. At the present, 1999, most computers have 32 bit processors, but some offer 64 bit processors. This means that the precision of a calculated result from a single program could vary from one brand of computer to another. One would like to have a portable precision control so as to get the same answer from different hardware; whereas some languages, like C++, specify three ranges of precision (with specific bit widths). Fortran provides default precision types as well as two functions to allow the user to define the "kind" of precision desired.

Table 2–2. Numeric Types on 32 Bit Processors

Type	*Bit Width*	*Significant Digits*	*Common Range*
integer	16	10	–32,768 to 32,767
real	32	6	-10^{37} to 10^{37}
double precision†	64	15	-10^{307} to 10^{307}
complex	2@32	2@6	two reals

†obsolete in F90, see `selected_real_kind`

Still, it is good programming practice to employ a precision that is of the default, double, or quad precision level. Table 2–2 lists the default precisions for 32 bit processors. The first three entries correspond to types *int*, *float*, and *double*, respectively, of C++. Examples of F90 integer constants are

```
–32      0      4675123      24_short      24_long
```

while typical real constant examples are

```
 –3.      0.123456      1.234567e+2      0.0      0.3_double
7.6543e+4_double       0.23567_quad    0.3d0
```

In both cases, we note that it is possible to impose a user-defined precision kind by appending an underscore (`_`) followed by the name of the integer variable that gives the precision kind number. For example, one could define

```
long = selected_int_kind(9)
```

to denote an integer in the range of -10^9 to 10^9, while

```
double = selected_real_kind(15,307)
```

defines a real with 15 significant digits with an exponent range of ± 307. Likewise, a higher precision real might be defined by the integer kind

```
quad = selected_real_kind(18,4932)
```

to denote 18 significant digits over the exponent range of ± 4932. If these kinds of precision are available on your processors, then the F90 types of "integer (long)," "real (double)," and "real (quad)" would correspond to the C++ precision types of "long int," "double," and "long double," respectively. If the processor cannot produce the requested precision, then it returns a negative number as the integer kind number. Thus, one should always check that the kind (i.e., the above integer values of long, double, or quad) is not negative, and report an exception if it is negative.

The old F77 intrinsic type of `DOUBLE PRECISION` has been declared obsolete, since it is now easy to set any level of precision available on a processor. Another way to always define a double precision real on any processor is to use the "kind" function such as

```
double = kind(1.0d0)
```

where the symbol '`d`' is used to denote the I/O of a double precision real. For completeness it should be noted that it is possible on some processors to define different kinds of character types, such as "greek" or "ascii", but in that case, the kind value comes before the underscore and the character string such as: `ascii_`"a string".

```
[ 1]  Module Math_Constants     ! Define double precision math constants
[ 2]   implicit none
[ 3]   ! INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND (15,307)
[ 4]     INTEGER, PARAMETER :: DP = KIND (1.d0) ! Alternate form
[ 5]  real(DP), parameter:: Deg_Per_Rad  = 57.295779513082320876798155_DP
[ 6]  real(DP), parameter:: Rad_Per_Deg  = 0.017453292519943295769237_DP
[ 7]
[ 8]  real(DP), parameter:: e_Value      =  2.71828182845904523560287_DP
[ 9]  real(DP), parameter:: e_Recip      =  0.367879441171442321595523_DP
[10]  real(DP), parameter:: e_Squared    =  7.389056098930650227230427_DP
[11]  real(DP), parameter:: Log10_of_e  =  0.434294481903251827651128_DP
[12]
[13]  real(DP), parameter:: Euler         =  0.5772156649015328606_DP
[14]  real(DP), parameter:: Euler_Log    = -0.5495393129816448223_DP
[15]  real(DP), parameter:: Gamma         =  0.577215664901532860606512_DP
[16]  real(DP), parameter:: Gamma_Log    = -0.549539312981644822337662_DP
[17]  real(DP), parameter:: Golden_Ratio =  1.618033988749894848_DP
[18]
[19]  real(DP), parameter:: Ln_2         =  0.693147180559945309417232l_DP
[20]  real(DP), parameter:: Ln_10        =  2.302585092994045684017991_5_DP
[21]  real(DP), parameter:: Log10_of_2  =  0.301029995663981195213738 9_DP
[22]
[23]  real(DP), parameter:: pi_Value     =  3.141592653589793238462643_DP
[24]  real(DP), parameter:: pi_Ln        =  1.144729885849400174143427_DP
[25]  real(DP), parameter:: pi_Log10     =  0.497149872694133854351268 3_DP
[26]  real(DP), parameter:: pi_Over_2   =  1.570796326794896619231322_DP
[27]  real(DP), parameter:: pi_Over_3   =  1.047197551196597746154214_DP
[28]  real(DP), parameter:: pi_Over_4   =  0.785398163397448309615660 8_DP
[29]  real(DP), parameter:: pi_Recip     =  0.318309886183790671537767 5_DP
[30]  real(DP), parameter:: pi_Squared   =  9.869604401089358618834491_DP
[31]  real(DP), parameter:: pi_Sq_Root  =  1.772453850905516027298167_DP
[32]
[33]  real(DP), parameter:: Sq_Root_of_2 =  1.4142135623730950488_DP
[34]  real(DP), parameter:: Sq_Root_of_3 =  1.7320508075688772935_DP
[35]
[36]  End Module Math_Constants
[37]
[38]  Program Test
[39]    use Math_Constants                  ! Access all constants
[40]    real :: pi                           ! Define local data type
[41]    print *, 'pi_Value: ', pi_Value      ! Display a constant
[42]    pi = pi_Value; print *, 'pi = ', pi  ! Convert to lower precision
[43]  End Program Test                      ! Running gives:
[44]    ! pi_Value: 3.1415926535897931      ! pi = 3.14159274
```

Figure 2.1: Defining Global Double Precision Constants

To illustrate the concept of a defined precision intrinsic data type, consider a program segment to make available useful constants such as *pi* (3.1415. . .) or Avogadro's number $(6.02\ldots \times 10^{23})$. These are real constants that should not be changed during the use of the program. In F90, an item of that nature is known as a PARAMETER. In Fig. 2.1, a selected group of such constants have been declared to be of double precision and stored in a MODULE named Math_Constants. The parameters in that module can be made available to any program one writes by including the statement "use math_constants" at the beginning of the program segment. The figure actually ends with a short sample program that converts the tabulated value of *pi* (line 23) to a default precision real (line 42) and prints both.

2.2 User Defined Data Types

While the above intrinsic data types have been successfully employed to solve a vast number of programming requirements, it is logical to want to combine these types in some structured combination that represents the way we think of a particular physical object or business process. For example, assume we wish to think of a chemical element in terms of the combination of its standard symbol, atomic number and atomic mass. We could create such a data structure type and assign it a name, say chemical_element, so that we can refer to that type for other uses just like we might declare a real variable. In F90 we would define the structure with a TYPE construct as shown below (in lines 3–7):

```
[ 1]  program create_a_type
[ 2]  implicit none
[ 3]    type chemical_element              ! a user defined data type
[ 4]      character (len=2) :: symbol
[ 5]      integer           :: atomic_number
[ 6]      real              :: atomic_mass
[ 7]    end type
```

Having created the new data type, we would need ways to define its values and/or ways to refer to any of its components. The latter is accomplished by using the component selection symbol "%". Continuing the above program segment we could write:

```
[ 8]    type (chemical_element) :: argon, carbon, neon    ! elements
[ 9]    type (chemical_element) :: Periodic_Table(109)     ! an array
[10]    real                    :: mass                    ! a scalar
[11]
[12]    carbon%atomic_mass   = 12.010          ! set a component value
[13]    carbon%atomic_number = 6               ! set a component value
[14]    carbon%symbol        = "C"            ! set a component value
[15]
[16]    argon = chemical_element ("Ar", 18, 26.98) ! construct element
[17]
[18]    read *, neon                              ! get "Ne" 10 20.183
[19]
[20]    Periodic_Table( 5) = argon         ! insert element into array
[21]    Periodic_Table(17) = carbon        ! insert element into array
[22]    Periodic_Table(55) = neon          ! insert element into array
[23]
[24]    mass = Periodic_Table(5) % atomic_mass     ! extract component
[25]
[26]    print *, mass                     ! gives 26.9799995
[27]    print *, neon                     ! gives Ne 10 20.1830006
[28]    print *, Periodic_Table(17)       ! gives C  6  12.0100002
[29]  end program create_a_type
```

In the above program segment, we have introduced some new concepts:

- define argon, carbon and neon to be of the `chemical_element` type (line 7).
- define an array to contain 109 `chemical_element` types (line 8).
- used the selector symbol, %, to assign a value to each of the components of the carbon structure (line 15).
- used the intrinsic "structure constructor" to define the argon values (line 15). The intrinsic construct or initializer function must have the same name as the user-defined type. It must be supplied with all of the components, and they must occur in the order that they were defined in the `TYPE` block.
- read in all the neon components, in order (line 17). [The '*' means that the system is expected to automatically find the next character, integer and real, respectively, and to insert them into the components of `neon`.]
- inserted argon, carbon and neon into their specific locations in the periodic table array (lines 19–21).
- extracted the `atomic_mass` of argon from the corresponding element in the `periodic_element` array (line 23).
- print the real variable, `mass` (line 25). [The '*' means to use a default number of digits.]
- printed all components of neon (line 26). [Using a default number of digits.]
- printed all the components of carbon by citing its reference in the periodic table array (line 27). [Note that the printed real value differs from the value assigned in line 12. This is due to the way reals are represented in a computer, and will be considered elsewhere in the text.]

A defined type can also be used to define other data structures. This is but one small example of the concept of code re-use. If we were developing a code that involved the history of chemistry, we might use the above type to create a type called *history* as shown below.

```
type (chemical_element)   :: oxygen

type history                      ! a second type using the first
  character (len=31)       :: element_name
  integer                  :: year_found
  type (chemical_element)  :: chemistry
```

```
end type history
type (history) :: Joseph_Priestley                  ! Discoverer
oxygen = chemical_element ("O", 76, 190.2)   ! construct element
Joseph_Priestley = history ("Oxygen", 1774, oxygen)  ! construct
print *, Joseph_Priestley ! gives Oxygen 1774 O 76 1.9020000E+02
```

Shortly we will learn about other important aspects of user-defined types, such as how to define operators that use them as operands.

2.3 Abstract Data Types

Clearly, data alone is of little value. We must also have the means to input and output the data, subprograms to manipulate and query the data, and the ability to define operators for commonly used procedures. The coupling or encapsulation of the data with a select group of functions that defines everything that can be done with the data type introduces the concept of an abstract data type (ADT). An ADT goes a step further in that it usually hides from the user the details of how functions accomplish their tasks. Only knowledge of input and output interfaces to the functions are described in detail. Even the components of the data types are kept private.

The word *abstract* in the term *abstract data type* is used to: 1) indicate that we are interested only in the essential features of the data type, 2) to indicate that the features are defined in a manner that is independent of any specific programming language, and 3) to indicate that the instances of the ADT are being defined by their behavior, and that the actual implementation is secondary. An ADT is an abstraction that describes a set of items in terms of a hidden or encapsulated data structure and a set of operations on that data structure.

Previously we created user-defined entity types such as the `chemical_element`. The primary difference between entity types and ADTs is that all ADTs include methods for operating on the type. While entity types are defined by a name and a list of attributes, an ADT is described by its name, attributes, encapsulated methods, and possibly encapsulated rules.

Object-oriented programming is primarily a data abstraction technique. The purpose of abstraction and data hiding in programming is to separate behavior from implementation. For abstraction to work, the implementation must be encapsulated so that no other programming module can depend on its implementation details. Such encapsulation guarantees that modules can be implemented and revised independently. Hiding of the attributes and some or all of the methods of an ADT is also important in the process. In F90 the `PRIVATE` statement is used to hide an attribute or a method; otherwise, both will default to `PUBLIC`. Public methods can be used outside the program module that defines an ADT. We refer to the set of public methods or operations belonging to an ADT as the public interface of the type.

The user-defined data type, as given above, in F90 is not an ADT even though each is created with three intrinsic methods to construct a value, read a value, or print a value. Those methods cannot modify a type; they can only instantiate the type by assigning it a value and display that value. (Unlike F90, in C or C++ a user-defined type, or "struct", does not have an intrinsic constructor method, or input/output methods.) Generally ADTs will have methods that modify or query a type's state or behavior.

From the above discussion we see that the intrinsic data types in any language (such as complex, integer and real in F90) are actually ADTs. The system has hidden methods (operators) to assign them values and to manipulate them. For example, we know that we can multiply any one of the numerical types by any other numerical type.

We do not know how the system does the multiplication, and we don't care. All computer languages provide functions to manipulate the intrinsic data types. For example, in F90 a square root function, named *sqrt*, is provided to compute the square root of a real or complex number. From basic mathematics you probably know that two distinctly different algorithms must be used and the choice depends on the type of the supplied argument. Thus, we call the *sqrt* function a generic function since its single name, *sqrt*, is used to select related functions in a manner hidden from the user. In F90 you can not take the square root of an integer; you must convert it to a real value and you receive back a real answer. The

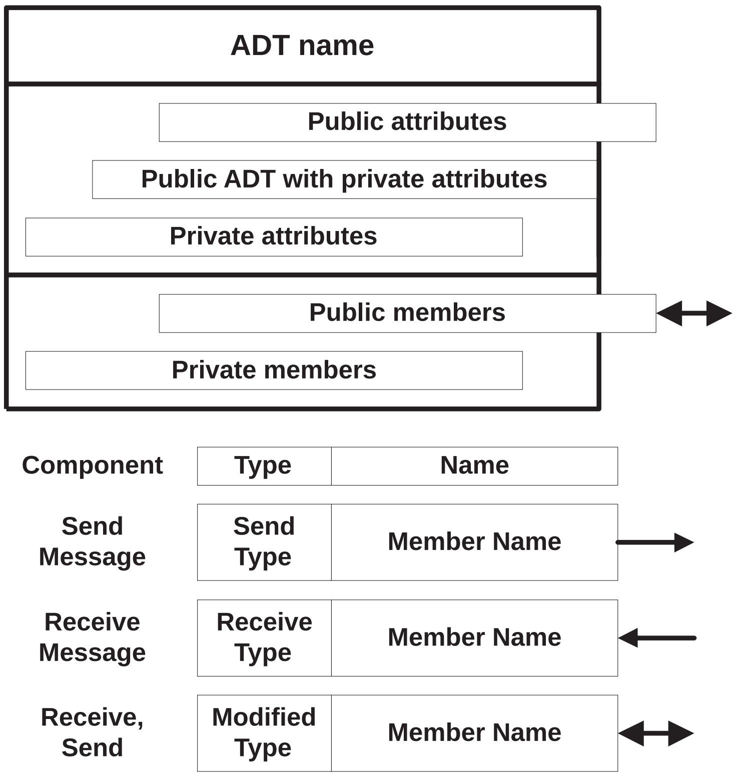

Figure 2.2: Graphical Representation of ADTs

above discussions of the methods (routines) that are coupled to a data type and describe what you can and can not do with the data type should give the programmer good insight into what must be done to plan and implement the functions needed to yield a relatively complete ADT.

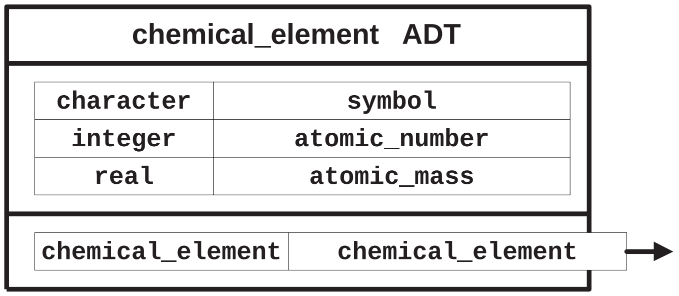

Figure 2.3: Representation of the Public Chemical_Element ADT

It is common to have a graphical representation of the ADTs and there are several different graphical formats suggested in the literature. We will use the form shown in Fig. 2.4 where a rectangular box begins with the ADT name and is followed by two partitions of that box that represent the lists of attribute data and associated member routines. Items that are available to the outside world are in sub-boxes that cross over the right border of the ADT box. They are the parts of the public interface to the ADT. Likewise those items that are strictly internal, or private, are contained fully within their respective partitions of the ADT box. There is a common special case where the name of the data type itself is available for external use, but its individual attribute components are not. In that case the right edge of the private attributes lists lie on the right edge of the ADT box. In addition, we will often segment the smallest box for an item to give its type (or the most important type for members) and the name of the item. Public member boxes are also supplemented with an arrow to indicate which take in information (<--), or send out information (-->). Such a graphical representation of the previous `chemical_element` ADT, with all its items public, is shown in Fig. 2.4.

The sequence of numbers known as Fibonacci numbers is the set that begins with one and two and where the next number in the set is the sum of the two previous numbers (1, 2, 3, 5, 8, 13, ...). A primarily private ADT to print a list of Fibonacci numbers up to some limit is represented graphically in Fig. 2.5.

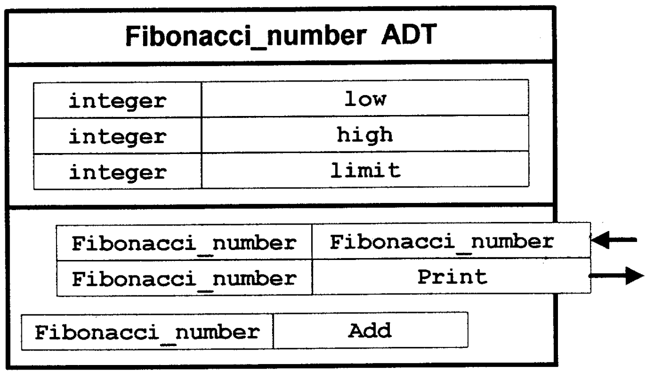

Figure 2.4: Representation of a Fibonacci_Number ADT

2.4 Classes

A class is basically the extension of an ADT by providing additional member routines to serve as *constructors*. Usually those additional members should include a *default constructor* which has no arguments. Its purpose is to assure that the class is created with acceptable default values assigned to all its data attributes. If the data attributes involve the storage of large amounts of data (memory) then one usually also provides a *destructor* member to free up the associated memory when it is no longer needed. F95 has an automatic deallocation feature which is not present in F90 and thus we will often formally deallocate memory associated with data attributes of classes.

As a short example we will consider an extension of the above Fibonacci_Number ADT. The ADT for Fibonacci numbers simply keeps up with three numbers (low, high, and limit). Its intrinsic initializer has the (default) name Fibonacci. We generalize that ADT to a class by adding a constructor named new_Fibonacci_number. The constructor accepts a single number that indicates how many values in the infinite list we wish to see. It is also a default constructor because if we omit the one optional argument it will list a minimum number of terms set in the constructor. The graphical representation of the Fibonacci_Number class extends Fig. 2.4 for its ADT by at least adding one public constructor, called new_Fibonacci_number, as shown in Fig. 2.5. Technically, it is generally accepted that a constructor should only be able to construct a specific object once. This differs from the intrinsic initializer which could be invoked multiple times to assign different values to a single user-defined type. Thus, an additional logical attribute has been added to the previous ADT to allow the constructor, new_Fibonacci_number, to verify that it is being invoked only once for each instance of the class. The coding for this simple class is illustrated in Fig. 2.6. There the access restrictions are given on lines 4, 5, and 7 while the attributes are declared on line 8 and the member functions are given in lines 13-33. The validation program is in lines 36–42, with the results shown as comments at the end (lines 44–48).

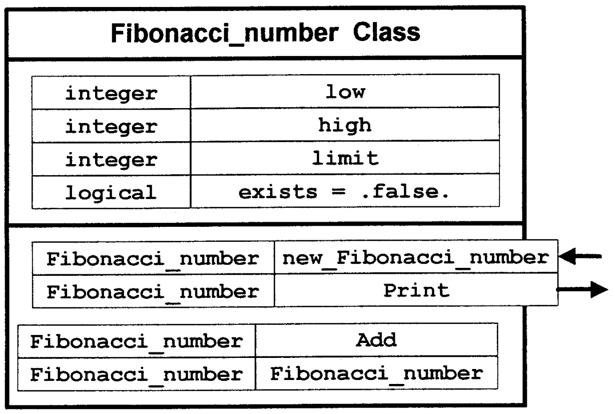

Figure 2.5: Representation of a Fibonacci_Number Class

```
! Fortran 90 OOP to print list of Fibonacci Numbers
Module class_Fibonacci_Number            ! file: Fibonacci_Number.f90
 implicit none
  public  :: Print                       ! member access
  private :: Add                         ! member access
  type Fibonacci_Number                 ! attributes
    private
    integer :: low, high, limit          ! state variables & access
  end type Fibonacci_Number

Contains                                 ! member functionality

  function new_Fibonacci_Number (max)  result (num) ! constructor
  implicit none
    integer, optional        :: max
    type (Fibonacci_Number) :: num
      num = Fibonacci_Number (0, 1, 0)                       ! intrinsic
      if ( present(max) ) num = Fibonacci_Number (0, 1, max) ! intrinsic
      num%exists = .true.
  end function new_Fibonacci_Number

  function Add (this) result (sum)
  implicit none
    type (Fibonacci_Number), intent(in) :: this      ! cannot modify
    integer                            :: sum
      sum = this%low + this%high ; end function add  ! add components

  subroutine Print (num)
  implicit none
    type (Fibonacci_Number), intent(inout) :: num    ! will modify
    integer                               :: j, sum ! loops
      if ( num%limit < 0 ) return                    ! no data to print
      print *, 'M  Fibonacci(M)'                     ! header
      do j = 1, num%limit                            ! loop over range
        sum = Add(num)     ; print *, j, sum         ! sum and print
        num%low = num%high ; num%high = sum          ! update
      end do ; end subroutine Print
End Module class_Fibonacci_Number

program Fibonacci                    !** The main Fibonacci program
implicit none
  use class_Fibonacci_Number         ! inherit variables and members
  integer, parameter       :: end = 8    ! unchangeable
  type (Fibonacci_Number) :: num
    num = new_Fibonacci_Number(end)      ! manual constructor
    call Print (num)                     ! create and print list
end program Fibonacci                    ! Running gives:

!  M  Fibonacci(M)   ; !  M  Fibonacci(M)
!  1 1               ; !  5 8
!  2 2               ; !  6 13
!  3 3               ; !  7 21
!  4 5               ; !  8 34
```

Figure 2.6: A Simple Fibonacci Class

2.5 Exercises

1. Create a module of global constants of common a) physical constants, b) common units conversion factors.

2. Teams in a Sports League compete in matches that result in a tie or a winning and loosing team. When the result is not a tie the status of the teams is updated. The winner is declared better that the looser and better than any team that was previously bettered by the loser. Specify this process by ADTs for the League, Team, and Match. Include a logical member function `is_better_than` which expresses whether a team is better than another.